Asynchronous Queue Machines with Explicit Forwarding

Lennart Beringer

Doctor of Philosophy

School of Informatics

University of Edinburgh

2002

Abstract

We consider computational models motivated by processors which exhibit architectural asynchrony and allow operands to bypass the register bank using a forwarding mechanism. We analyse the interaction between asynchrony and forwarding, derive constraints on the usage of forwarding for various models of operation, and study consequences for compilers targeting such processors.

Our approach to reasoning about processor behaviour is programming language based. We introduce an assembly language in which forwarding is explicitly visible. Operational models corresponding to processor abstractions are expressed as structural operational semantics for this language. The benefits of this approach for defining program execution and for relating processor models formally are demonstrated. Furthermore, we study the restrictions on the class of admissible programs for each operational model. Under our programming language perspective, these constraints are expressed as static semantics and formalised as type systems. Suitability of forwarding schemes for particular models of operation follows from soundness and completeness results which are established by standard programming language proof techniques. Well-typed programs are structurally correct and cannot experience run-time errors due to ill usage of the forwarding mechanism.

Exposing asynchrony and forwarding to the programmer allows a compiler to optimise forwarding behaviour by scheduling operands. We show how program analysis can decide which values to communicate through registers and which ones to forward. The analysis is expressed as a dataflow problem for an intermediate language and is proven sound with respect to a dynamic semantics. Solutions to the dataflow equations yield translations into the assembly language which are functionally faithful to the operational semantics and also structure-preserving as resulting programs are well-typed. The theoretical development of the translation is complemented by a prototypical implementation. Experimental results are included for a symbolic conversion of Java virtual machine code into the intermediate language, indicating that application programs contain sufficient opportunities for forwarding to make our approach viable.

In conclusion, we demonstrate the benefits of a programming language based view for reasoning about programs targeting architectures with asynchrony and forwarding.

Acknowledgements

First and foremost I would like to thank Colin Stirling, whose continuous guidance and support has been invaluable. I admire his ability to foresee the scientific development of my work, and many problems en route could not have been solved without his insight and experience. Colin's commitment and encouragement at times when things did not look so bright is particularly acknowledged.

Secondly, I am grateful to Ian Stark, who joined our weekly meetings during the final year, and whose knowledgeable expertise (not only) on programming language topics guided the development of the later parts of this thesis. Colin and Ian also read several versions of this thesis, and their detailed feedback on content and presentation helped enormously to bring this document into its final shape.

I would like to thank D K Arvind for (twice) attracting me to Edinburgh, and for introducing me to forwarding and asynchrony.

Like many earlier generations of students, I experienced the atmosphere of LFCS and ICSA as extremely stimulating, both scientifically and socially. I am grateful to all their members for their encouragement, and to the secretarial, administrative and technical staff for providing a smooth organisational environment. Of course, my time in Edinburgh would not have been half as enjoyable had it not been for the fellow PhD students, flatmates and friends from EUMS and the resulting numerous social activities and discussions.

Financially, I am grateful to the Division of Informatics for covering my fees, and to the German Academic exchange Service (DAAD) for a one-year scholarship under the HSP III program, reference number D/98/04840. During the last months, I have been employed by the EU-funded research project *Mobile Resource Guarantees* (MRG), and I am indebted to its leaders Don Sannella and Martin Hofmann, as well as all other participants, for giving me the time to finish this thesis.

Paul Taylor's `prooftree` package was used for typesetting proof rules and derivation trees.

I dedicate this thesis to my parents for their continuing support and encouragement.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise, and that this work has not been submitted for any other degree or professional qualification. An earlier version of the material in Chapters 3 and 4 was presented at the Fourth International Symposium on Theoretical Aspects of Computer Software (TACS'01) [Ber01].

Table of Contents

1 Introduction **1**

1.1 Programming language based reasoning 2

1.2 Program analysis for scheduling communication 3

1.3 Summary of contributions 4

1.4 Synopsis 6

2 Background and motivation **9**

2.1 Asynchronous processors 10

2.2 Forwarding and compounding 14

2.2.1 Little or no forwarding 14

2.2.2 Register bypass unit 14

2.2.3 Forwarding in the control unit 15

2.2.4 Compiler-based forwarding 15

2.2.5 Related synchronous architectures 19

2.3 Asynchronous queue machines 20

2.4 Processor design and verification 22

2.5 Programming language based reasoning 24

2.6 Related work 27

2.6.1 Programming language technology and processor design . . . 27

2.6.2 Typing in intermediate and low-level languages 29

2.6.3 Program analysis frameworks 34

2.7 Notation 35

3 Sequential queue machines **39**
3.1 ALEF: a language for explicit forwarding 40
3.2 Dynamic semantics 43
3.3 Static semantics 50
3.4 Principal types 57
3.5 Program execution 60
3.5.1 Dynamic semantics 61
3.5.2 Static semantics 64
3.5.3 Type inference 66
3.6 Discussion 72
3.6.1 Summary and outlook 74

4 Asynchronous queue machines **75**
4.1 Syntax and dynamic semantics 76
4.2 Determinism, safety and completeness 81
4.3 Dealing with non-determinism 86
4.3.1 A static semantics for determinism 87
4.3.2 Pipeline-dependencies 96
4.3.3 Data-dependencies 99
4.3.4 Other sequentialisation mechanisms 105
4.4 Program execution 106
4.4.1 Dynamic semantics 107
4.4.2 Static semantics 108
4.5 Finite operand queues 113
4.5.1 Distributed execution 119
4.6 AQM's and Tomasulo's algorithm 123
4.6.1 Tomasulo's algorithm 123
4.6.2 Mapping to AQM's 127
4.6.3 Limitations 127
4.6.4 Conclusion 130
4.7 Discussion 130
4.7.1 Summary and outlook 133

5 Queue machines with registers **135**
5.1 Sequential execution . . . 137
5.1.1 Dynamic semantics . . . 137
5.1.2 Static semantics . . . 140
5.1.3 Program execution . . . 148
5.2 Distributed execution . . . 158
5.2.1 Dynamic semantics . . . 158
5.2.2 Determinism, safety and completeness . . . 160
5.2.3 A static semantics for determinism . . . 160
5.2.4 Program execution . . . 164
5.3 Discussion . . . 166
5.3.1 AQM's and compounding . . . 167
5.3.2 Conclusion and outlook . . . 169

6 Intermediate program analysis **171**
6.1 Static single assignment . . . 172
6.2 An intermediate language IL . . . 174
6.2.1 Syntax and regular programs . . . 175
6.2.2 Programs in Static Single Assignment form . . . 178
6.2.3 Dynamic semantics . . . 182
6.3 Analysis of variable usage . . . 186
6.3.1 Dataflow analysis for liveness . . . 188
6.3.2 Usage analysis for regular programs . . . 191
6.3.3 Usage analysis for SSA . . . 200
6.4 Soundness of usage analysis . . . 205
6.4.1 Non-standard dynamic semantics . . . 205
6.4.2 Soundness for regular programs . . . 211
6.4.3 Soundness for SSA programs . . . 218
6.4.4 Discussion . . . 224

7 Translating intermediate programs into ALEF **227**
7.1 Deciding forwardability . . . 228

7.2 Eliminating Φ-instructions . . . 230
7.3 Translation . . . 238
7.4 Type-correctness . . . 241
7.5 Allocating operand queues and registers . . . 255
7.6 Functional correctness . . . 259
7.7 Discussion . . . 281
7.7.1 Summary of the compilation . . . 281
7.7.2 Extensions and future work . . . 282
7.7.3 Conclusion . . . 287

8 Implementation **289**
8.1 Analysis of JVM bytecode . . . 290
8.1.1 Java virtual machine code . . . 290
8.1.2 Translating each stack position into a variable . . . 292
8.1.3 Translating each push operation into a variable . . . 297
8.1.4 Implemented instruction set . . . 301
8.2 Results . . . 302
8.2.1 Conversion into IL . . . 302
8.2.2 Translation into ALEF . . . 305
8.3 Discussion . . . 309

9 Discussion **313**
9.1 Summary . . . 313
9.2 Shortcomings and future work . . . 314
9.2.1 Full implementation . . . 314
9.2.2 Extensions of the computational model . . . 315
9.2.3 Connections to linear logic and other intermediate forms . . . 317
9.2.4 Explicitness of architectural features . . . 318
9.2.5 Formalisations . . . 319
9.3 A future application . . . 321

Bibliography **323**

List of Figures

2.1 Sutherland's micro-pipeline 12
2.2 Compoundings for [AM99]'s example program. 16
2.3 Verification against ISA using abstraction functions. 24

3.1 General organisation of processors suitable for ALEF 41
3.2 Visualisation of program (3.1) 43
3.3 Architectural model for sequential dynamic semantics 44
3.4 Dynamic semantics of opcodes 46
3.5 Sequential dynamic semantics of instruction sequences 46
3.6 Example derivation for $\mathsf{C} \xrightarrow{t} \mathsf{D}$. 47
3.7 Proof of determinacy for $t = s\,r$. 49
3.8 Sequential dynamic semantics including jumps 62
3.9 Example derivation for $\mathsf{C} \xrightarrow{t} \mathsf{D}$ including jumps. 62
3.10 Sequential dynamic semantics for programs $\mathtt{P} = (\mathtt{N}, \mathtt{A})$. 63
3.11 Example program for type inference via unification 71

4.1 Architectural model for distributed dynamic semantics 77
4.2 Execution of a non-deterministic program 81
4.3 Execution traces for π_s (left) and π_t (right). 84
4.4 Type system for deterministic execution 88
4.5 Confluence of u and t. 96
4.6 Type system for finite operand queues 115
4.7 Interleavings of program (4.9) for various length restrictions. 120
4.8 Tomasulo's algorithm 124
4.9 Snapshot after issuing instruction $[\mathtt{1}]$. 125

4.10 Snapshot after issuing instructions [1] to [5]. 126
4.11 Snapshot after issuing all instructions of program (4.10). 126
4.12 Example for instruction execution in zero time. 129

5.1 Architecture for sequential dynamic semantics with registers 138
5.2 Sequential execution with registers: dynamic semantics 139
5.3 Sequential dynamic semantics including jumps and registers 149
5.4 Dynamic semantics for programs involving registers 165

6.1 Intermediate language IL: syntax 175
6.2 Relationship between dataflow solutions, $\Rightarrow_\alpha$ and $\rightarrow$ 216

7.1 States Π_i and Δ_i. 235
7.2 Change of $\mathsf{Q}(q)$ when executing $[\![ass]\!]_\rho = \mathtt{id}\, q\, q$. 270
7.3 Compilation from IL to ALEF 281

List of Tables

3.1 Type system 52
3.2 Type system including jumps 64

6.1 Dataflow information for programs (6.5) to (6.8). 197
6.2 Equations (6.13) applied to program (6.5). 198
6.3 Simplification of Table 6.2 199
6.4 Equations (6.13) applied to program (6.6). 199

7.1 Translation $[\![\mathtt{P}]\!]_\rho$ for IL program $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ 239
7.2 Type system for extended instruction set 239

8.1 Operational semantics of some JVML instructions 291
8.2 Translation scheme A for non-jumps 293
8.3 Translation scheme A for jumps 294
8.4 Translation scheme B for non-jumps 297
8.5 Translation scheme B for jumps 298
8.6 Implemented JVML instructions 302
8.7 Linpack results for scheme A: Translation JVM → IL 303
8.8 Linpack results for scheme B: Translation JVM → IL 304
8.9 Linpack results for scheme A: Translation JVM → ALEF 305
8.10 Register/queue access operations (scheme A) 307
8.11 Linpack results for scheme B: Translation JVM → ALEF 308
8.12 Register/queue access operations (scheme B) 310

Chapter 1

Introduction

In this thesis we study computational models of asynchronous processors with operand forwarding. Using this architectural feature, instructions may communicate operands directly by sending them to the functional unit where they will be consumed, bypassing the register bank. In synchronous architectures, forwarding allows one to schedule instructions more aggressively as the number of pipeline-slots separating the producer of an operand from its consumer may be reduced [Fly95]. In these architectures, forwarding may for example be implemented using Tomasulo's algorithm where data is broadcast to all functional units and consumed using a tag-matching operation [Tom67]. In *asynchronous* architectures, broadcasting is undesirable as it globally synchronises operations and couples functional units too tightly. On the other hand, a *distributed* forwarding regime where operands may be sent to specific destinations fits much better with the data-driven nature of asynchronous computation. Consequently, several recent asynchronous architectures explored different forwarding schemes or closely related approaches. However, the interaction between operand forwarding and asynchronous computation is complex and difficult to reason about due to the distribution of asynchronous state. As a result, forwarding schemes are often implemented in an ad-hoc manner without being complemented by detailed studies of the legality of forwarding schemes or the resulting programming model.

1.1 Programming language based reasoning

In order to fill this gap, we propose a programming language based approach where forwarding is explicitly visible to the programmer. While some architectures allow forwarding information to occur in the assembly language in order to expose scheduling opportunities to the compiler, a systematic investigation into the *structural* requirements for code using forwarding has not yet been provided. For example, the SCALP asynchronous architecture admits programs which use the forwarding mechanism in unsafe ways, potentially leading to a deadlock of the processor, operand overflow, operand starvation or non-determinism [End96]. Our programming language approach allows one to eliminate these hazards before runtime by characterising safe code, resulting in *proven* execution safety. Programs which might run into hazards are rejected before an attempt is made to execute them. We achieve the necessary mathematical rigour by following the approach of modern programming language design which complements the syntactic definition of a language by structural descriptions of runtime behaviour and of static program properties. These static and dynamic semantics allow us to reason about program characteristics in an implementation-independent way, to compare the operational behaviour of implementations systematically and to relate syntactically enforceable program properties to runtime behaviour. Often formalised as type systems, static semantics aim to eliminate all dynamic hazards under consideration at compile time, admitting only those programs for execution which could be proven to be free of hazards. As a result, not only is the safety of program execution ensured, but also performance benefits are observed as runtime checks and routines for recovering from hazardous situations need not be implemented.

As a test-bed for reasoning about forwarding schemes we define a small assembly language for explicit forwarding called ALEF. Different dynamic semantics are given to ALEF, corresponding to different abstractions of processor behaviour. The first dynamic semantics models sequential execution similar to conventional descriptions of the instruction set architecture (ISA). Later, we consider (interleaved) parallel execution and execution under limitations on hardware resources. The dynamic semantics are compared to each other formally, and give rise to different error conditions. Each model's characteristic hazards are subsequently eliminated by typing calculi. Soundness results are included, showing that well-typed programs will indeed not exhibit the runtime hazards under consideration.

1.2 Program analysis for scheduling communication

Exposing forwarding to the programmer allows the compiler to schedule the communication by deciding for each operand by which mechanism it should be sent. The architectures we concentrate on implement forwarding using operand *queues* where the head value is removed from the queue during consumption. Hence the compiler's options for scheduling are limited as not all values are suitable for forwarding. In the second part of the thesis we analyse these restrictions and develop the program analysis necessary for detecting the forwardability of values in an intermediate language. We show that the forwarding discipline imposes linearity conditions on the usage of intermediate values which may be expressed using dataflow equations. We consider an operational semantics of the intermediate language which captures the essence of forwarding by restricting reading and writing capabilities of program variables. We then prove the dataflow equations sound with respect to this operational model. The earlier insight into the structural requirements on assembly programs is subsequently exploited as we demonstrate how dataflow solutions in the intermediate language correspond to well-typedness on the assembly level. In particular, we provide a translation from the intermediate language into the assembly language which is both functionally correct and structure preserving, including the allocation of operand queues and registers. We thus demonstrate that forwarding may be lifted from the architectural level to the level of program analysis. This links our work to current research in the programming language community such as typed compilation and typed assembly languages, as well as to recent work on the relationship between different program analysis frameworks [NNH99]. On the other hand, our work shows how programming language concepts may be used to raise the level of reasoning about computer architectures. We expect this to be relevant for other architectures which expose the communication structure to the programmer such as (reconfigurable) field-programmable gate arrays (FPGA's [CH02]), transport-triggered architectures [CM91] and the Raw model of computation [TKM+02]. In all of these models, safety of execution relies on the coordinated usage of the underlying hardware which is susceptible to structural properties of code.

Finally, we provide a prototypical implementation of our compilation approach, consisting of the dataflow analysis in the intermediate language, queue and register allocation for the forwardable and non-forwardable entities respectively, and the translation into ALEF. By linking our implementation to two symbolic conversions of Java virtual machine code [LY97] into our intermediate language, we give experimental results for well-known benchmark programs. This allows us to address the question whether the requirement of linear usage limits the forwarding opportunities to an extent which makes forwarding inviable. Both the conceptual analysis and the experimental results indicate that indeed a huge number of forwarding opportunities exist in application programs as virtually the whole JVM operand stack may be turned into forwarding. Under the assumption that forwarding queues may be built in a more energy-efficient way than large register banks with multiple read/write ports, our work hence provides a means to reduce the power consumption in virtual machine implementations which compile bytecode into machine language [Sun01], [AAB+00]. A particular benefit should result for mobile JVM's running on personal digital assistants (PDA's) or mobile phones where battery lifetime is limited.

1.3 Summary of contributions

On a pragmatic level, our main contribution consists of an argument for a programming language based approach to reasoning about processor architecture, compiler and instruction set design. The need for such a methodology is outlined in Chapter 2, motivated by the complexity of modern system architecture. The key ingredients are operational semantics for describing processor behaviour and static semantics for characterising some program properties. Although these ingredients have been of central importance in the programming language community, their application to processor architectures has to our knowledge not yet been explored.

We exercise our methodology by applying it to the forwarding of operands in processors with instruction-level asynchrony. The second contribution thus consists of a novel abstract machine model, *asynchronous queue machines* (AQM's). AQM's unify existing architectures with respect to forwarding, highlighting their similarities and differences.

On a more technical level, the first part (Chapters 3 to 5) provides

- an assembly language for AQM's called ALEF (Assembly Language for Explicit Forwarding). Exposing forwarding to the programmer/compiler, ALEF facilitates the usage of programming language technology and serves as our main vehicle for analysing forwarding behaviour and linking it to program analysis.

- various dynamic semantics for ALEF, each corresponding to a particular aspect of processor behaviour. These semantics demonstrate the flexibility gained by employing programming language notions for defining processor abstractions uniformly and comparing them formally.

- static semantics which eliminate structurally incorrect programs before runtime. As each dynamic semantics gives rise to characteristic error situations, the benefits of statically enforcable safety conditions become apparent. These conditions are expressed as type systems which are tied to the dynamic semantics by soundness and completeness results.

In order to demonstrate the approach without being overwhelmed by implementation details, we present simplified processor models and a deliberately restricted language. Starting from a sequential model of operation for pure forwarding (sequential AQM's), we add other features (concurrency, hardware limitations, registers) as we proceed. At each step, dynamic and static semantics arise naturally from the corresponding notions of the previous model. This demonstrates the advantages of a structured approach, even more so as proofs of correctness generalise equally well. For example, in the model for concurrent operation (Chapter 4), we distinguish between the derivation of conditions necessary for correct execution and means for discharging such constraints. This separation reappears when registers are introduced in Chapter 5: as the sources of error conditions are generalisations of the ones examined in Chapter 4, the means for discharging the constraints are seen to be correct by invoking the earlier results.

The second part (Chapters 6 to 8) substantiates the claim that a programming language based approach is beneficial for linking processor behaviour to program analysis and compiler optimisation. We provide

- the program analysis for deciding which operand communication mechanism should be used for each intermediate result. The analysis is presented as a dataflow problem for an intermediate language, and is formally proven sound with respect to a non-standard operational semantics (Chapter 6).

- a compilation route from intermediate code into ALEF which is based on solutions to the dataflow equations and includes an allocation phase for registers and queue names. We prove functional correctness by relating the dynamic semantics of the intermediate language to the dynamic semantics of ALEF. The translation is also shown to be structure preserving as each dataflow solution yields ALEF code which is well-typed according to the linear type system of Chapters 3 to 5. We thus lay the foundations for a verifiable compilation from high-level code into ALEF, similar to recent advances in the compilation of functional programming languages (Chapter 7).

- a prototypical implementation of the compilation process (Chapter 8). Although concrete instructions are replaced by symbolic operations, program analysis and translation proceed as outlined in Chapters 6 and 7, starting from Java virtual machine (JVM) code. The resulting ALEF programs indicate the multitude of forwarding opportunities in application programs.

The last item thus provides modest experimental evidence for our approach, complementing the more theoretical justification presented in the preceding chapters.

1.4 Synopsis

The thesis is structured as follows. Chapter 2 describes the background and motivation for this work, summarising trends in (asynchronous) processor architecture, arguing the applicability of programming language concepts, discussing related work, and introducing some notational conventions.

The assembly language for explicit forwarding ALEF is introduced in Chapter 3. It is equipped with an operational semantics where instructions are executed sequentially. In order to understand the basic properties of forwarding in isolation, *all* operands are forced to be communicated via the forwarding mechanism, i.e. no support for registers is provided. The characteristic runtime hazards are discussed, and subsequently eliminated by a static semantics consisting of a type system based on notation from linear logic [Gir86]. Types abstractly characterise the shape of configurations, given by the number of items in each operand queue.

Chapter 4 considers alternative dynamic semantics which are closer to processor implementations. First, we consider a distributed model of execution where instructions await their operands in instruction queues, and head instructions may execute as soon as all their operands have arrived. The resulting out-of-order execution is purely governed by the delays inside the functional units, on which no assumptions are made. Consequently, an additional class of runtime hazards is observed as instructions compete for writing access to operand queues. The dynamic semantics is formally related to the earlier sequential dynamic semantics. Instead of modifying the operational semantics until full compatibility between the two models is achieved we *characterise* the programs for which compatibility holds. This approach is motivated by the desire to constrain the operational behaviour as little as possible, i.e. make no compromises regarding distributed asynchrony. Programs which fulfil the compatibility conditions are considered *safe* for distributed execution, and the earlier typing system is extended to eliminate unsafe programs. As the analysis is split into a phase of *deriving* conditions which must be fulfilled for safe execution and a phase of *discharging* conditions which *are* fulfilled, a scalable approach is taken where more programs may be proven safe at the cost of a more involved program analysis.

Secondly, we consider operational models for implementations where operand queues are of finite length. Again, the resulting restrictions are treated as conditions on the programs, and are indeed shown to be related to the properties necessary for safe distributed execution.

Chapter 5 introduces registers and reexamines the earlier operational models correspondingly. The robustness of our approach is demonstrated as extending the static semantics in the expected way proves sufficient for guaranteeing hazard-free execution. In particular, the characteristic property of registers to allow the repeated consumption of values is reflected in the linear typing calculi by the expected exponentials.

Chapters 6 and 7 describe the program analysis for intermediate code and the translation of intermediate code into ALEF. We introduce a small intermediate language with assignments and branch instructions, and develop the dataflow equations for detecting single usage of program variables. We also consider programs in static single assignment (SSA) form [AWZ88], and show how the dataflow equations may be extended appropriately. In both cases, the dataflow equations are proven sound with respect to a dynamic semantics of the intermediate language which restricts the read/write capabilities of some variables similarly to the low-level forwarding. On the other hand, the semantics are also related to the standard dynamic semantics of the the intermediate language. Chapter 7 presents the translation of programs from the intermediate language into ALEF. We show how each solution to the dataflow equations results in a different ALEF program, and prove that the result is always well-typed with respect to the calculus of Chapter 5. The corresponding formal proof relates intermediate-level structure (dataflow equations) to low-level structure (assembly-level typing calculus), independently of the concrete allocation of operand queues and registers. Subsequently, we treat the *functional* correctness of the translation. We characterise when forwardable entities need to be mapped to different queues, and obtain operand queue allocations by colouring the corresponding conflict graph. The proof of functional correctness finally shows that certain invariants hold during ALEF program execution, corresponding to properties in the execution of the intermediate program.

Chapter 8 presents the experimental results. We discuss the two conversion schemes from JVM code into the intermediate language and compare their effectiveness for transforming operand stack communicated values into forwardable entities, based on the program analysis and the translation introduced in Chapters 6 and 7.

Finally, we conclude in Chapter 9 by discussing shortcomings and possible extensions of our work and outlining some topics for future research.

Chapter 2

Background and motivation

This chapter provides the background and motivation for the work presented in this thesis, outlines our approach, and discusses related work. We touch on a number of areas, ranging from asynchrony and hardware verification to programming language technology and program analysis.

The motivation for our research arises from the desire to reason about operand forwarding in asynchronous processors. Although we are not aiming to justify circuit-level asynchrony from a hardware design perspective, we summarise some arguments for asynchronous operation in Section 2.1 in order to explain the concepts underlying our later computational models. We then discuss the various schemes of forwarding occurring in recent architectures, leading from no forwarding via exclusively hardware-based forwarding to explicit forwarding and compounding (Section 2.2). Armed with this overview, Section 2.3 introduces asynchronous queue machines as a unifying model of computation for globally asynchronous architectures with forwarding. Subsequently, we discuss current approaches to processor verification in Section 2.4. This is followed by motivating why a programming language based approach may be more suitable for understanding forwarding conceptually, complementing the existent verification techniques (Section 2.5). Related work is discussed in Section 2.6, concentrating on applications of structural techniques to hardware design, and of type systems to assembly-level languages and compilation. Finally, Section 2.7 introduces notational conventions employed throughout the thesis.

2.1 Asynchronous processors

The operational models we consider in this thesis are inspired by processor architectures which employ *architectural asynchrony*, i.e. the activities of functional components of a processor are not centrally synchronised by a global clock. Instead, components may execute their tasks independently at different speed. Some effects of architectural asynchrony occur in globally synchronous systems, such as out-of-order execution in super-scalar architectures or multi-processor chips. The archetypical situation for architectural asynchrony however arises if components are clocked at different rates and communicate by handshaking (the locally synchronous, globally asynchronous regime) or if the system clock is removed altogether, resulting in a purely asynchronous regime. In the latter case, execution delays may be influenced by environmental factors such as voltage and temperature, variations during fabrication, or the actual data to be processed.

Both forms of asynchrony are being investigated as potential alternatives to traditional hardware design philosophies, motivated by the growing design complexity for globally synchronous systems. Advocates such as [BS95], [DN97], [Hau95] and [Man00] argue that a number of technological drawbacks found in conventional design methodologies may be overcome using global asynchrony:

Cost of computation versus cost of communication. As the integration density continues to increase, the relative cost of communicating values (wiring) dominates the cost of actually performing computation. The absence of a central clock removes the need to communicate clock signals and thus decreases the chip area needed for routing.

Absence of clock skew. Ensuring that the clock signal arrives at all components of a chip at the same time is one of the most time-consuming tasks during chip design. Timing closure is often achieved by using gated clocks and clock trees, with negative effects on power consumption or die size. Asynchronous systems do not rely on the arrival of a signal at a gate at a globally determined point in time, so the clock skew problem does not arise.

Average case performance. The clock speed of a synchronous system is determined by most pessimistic assumptions on the speed of individual components, propagation delays, incoming data and environmental conditions. Unless the clock speed can be adapted at runtime, presence of more favourable situations cannot be exploited, and time and energy are wasted. Asynchronous systems often allow a computation to proceed at its natural speed and can thus adapt to various runtime conditions, resulting in expected average-case rather than worst-case performance.

Modularity and modifiability. As decoupled components do not rely on global timing assumptions, they can be designed in a more modular way. The performance of a design can be improved more easily by replacing bottleneck components by improved implementations, without having to reevaluate the timing constraints or re-routing other areas of the chip. Consequently, the life-time of a design is extended, leading to a more favourable cost-per-design ratio.

Power consumption. Although modern synchronous systems switch off or slow down unused parts of a chip when no useful work is performed, the overhead of the corresponding regulation mechanisms disappears if the activity of gates is determined locally, as is the case in globally asynchronous systems.

Security. By probing and observing characteristic current spikes, an outside attacker can identify executed instructions and analyse the data of a computation. Successful experiments on retrieving the complete keys of commonly used smart-card security protocols (RSA, DES) by probing the electro-magnetic (EM) characteristics have been performed [GMO01]. It is argued in [MAK00] that asynchronous systems blur patterns in the curves of current or EM radiation and thus help inhibiting attacks based on power analysis. Furthermore, the vulnerability of processors to modulation of clock and power by applying additional external signals and thus changing the processor's behaviour is reduced under the asynchronous regime.

Most of these advocated advantages have been validated only in isolation or for special-purpose applications, and are counterbalanced by increased transistor counts, negatively effecting die area, power consumption and design complexity. In addition,

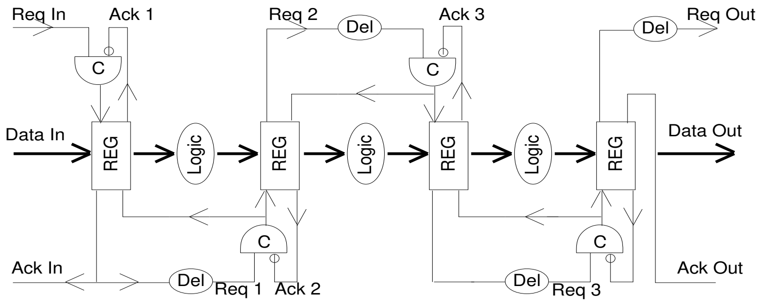


Figure 2.1: Sutherland's micro-pipeline

asynchronous systems need non-standard encoding techniques for both signals (handshaking protocols, transition signalling) and data (1-out-of-*n*, ...) and are difficult to verify and test [LGS95][HBB94]. For these reasons, most mainstream hardware design is done using synchronous design styles.

Nevertheless, circuit-level design of asynchronous systems has improved over the last decade with the advent of specific design methodologies. These are often based on concurrency formalisms such as CSP [Mar86] [Bur87], CCS [LBM96] [BM00] or Petri nets [SKLY97] [YGL00] [CM99a] and have partially resulted in design and synthesis tools. Some of these formal methods have been validated during the design of full processors [MLM^{+}97] [DGY93] or parts thereof [MLM99] [CM99a].

On a higher level, novel processor architectures have been proposed [Pav94] [GFC99] [RB95] [SSM94] [Mul01] [OIU^{+}01] [WA01], most of which are based on Sutherland's micro-pipeline [Sut89].

This basic processing structure (see Figure 2.1) consists of a series of storage elements realised by asynchronous composition and controlled by specific logic gates (Muller C gates). In the absence of circuitry between the stages, the micro-pipeline acts as an *elastic* FIFO queue as the number of values held in the queue may vary dynamically. If logic functionality is inserted between the stages, the micro-pipeline acts like a conventional processor pipeline, but delays are determined dynamically. A functional component may start to process data incoming from its left neighbour as soon as it has communicated the previous result to the neighbour to its right. If some stages are left without circuitry, an elastic processing pipeline is obtained holding a varying number of items. The actions of each stage are governed by local constraints such as the availability of data, and buffering between the stages decouples the progress in adjacent functional units, resulting in a self-adapting throughput. For example, a long latency for a specific instruction in the first pipeline stage may be compensated by short latency for the same instruction in the second stage. In contrast, a synchronous pipeline would artificially extend the latency of the second stage to match the latency of the first stage, resulting in a reduced throughput. Concerning the *design complexity*, it could be argued that synchronous design violates one of the principles of modern processor design outlined in [HP96]: to make the frequent case fast and the infrequent cases (at least) correct. Assuming that the above latency behaviour occurs for a small percentage of instructions, design time and hardware spent on improving the speed of the first stage will never pay off, but leaving it unimproved wastes runtime performance in the common case.

The processors described in [SSM94] and [WA01] use counterflow pipelines where instructions and values travel in opposite directions. The rotary pipeline processor [MRW96] employs a circular datapath which values traverse clock-wise, visiting the respective functional units. Most other architectures mentioned above take a less radical approach and stick to the conventional order of functional stages as found in a RISC pipeline or employ a super-scalar approach by parallelising either functional units in the execution stage or full pipelines. The resulting out-of-order execution is often complemented by write-back stages containing reordering mechanisms, allowing precise exception handling and hiding execution delays from the programmer. Most of these architectures also employ some form of last result reuse, register bypassing or operand forwarding as outlined in the next section.

A recent article motivating architectural asynchrony in the context of dataflow and multi-threaded architectures can be found in [SRU01].

Asynchrony has also been proposed as a feature of biologically inspired reconfigurable electronic systems in the area of embryonics [JT01].

2.2 Forwarding and compounding

We concentrate our analysis on the high-level communication of operands between instructions. In traditional (synchronous) ISA's, all operands are communicated through registers or memory [HP96]. Modern pipelined processor implementations however use an additional mechanism called *operand forwarding* or *register bypassing* as a fast mechanism to communicate operands. Avoiding the latency of register communication, the distance between producer and consumer in the pipeline can be reduced and data-dependent instructions can be executed in close succession [HP96] [Fly95]. As a further consequence the demand on the register bank to provide instructions with operands is reduced. As more instruction level parallelism (ILP) is exploited, this allows one to use register files with less read/write-ports, reducing hardware complexity and processor area. Often, either the control unit or the register bank contain special hardware to detect forwarding opportunities, based on the Tomasulo-algorithm [Tom67], score-boards [Tho64] or bypassing logic.
In the context of asynchrony, forwarding is a natural means to capture the data-driven aspect of computation, and asynchronous architectures have consequently explored a wide spectrum of forwarding techniques.

2.2.1 Little or no forwarding

Most early asynchronous architectures did not implement any means of data forwarding [DGY93] or relied on register locking (Amulet1, [Pav94]), scoreboards [RB95], local reuse of a previously computed result inside a functional unit [FDG+96], [RB95] and reuse of values previously loaded from memory [FDG+96]. The recent DCAP-architecture [WA01] concentrates on the implementation of dynamic scheduling, does not include any forwarding and employs register renaming and other techniques in the control unit to resolve data dependencies.

2.2.2 Register bypass unit

A hardware-based solution consists of a bypass unit in front of the register bank which holds recent results and serves all operand requests for values it contains. Such a solution is employed in [MLM+97], but might not scale well as the number of operand requests grows. The design of [MLM+97] also presents a simple mechanism for re-ordering the instructions before the write-back stage: during (in-order) instruction issue, a tag is entered into a FIFO queue which allows the write-back stage to poll the functional units for results in the same order in which the instructions were issued. A number of low-level design decisions were explored during the development reported in [MLM+97], and a CSP-like formalism was employed.

2.2.3 Forwarding in the control unit

Under this regime, the control or instruction issue unit identifies opportunities for forwarding when decoding an instruction. Based on an analysis of data-dependencies, forwarding requests or bypassing tags are generated dynamically. An improved queue implementation proposed in [GG97] reducing power consumption and easing reading and writing access was implemented in Amulet3 [GFC99]. The Cascade-ALU [OIU+01] and some schemes in [Mul01] also employ control-unit based forwarding which has the advantage of finding more forwarding opportunities than static, compiler-based methods. The downside is runtime overhead in terms of speed and power consumption as additional hardware for detecting dependencies and avoiding deadlocks is employed.

2.2.4 Compiler-based forwarding

Several authors proposed to include forwarding information in the instruction code, either explicitly [End96] or by annotations [Mul01] or special interpretations of register names [RB95]. In all of these cases, the responsibility to schedule operand forwarding lies with the compiler, and consequently the hardware is vulnerable to compiler-errors leading to unsafe, deadlocking or functionally incorrect usage of the forwarding mechanism.

In FRED [RB96] [Ric96], the register name `r1` carries a special meaning by allowing access to a queue of values, which delivers a different value each time it is accessed. Execution of branches is split into an evaluation of the branch condition and an operation reading the branch outcome and performing the jump. Although the mechanism

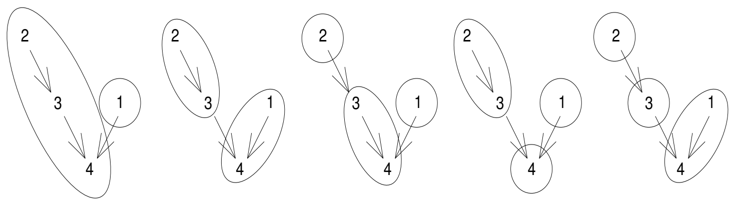


Figure 2.2: Compoundings for [AM99]'s example program.

was considered interesting enough to be included in FRED, it was not given a prominent status in the further development and an evaluation of the usage of this mechanism is not provided.

2.2.4.1 Compounding

The *compounding* approach [AM00] [AM99] [Mul01] identifies forwarding opportunities in the data-dependence graph of programs by grouping adjacent instructions. For example, the program

```
(1) r5 = mem[r2+4]
(2) r1 = mem[r2]
(3) r2 = r1 * 321
(4) r3 = r2 + r5
```

(taken from [AM99]) can be compounded as indicated in Figure 2.2. At runtime, encircled data-dependencies result in forwarding and cross-compound dependencies in register usage. A single bit in the instruction code suffices to indicate compound membership provided that only syntactically neighbouring instructions can inhabit the same compound – the second and fifth compoundings in Figure 2.2 violate this requirement. If instructions may be reordered during compilation, arbitrary adjacency relations may thus result in forwarding, as long as each compound is a *linear* chain of instructions and the graph of compound instructions and inter-compound dependencies is acyclic. [Mul01] proposed a number of implementations for dynamic scheduling and forwarding, and explored their effect on overall performance for varying latencies of functional units and registers. The motivation for our investigation into forwarding grew from the desire to characterise legal compoundings more abstractly, allow non-linear shapes, and extend compounding to forwarding across basic block boundaries.

2.2.4.2 SCALP

In SCALP [End96], identifiers of input ports of functional units may appear in the opcode, as in

```
add -> mul_b
```

This instruction would add the two values at the (implicit) input ports `a` and `b` of the functional unit `ALU` and send the result to port `b` of functional unit `MUL`. Incorporating the destination of forwarded values explicitly was motivated by the desire for high power efficiency, realised by high code density and asynchronous parallelism. The resulting paradigm is more flexible than simple compounding as the need to restrict forwarding to neighbouring instructions is removed. Consider for example the program

```
(1) ld 4 -> mul_a;
(2) ld 2 -> alu_a;
(3) ld 3 -> mul_b;
(4) inc -> mem_a;
(5) mul -> mem_b
(6) st
```

where instruction `(4)` consumes the result of instruction `(2)`. As input ports are generalised to operand queues, functional units become decoupled, and availability of several input ports at functional units enables compounds where forwarding chains merge. For example, the multiplication instruction `(5)` in the above program obtains operands from `(1)` and `(3)`. On the other hand, introduction of a *duplication* instruction where a value is sent to two destinations allows us to implement branching compounds. In the program

```
(1) ld 2 -> alu_a;
(2) ld 4 -> mul_a;
(3) dupl -> mem_a, mul_b;
(4) mul -> mem_b;
(5) st
```

the duplication instruction for the functional unit `ALU` duplicates the value `2`, sending it to `mem_a` and `mul_b`.

Finally, forwarding across basic block boundaries becomes possible as shown in the program

```
(1) ld 2 -> mul_a;
(2) ld 4 -> alu_a;
(3) ld 7 -> mul_b;
(4) mul -> mul_a;
(5) dec -> alu_a;
(6) dupl -> bu_a alu_a;
(7) ifz (3) (8);
(8) . . .
```

Together, these generalisations allow more generous forwarding patterns than linear compounds, paid for by fixing the syntactic order of instructions when compared to the instruction dependence graph and by a more costly encoding of forwarding information (names of input ports instead of a single bit).

The specific implementation pursued in SCALP did not fulfil the motivating expectations of low power consumption and high code density. Architecturally, one of the reasons appears to be the decision to model the register bank as another functional unit, which requires additional transfer instructions for each register access. From our point of view, a more important question left unanswered in [End96] is about the programming model resulting from explicit forwarding. As the structure of the implementing hardware becomes visible to the programmer, the processor is sensitive to fine-grained properties of compiled code. The discussion in [End96] lists a number of illegal programs which should not be executed because they would run into hazards such as non-determinism, deadlock or queue over- or underflow. For some hazards, ad-hoc solutions are proposed such as the existence of sequentialisation instructions, while other errors are essentially assumed not be present in programs. Given that explicit forwarding admits more generous forwarding schemes than compounding, structural incorrectness of code becomes difficult to detect. If runtime hazards cannot be tolerated, techniques are consequently needed for reasoning about the legality of forwarding schemes.

2.2.5 Related synchronous architectures

We briefly mention two architectural approaches from the synchronous processor design community related to forwarding.

2.2.5.1 Register files with queues in VLIW processors

Motivated by the increase of hardware complexity and processor area of traditional register files in VLIW architectures, queue register files have been proposed in [FLT97], with further exploration reported in [FLT98]. The results presented indicate the competitiveness of operand queues as the architecture is scaled, provided the right organisational structure is chosen. As mapping of instructions to functional units in VLIW processors is performed by the compiler and legality of operand queue usage depends on this mapping, the authors present a condition when instructions may share an operand queue. This appears to be similar to the task we are considering in Chapter 4, but the solution provided cannot be transferred to our models of computation as VLIW architectures behave highly synchronously. Indeed, scheduling for VLIW architectures is based on a detailed model of instruction delays, given by the number of cycles required for a particular instruction and functional unit. The success of the analysis presented in [FLT97] thus depends on the ability to characterise the availability of operands and functional units at particular cycle times.

2.2.5.2 Transport-triggered architectures

In *transport-triggered architectures* (TTA, [CM91]), only the movement of operands is explicitly specified by instructions, while the operations themselves are implicitly triggered by the availability of operands at input ports of functional units. The following example program taken from [MHC97]

$$
\begin{array}{lcl}
(\mathtt{CNST\text{-}10{:}\ ADD}^{\mathtt{O}} & \| & \mathtt{CNST\text{-}20{:}\ ADD}^{\mathtt{T}}) \\
(\mathtt{ADD}^{\mathtt{R}}\mathtt{{:}\ ADD}^{\mathtt{O}} & \| & \mathtt{CNST\text{-}30{:}\ ADD}^{\mathtt{T}}) \\
(\mathtt{ADD}^{\mathtt{R}}\mathtt{{:}\ REG}^{\mathtt{T}} & \| & \mathtt{NOP})
\end{array}
$$

is an optimised program for a clocked architecture where the data bus can transfer two values in each cycle. The first pair of instructions delivers two constant values to the input ports `O` and `T` of functional unit `ADD`. In the second cycle, their sum is transferred from the result port `R` of `ADD` back to the input port `O` and a third constant is delivered to port `T`. The third pair of instructions writes the overall sum to a register.

Realised so far only in synchronous hardware, the programming paradigm offered by TTA is radically different from that of conventional architectures. For deadlock-free scheduling of TTA programs, reasoning mechanisms for structural properties of TTA code would be beneficial, but have apparently not been developed so far.

2.3 Asynchronous queue machines

Despite the differences between the implementations of forwarding, correct execution in all architectures mentioned relies on an understanding of the interaction between operand communication and concurrent instruction execution. In order to study these interactions systematically, models of computation are necessary which abstract from architectural details, concentrating on the relevant issues. This thesis proposes *asynchronous queue machines* (AQM's) as a family of such models, and studies their suitability for the task at hand. AQM's consist of a set of functional units located in parallel to each other, where instructions exchange operands by inserting them into (and reading them out of) *operand queues*. Different functional units may implement various sets of operations and operate independently of each other. Internally, they may be realised synchronously or asynchronously, and the outside environment may not make any assumptions regarding their latency, apart from finiteness. Operand queues are expected to be FIFO-queues, and architectures subscribing to the AQM regime may impose additional disciplines on the way operand queues may be used. For example, a processor may restrict the capability to read from/write to an operand queue to instructions executing on a particular functional unit, or may complement the operand queues by a set of registers. In fact, the queue machines considered in this thesis implement operand access destructively, i.e. the head value of an operand queue is deleted from the queue during its consumption.

Studying AQM's amounts to reasoning about processor behaviour and program execution under unknown latencies of functional units, with particular emphasis being paid

on the operand queues and their usage by an application program. The granularity of models of processor behaviour and the way in which a program uses the forwarding mechanism influences the resulting properties. One way to deal with this would be to fix a particular model of operation and a certain forwarding policy (such as an asynchronous implementation of a score-board) and to verify properties of the resulting architecture. Instead, we aim at reasoning about *forwarding schemes*, i.e. we develop conditions on forwarding policies. Verification under this perspective consequently means to show that any program which respects a particular forwarding policy fulfils the runtime properties under consideration.
We argue that such a general approach is beneficial for understanding the interaction between forwarding and asynchronous operation, for the following reasons.

- Firstly, our results are not specifically tied to any particular algorithm for deciding where forwarding should be used. In fact, we are not concerned with the task to implement or verify any such algorithm but concentrate on the resulting forwarding scheme directly. Indeed, we argue that once one understands the constraints on forwarding, one may compare the efficiency of various hardware- or software-based algorithms systematically, without having to verify each implementation individually. For example, the task to allocate operand queues to forwarded values may be performed by the processor (for example in the control unit) or the compiler (under knowledge of the hardware resources), but this design decision is independent from the legality of the resulting forwarding scheme.

- Secondly, we are able to study several models of processor behaviour and to compare the resulting restrictions on forwarding schemes. This enables system-level engineers to study design alternatives using a consistent formalism for various processor models. Reasoning about the consequences on forwarding (and other aspects) may thus influence design decisions.

- Thirdly, as no assumptions are made regarding the delays of functional units, large parts of our analysis remain valid when functional components are upgraded or functionality is added.

In this thesis, we consider various models of AQM operation and explore their forwarding constraints. We study sequential instruction execution as is captured by an instruction set architecture description as well as distributed instruction execution as found in (idealised) super-scalar implementations. In addition, we consider AQM's with and without additional registers, and with operand queues of infinite and finite length.

Each of these operational models gives rise to specific conditions on the way forwarding is used, which manifest themselves in characteristic classes of runtime hazards. These are similar to error conditions occuring in SCALP or situations where forwarding has to be cancelled dynamically in the compounding approach. This demonstrates that our approach is useful in highlighting common properties of these architectures.

Before outlining our approach to verifying forwarding schemes in more detail, we discuss some aspects of processor verification.

2.4 Processor design and verification

The development of hardware architectures and the subsequent implementation of compilers and processors centers around the instruction set architecture (ISA). This interface between hard- and software defines the set of available instructions, their encodings, types and sizes of operands, and memory addressing schemes. Modern approaches [HP96] advocate to evaluate instruction set decisions by quantitative simulation of benchmark and application programs. During the validation of an architecture, this approach is useful for performance optimisation and testing. However, different concepts are needed to verify functional correctness, and processor verification is consequently a well established area of research. Indeed, techniques such as abstract state machines [CCL+97], (higher-order) logics [Mel88], theorem proving [CRSS94], model checking [BCL+94] [VBF+97] and symbolic simulation [SB93] have been successfully applied on various levels of abstraction and represent the most prominent application area of formal methods. For a comparison of some of these techniques, see [Seg93] [Kro97].

Hardware verification has traditionally been separated into verification of the data path and that of the control logic. Theorem proving and (higher-order) logics have been particularly successful in the former area while control issues have traditionally been tackled using model checking approaches. One of the reasons for this separation is the high regularity of control logic, enabling a fully mechanised exploration of the state space. The size of the state space may be reduced by symbolic techniques or compositional methods. In contrast, theorem proving usually involves considerable manual intervention. A variety of application problems has been successfully tackled, such as algorithms of Intel's PentiumPro's floating point instructions [OZGS99] or asynchronous micro-pipelines [BFGW97]. Recently, several proposals for merging the strengths of theorem proving and model checking have been put forward [Uri00] [Amj01] [SS99]. Symbolic simulation [SB93] arises from model checking by factoring the state space using data abstraction and restricting the expressiveness of the logic, such that a property may be proven by a simulation over the abstracted system.

At the architectural level, Burch and Dill [BD94] introduced abstraction functions between states of an implementation and ISA states. These are used in correctness proofs by showing that diagrams between the ISA level specification and the microprocessor implementation commute (see Figure 2.3). In order to verify a single instruction step, the processor is virtually stalled by *flushing* the pipeline: a sequence of micro-steps is considered (horizontal steps in Figure 2.3) where no issuing takes place but currently executing instructions may complete. Correctness is obtained if the two compositions of flushing and issuing one instruction (vertical steps) commute. By employing uninterpreted function symbols instead of concrete representations, the verification complexity is reduced. Burch and Dill's approach was extended and modified by several groups who generate abstraction functions automatically, define completion functions for unfinished instructions and improve mechanisation [Cyr93] [SRC97] [SJD98] [SM95] [HSG98] [HSG99]. These improvements enabled the verification of (synchronous) microprocessors with realistic features such as pipelining, out-of-order execution and branch prediction. Recent combinations with model checking aim at improved automatic abstraction functions, compositional reduction and induction over cycle time [BBCZ98], [JM01].

Damm and Pnueli proposed an alternative approach for verifying out-of-order exe-

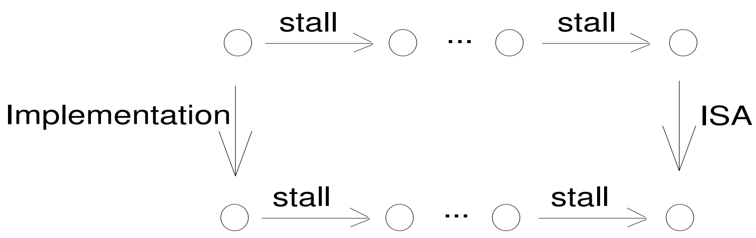


Figure 2.3: Verification against ISA using abstraction functions.

cutions using synchronous transition systems [DP97]. Interpretative approaches have been proposed [Moo98] and were applied to commercial microprocessors [BKM96].

2.5 Programming language based reasoning

The above techniques are very well suited for verifying that a processor implementation satisfies the specification of a processor. In particular, the approach using abstraction functions (Figure 2.3) ensures correctness of pipelined architectures with respect to the sequential ISA semantics. This method can consequently be used for verifying forwarding algorithms implemented in hardware. In fact, [BBCZ98] and [HGS99] use Tomasulo's algorithm to demonstrate their method. The correctness resulting from Figure 2.3 is indeed strong: for all application programs, the processor implementation (or its abstraction as formalised in the small-step transition system) shows the same observable behaviour as its ISA specification.

In this thesis however, we do not aim at proving such a strong correctness result for a particular processor implementation, but wish to reason about forwarding schemes more generally. In particular, the above verification method appears not applicable to the compiler-based approaches to forwarding, as correctness of execution depends as much on properties of the program as it does on properties of the processor. In order to give structural properties of a program a more prominent status, we advocate an approach based on programming language technology, with the following ingredients. First, we make forwarding explicitly visible at the ISA level by including operand queues in the syntax of the assembly language. This turns forwarding from a processor-internal feature into a programmable mechanism to schedule communication. Logically extending SCALP's concept of explicit forwarding, we make both the destination *and the source* operand queue(s) of an instruction visible and give operand queue names the same status as register names.

Second, the explicitness of forwarding in the programming language enables the application of reasoning techniques from the programming language design community. Models of processor behaviour may be expressed as dynamic semantics for the assembly language, as demonstrated by models for sequential (ISA-like) and distributed (super-scalar-like) execution. Indeed, as the dynamic semantics will be given in structural operational semantics (SOS) form [Plo81], we relate processor models structurally and prove properties by structural induction. At first sight, the presence of *several* dynamic semantics might appear to deviate from the usage of SOS in programming language design, where a single formal definition is aimed at [MTHM97]. However, the process of evaluating design alternatives with respect to compilation, hardware implementation and runtime performance should not be compared with a final programming language, but with the process of *designing* one. Indeed, exploration of design alternatives regarding programming language constructs involves a number of dynamic semantics and occasionally several experimental languages.

The benefit of our approach becomes apparent in the third step as we express forwarding schemes as program properties. Instead of *verifying processor behaviour*, we *classify the programs* according to which runtime behaviour they exhibit. By complementing the dynamic semantics by static semantics (type systems), a better understanding of the interplay between operational model and forwarding is obtained. As the runtime hazards associated with the respective operational models manifest themselves as properties of the corresponding SOS systems (or incompatibilities between such systems), we design type systems for characterising programs which are safe for execution: a program which passes the type checker is guaranteed not to experience the runtime hazard under consideration. Thanks to the structural formalisation of dynamic and static semantics, the soundness of the typing calculi may be proven formally, using well-established proof methods such as structural induction [Bur69].

The separation between static and dynamic semantics meets our earlier requirement that the *implementation* of a forwarding algorithm and the *legality* of the resulting forwarding scheme be separated. Indeed, while our calculi can also be employed for type inference, the decision how operand queues are used may also be made elsewhere. If our language with operand queues is chosen as the visible assembly language, we may in fact accept code from arbitrary sources, provided that we type-check it before execution. Alternatively, it is possible to understand the typing calculi as a specification of a hardware implementation of forwarding. For example, a forwarding implementation in the control unit may be considered correct if its output (the program as issued to the respective functional units) is well-typed. Although this aspect is not explored further in this thesis, we expect that it allows one to partition the verification of processors with hardware forwarding into a functional verification of the algorithm implemented in the control unit and an invocation of our results for the operational model realised in the data path.

The fourth aspect of our approach consists of its mediating role linking architecture and compilation. This benefit is demonstrated in the second part of this dissertation, where we show how software-based forwarding may be implemented by program analysis, formalised as a dataflow problem. While in general the output of such a compiler may be treated like any other program by type-checking it before execution, our earlier insight into forwarding is employed for eliminating this check. Indeed, code emitted from a compiler using our forwardability analysis is guaranteed to be well-typed and hence structurally safe for execution.

Finally, the fifth aspect of programming language technology consists in the ability to modify operational models and typing calculi quickly, and to extend them with annotations. This is of particular interest during the system design phase when configurational choices need to be explored. For example, by extending an SOS system with cost models comprising, say, measures for energy consumption in operand queues and registers, performance may be evaluated as required by [HP96]. A system designer may thus explore various operand communication schemes and optimise the corresponding scheduling algorithms using a consistent framework with solid formal foundations.

2.6 Related work

We now discuss related work on applying programming language concepts to hardware design, and on types in compilation and low-level languages.

2.6.1 Programming language technology and processor design

Immediately after the inception of structural operational semantics, Cardelli [Car82] explored its applicability to hardware description, layout, and VLSI design using algebraic techniques. Most of this work is concerned with lower level abstractions, and with the verification of properties regarding layout and timing. Since then, applications of structural operational semantics in the area of hardware have mostly concentrated on description languages such as ELLA [Goo93], VHDL [vT93], [TE01], Verilog [BJQ00] or Esterel [BG92] by following [Plo81]'s general approach of using SOS for defining the meaning of languages. As is the case for higher programming languages [Win93], SOS and typing turn out to be effective for defining hardware description languages precisely, reasoning about their properties and uncovering errors and ambiguities of earlier (and usually informal) descriptions. The resulting semantics may hence serve as specifications for hardware development tools such as simulators and synthesisers. While it is possible to obtain a precise description of a processor architecture by specifying its behaviour in a hardware description language, the programming language based reasoning we advocate acts on the meta-level of such a formalisation. In particular, abstract entities such as register or queue names, assembly instructions or operands are only available implicitly, rather than syntactically. The above work does therefore not solve the task we set.

Term rewriting systems (TRS's) were used by Shen, Arvind et al. in a series of papers and technical reports for designing, verifying and comparing processor models with various architectural features including super-scalar and out-of-order execution [SA98b] [SA98a] [PHA98] [AS99]. This formalism expresses the above abstract entities explicitly and hence shares the advantages of our method with respect to the first two aspects mentioned in the previous section. On the other hand, the proof method consists of showing a confluence result similar to the one of Figure 2.3. Thus, the cited work still aims at verifying hardware behaviour rather than classifying programs and reasoning about their execution. Consequently, the later steps (in particular complementing the dynamic semantics by static semantics and providing a link to compile-time analysis) are not provided. On the other hand, the authors report in [HA99] how the TRS descriptions can be transformed for generating hardware. This is a task we have not attempted, but we expect that the process could also be carried out starting from SOS descriptions.

The structured operational description of processor behaviour most related to our approach is represented by [MHC97]. This work gives an SOS description of transport-triggered architectures where the structure of the semantics follows the structure of the architecture and the hierarchy of entities is mirrored in the syntax of instructions. In contrast to VLIW architectures where the width of an instruction is determined by the number (and kinds of) functional units, instructions in transport-triggered architectures are parallel compositions of (conditional) *moves* and consist of as many parallel components as there are transport buses. For example, the program we gave in Section 2.2.5.2 is suitable for an architecture with two transport buses. Individual move operations are composed of a guard (for implementing conditional instructions) and one source and target socket each. Source sockets may be constant values or output ports of functional units while target sockets may be data or trigger input ports of functional units, where a value at a port of the latter sort may be a data item or a trigger for initiating the execution of an operation. Semantically, the effect of a move consists of transferring a value from the socket indicated in its source component to the one given in the target component, provided that the guard evaluates to true. Only a single dynamic semantics is given in [MHC97], with execution modelled as follows. An individual *move* operation consists of two phases. A *source* phase executes the operand access as required by the source component, resulting in the corresponding value. The second step combines this value with the current state, yielding a substitution which indicates the necessary update to the state, as required by the target component. An *instruction* is executed by performing the constituent move operations concurrently and then applying the resulting substitutions to the initial state. A *program* execution step first executes the instruction pointed to by the program counter (which is subse-

quently increased), and then performs additional state modification corresponding to the functional operations inside the functional units.
The semantics is illustrated using some small example programs, and some initial properties are proven by giving the derivations trees. Regarding well-definedness of the substitutions, the authors note that conflicts may arise if several updates to a particular socket are required in the same cycle. These conflicts are in fact similar to one class of errors we treat in Chapter 4, with the difference that conflicts in TTA's can only occur between components of the same instruction as operation is synchronous and cycle-based. The authors remark that code legality hence relies on the ability of the compiler to schedule code in a way which avoids conflicts and mention the possibility to enforce this property using a static semantics. However, no details are given in [MHC97], and the topic was apparently not pursued any further. Likewise, the ability of a static semantics to enforce correctness in the case of a non-fully connective move network is mentioned, but not elaborated on.

2.6.2 Typing in intermediate and low-level languages

Static type systems for ensuring safety of execution as early as possible have recently found their way from high-level programming languages to intermediate and low-level languages.

2.6.2.1 Types in compilation (TIC)

Inclusion of types during the compilation process (TIC) has been advocated and explored in the FOX and FLINT projects [TMC$^+$96] [PCHS00] [SA95] [Sha01]. While the FOX work, as well as [SA95], concentrates on compilation of functional languages (in particular, ML), the FLINT project [Sha01] aims more generally to provide a common, typed intermediate language for various higher-order, typed languages, including ML, Haskell and Java. In both cases, typed intermediate languages preserve high-level typing information throughout the compilation (including optimisation phases), with additional type information resulting from program analysis. As our program analysis is formalised in terms of dataflow equations, the typed-intermediate-languages concept does not apply directly. However, in the light of similarities between dataflow and typing (see below) it may be possible to transfer our analysis to a typed framework. In particular, this might allow one to recast our compilation into assembly code with the conversion of dataflow properties to typing statements as a type-preserving translation. We will briefly return to this point in the discussion.

2.6.2.2 Typed assembly languages (TAL)

On even lower levels, typed assembly languages have been designed which enforce well-behavedness of code with respect to heap allocation, data layout, runtime stack and type-safe operand behaviour [CM99b] [AC01]. The TAL approach annotates assembly code with information for tracking the types of register contents and heap locations, including initialisation [CM99b]. In particular, by using polymorphic types, one may type code which is applied to data of various shapes. The concept was extended to stack-based languages in [MCGW98], with subsequent application to an Intel-X86-like architecture [MCG$^+$99]. [MWCG98] showed how TAL may serve as the target language of a typed compilation, consisting of a number of (type-preserving) compilation steps. This work has demonstrated the applicability of typing at low levels, and we see the processor-architecture-based motivation as complementing it. In particular, we expect that aspects of polymorphism may be useful to our work when including code blocks such as procedures or function calls.

Xi and Harper's DTAL extends TAL by a limited form of dependent types, allowing dynamic array bounds checks and tag checks for sum types to be eliminated [XH01].

As all typed-assembly-language work is approached from the compiler's point of view, it is not surprising that all only single dynamic semantics are considered. The usage of several dynamic semantics motivated by our processor-architecture driven viewpoint thus adds a new facet to typing of assembly languages adds further motivation to the general concept.

2.6.2.3 Typing of Java bytecode

Typing in low-level languages received a further stimulus from the introduction of the Java virtual machine (JVM). As the safety of Java virtual machine language (JVML) programs represents a cornerstone of the JVM's architecture, types have been used for verifying operand stack behaviour, object initialisation, nesting of subroutines and method invocation [SA98c] [FM98] [FM99] [HT98]. Processor-architecture-inspired AQM's differ from the abstract machine model of the JVM in various respects, such as the differences between a *single stack* and *distributed queues* and single and multiple dynamic semantics. However, general similarities exists (as exemplified by our implementation in Chapter 8), and the treatment of composition of typing statements for basic blocks in Chapter 3 was in fact motivated by [SA98c].

2.6.2.4 Ohori's assembly-level Curry-Howard isomorphism

As a foundation of typed compilation, Ohori [Oho99a] presents an assembly-level isomorphism between typed low-level languages and formal proof systems. The low-level language is that of A-normal forms, an intermediate format closely related to continuation-passing style (CPS) [FSDF93]. Translating typed λ-calculus terms into this language is first shown to amount to a proof transformation taking a proof in natural deduction form to a proof in a restricted form of a Gentzen-style sequent calculus. Then, the evaluation of A-normal form programs using runtime environments is shown to correspond precisely to proof reduction in the sequent calculus. As [Oho99a] argues, this relation between types and formulae models low-level execution of code more precisely than the correspondence between high-level function application and cut elimination in the usual Curry-Howard isomorphism.

Complementing this work, [Oho99b] proposes a general style of logical abstract machines, the *sequential sequent calculus* (SSC). This proof system represents machine-level programs in a linear form where each instruction is modelled by a corresponding proof rule and rule applications are composed in program order. Dynamic program behaviour consequently amounts to proof reduction as the execution of an instruction eliminates the last inference step. The concept is presented for register-based and stack-based architectures, and code generation is related to proof transformation. The stack-based architecture was subsequently applied to a subset of the Java virtual machine, yielding an elegant de-compilation algorithm from JVML into a functional language [KO01].

The typing of our assembly language follows a more traditional rule format with an explicit cut rule rather than an internalisation of the cut as rule application. As intermediate work towards this dissertation, we formalised some aspects of our typing calculus in SSC, but the details of this work and an understanding of the conversion between the two styles are left for future research.

2.6.2.5 Proof-carrying code

Building on the technology of intermediate and assembly level typing, *proof-carrying code* (PCC) has recently emerged as a novel technology for formally certifying programs [Nec97]. Instead of relying on trusted third parties for authentication, a formal proof object asserting intrinsic properties of a program is bundled with the code using type-theoretic concepts [BM92]. The task of program verification is then split into two phases. The code *producer* (e.g. compiler) constructs the proof together with the program, and bundles them together. Depending on the specification logic, proof construction may either be done fully automatically or involve manual intervention, possibly supported by verification condition generators, theorem provers or other tools. The code *consumer* (e.g. execution engine) validates the consistency of the transmitted code-proof pair. Thanks to the type-theoretic technology, the second step amounts to type-checking, which is often more feasible than proof inference or full verification. From the point of view of the code consumer, the means employed for constructing the proof is irrelevant – indeed, the proof does not need to originate from the code producer at all as long as it matches the code's properties.

Up to now, PCC has been mostly used for ensuring security properties, such as restricting the access to local data or bounding the consumption of computational resources. In the context of Java, additional security issues arise from code mobility, such as the choice of class loader used, and from class file verification. Specifying security policies in a (type-theoretic) specification logic allows one to employ PCC for enforcing safety of execution. Dynamically transmitted code is rejected by the virtual machine's verifier unless it is equipped with a proof that certifies obedience to the safety policy. Third-party intervention during code transmission leading to code which violates the security model is detected. The intervention will manifest itself either by the proof not matching the program's properties any longer or by an inconsistency between the proof and the specified security properties. In contrast, outside intervention which leaves the security aspects untouched (or manipulates code and proof in a consistent and secure way) is not detected. Proof-carrying transmitted code may hence still behave functionally incorrect, but is guaranteed to respect the security policy.

Several extensions of the original PCC approach are currently being explored. These use sophisticated type systems for reducing the size of the trusted code base (verification condition generator, proof checker, ...) and logical embeddings of assembly and intermediate languages into higher-order theorem provers [AF00] [MA00] [App01] [SSTP02]. As is the case for earlier work such [Cur92], we expect that these embeddings may be relevant for a formalisation of our approach. In particular, sophisticated type systems should be beneficial for verifying programs with respect to architectures where not all forwarding paths are available and for developing typed compilation schemes starting from functional intermediate languages. These aspects are not treated in this dissertation but represent appropriate topics for future work, and we will elaborate on some ideas in our concluding discussion.

2.6.2.6 Typed register allocation

Two pieces of research consider typed approaches to register allocation. Thiemann [Thi98] presents *implementation types* which make resource allocation and conversion explicit. Register allocation is formalised in a type-directed way, and the transfer of values between registers and memory is tracked syntactically. While the source language consist of a restricted class of A-normal forms, the target code comprises a generic assembly language with simple register operations and immediate values. In order to model not only the assignment of locations to values, the type system is formalised using *effects* capturing those registers which are affected by the execution of an instruction, or by a function call. According to [Thi98], implementation types allow flexible calling conventions for function calls (caller-saved, callee-saved, ...) and support *lightweight closures* where function representations store only parts of their environment [SW97]. The last aspect should be relevant in an extension of our work for compiling functional languages, as will be briefly discussed in Chapter 9. As register allocation in our work represents only a minor aspect and our intermediate language is not as well-structured as A-normal forms (though related to it), we did not formalise our allocation using type and effect systems.

The same remark holds for the work of Agat [Aga97] who presents a type and effect system for register usage in assembly languages. His motivation is to guarantee legality of register allocation schemes for functional languages. Assembly programs are expressed in a λ-calculus like notation, where for example a program

$$\texttt{mov}\ 3, \%\texttt{r8}\ \texttt{mov}\ 5, \%\texttt{r6}\ \texttt{add}\ \%\texttt{r8}, \%\texttt{r6}, \%\texttt{r2}$$

is expressed as

$$\texttt{let}\ x = 3_{\texttt{R8}}\ \texttt{in}\ \texttt{let}\ y = 5_{\texttt{R6}}\ \texttt{in}\ \texttt{ADD}_{\texttt{R8},\texttt{R6},\texttt{R2}}\, x\, y$$

This program is assigned the type $\mathit{Int}_{\texttt{R2}}!\{\texttt{R2},\texttt{R6},\texttt{R8}\}$, indicating that the result (of type *Int*) will be delivered in register $\texttt{R2}$, and that all registers used for producing the result are amongst $\texttt{R2}, \texttt{R6}$ and $\texttt{R8}$. This mechanism is extended to (higher-order) functions and closures, and soundness of the corresponding typing system with respect to a dynamic semantics with (finitely many) registers and an operand stack is proven. Again, we expect that this calculus might provide guidance for reasoning about function calls and a more formal treatment of allocation than ours.

It would also be interesting to explore which modifications are needed to the calculi presented in [Thi98] and [Aga97] for treating (linear) forwarding or operand queue allocation.

2.6.3 Program analysis frameworks

The second part of this thesis illustrates the benefit of explicit forwarding for scheduling communication in the compiler. The program analysis presented uses the concept of dataflow equations which are a well-known technology in compiler construction [App98a]. The recent book [NNH99] stresses the similarities between dataflow analysis and other program analysis techniques such as abstract interpretation, typing and constraint-based analysis. The techniques are presented using a uniform language, with operational models given in SOS form. Included in [NNH99] is a soundness proof of the dataflow equations for liveness analysis which shares similarities with our approach to usage analysis. We will discuss these similarities but also differences in Chapter 6.

2.7 Notation

We follow the tradition of presenting programming language technology using formal derivation calculi. These are based on formal languages of *judgements* (*sequents*) whose grammatical structure depends on the purpose of the calculus. In our case, operational semantics of assembly languages will be defined using judgements of the form $\mathsf{C} \xrightarrow{t} \mathsf{D}$ where C and D are configurations and t is a piece of assembly code. Typing calculi will employ judgements of the form $\Gamma \vdash t : \tau$ where τ denotes a type, Γ an (optional) set of typing assumptions (also called a typing context) and t is again a program fragment.

Judgements are related to each other by a set of rules

$$\text{Name}\ \frac{H_1 \quad \dots \quad H_n}{C}$$

linking hypothetical judgements $H_1, \dots, H_n$ to concluding judgement C. Since most derivation systems contain large (often infinite) sets of rules, derivation systems are usually presented as a set of rule *schemata*

$$\text{Name}\ \frac{H_1 \quad \dots \quad H_n}{C}\ \textit{side conditions}$$

where judgements may contain meta-variables and side-conditions govern instantiation: any substitution of appropriate terms for the meta-variables satisfying the side conditions yields a valid rule, and the denoted derivation system is determined by the set of all such instantiations.

Rules for which the set of hypothetical judgements is empty (i.e. $n = 0$ holds) are called *axioms*, and instantiations are recursively combined to derivation trees with final sequents (roots) C:

- any instantiation of an axiom with concluding judgement C is a derivation tree with root C, and

- an instantiation of a rule with conclusion C and hypothesis $H_1, \ldots, H_n$ yields a derivation for C if there are trees T_i with final sequents H_i for all $1 \leq i \leq n$.

The purpose of the derivation rules thus consists of constructively defining subsets of judgements comprising exactly those sequents C which are derivable (i.e. for which a derivation tree with root C exists). In our case, judgements $\mathsf{C} \xrightarrow{t} \mathsf{D}$ will define the dynamic semantics of our assemble language: derivability of this sequent will encode the fact that executing program t in initial configuration C leads to final configuration D. Likewise, the typing calculi will implicitly define a subset of programs, the set of well-typed programs.
The structure of derivations may subsequently be exploited for proving properties of derivable judgements, using the principle of (rule) induction: for showing that a property ϕ holds for all judgements defined by a derivation system with rule schemata $\mathrm{R}_1, \ldots, \mathrm{R}_\mathrm{n}$, it suffices to show the following

- ϕ holds for all judgements which are final sequents of axiom instantiations, and
- ϕ holds for any conclusion C of a rule with hypothesis $H_1, \ldots, H_n$ provided that ϕ holds for all hypothesis H_i.

In our case, we will inductively link typing calculi and dynamic semantics in order to eliminate runtime hazards: we will prove that the dynamic execution of a program t fulfils certain properties whenever a judgement $\Gamma \vdash t : \tau$ is derivable for some Γ and τ.
Often, judgements involve a syntactic category $\mathcal{C}$ such that each grammatical form of terms in $\mathcal{C}$ occurs in the conclusion of at most one rule schema. This restricts the set of rules which may have been applied in the last (closest to the root) derivation step, which may be exploited to simplify a proof using rule induction.
The grammatical structure of terms of a particular syntactic category $\mathcal{C}$ is also exploited in *structural* induction. Here, hypothesis in a derivation system for judgements with respect to terms of $\mathcal{C}$ may consist of judgements in a derivation system for judgements for terms of a different syntactic category $\mathcal{D}$, where $\mathcal{D}$-terms occur as components of (some) $\mathcal{C}$ terms. In this case, proving a property ϕ of $\mathcal{C}$-terms involves proving some related property ψ for $\mathcal{D}$-terms. For example, our dynamic semantics $\mathsf{C} \xrightarrow{t} \mathsf{D}$ will be defined in terms of a relation $\mathsf{C} \xrightarrow{\mu} \mathsf{D}$ where μ is a syntactic constituent of program t.

A proof of a property regarding judgements $\mathsf{C} \xrightarrow{t} \mathsf{D}$ consequently rests on a proof of a related property for judgements $\mathsf{C} \xrightarrow{\mu} \mathsf{D}$.

The following conventions are followed throughout this dissertation.

The set of words over a syntactic category $\mathcal{C}$ is denoted by $\mathcal{C}^*$, and is ranged over by variables $w, v \ldots$, obtained by juxtaposing elements from $\mathcal{C}$. The empty word is denoted by λ, and the length of a word $w = c_1 \ldots c_n$ is $|w| = n$, with $|\lambda| = 0$.

Binary relations $\mathcal{R}$ are usually written in infix notation. We denote the reflexive and transitive closure of $\mathcal{R}$ by $\mathcal{R}^*$ and the transitive closure by $\mathcal{R}^+$.

Functions between sets A and B are denoted by $f : A \to B$, with domain $dom\ f = A$ and codomain $cod\ f = B$. Partial maps with finite domain are denoted by $f : A \rightharpoonup B$, and $f[a \mapsto b]$ is used for both maps and functions to denote the modified function f' which maps a to b and acts like f on all other elements from A.

$\mathcal{P}(S)$ denotes the powerset of a set S, and for a directed graph $G = (V, E)$ with vertices V and edges $E \subset V \times V$, the sets of predecessors and successors of $v \in V$ are given by $preds(v) = \{u \in V \mid (u, v) \in E\}$ and $succs(v) = \{u \in V \mid (v, u) \in E\}$, respectively.

Chapter 3

Sequential queue machines

The first step in understanding forwarding consists of defining an assembly language in which forwarding information is captured explicitly in the syntax and thus exposed to the programmer. The benefit of this formalisation is that different dynamic semantics may be related to each other and to static semantics expressing well-formedness conditions. Reasoning techniques for programming languages can be used to prove these relationships. Furthermore, explicitness of forwarding information represents a well-defined interface to the compiler which may thus optimise the forwarding behaviour. Concepts such as low-level typing systems can be used for proving structural correctness of code emitted by a compiler and for giving hints to the runtime system regarding constraints for dynamic scheduling.

This chapter consequently introduces ALEF, an assembly language for explicit forwarding. In ALEF, operand queue names and register names occur syntactically in the same position, so a compiler can choose which mechanism to use for each operand. A dynamic semantics for ALEF is given which models sequential execution similar to the instruction set architecture of a processor. This model aims at understanding the basic requirements of forwarding schemes. For this analysis, registers can be disregarded and the dynamic semantics only supports operands which are sent through forwarding queues. At first, only straight-line code is considered, i.e. code which does not contain jump instructions. Its execution can encounter runtime errors which inhibit further execution, similar to deadlocks in SCALP programs. As a first example of the benefits of a programming language based view we present a static semantics for eliminating these errors using a type system based on notation from linear logic [Gir86]. Types abstract from the order of operands in operand queues and from individual values. We then extend dynamic and static semantics to programs with jumps, following an approach introduced by Stata and Abadi [SA98c] when combining the typings of basic blocks. We require that input and output types of neighbouring basic blocks match, so that the number of elements in operand queues is statically bound.

Static and dynamic semantics are related by soundness theorems which guarantee that well-typed programs are free of dynamic errors. In the case of straight-line code, a completeness theorem complements soundness, stating that any error-free program is indeed typable. We also show that the type system admits a notion of principal types, from which any other typing for the same program may be obtained. The corresponding type inference task for basic blocks is solved by unification.

Synopsis We start by defining the syntax of ALEF in Section 3.1. Subsequently, we restrict our attention to programs without registers or branch instructions. The dynamic semantics for the restricted set of programs is defined in Section 3.2. Its characteristic dynamic errors motivate a static semantics which is introduced in Section 3.3. Soundness and completeness with respect to the operational model are discussed, proving that programs which pass the type system will not experience the dynamic hazards under consideration. In Section 3.4, we show that the type system admits a notion of principal types. For each well-typed program t we can single out a type of which all other types for t are generalisations. In Section 3.5, we generalise our approach to programs involving branch instructions. Again, we first give a dynamic semantics, before introducing the static semantics and discussing type inference. Finally, Section 3.6 summarises the achievements and discusses some additional aspects.

3.1 ALEF: a language for explicit forwarding

ALEF, the assembly language for explicit forwarding, is intended for execution on architectures whose general organisation is shown in Figure 3.1. Functional units of different types occur in parallel, each executing a specific subset of the instructions.

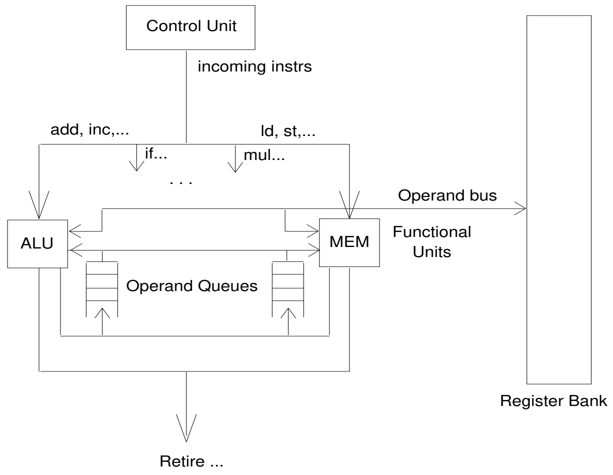


Figure 3.1: General organisation of processors suitable for ALEF

Instructions are sent to their functional unit by the control unit and may exchange operands through registers or named operand queues before retiring after execution. The syntax of ALEF is given by the following grammar where instructions *ins* are pairs $[n]code$ of unique labels $n \in \mathbf{N}$ and opcodes with operand queue names q and registers r as arguments. Some instructions are parametric in the type *fu* of the functional unit they execute on. Operand values $a \in Val$ and memory locations both range over integers.

$$
\begin{array}{rcl}
a \in Val & ::= & \ldots\ \mathtt{-1} \mid \mathtt{0} \mid \mathtt{1}\ \ldots \\
n \in \mathbf{N} & ::= & \mathtt{1} \mid \mathtt{2} \mid \ldots \\
q \in Q & ::= & \mathtt{q}_n \\
r \in R & ::= & \mathtt{r}_n \\
op \in OP & ::= & q \mid r \\
fu \in FU & ::= & \mathtt{ALU} \mid \mathtt{MEM} \mid \mathtt{MUL} \mid \mathtt{BU} \\
code \in Opcode & ::= & \mathtt{dec}\ op_1\ op_2 \mid \mathtt{add}\ op_1\ op_2\ op_3 \mid \mathtt{mul}\ op_1\ op_2\ op_3 \\
 & \mid & \mathtt{ldc}\ a\ op \mid \mathtt{ld}\ op_1\ op_2 \mid \mathtt{st}\ op_1\ op_2 \mid \mathtt{id}^{fu}\ op_1\ op_2 \\
 & \mid & \mathtt{dupl}^{fu}\ op_1\ op_2\ op_3 \mid \mathtt{skip}^{fu}\ op \mid \mathtt{if}\ op\ n_1\ n_2 \mid \mathtt{jmp}\ n \\
ins \in Instr & ::= & [n]code \\
t, s, \ldots \in Iseq & ::= & \varepsilon \mid ins \mid st
\end{array}
$$

The first three opcode forms are arithmetic instructions for decrementing, adding and multiplying values. For example, the instruction `add` q_1 q_2 q_2 removes the head values from operand queues q_1 and q_2 and inserts their sum into q_2. The next three instructions are memory instructions. `ldc` *a op* inserts the value *a* into *op* while `ld` op_1 op_2 interprets the value of op_1 as memory address and inserts the value found at that location into op_2. `st` op_1 op_2 stores the value found in op_2 at the location given by the value from op_1. The next three instruction forms are parametric in the functional unit. For each parameter *fu*, $\texttt{id}^{fu}$ transfers the value of op_1 to op_2 while $\texttt{dupl}^{fu}$ in addition sends a copy of it to op_3 and $\texttt{skip}^{fu}$ simply consumes the value read from *op*. The last two instructions are conditional and unconditional jumps and will be dealt with in Section 3.5.

Instruction sequences $s, t, \ldots$ are lists of instructions with ε denoting the empty sequence. It can be proven that composition behaves associatively and ε neutrally for all notions in this thesis, and instruction sequences are consequently often treated as flat programs $ins_1 \ldots ins_n$. Occasionally, we omit labels of instructions and confuse opcodes and instructions.

When compared to simple compounds, ALEF imposes fewer restrictions on the forwarding discipline. Firstly, forwarding between instructions which are not syntactical neighbours is possible. Secondly, a program may use forwarding across basic block boundaries. Thirdly, the linearity restriction is lifted as binary instructions may obtain both operands through forwarding and a `dupl` instruction may send its result to two operand queues. The program

$$\begin{array}{ll} [1]\texttt{ldc}\ 4\ q_1 & [5]\texttt{add}\ q_1\ q_2\ q_1 \\ [2]\texttt{ldc}\ 2\ q_2 & [6]\texttt{dec}\ r_1\ r_1 \\ [3]\texttt{ldc}\ 7\ r_1 & [7]\texttt{if}\ r_1\ 8\ 4 \\ [4]\texttt{dupl}\ q_2\ q_2\ q_2 & [8]\texttt{add}\ q_2\ q_1\ r_1 \end{array} \tag{3.1}$$

contains examples for all three forwarding patterns. The Java excerpt

```
int i = 4; int j = 2;
for (int k=7; k>0; k--){i = i+j;}
k = i+j;
```

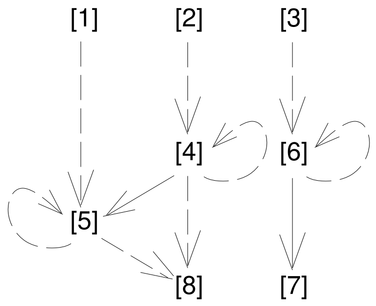


Figure 3.2: Visualisation of program (3.1)

may result in this code, and Figure 3.2 shows a visualisation of the forwarding behaviour, where forwardings across basic block boundaries are shown as dashed arrows and solid arrows represent forwardings inside basic blocks.

When compared to SCALP, ALEF specifies not only the destination of instruction's results but also the sources of the operands. In SCALP, a typical binary instruction such as `add` would expect its operands at the two input ports `alu_a` and `alu_b` of its functional unit. In contrast, the source queue of forwarded operands is explicitly visible in ALEF and different `add` instructions may well obtain their operands from different queues.

The remainder of this chapter as well as Chapter 4 aim at deriving basic principles of explicit forwarding. These can be examined most purely in the absence of registers, and the following sections hence completely ignore registers and assume that all operand fields *op* are of the form *q*.

3.2 Dynamic semantics

The dynamic semantics for sequential execution corresponds to the instruction set architecture (ISA) of a processor. A simplified architectural model is shown in Figure 3.3. The structure of forwarding paths in modern architectures motivate an asymmetric access policy for operand queues [Fly95]. While instructions can send their result to any operand queue, they can obtain their operands only from queues which are associated to their functional unit. No reduction is possible if an instruction attempts to read

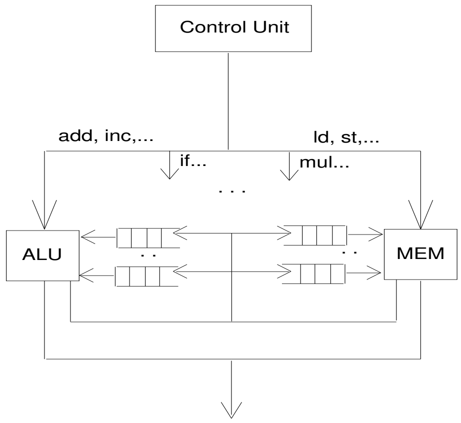


Figure 3.3: Architectural model for sequential dynamic semantics

from a foreign operand queue. We let $\gamma(q)$ denote the functional unit associated to q. In SCALP [End96], instructions in fact do not specify the source queues of operands at all but implicitly consume values from the input ports of their functional unit.

The dynamic semantics executes one instruction at a time, determined by a relation $\mathsf{C} \xrightarrow{t} \mathsf{D}$ where an instruction sequence t relates initial configuration C and final configuration D. In this section, configurations are pairs (Q,M) consisting of total maps $\mathsf{Q} : Q \to Val^*$ and $\mathsf{M} : Val \to Val$. The queue configuration Q assigns a word of values to each operand queue, while M represents the memory state by mapping addresses to values. We write $\mathsf{Q}(q)$ to represent Q's entry at q and $\mathsf{Q}[q \mapsto w]$ for the queue configuration which agrees with Q everywhere except q and maps q to the word w. For the memory component M, we let $\mathsf{M}(a)$ denote the value at address a in M and write $\mathsf{M}[a \mapsto b]$ for the memory M updated by value b at address a. Often, we only mention the non-trivial parts of queue configurations by writing $[q_1 \mapsto w_1, \dots, q_n \mapsto w_n]$ for the Q with $\mathsf{Q}(q_i) = w_i$ and $\mathsf{Q}(q) = \lambda$ for $q \notin \{q_1, \dots, q_n\}$. A similar convention is used for memories M.

The dynamic semantics is defined structurally, based on (semantic) micro-instructions $read(fu)$ and $write$ which are ranged over by μ. These model the low-level access to operand queues, and their actions are parametrised by the operand queue they access and the value they operate on.

$$\text{WR}\,\frac{}{w \xrightarrow{(write,a,q)} wa} \qquad \text{RD}\,\frac{}{aw \xrightarrow{(read(fu),a,q)} w}\ \gamma(q) = fu$$

The enqueueing operation (rule WR) appends the value a to the word $w \in Val^*$ and carries no side condition. Dequeuing (rule RD) removes the head value a of a word aw. The rule is only applicable if the side condition $\gamma(q) = fu$ holds and the current queue content is of the form aw.

The effect of micro-instructions is promoted to the level of configurations by

$$\mu\,\frac{\mathsf{Q}(q) \xrightarrow{(\mu,a,q)} w}{(\mathsf{Q},\mathsf{M}) \xrightarrow{(\mu,a,q)} (\mathsf{Q}[q \mapsto w],\mathsf{M})}$$

and the dynamic semantics of opcodes is given in terms of micro-instructions by the rules CODE in Figure 3.4. Finally, the dynamic semantics of instruction sequences is defined in terms of the relation $\mathsf{C} \xrightarrow{code} \mathsf{D}$ in the rules INSTR and COMP (Figure 3.5).

Example. Let t be the program $[1]\mathtt{ldc}\ 4\ \mathtt{q}_1\ [2]\mathtt{add}\ \mathtt{q}_1\ \mathtt{q}_1\ \mathtt{q}_2$, and the configurations $\mathsf{C}, \ldots, \mathsf{G}$ be given by

$$\begin{aligned}
\mathsf{C} &= ([\mathtt{q}_1 \mapsto 3, \mathtt{q}_2 \mapsto 5], \mathsf{M}) \\
\mathsf{D} &= ([\mathtt{q}_1 \mapsto \lambda, \mathtt{q}_2 \mapsto 5\,7], \mathsf{M}) \\
\mathsf{E} &= ([\mathtt{q}_1 \mapsto 3\,4, \mathtt{q}_2 \mapsto 5], \mathsf{M}) \\
\mathsf{F} &= ([\mathtt{q}_1 \mapsto 4, \mathtt{q}_2 \mapsto 5], \mathsf{M}) \\
\mathsf{G} &= ([\mathtt{q}_1 \mapsto \lambda, \mathtt{q}_2 \mapsto 5], \mathsf{M})
\end{aligned}$$

where M is arbitrary. Figure 3.6 shows a derivation for $\mathsf{C} \xrightarrow{t} \mathsf{D}$, proving that executing t in configuration C leads to configuration D provided that $\gamma(\mathtt{q}_1) = \mathtt{ALU}$ holds. For typographical reasons, the side conditions are shown in the position of the hypothesis and rule names have been omitted. $\diamond$

The final configuration D in the above example is uniquely determined once C and t are fixed, as the sequential dynamic semantics is deterministic.

$$\text{LDC}\,\frac{\mathsf{C} \xrightarrow{(write,a,op)} \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{ldc}\ a\ op} \mathsf{D}} \qquad \text{DEC}\,\frac{\mathsf{C} \xrightarrow{(read(\mathtt{ADD}),a,op_1)} \mathsf{E} \quad \mathsf{E} \xrightarrow{(write,a-1,op_2)} \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{dec}\ op_1\ op_2} \mathsf{D}}$$

$$\text{SKIP}\,\frac{\mathsf{C} \xrightarrow{(read(fu),a,op_1)} \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{skip}^{fu}\ op_1} \mathsf{D}} \qquad \text{ID}\,\frac{\mathsf{C} \xrightarrow{(read(fu),a,op_1)} \mathsf{E} \quad \mathsf{E} \xrightarrow{(write,a,op_2)} \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{id}^{fu}\ op_1\ op_2} \mathsf{D}}$$

$$\text{LD}\,\frac{\mathsf{C} \xrightarrow{(read(\mathtt{MEM}),a,op_1)} (\mathsf{Q},\mathsf{M}) \quad (\mathsf{Q},\mathsf{M}) \xrightarrow{(write,\mathsf{M}(a),op_2)} \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{ld}\ op_1\ op_2} \mathsf{D}}$$

$$\text{ST}\,\frac{\mathsf{C} \xrightarrow{(read(\mathtt{MEM}),a,op_1)} \mathsf{D} \quad \mathsf{D} \xrightarrow{(read(\mathtt{MEM}),b,op_2)} (\mathsf{Q},\mathsf{M})}{\mathsf{C} \xrightarrow{\mathtt{st}\ op_1\ op_2} (\mathsf{Q},\mathsf{M}[a \mapsto b])}$$

$$\text{ADD}\,\frac{\mathsf{C} \xrightarrow{(read(\mathtt{ALU}),a,op_1)} \mathsf{E} \quad \mathsf{E} \xrightarrow{(read(\mathtt{ALU}),b,op_2)} \mathsf{F} \quad \mathsf{F} \xrightarrow{(write,a+b,op_3)} \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{add}\ op_1\ op_2\ op_3} \mathsf{D}}$$

$$\text{MUL}\,\frac{\mathsf{C} \xrightarrow{(read(\mathtt{MUL}),a,op_1)} \mathsf{E} \quad \mathsf{E} \xrightarrow{(read(\mathtt{MUL}),b,op_2)} \mathsf{F} \quad \mathsf{F} \xrightarrow{(write,a*b,op_3)} \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{mul}\ op_1\ op_2\ op_3} \mathsf{D}}$$

$$\text{DUPL}\,\frac{\mathsf{C} \xrightarrow{(read(fu),a,op_1)} \mathsf{E} \quad \mathsf{E} \xrightarrow{(write,a,op_2)} \mathsf{F} \quad \mathsf{F} \xrightarrow{(write,a,op_3)} \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{dupl}^{fu}\ op_1\ op_2\ op_3} \mathsf{D}}$$

Figure 3.4: Dynamic semantics of opcodes

$$\text{INSTR}\,\frac{\mathsf{C} \xrightarrow{code} \mathsf{D}}{\mathsf{C} \xrightarrow{[n]code} \mathsf{D}} \qquad \text{COMP}\,\frac{\mathsf{C} \xrightarrow{s} \mathsf{E} \quad \mathsf{E} \xrightarrow{t} \mathsf{D}}{\mathsf{C} \xrightarrow{st} \mathsf{D}}$$

Figure 3.5: Sequential dynamic semantics of instruction sequences

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{3 \xrightarrow{(\mathit{write},4,\mathsf{q}_1)} 3\,4}
}{\mathsf{C} \xrightarrow{(\mathit{write},4,\mathsf{q}_1)} \mathsf{E}}
}{\mathsf{C} \xrightarrow{\mathtt{ldc}\ 4\ \mathsf{q}_1} \mathsf{E}}
}{\mathsf{C} \xrightarrow{[1]\mathtt{ldc}\ 4\ \mathsf{q}_1} \mathsf{E}}
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{\gamma(\mathsf{q}_1)=\mathtt{ALU}}{3\,4 \xrightarrow{(\mathit{read}(\mathtt{ALU}),3,\mathsf{q}_1)} 4}
}{\mathsf{E} \xrightarrow{(\mathit{read}(\mathtt{ALU}),3,\mathsf{q}_1)} \mathsf{F}}
\quad
\dfrac{
\dfrac{\gamma(\mathsf{q}_1)=\mathtt{ALU}}{4 \xrightarrow{(\mathit{read}(\mathtt{ALU}),4,\mathsf{q}_1)} \lambda}
}{\mathsf{F} \xrightarrow{(\mathit{read}(\mathtt{ALU}),4,\mathsf{q}_1)} \mathsf{G}}
\quad
\dfrac{
\dfrac{}{5 \xrightarrow{(\mathit{write},7,\mathsf{q}_2)} 5\,7}
}{\mathsf{G} \xrightarrow{(\mathit{write},3+4,\mathsf{q}_2)} \mathsf{D}}
}{\mathsf{E} \xrightarrow{\mathtt{add}\ \mathsf{q}_1\ \mathsf{q}_1\ \mathsf{q}_2} \mathsf{D}}
}{\mathsf{E} \xrightarrow{[2]\mathtt{add}\ \mathsf{q}_1\ \mathsf{q}_1\ \mathsf{q}_2} \mathsf{D}}
}{\mathsf{C} \xrightarrow{[1]\mathtt{ldc}\ 4\ \mathsf{q}_1\ [2]\mathtt{add}\ \mathsf{q}_1\ \mathsf{q}_1\ \mathsf{q}_2} \mathsf{D}}
$$

Figure 3.6: Example derivation for $\mathsf{C} \xrightarrow{t} \mathsf{D}$.

Proposition 1. *(Determinacy of sequential execution) If* $\mathsf{C} \xrightarrow{t} \mathsf{D}$ *and* $\mathsf{C} \xrightarrow{t} \mathsf{E}$ *then* $\mathsf{D} = \mathsf{E}$.

$$
\begin{tikzcd}
 & \mathsf{C} \arrow[dl, "t"'] \arrow[dr, "t"] & & & \\
\mathsf{D} & & \mathsf{E} & \Rightarrow & \mathsf{D} = \mathsf{E}
\end{tikzcd}
$$

Proof. We perform a structural induction on t. We first show determinacy of the rules WR, RD and μ. Next, we show that each rule CODE is deterministic and promote this result to the instruction level by rule INSTR. Finally, we perform the induction on the structure of t.

WR **Claim 1:** If $w \xrightarrow{(\mathit{write},a,q)} u$ and $w \xrightarrow{(\mathit{write},a,q)} v$ then $u = v$.

Proof For $w \xrightarrow{(\mathit{write},a,q)} u$, the definition of rule WR implies $u = wa$, and for $w \xrightarrow{(\mathit{write},a,q)} v$ the definition of WR implies $v = wa$, hence $u = v$ holds.

RD **Claim 2:** If $w \xrightarrow{(\mathit{read}(\mathit{fu}),a,q)} u$ and $w \xrightarrow{(\mathit{read}(\mathit{fu}),b,q)} v$ then $a = b$ and $v = u$.

Proof For $w \xrightarrow{(\mathit{read}(\mathit{fu}),a,q)} u$ and $w \xrightarrow{(\mathit{read}(\mathit{fu}),b,q)} v$, the definition of rule RD implies $w = au$ and $w = bv$, hence $au = w = bv$ follows and thus $a = b$ and $v = u$.

μ There are two claims.

Claim 3: If $\mathsf{C} \xrightarrow{(\mathit{write},a,q)} \mathsf{D}$ and $\mathsf{C} \xrightarrow{(\mathit{write},a,q)} \mathsf{E}$ then $\mathsf{D} = \mathsf{E}$.

Proof Writing $\mathsf{C} = (\mathsf{Q}, \mathsf{M})$, the definition of rule μ implies that there are w and v with

- $\mathsf{Q}(q) \xrightarrow{(write,a,q)} w$ and $\mathsf{D} = (\mathsf{Q}[q \mapsto w], \mathsf{M})$ and
- $\mathsf{Q}(q) \xrightarrow{(write,a,q)} v$ and $\mathsf{E} = (\mathsf{Q}[q \mapsto v], \mathsf{M})$.

By claim 1 (determinacy of WR) $w = v$ follows, hence $\mathsf{Q}[q \mapsto w] = \mathsf{Q}[q \mapsto v]$ and $\mathsf{D} = \mathsf{E}$.

Claim 4: If $\mathsf{C} \xrightarrow{(read(fu),a,q)} \mathsf{D}$ and $\mathsf{C} \xrightarrow{(read(fu),b,q)} \mathsf{E}$ then $\mathsf{D} = \mathsf{E}$ and $a = b$.

Proof For $\mathsf{C} = (\mathsf{Q}, \mathsf{M})$, $\mathsf{D} = (\mathsf{P}, \mathsf{N})$ and $\mathsf{E} = (\mathsf{R}, \mathsf{L})$, the definition of rule μ implies that there are w and v with

- $\mathsf{Q}(q) \xrightarrow{(read(fu),a,q)} w$ and $(\mathsf{P}, \mathsf{N}) = (\mathsf{Q}[q \mapsto w], \mathsf{M})$ and
- $\mathsf{Q}(q) \xrightarrow{(read(fu),b,q)} v$ and $(\mathsf{R}, \mathsf{L}) = (\mathsf{Q}[q \mapsto v], \mathsf{M})$.

By claim 2 (determinacy of RD) $a = b$ and $w = v$ follow, hence $\mathsf{D} = (\mathsf{P}, \mathsf{N}) = (\mathsf{Q}[q \mapsto w], \mathsf{M}) = (\mathsf{Q}[q \mapsto v], \mathsf{M}) = (\mathsf{R}, \mathsf{L}) = \mathsf{E}$ and $a = b$.

CODE. **Claim 5:** If $\mathsf{C} \xrightarrow{code} \mathsf{D}$ and $\mathsf{C} \xrightarrow{code} \mathsf{E}$ then $\mathsf{D} = \mathsf{E}$.

Proof The claim is proven be a case distinction on *code*. We treat the rule ADD for $code = \texttt{add}\ op_1\ op_2\ op_3$ explicitly, the other cases being similar.

Case $code = \texttt{add}\ op_1\ op_2\ op_3$**.** For $\mathsf{C} \xrightarrow{code} \mathsf{D}$ and $\mathsf{C} \xrightarrow{code} \mathsf{E}$ there are by rule ADD configurations F, G, H and I such that

$$\frac{\mathsf{C} \xrightarrow{(read(\texttt{ALU}),a_1,op_1)} \mathsf{F} \quad \mathsf{F} \xrightarrow{(read(\texttt{ALU}),b_1,op_2)} \mathsf{G} \quad \mathsf{G} \xrightarrow{(write,a_1+b_1,op_3)} \mathsf{D}}{\mathsf{C} \xrightarrow{\texttt{add}\ op_1\ op_2\ op_3} \mathsf{D}}$$

and

$$\frac{\mathsf{C} \xrightarrow{(read(\texttt{ALU}),a_2,op_1)} \mathsf{H} \quad \mathsf{H} \xrightarrow{(read(\texttt{ALU}),b_2,op_2)} \mathsf{I} \quad \mathsf{I} \xrightarrow{(write,a_2+b_2,op_3)} \mathsf{E}}{\mathsf{C} \xrightarrow{\texttt{add}\ op_1\ op_2\ op_3} \mathsf{E}}$$

Applying the second claim for rule μ (claim 4) twice yields first $\mathsf{F} = \mathsf{H}$ and $a_1 = a_2$ and then $\mathsf{G} = \mathsf{I}$ and $b_1 = b_2$. Consequently, $a_1 + b_1 = a_2 + b_2$, and applying the first claim for rule μ (claim 3) yields $\mathsf{D} = \mathsf{E}$.

Cases $code \neq \texttt{add}\ op_1\ op_2\ op_3$**.** Similar.

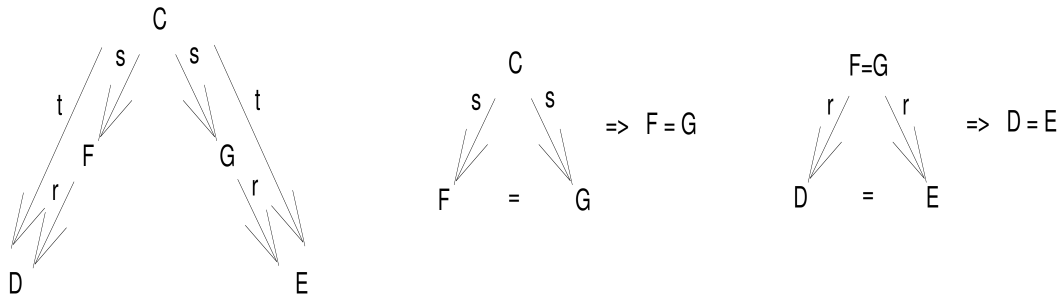


Figure 3.7: Proof of determinacy for $t = s\,r$.

INSTR **Claim 6:** If $\mathsf{C} \xrightarrow{ins} \mathsf{D}$ and $\mathsf{C} \xrightarrow{ins} \mathsf{E}$ then $\mathsf{D} = \mathsf{E}$.

> **Proof** For $ins = [n]code$, $\mathsf{C} \xrightarrow{ins} \mathsf{D}$ and $\mathsf{C} \xrightarrow{ins} \mathsf{E}$, rule INSTR implies $\mathsf{C} \xrightarrow{code} \mathsf{D}$ and $\mathsf{C} \xrightarrow{code} \mathsf{E}$, so applying the statements for rules CODE (claim 5) yields $\mathsf{D} = \mathsf{E}$.

Finally, the structural induction for showing that $\mathsf{C} \xrightarrow{t} \mathsf{D}$ and $\mathsf{C} \xrightarrow{t} \mathsf{E}$ implies $\mathsf{D} = \mathsf{E}$ consists of three cases.

If $t = \varepsilon$, then the preconditions $\mathsf{C} \xrightarrow{t} \mathsf{D}$ and $\mathsf{C} \xrightarrow{t} \mathsf{E}$ are false as no rule for deriving $\mathsf{C} \xrightarrow{\varepsilon} \mathsf{D}$ exists.

If $t = ins$, then the last rule used in the derivations for $\mathsf{C} \xrightarrow{t} \mathsf{D}$ and $\mathsf{C} \xrightarrow{t} \mathsf{E}$ is INSTR, so $\mathsf{D} = \mathsf{E}$ follows from claim 6.

If $t = s\,r$, then (see Figure 3.7) the last rule in the derivations for $\mathsf{C} \xrightarrow{t} \mathsf{D}$ and $\mathsf{C} \xrightarrow{t} \mathsf{E}$ must have been COMP, so there are F and G such that

$$\frac{\mathsf{C} \xrightarrow{s} \mathsf{F} \quad \mathsf{F} \xrightarrow{r} \mathsf{D}}{\mathsf{C} \xrightarrow{s\,r} \mathsf{D}} \qquad \text{and} \qquad \frac{\mathsf{C} \xrightarrow{s} \mathsf{G} \quad \mathsf{G} \xrightarrow{r} \mathsf{E}}{\mathsf{C} \xrightarrow{s\,r} \mathsf{E}}$$

By applying the induction hypothesis twice we obtain first $\mathsf{F} = \mathsf{G}$ and then $\mathsf{D} = \mathsf{E}$.

$\square$

Programs fail to execute (i.e. do not have a derivation) whenever a side condition of rule RD is not fulfilled. Failure of the condition $\gamma(op) = fu$ represents a (static) operand queue mismatch where an instruction tries to access an operand queue which is not connected to its FU. For example, program (3.2)

$$[1]\texttt{ldc}\ 4\ \mathsf{q}_2\ [2]\texttt{dec}\ \mathsf{q}_2\ \mathsf{q}_1 \tag{3.2}$$

deadlocks after executing the first instruction unless $\gamma(\mathsf{q}_2) = \texttt{ALU}$ holds. In the SOS system, all derivations $\mathsf{C} \xrightarrow{[1]\texttt{ldc}\ 4\ \mathsf{q}_2\ [2]\texttt{dec}\ \mathsf{q}_2\ \mathsf{q}_1} \mathsf{D}$ contain an axiom with the side condition $\gamma(\mathsf{q}_2) = \texttt{ALU}$.

Failure of the second condition is a dynamic error which occurs if previous instructions or the initial configuration did not provide sufficiently many operands. For example, for $\mathsf{C} = ([\mathsf{q}_2 \mapsto 3], \mathsf{M})$ the program

$$[1]\texttt{ldc}\ 4\ \mathsf{q}_2\ [2]\texttt{add}\ \mathsf{q}_2\ \mathsf{q}_1\ \mathsf{q}_1 \tag{3.3}$$

deadlocks after executing the first instruction as the addition can not be executed. No D exists such that $\mathsf{C} \xrightarrow{t} \mathsf{D}$ is derivable. We call this error starvation or a deadlock due to operand queue underflow. Other configurations might well have derivations. For example, for configurations $\mathsf{E} = ([\mathsf{q}_1 \mapsto 5], \mathsf{M})$ and $\mathsf{F} = ([\mathsf{q}_1 \mapsto 9], \mathsf{M})$ we can derive $\mathsf{E} \xrightarrow{[1]\texttt{ldc}\ 4\ \mathsf{q}_2\ [2]\texttt{add}\ \mathsf{q}_2\ \mathsf{q}_1\ \mathsf{q}_1} \mathsf{F}$.

Both kinds of hazards are undesirable as they occur at runtime, when other parts of the program have successfully completed. In order to avoid costly monitoring of program execution or deadlock resolution at runtime, one would like to eliminate both hazards prior to program execution. Both example programs are *syntactically* correct, hence more sophisticated technology is needed to discover that they should not be executed. The following static semantics performs the corresponding program analysis, formalised as a type system. Derivations show the requirements of a program: the shape of admissible initial configurations is indicated in the final typing judgement and all necessary relations $\gamma(q) = fu$ are visible in axioms.

3.3 Static semantics

The static semantics appropriate for the sequential dynamic semantics eliminates illegal operand queue access and deadlock due to operand queue underflow. It is formalised as a type system, where each instruction is assigned a type characterising its net-effect on the number of items in operand queues. As the type system aims at abstracting from the order of items in queues and the values of items, we use a linear notation based on Girard's linear logic [Gir86].

Judgements are of the form $t : A \multimap B$ where t is an instruction sequence and A and B are formal products over the set of operand identifiers *op*.

$$\begin{aligned} A, B \ldots \in \mathit{Product} \quad &::= \quad \mathbf{1} \mid op \otimes A \\ \tau \in \mathit{Type} \quad &::= \quad A \multimap A \end{aligned}$$

Linear operators $\otimes$ and $\multimap$ are motivated by the fact that the *multiplicity* of items in an operand queue matters. We treat $\otimes$ associatively and commutatively with $\mathbf{1}$ as neutral element, and for $i \geq 0$ we write op^i for $\underbrace{op \otimes \ldots \otimes op}_{i}$, with $op^0 = \mathbf{1}$. Types thus abstract from the particular values of operands and from the order of items in each queue. Occasionally, we say that *op divides A*, *occurs in A* or *is a factor of A* if A can be written as $A = op \otimes B$ for some B.

For each instruction form *ins*, the type system contains an axiom

$$\text{AX}\,\frac{}{[n]code : X \otimes A_{code} \multimap X \otimes B_{code}}\, SC(code)$$

where products A_{code} and B_{code} are given in Table 3.1, product X is arbitrary and the side condition $SC(code)$ is

$$\gamma(q_1) = \ldots = \gamma(q_n) = FU(code)$$

for $A_{code} = q_1 \otimes \ldots \otimes q_n$.

Typed program fragments are combined using a cut rule.

$$\text{CUT}\,\frac{s : A \multimap B \qquad t : B \multimap C}{s\,t : A \multimap C}$$

The type system fulfils the following property.

Proposition 2. *If $t : A \multimap B$ and $t : C \multimap D$ then $A = C$ iff $B = D$.*

Proof. Induction on the structure of t.

code	$FU(code)$	A_{code}	B_{code}
`dec` $op_1\ op_2$	`ALU`	op_1	op_2
`add` $op_1\ op_2\ op_3$	`ALU`	$op_1 \otimes op_2$	op_3
`mul` $op_1\ op_2\ op_3$	`MUL`	$op_1 \otimes op_2$	op_3
`ldc` $a\ op$	`MEM`	$\mathbf{1}$	op
`ld` $op_1\ op_2$	`MEM`	op_1	op_2
`st` $op_1\ op_2$	`MEM`	$op_1 \otimes op_2$	$\mathbf{1}$
$\mathtt{id}^{fu}\ op_1\ op_2$	fu	op_1	op_2
$\mathtt{dupl}^{fu}\ op_1\ op_2\ op_3$	fu	op_1	$op_2 \otimes op_3$
$\mathtt{skip}^{fu}\ op$	fu	op	$\mathbf{1}$

Table 3.1: Type system

If $t = \varepsilon$**,** no derivations for $t : A \multimap B$ and $t : C \multimap D$ exist and the claim is trivially fulfilled.

If $t = ins$**,** the typing judgements $t : A \multimap B$ and $t : C \multimap D$ were obtained by the rule AX, so there are X and Y such that for $ins = [n]code$ the equalities $A = A_{code} \otimes X$, $B = B_{code} \otimes X$, $C = A_{code} \otimes Y$, $D = B_{code} \otimes Y$ hold. Consequently, $A = C$ implies $X = Y$ and hence $B = D$, and $B = D$ implies $X = Y$ and hence $A = C$.

If $t = s\,r$**,** the last rule in the derivations for $t : A \multimap B$ and $t : C \multimap D$ was CUT, so there are E and F such that $s : A \multimap E$ and $r : E \multimap B$ and $s : C \multimap F$ and $r : F \multimap D$. Consequently, for $A = C$ the induction hypothesis yields first $E = F$ and then $B = D$, and for $B = D$ the induction hypothesis yields first $E = F$ and then $A = C$.

$\square$

The type system eliminates both sources of runtime hazards. For example, program (3.2) can only be typed if the condition $\gamma(\mathtt{q}_2) = \mathtt{ALU}$ is fulfilled. Likewise, a typing $t : A \multimap B$ for program (3.3) requires A to contain at least the factor $\mathtt{q}_1$, indicating that any initial configuration should at least contain one value in that queue.
Formally, the static semantics and the sequential dynamic semantics are linked using products which represent the shape of configurations.

Definition 1. *The* shape *of* $\mathsf{C} = (\mathsf{Q}, \mathsf{M})$ *is* $shape(\mathsf{C}) = \otimes_{q \in Q} q^{|\mathsf{Q}(q)|}$.

The following result shows soundness and completeness of the type system.

Theorem 1. *Let t be an instruction sequence.*

1. *If* $t : A \multimap B$ *and* $shape(\mathsf{C}) = A$ *then there is a unique* D *such that* $\mathsf{C} \xrightarrow{t} \mathsf{D}$*, and* $shape(\mathsf{D}) = B$ *holds.*

2. *If* $\mathsf{C} \xrightarrow{t} \mathsf{D}$ *then* $t : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.

$$\begin{tikzcd} \mathsf{C} \arrow[r, "t"] \arrow[d, dashed, "shape"'] & \mathsf{D} \arrow[d, dashed, "shape"] \\ t : A \arrow[r, "", multimap] & B \end{tikzcd}$$

Proof. Both parts are shown separately, and the proofs follow the syntactic structure.

1. We first prove the corresponding claims for rules WR, RD, μ and CODE.

RD **Claim 1:** If $w = av$ and $\gamma(q) = fu$ then there are unique u and b with $w \xrightarrow{(read(fu),b,q)} u$. Furthermore, $|u| = |w| - 1$ holds.

Proof For $w = av$ and $\gamma(q) = fu$ both side conditions of rule RD are fulfilled and the claim holds for the unique $a = b$ and $u = v$.

WR **Claim 2:** For arbitrary w, a and q there is a unique u such that $w \xrightarrow{(write,a,q)} u$. Furthermore, $|u| = |w| + 1$ holds.

Proof Rule WR does not carry any side conditions, so the claim follows for $u = wa$.

μ There are two claims.

Claim 3: If $shape(\mathsf{C}) = q \otimes X$ and $\gamma(q) = fu$ then there are unique D and a with $\mathsf{C} \xrightarrow{(read(fu),a,q)} \mathsf{D}$. Furthermore, $shape(\mathsf{D}) = X$ holds.

Proof For $\mathsf{C} = (\mathsf{Q}, \mathsf{M})$ and $shape(\mathsf{C}) = q \otimes X$ we have $|\mathsf{Q}(q)| \geq 1$, so $\mathsf{Q}(q) = aw$ for some unique a and w. By claim 1, there are unique u and b with $\mathsf{Q}(q) \xrightarrow{(read(fu),b,q)} u$, and $|u| = |\mathsf{Q}(q)| - 1$ holds. Hence, $\mathsf{C} \xrightarrow{(read(fu),a,q)} \mathsf{D}$ is derivable for $\mathsf{D} = (\mathsf{Q}[q \mapsto u], \mathsf{M})$, and uniqueness of D and a follow as in Proposition 1.

Furthermore, $|\mathsf{Q}[q \mapsto u](q)| = |u| = |\mathsf{Q}(q)| - 1$ holds and $|\mathsf{Q}[q \mapsto u](q')| = |\mathsf{Q}(q')|$ for $q' \neq q$, hence $shape(\mathsf{D}) \otimes q = shape(\mathsf{C}) = q \otimes X$ and therefore $shape(\mathsf{D}) = X$.

Claim 4: If $shape(\mathsf{C}) = A$ and q and a arbitrary, then there is a unique D with $\mathsf{C} \xrightarrow{(write,a,q)} \mathsf{D}$. Furthermore, $shape(\mathsf{D}) = A \otimes q$ holds.

Proof For $\mathsf{C} = (\mathsf{Q}, \mathsf{M})$ there is by claim 2 a unique u with $\mathsf{Q}(q) \xrightarrow{(write,a,q)} u$, and $u = |\mathsf{Q}(q)| + 1$ holds. Hence, $\mathsf{C} \xrightarrow{(write,a,q)} \mathsf{D}$ is derivable for $\mathsf{D} = (\mathsf{Q}[q \mapsto u], \mathsf{M})$ and uniqueness of D follows as in Proposition 1.

Furthermore, $|\mathsf{Q}[q \mapsto u](q)| = |u| = |\mathsf{Q}(q)| + 1$ holds and $|\mathsf{Q}[q \mapsto u](q')| = |\mathsf{Q}(q')|$ for $q' \neq q$, hence $shape(\mathsf{D}) = shape(\mathsf{C}) \otimes q = A \otimes q$.

CODE **Claim 5:** If $[n]code : A \multimap B$ and $shape(\mathsf{C}) = A$ then there is a unique D such that $\mathsf{C} \xrightarrow{code} \mathsf{D}$. Furthermore, $shape(\mathsf{D}) = B$ holds.

Proof We treat the representative case ADD explicitly.

Case $code = \mathtt{add}\ op_1\ op_2\ op_3$**.** For $[n]\mathtt{add}\ op_1\ op_2\ op_3 : A \multimap B$, the typing axiom AX and Table 3.1 guarantee $\gamma(op_1) = \gamma(op_2) = \mathtt{ALU}$ and $A = op_1 \otimes op_2 \otimes X$ and $B = op_3 \otimes X$ for some X. By the assumption that all operand identifiers op are operand queue identifiers q, there are q_1, q_2 and q_3 such that $q_i = op_i$ holds for $i \in \{1,2,3\}$. Therefore, $shape(\mathsf{C}) = A = q_1 \otimes q_2 \otimes X$ and $\gamma(q_1) = \gamma(q_2) = \mathtt{ALU}$, so by claim 3 there are unique E and a such that $\mathsf{C} \xrightarrow{(read(\mathtt{ALU}),a,q_1)} \mathsf{E}$, and $shape(\mathsf{E}) = q_2 \otimes X$ holds. Applying claim 3 again yields the existence of unique F and b with $\mathsf{E} \xrightarrow{(read(\mathtt{ALU}),b,q_2)} \mathsf{F}$, and $shape(\mathsf{F}) = X$ holds. By applying claim 4, there is a unique D with $\mathsf{F} \xrightarrow{(write,a+b,q_3)} \mathsf{D}$, and $shape(\mathsf{D}) = X \otimes q_3$ holds. Consequently, the following derivation is possible

$$\mathrm{ADD}\ \frac{\mathsf{C} \xrightarrow{(read(\mathtt{ALU}),a,q_1)} \mathsf{E} \quad \mathsf{E} \xrightarrow{(read(\mathtt{ALU}),b,q_2)} \mathsf{F} \quad \mathsf{F} \xrightarrow{(write,a+b,q_3)} \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{add}\ q_1\ q_2\ q_3} \mathsf{D}}$$

Using $q_i = op_i$ for $i \in \{1,2,3\}$ we thus obtain a derivation for $\mathsf{C} \xrightarrow{code} \mathsf{D}$, and $shape(\mathsf{D}) = X \otimes q_3 = X \otimes op_3 = B$ holds. Uniqueness of D with respect to $\mathsf{C} \xrightarrow{code} \mathsf{D}$ follows as in Proposition 1.

Cases $code \neq$ `add` $op_1\ op_2\ op_3$**.** Similar.

For proving the main claim, we perform a structural induction on t.

If $t = \varepsilon$**,** no judgement $t : A \multimap B$ is derivable, so the claim is trivially fulfilled.

If $t = ins$**,** then claim 5 guarantees that there is a unique D such that $\mathsf{C} \xrightarrow{code} \mathsf{D}$ holds, where $ins = [n]code$. By rule INSTR we obtain $\mathsf{C} \xrightarrow{ins} \mathsf{D}$. Furthermore, claim 5 guarantees $shape(\mathsf{D}) = B$, and uniqueness of D with respect to $\mathsf{C} \xrightarrow{ins} \mathsf{D}$ follows from Proposition 1.

If $t = s\,r$**,** then the typing rule CUT guarantees that there is a C such that $s : A \multimap C$ and $r : C \multimap B$, and a simple structural induction shows that C is unique. Applying the induction hypothesis to s and r yields unique E and D for $\mathsf{C} \xrightarrow{s} \mathsf{E}$ and $\mathsf{E} \xrightarrow{r} \mathsf{D}$, and $shape(\mathsf{E}) = C$ and $shape(\mathsf{D}) = B$ hold, so $\mathsf{C} \xrightarrow{s\,r} \mathsf{D}$. Uniqueness of D with respect to $\mathsf{C} \xrightarrow{s\,r} \mathsf{D}$ follows from Proposition 1.

2. Again, the claim follows by induction on t, based on claims for the rules WR, RD, μ and CODE.

RD **Claim 1:** If $w \xrightarrow{(read(fu),a,q)} v$ then $w = av$ and $\gamma(q) = fu$ and $|w| = |v| + 1$ hold.

Proof For $w \xrightarrow{(read(fu),a,q)} v$, w must have the form $w = av$ by rule RD and the side condition $\gamma(q) = fu$ must hold. Hence $|w| = |av| = |v| + 1$.

WR **Claim 2:** If $w \xrightarrow{(write,a,q)} v$ then $v = wa$ and $|v| = |w| + 1$.

Proof For $w \xrightarrow{(write,a,q)} v$, v must have the form $v = wa$ by rule WR, so $|v| = |wa| = |w| + 1$ holds.

μ There are two claims.

Claim 3: If $\mathsf{C} \xrightarrow{(read(fu),a,q)} \mathsf{D}$ then $shape(\mathsf{C}) = shape(\mathsf{D}) \otimes q$ and $\gamma(q) = fu$.

Proof For $\mathsf{C} \xrightarrow{(read(fu),a,q)} \mathsf{D}$ to hold, rule μ requires $\mathsf{Q}(q) \xrightarrow{(read(fu),a,q)} w$ for $\mathsf{C} = (\mathsf{Q},\mathsf{M})$ and $\mathsf{D} = (\mathsf{Q}[q \mapsto w],\mathsf{M})$. By claim 1, $\gamma(q) = fu$ and $\mathsf{Q}(q) = aw$

and $|\mathsf{Q}(q)| = |w|+1$ hold. Hence, we obtain

$$\begin{aligned} shape(\mathsf{C}) &= \otimes_{p\neq q} p^{|\mathsf{Q}(p)|} \otimes q^{|w|+1} \\ &= \otimes_{p\neq q} p^{|\mathsf{Q}(p)|} \otimes q^{|w|} \otimes q \\ &= \otimes_{p\neq q} p^{|\mathsf{Q}[q\mapsto w](p)|} \otimes q^{|w|} \otimes q \\ &= shape(\mathsf{D}) \otimes q. \end{aligned}$$

Claim 4: If $\mathsf{C} \xrightarrow{(write,a,q)} \mathsf{D}$ then $shape(\mathsf{D}) = shape(\mathsf{C}) \otimes q$.

Proof For $\mathsf{C} \xrightarrow{(write,a,q)} \mathsf{D}$ to hold, rule μ requires $\mathsf{Q}(q) \xrightarrow{(write,a,q)} w$ for $\mathsf{C} = (\mathsf{Q},\mathsf{M})$ and $\mathsf{D} = (\mathsf{Q}[q\mapsto w],\mathsf{M})$. By claim 2, $w = \mathsf{Q}(q)a$ and $|w| = |\mathsf{Q}(q)|+1$ hold. Hence, we obtain

$$\begin{aligned} shape(\mathsf{D}) &= \otimes_{p\neq q} p^{|\mathsf{Q}[q\mapsto w](p)|} \otimes q^{|w|} \\ &= \otimes_{p\neq q} p^{|\mathsf{Q}(p)|} \otimes q^{|\mathsf{Q}(q)|+1} \\ &= \otimes_{p\neq q} p^{|\mathsf{Q}(p)|} \otimes q^{|\mathsf{Q}(q)|} \otimes q \\ &= shape(\mathsf{C}) \otimes q. \end{aligned}$$

CODE **Claim 5:** If $\mathsf{C} \xrightarrow{code} \mathsf{D}$ then $[n]code : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.

Proof The claim is proven by a case distinction on *code*, and we treat the case ADD explicitly.

Case $code = \mathtt{add}\ op_1\ op_2\ op_3$**.** For $\mathsf{C} \xrightarrow{\mathtt{add}\ op_1\ op_2\ op_3} \mathsf{D}$, the definition of rule ADD yields the existence of configurations E and F such that

$$\mathsf{C} \xrightarrow{(read(\mathtt{ALU}),a,q_1)} \mathsf{E}, \quad \mathsf{E} \xrightarrow{(read(\mathtt{ALU}),b,q_2)} \mathsf{F} \text{ and } \mathsf{F} \xrightarrow{(write,a+b,q_3)} \mathsf{D}$$

hold, where $q_i = op_i$ holds for $i \in \{1,2,3\}$ by the assumption that all operand identifiers are queue names. By applying claim 3 to the statement $\mathsf{E} \xrightarrow{(read(\mathtt{ALU}),b,q_2)} \mathsf{F}$ we obtain $shape(\mathsf{E}) = shape(\mathsf{F}) \otimes q_2$ and $\gamma(q_2) = \mathtt{ALU}$. Likewise, applying claim 3 to $\mathsf{C} \xrightarrow{(read(\mathtt{ALU}),a,q_1)} \mathsf{E}$ yields $shape(\mathsf{C}) = shape(\mathsf{E}) \otimes q_1$ and $\gamma(q_1) = \mathtt{ALU}$. Applying claim 4 to $\mathsf{F} \xrightarrow{(write,a+b,q_3)} \mathsf{D}$ results in $shape(\mathsf{D}) = shape(\mathsf{F}) \otimes q_3$.
Summarising, we have

$$\begin{aligned} shape(\mathsf{C}) &= shape(\mathsf{E}) \otimes q_1 = shape(\mathsf{F}) \otimes q_2 \otimes q_1, \\ shape(\mathsf{D}) &= shape(\mathsf{F}) \otimes q_3 \end{aligned}$$

and $\gamma(q_1) = \gamma(q_2) = \mathtt{ALU} = FU(\mathtt{add}\ op_1\ op_2\ op_3)$. Using the weakening $X = shape(\mathsf{F})$, axiom AX yields $[n]\mathtt{add}\ op_1\ op_2\ op_3 : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.

Cases $code \neq \mathtt{add}\ op_1\ op_2\ op_3$**.** Similar.

For proving the main claim, we again perform a structural induction on t.

If $t = \varepsilon$**,** no judgement $\mathsf{C} \xrightarrow{t} \mathsf{D}$ is derivable, so the claim is trivially fulfilled.

If $t = ins = [n]code$**,** then the last rule used for deriving $\mathsf{C} \xrightarrow{t} \mathsf{D}$ must have been INSTR, so we must have $\mathsf{C} \xrightarrow{code} \mathsf{D}$, and applying claim 5 yields $ins : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.

If $t = s\,r$**,** then the last rule in the derivation for $\mathsf{C} \xrightarrow{t} \mathsf{D}$ must have been COMP, so there is an E with $\mathsf{C} \xrightarrow{s} \mathsf{E}$ and $\mathsf{E} \xrightarrow{r} \mathsf{D}$. Applying the induction hypothesis yields $s : shape(\mathsf{C}) \multimap shape(\mathsf{E})$ and $r : shape(\mathsf{E}) \multimap shape(\mathsf{D})$, so the rule CUT implies $s\,r : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.

□

The type system presented in this section is based on the same architectural information as the operational model: the mapping of instructions to functional units and the binding of operand queues to functional units are identical. While this architectural transparency is useful during system design, type-checkers in realistic compilers are not expected to have full architectural information available. Instead, they will derive architectural conditions which a processor has to fulfil in order to be admissible. We will discuss an extension of our framework which might support such reasoning in Chapter 9.

3.4 Principal types

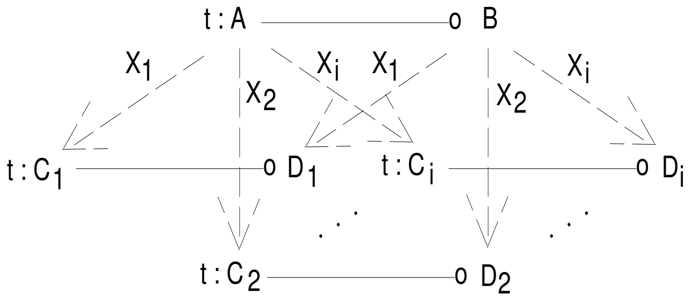
The type system admits a derived weakening rule

$$\text{WK}\,\frac{t : A \multimap B}{t : A \otimes X \multimap B \otimes X}$$

where X is arbitrary. Operationally, typings have different effects on the function a program calculates. For example, if program (3.2) is given type $\mathbf{1} \multimap \mathtt{q}_1$ and $\mathtt{C}$ is a configuration of shape $\mathbf{1}$, the first instruction will insert the value $\mathtt{4}$ into the (initially empty) queue $\mathtt{q}_2$, and the second instruction will decrement this value by one and hence deposit $\mathtt{3}$ in $\mathtt{q}_1$. If a typing $A \multimap B$ is chosen where $A = \mathtt{q}_2 \otimes X$ for some X, then a configuration of shape A will contain an initial value in $\mathtt{q}_2$. The second instruction will consequently decrement that initial value rather than the value provided by `ldc 4` $\mathtt{q}_2$. Hence, a different input-output behaviour is observed.

The *legality* of a program with respect to sequential execution is not affected by the weakening rule. For straight-line code, the following notion of principal type can be defined.

Definition 2. *A typing $t : A \multimap B$ is* principal *if for every typing $t : C \multimap D$ there is an X such that $C = A \otimes X$ and $D = B \otimes X$.*



Every well-typed program has a unique principal type which can be obtained by type inference using a modified type system with judgements of the form $t :: A \multimap B$. Axioms are of the non-weakened form

$$\text{P-AX}\ \frac{}{[n]code :: A_{code} \multimap B_{code}}\ SC(code)$$

and composition performs the minimal weakening necessary for a cut.

$$\text{P-CUT}\ \frac{s :: A \multimap B \qquad t :: C \multimap D}{st :: A \otimes U \multimap V \otimes D}\ B/C = V/U$$

Here (V,U) is the cancellation of (B,C), defined as follows.

Definition 3. *Products A and B are* relatively prime*, written $prime(A,B)$, if the factors of A and B are disjoint.*

Types V and U are called the cancellation *of A and B, written $A/B = V/U$, if V and U are relatively prime and $U \otimes A = V \otimes B$.*

Occasionally, we write $A/B =_C V/U$ where the greatest common divisor C is given by the (unique) product such that $A = C \otimes V$ and $B = C \otimes U$.

Proposition 3. *$t :: A \multimap B$ iff $t : A \multimap B$ is the principal typing for t.*

Proof. Structural induction on t.

If $t = \varepsilon$, there is no derivation for $t :: A \multimap B$ or $t : A \multimap B$, so the claim is trivially fulfilled.

If $t = [n]code$, we perform a case distinction on *code*.

Case $code = \texttt{add}\ op_1\ op_2\ op_3$**.** If $t :: A \multimap B$ then the side condition $SC(code)$ is fulfilled and $A = op_1 \otimes op_2$ and $B = op_3$. Take $X = \mathbf{1}$ to obtain a valid typing $t : A \multimap B$. Suppose $t : C \multimap D$ is another typing for t, then by the typing rule for `add` we have $C = op_1 \otimes op_2 \otimes Y$ and $D = op_3 \otimes Y$ for some Y, so $C \multimap D$ can be obtained from $t : A \multimap B$ by weakening.

Conversely, suppose $t : A \multimap B$ is the principal typing for t. Then $t : A \multimap B$ holds, so $\gamma(op_1) = \gamma(op_2) = \texttt{ALU}$ and $A = op_1 \otimes op_2 \otimes X$ and $B = op_3 \otimes X$ for some X. Take $C = op_1 \otimes op_2$ and $D = op_3$, then $t :: C \multimap D$ and $t : C \multimap D$ hold since $\gamma(op_1) = \gamma(op_2) = \texttt{ALU}$. Since $t : A \multimap B$ is the principal typing for t, there must by a Y such that $op_1 \otimes op_2 = C = A \otimes Y = op_1 \otimes op_2 \otimes X \otimes Y$ and $op_3 = D = B \otimes Y = op_3 \otimes X \otimes Y$. Consequently, $X = Y = \mathbf{1}$ and $A = C$ and $B = D$.

Cases $code \neq \texttt{add}\ op_1\ op_2\ op_3$**.** Similar.

If $t = s\,r$, then $t :: A \multimap B$ implies that there are C, D, E, F such that

$$\text{P-CUT}\ \frac{s :: C \multimap D \quad r :: E \multimap F}{t :: C \otimes U \multimap V \otimes F}\ D/E = V/U$$

where $A = C \otimes U$ and $B = V \otimes F$. Consequently, $U \otimes D = V \otimes E$. By induction hypothesis, we have $s : C \multimap D$ and $r : E \multimap F$, so $s : C \otimes U \multimap D \otimes U$ and $r : E \otimes V \multimap F \otimes V$ by weakening. By the cut rule, we obtain $s\,r : C \otimes U \multimap F \otimes V$, i.e. $s\,r : A \multimap B$ as a valid typing. Suppose $s\,r : G \multimap H$. Then there is an I such that $s : G \multimap I$ and $r : I \multimap H$. By induction hypothesis, $s :: C \multimap D$ and $r :: E \multimap F$ are principal, so there are X and Y with $G = C \otimes X$, $I = D \otimes X$, $I = E \otimes Y$ and $H = F \otimes Y$. Consequently, $D \otimes X = I = E \otimes Y$. Since V and U are prime and fulfil $U \otimes D = V \otimes E$, it can be shown that there must be a Z with $X = U \otimes Z$ and $Y = V \otimes Z$. Hence, $G = C \otimes X = C \otimes U \otimes Z = A \otimes Z$ and $H = F \otimes Y = F \otimes V \otimes Z = B \otimes Z$, so $t : G \multimap H$ can be obtained from $t : A \multimap B$ by weakening.

Conversely, let $s\,r : A \multimap B$ be the principal typing for t. Then there is a C such that $s : A \multimap C$ and $r : C \multimap B$. Let $s : D \multimap E$ and $r : F \multimap G$ be the principal typings for s and r, respectively. Then by induction hypothesis, $s :: D \multimap E$ and $r :: F \multimap G$ hold, and by principality there are X and Y such that $A = D \otimes X$, $E \otimes X = C = F \otimes Y$ and $B = G \otimes Y$. Let $E/F = V/U$. Then $s\,r :: D \otimes U \multimap V \otimes G$. Since V and U are prime, there is a Z such that $U \otimes Z = X$ and $V \otimes Z = Y$. Consequently, $A = D \otimes X = D \otimes U \otimes Z$ and $B = G \otimes Y = G \otimes V \otimes Z$. Since $s\,r :: D \otimes U \multimap V \otimes G$ implies $s\,r : D \otimes U \multimap V \otimes G$, we have $Z = 1$ by the principality of $s\,r : A \multimap B$. Thus $s\,r :: A \multimap B$.

$\square$

We will see in the next section that weakening plays an important role for programs involving jump instructions.

3.5 Program execution

In this section we consider the language including the conditional and unconditional jumps `if` *op* n_1 n_2 and `jmp` n, but no registers. Both dynamic and static semantics are defined in terms of *basic blocks*, i.e. sequences of instructions where at most the last instruction is a jump instruction and no instruction except the first one is the target of an incoming control-flow arrow.

Formally, a program $\mathtt{P} = (\mathtt{N}, \mathtt{A})$ consists of a partial map $\mathtt{N} : \mathbf{N} \rightharpoonup \mathit{Iseq}$, together with an association $\mathtt{A} \subset \mathit{dom}\,\mathtt{N} \times \mathit{dom}\,\mathtt{N}$ such that

- the labelling of instructions is unique
- the first instruction of $\mathtt{N}(n)$ has label $[n]$
- at most the last instruction of $\mathtt{N}(n)$ is a jump instruction
- $\mathtt{A}$ contains exactly those pairs (n,m) where the opcode of last instruction in $\mathtt{N}(n)$ is either $\mathtt{jmp}\ m$ or $\mathtt{if}\ op\ n_1\ n_2$ with $m \in \{n_1, n_2\}$.

3.5.1 Dynamic semantics

We modify the dynamic semantics from Section 3.2 by extending configurations by a third component. In a configuration $(\mathsf{Q},\mathsf{M},\widetilde{n})$ the component $\widetilde{n} \in \mathbf{N} \cup \{nil\}$ represents the optional label of the basic block to be executed next. Operand queue configuration Q and memory component M are as before, and the effect of opcodes is still defined in terms of the relation $(\mathsf{Q},\mathsf{M}) \xrightarrow{(\mu,a,q)} (\mathsf{P},\mathsf{N})$. The dynamic semantics for instructions is given in Figure 3.8, embedding the rules from Figure 3.4 by the rule INJ. Additional rules define the meaning of jump instructions. These insert the branch target into the third component of a configuration, possibly depending on an evaluation of a branch condition. Both jump instructions are mapped to the functional unit $\mathtt{BU}$ (*branch unit*), and there are two rules for the opcode $\mathtt{if}\ op\ n_1,\ n_2$, one for each possible outcome of the branch condition. All jump instructions require the third component of their initial configuration to be empty, in accordance with the definition of the rules for issuing basic blocks which will be given shortly. The rules INSTR and COMP are formally identical to the rules in Figure 3.5, but now relate configurations of the extended form. A typical derivation for a statement $\mathsf{C} \xrightarrow{t} \mathsf{D}$ is shown in Figure 3.9, where

$$\begin{aligned} \mathsf{C} &= ([\,],\mathsf{M},nil) \\ \mathsf{E} &= ([\mathtt{q}_1 \mapsto 4],\mathsf{M},nil) \\ \mathsf{D} &= ([\,],\mathsf{M},5) \end{aligned}$$

Determinacy of sequential execution carries over to the extended dynamic semantics, i.e. $\mathsf{D} = \mathsf{E}$ holds provided that $\mathsf{C} \xrightarrow{t} \mathsf{D}$ and $\mathsf{C} \xrightarrow{t} \mathsf{E}$.

Programs $\mathtt{P}$ are executed by issuing the constituent basic blocks according to the outcome of branches. The corresponding relation is $\mathsf{C} \xrightarrow{w} \mathsf{D}$ where $w \in \mathbf{N}^*$ is a word over

$$\text{IF-T}\ \frac{(\mathsf{Q},\mathsf{M}) \xrightarrow{(\mathit{read}(\mathtt{BU}),0,\mathit{op})} (\mathsf{P},\mathsf{N})}{(\mathsf{Q},\mathsf{M},\mathit{nil}) \xrightarrow{\mathtt{if}\ \mathit{op}\ n_1\ n_2} (\mathsf{P},\mathsf{N},n_1)}$$

$$\text{IF-F}\ \frac{(\mathsf{Q},\mathsf{M}) \xrightarrow{(\mathit{read}(\mathtt{BU}),a,\mathit{op})} (\mathsf{P},\mathsf{N})}{(\mathsf{Q},\mathsf{M},\mathit{nil}) \xrightarrow{\mathtt{if}\ \mathit{op}\ n_1\ n_2} (\mathsf{P},\mathsf{N},n_2)}\ a \neq 0$$

$$\text{JMP}\ \frac{--}{(\mathsf{Q},\mathsf{M},\mathit{nil}) \xrightarrow{\mathtt{jmp}\ n} (\mathsf{Q},\mathsf{M},n)}$$

$$\text{INJ}\ \frac{(\mathsf{Q},\mathsf{M}) \xrightarrow{\mathit{code}} (\mathsf{P},\mathsf{N})}{(\mathsf{Q},\mathsf{M},\widetilde{n}) \xrightarrow{\mathit{code}} (\mathsf{P},\mathsf{N},\widetilde{n})}\ \mathit{FU}(\mathit{code}) \neq \mathtt{BU}$$

$$\text{INSTR}\frac{\mathsf{C} \xrightarrow{\mathit{code}} \mathsf{D}}{\mathsf{C} \xrightarrow{[n]\mathit{code}} \mathsf{D}} \qquad\qquad \text{COMP}\frac{\mathsf{C} \xrightarrow{s} \mathsf{E} \quad \mathsf{E} \xrightarrow{t} \mathsf{D}}{\mathsf{C} \xrightarrow{st} \mathsf{D}}$$

Figure 3.8: Sequential dynamic semantics including jumps

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\lambda \xrightarrow{(\mathit{write},4,\mathsf{q}_1)} 4}}{([\,],\mathsf{M}) \xrightarrow{(\mathit{write},4,\mathsf{q}_1)} ([\mathsf{q}_1 \mapsto 4],\mathsf{M})}}{([\,],\mathsf{M}) \xrightarrow{\mathtt{ldc}\ 4\ \mathsf{q}_1} ([\mathsf{q}_1 \mapsto 4],\mathsf{M})}}{\mathsf{C} \xrightarrow{\mathtt{ldc}\ 4\ \mathsf{q}_1} \mathsf{E}}}{\mathsf{C} \xrightarrow{[1]\mathtt{ldc}\ 4\ \mathsf{q}_1} \mathsf{E}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\gamma(\mathsf{q}_1) = \mathtt{BU}}{4 \xrightarrow{(\mathit{read}(\mathtt{BU}),4,\mathsf{q}_1)} \lambda}}{([\mathsf{q}_1 \mapsto 4],\mathsf{M}) \xrightarrow{(\mathit{read}(\mathtt{BU}),4,\mathsf{q}_1)} ([\,],\mathsf{M})}}{\mathsf{E} \xrightarrow{\mathtt{if}\ \mathsf{q}_1\ 3\ 5} \mathsf{D}}\ 4 \neq 0}{\mathsf{E} \xrightarrow{[2]\mathtt{if}\ \mathsf{q}_1\ 3\ 5} \mathsf{D}}}{\mathsf{C} \xrightarrow{[1]\mathtt{ldc}\ 4\ \mathsf{q}_1\ [2]\mathtt{if}\ \mathsf{q}_1\ 3\ 5} \mathsf{D}}$$

Figure 3.9: Example derivation for $\mathsf{C} \xrightarrow{t} \mathsf{D}$ including jumps.

$$\text{EXECUTE}\,\frac{(\mathsf{Q},\mathsf{M},\mathit{nil}) \xrightarrow{\mathtt{N}(n)} \mathsf{D}}{(\mathsf{Q},\mathsf{M},n) \xrightarrow{n} \mathsf{D}} \qquad \text{COMPOSE}\,\frac{\mathsf{C} \xrightarrow{v} \mathsf{E} \quad \mathsf{E} \xrightarrow{w} \mathsf{D}}{\mathsf{C} \xrightarrow{vw} \mathsf{D}}$$

Figure 3.10: Sequential dynamic semantics for programs $\mathtt{P} = (\mathtt{N}, \mathtt{A})$.

N. The rules for $\mathsf{C} \xrightarrow{w} \mathsf{D}$ are given in Figure 3.10. The rule EXECUTE represents the second component of the interplay between the execution of jump instructions and the issuing of basic blocks. As issuing the instructions of the target block deletes the basic block label n in the third component of the initial configuration, a jump at the end of $\mathtt{N}(n)$ will again be able to execute.
By an induction on $|w| > 0$ one can show that program execution is deterministic.

Proposition 4. *If* $\mathsf{C} \xrightarrow{w} \mathsf{D}$, $\mathsf{C} \xrightarrow{v} \mathsf{E}$ *and* $|w| = |v| > 0$ *then* $v = w$ *and* $\mathsf{D} = \mathsf{E}$.

Proof. For $|w| = 1$, $w = n$ holds for some n and $\mathsf{C} \xrightarrow{n} \mathsf{D}$ is only derivable if C has the form $(\mathsf{Q},\mathsf{M},n)$ with $\mathtt{N}(n) = t$ and $(\mathsf{Q},\mathsf{M},\mathit{nil}) \xrightarrow{t} \mathsf{D}$. Likewise, $v = m$ holds for some m and $\mathsf{C} \xrightarrow{m} \mathsf{E}$ is only derivable if C has the form $(\mathsf{Q},\mathsf{M},m)$ with $\mathtt{N}(m) = s$ and $(\mathsf{Q},\mathsf{M},\mathit{nil}) \xrightarrow{s} \mathsf{E}$. Since C is unique it must be $n = m$, so $v = w$ and $t = \mathtt{N}(n) = \mathtt{N}(m) = s$, and therefore $\mathsf{D} = \mathsf{E}$ by (generalised) Proposition 1.
For $|w| = |v| > 1$, there are non-empty w_1, w_2 and F and non-empty v_1, v_2 and G such that the last steps in the derivations for $\mathsf{C} \xrightarrow{w} \mathsf{D}$ and $\mathsf{C} \xrightarrow{v} \mathsf{E}$ are

$$\text{COMPOSE}\,\frac{\mathsf{C} \xrightarrow{w_1} \mathsf{F} \quad \mathsf{F} \xrightarrow{w_2} \mathsf{D}}{\mathsf{C} \xrightarrow{w} \mathsf{D}} \quad \text{and} \quad \text{COMPOSE}\,\frac{\mathsf{C} \xrightarrow{v_1} \mathsf{G} \quad \mathsf{G} \xrightarrow{v_2} \mathsf{E}}{\mathsf{C} \xrightarrow{v} \mathsf{E}}$$

Without loss of generality, $|w_1| = |v_1|$ holds – any derivation where the leftmost branch $\mathsf{C} \xrightarrow{u} \mathsf{H}$ does not fulfil $|u| = 1$ can be transformed into a derivation for $\mathsf{C} \xrightarrow{u} \mathsf{H}$ where the leftmost branch does fulfil $|u| = 1$ by reordering the axioms. Hence, we obtain $w_1 = v_1$ and $\mathsf{F} = \mathsf{G}$ by induction hypothesis as w_1 and v_1 are non-empty. Also, $|w_2| = |w| - |w_1| = |v| - |v_1| = |v_2|$ follows, so by applying the induction hypothesis again we obtain $\mathsf{D} = \mathsf{E}$. □

By the definition of programs, an execution will never attempt to issue a basic block for which no code exists. However, programs might experience deadlocks due to operand queue underflow at the boundary between basic blocks. The following section presents an extension of the static semantics which eliminates these errors.

$code$	$FU(code)$	A_{code}	B_{code}
$\texttt{if}\ op\ n_1\ n_2$	BU	op	$\mathbf{1}$
$\texttt{jmp}\ n$	BU	$\mathbf{1}$	$\mathbf{1}$

Table 3.2: Type system including jumps

3.5.2 Static semantics

Basic blocks can be typed using the type system from Section 3.3 if Table 3.1 is extended by the data given in Table 3.2. Instantiating the rule AX correspondingly amounts to the explicit rules

$$\text{AX-IF}\frac{}{[n]\texttt{if}\ op\ n_1\ n_2 : X \otimes op \multimap X}\ \gamma(op) = \texttt{BU}$$

and

$$\text{AX-JMP}\frac{}{[n]\texttt{jmp}\ m : X \multimap X}$$

with $X = \mathbf{1}$ for the calculus of principal types.

Theorem 1 (soundness and completeness for sequential execution) can be generalised to the bodies of basic blocks where $shape(\mathsf{C}) = shape(\mathsf{Q},\mathsf{M})$ for $\mathsf{C} = (\mathsf{Q},\mathsf{M},\tilde{n})$ generalises Definition 1.

For programs $\texttt{P} = (\texttt{N},\texttt{A})$ to be executable, the interfaces of basic blocks must be compatible. We follow an approach taken by Stata-Abadi [SA98c] where the shapes of basic blocks linked by a control flow dependency must match. All operands expected by a basic block must be provided by earlier basic blocks and all operands delivered must be either consumed or passed on to successor basic blocks. For example, if $\texttt{N}(1)$, $\texttt{N}(3)$ and $\texttt{N}(5)$ are of the form

$$\begin{array}{lcl} \texttt{N}(1) & = & [1]\texttt{ldc}\ 4\ \mathsf{q}_1\ [2]\texttt{jmp}\ 5 \\ \texttt{N}(3) & = & [3]\texttt{ldc}\ 7\ \mathsf{q}_1\ [4]\texttt{jmp}\ 5 \\ \texttt{N}(5) & = & [5]\texttt{dec}\ \mathsf{q}_1\ \mathsf{q}_2\ [6]\texttt{skip}\ \mathsf{q}_2 \end{array}$$

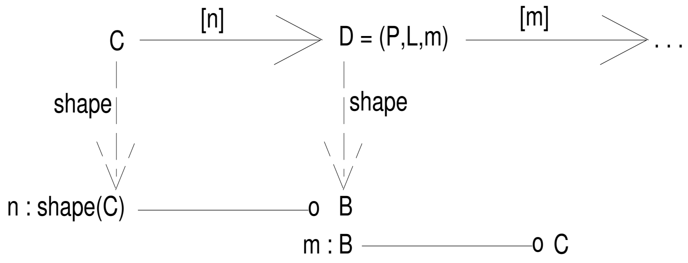
then $\texttt{N}(1)$ and $\texttt{N}(3)$ each provide exactly the right number of operands for $\texttt{N}(5)$.

Formally, Stata-Abadi's condition means that $B_n = A_m$ should hold whenever $\texttt{N}(n) : A_n \multimap B_n$, $\texttt{N}(m) : A_m \multimap B_m$ and $(n,m) \in \texttt{A}$. We hence introduce a judgement of the form $\Sigma \vdash \mathtt{P} : \circ$ where Σ is a *type environment* mapping basic block labels to types. For each basic block $\mathtt{N}(n)$, Σ has exactly one entry $n : A \multimap B$ where $\mathtt{N}(n) : A \multimap B$ must hold. The rule for $\Sigma \vdash \mathtt{P} : \circ$ then requires the above equality to hold for all arcs.

$$\text{PROG}\frac{\begin{array}{c}\Sigma = \cup_{n \in dom\,\mathtt{N}}\{n : A_n \multimap B_n\} \\ (n,m) \in \mathtt{A} \text{ implies } B_n = A_m\end{array}}{\Sigma \vdash \mathtt{P} : \circ}\ \mathtt{P} = (\mathtt{N}, \mathtt{A})$$

Consequently, basic blocks can be composed as required by the outcome of branches.

Proposition 5. *Let $\Sigma \vdash \mathtt{P} : \circ$, $\mathsf{C} = (\mathsf{Q}, \mathsf{M}, n)$ and let $n : shape(\mathsf{C}) \multimap B$ be the entry for n in Σ. Then there is a unique D such that $\mathsf{C} \xrightarrow{n} \mathsf{D}$. Furthermore, $shape(\mathsf{D}) = B$ holds, and if $\mathsf{D} = (\mathsf{P}, \mathsf{L}, m)$ and $m \neq nil$ then $m : B \multimap C$ holds for m's entry in Σ and some C.*



Proof. The definition of Σ implies $\mathtt{N}(n) : shape(\mathsf{C}) \multimap B$, so from Theorem 1 (extended to ternary configurations) we obtain $(\mathsf{Q}, \mathsf{M}, nil) \xrightarrow{\mathtt{N}(n)} \mathsf{D}$ for a unique D, and $shape(\mathsf{D}) = B$ holds. The rule EXECUTE consequently guarantees $\mathsf{C} \xrightarrow{n} \mathsf{D}$ as claimed. If D has the form $(\mathsf{P}, \mathsf{L}, m)$ with $m \neq nil$, then $\mathtt{N}(n)$ contains an instruction `jmp` m or `if` $op\ m_1\ m_2$ with $m \in \{m_1, m_2\}$. By the definition of programs $\mathtt{P}$ there is at most one such instruction in $\mathtt{N}(n)$, and it must be the last instruction, so $(n,m) \in \mathtt{A}$ holds, and $m \in dom\,\mathtt{N}$. Hence, there is an entry $m : A \multimap C$ in Σ, and by the second condition of the rule PROG, $\Sigma \vdash \mathtt{P} : \circ$ implies $B = A$. $\square$

Hence, programs either do not terminate due to loops or terminate in a well-defined configuration. No runtime errors related to the usage of the forwarding mechanism occur.

The type system is restrictive in that only one entry for each block is allowed in Σ. This is motivated by the desire to control the shape of final configurations. Likewise, a more generous type system where constraints $B_n = A_m$ are replaced by constraints $B_n = A_m \otimes$

X for arbitrary X would still guarantee the executability of programs. However, loops would be admitted which increase the size of a configuration in each iteration. Since the number of iterations can in general not be determined statically, the restriction $X = \mathbf{1}$ is necessary for controlling the size of configurations.

3.5.3 Type inference

Starting from typings $\mathtt{N}(n) : A_n \multimap B_n$ for the bodies of basic blocks, the typing rule for $\Sigma \vdash \mathtt{P} : \circ$ requires that the types of neighbouring basic blocks be *unifiable*. This means that factors X_i and X_j should exist such that for the weakenings $A_i \otimes X_i \multimap B_i \otimes X_i$ and $A_j \otimes X_j \multimap B_j \otimes X_j$, the equation $B_i \otimes X_i = A_j \otimes X_j$ holds whenever $(i, j) \in \mathtt{A}$. Type inference has to (dis)prove the existence of factors X_i, and we employ the following notion of unifier for this task.

Definition 4. *Let* $\mathtt{N}$ *be a partial map* $\mathtt{N} : \mathbf{N} \rightharpoonup Iseq$*, dom* $\mathtt{N} = \{n_1, \ldots, n_k\}$, $\mathtt{A} \subset dom\, \mathtt{N} \times dom\, \mathtt{N}$ *and for* $i \leq k$ *let* $\mathtt{N}(n_i) : A_i \multimap B_i$*. A tuple* $\vec{X} = (X_1, \ldots, X_k)$ unifies $\mathtt{N}$ for $\mathtt{A}$ *if* $(n_i, n_j) \in \mathtt{A}$ *implies* $B_i \otimes X_i = A_j \otimes X_j$*.*

The following proposition shows the appropriateness of unifiers for type inference.

Proposition 6. *Let* $\mathtt{P} = (\mathtt{N}, \mathtt{A})$ *be a program.*

1. *If* $\vec{X}$ *is a unifier for* $\mathtt{N}$ *and* $\mathtt{A}$ *and* $\Sigma = \cup_{n_i \in dom\, \mathtt{N}} n_i : A_i \otimes X_i \multimap B_i \otimes X_i$ *then* $\Sigma \vdash \mathtt{P} : \circ$*.*

2. *If* $\Sigma \vdash \mathtt{P} : \circ$ *and* $\Sigma = \cup_{n_i \in dom\, \mathtt{N}} n_i : A_i \multimap B_i$ *then* $\vec{\mathbf{1}}$ *unifies* $\mathtt{N}$ *for* $\mathtt{A}$*.*

Proof.

1. For $\Sigma = \cup_{n_i \in dom\, \mathtt{N}} n_i : A_i \otimes X_i \multimap B_i \otimes X_i$ and $(n_i, n_j) \in \mathtt{A}$ we have $B_i \otimes X_i = A_j \otimes X_j$ since $\vec{X}$ unifies $\mathtt{N}$ for $\mathtt{A}$. Consequently, all conditions in the rule for $\Sigma \vdash \mathtt{P} : \circ$ are fulfilled.

2. By the rule for $\Sigma \vdash \mathtt{P} : \circ$, $B_i = A_j$ must hold whenever $(n_i, n_j) \in \mathtt{A}$, where $\Sigma = \cup_{n_i \in dom\, \mathtt{N}} n_i : A_i \multimap B_i$. Consequently for $X_i = X_j = \mathbf{1}$ we have $B_i \otimes X_i = A_j \otimes X_j$ for all $(n_i, n_j) \in \mathtt{A}$.

$\square$

Of particular interest is the *minimal* unifier.

Definition 5. *For k-tuples $\vec{Y}$ and $\vec{Z}$ let $\vec{Y} \otimes \vec{Z} = (Y_1 \otimes Z_1, \dots, Y_k \otimes Z_k)$.*
If $\vec{X}$ unifies `N` *for* `A`*, it is* minimal *if for every $\vec{Y}$ which unifies* `N` *for* `A` *there is a $\vec{Z}$ such that $\vec{X} \otimes \vec{Z} = \vec{Y}$.*

For given `N` and `A` at most one minimal unifier exists.

Lemma 1. 1. *If $\vec{X}$ and $\vec{Y}$ unify* `N` *for* `A` *and $\vec{X} = \vec{Y} \otimes \vec{Z}$ for some $\vec{Z}$, then $Z_i = Z_j$ holds for all $(n_i, n_j) \in$* `A`*.

2. *If $\vec{X}$ and $\vec{Y}$ are minimal unifiers for* `N` *and* `A` *then $\vec{X} = \vec{Y}$.*

Proof. 1. Let $\vec{X}$ and $\vec{Y}$ unify `N` for `A` and let $\vec{X} = \vec{Y} \otimes \vec{Z}$. Then for $(n_i, n_j) \in$ `A` we have

$$A_j \otimes Y_j \otimes Z_j = A_j \otimes X_j = B_i \otimes X_i = B_i \otimes Y_i \otimes Z_i = A_j \otimes Y_j \otimes Z_i$$

and thus $Z_j = Z_i$. Consequently, $Z_i = Z_j$ whenever (n_i, n_j) is in the transitive closure of `A`. Closure under symmetry of `A` is trivial.

2. Since $\vec{X}$ is minimal and $\vec{Y}$ unifies `N` for `A`, there is a $\vec{Z}$ such that $\vec{X} \otimes \vec{Z} = \vec{Y}$. Likewise, minimality of $\vec{Y}$ implies the existence of a $\vec{W}$ such that $\vec{Y} \otimes \vec{W} = \vec{X}$. Together, we have $\vec{X} = \vec{Y} \otimes \vec{W} = \vec{X} \otimes \vec{Z} \otimes \vec{W}$, i.e. $X_i = X_i \otimes Z_i \otimes W_i$ for all i. This can only be fulfilled for $W_i = Z_i = \mathbf{1}$, so $X_i = Y_i$ as required.

□

The following theorem shows how to obtain a minimal unifier for `A`$\cup \{(n,m)\}$ from one for `A` provided that it exists.

Theorem 2. *Suppose $\vec{X}$ is the minimal unifier for* `N` *and* `A` *and for $n, m \in dom$* `N` *let $a = (m,n) \notin$* `A`*. Let*

$$S = \{l \,|\, (l,m) \in \texttt{A}^*\},\ T = \{l \,|\, (l,n) \in \texttt{A}^*\}$$

and $(B_m \otimes X_m)/(A_n \otimes X_n) = V/U$. Then

1. *$\vec{X}$ is the minimal unifier for* `A`$\cup\{a\}$ *if $a \in$* `A`* *and $V \otimes U = \mathbf{1}$.*

2. `A`$\cup\{a\}$ *has no unifier if $a \in$* `A`* *and $V \otimes U \neq \mathbf{1}$.*

3. *if $a \notin \mathtt{A}^*$, then $S \cap T = \emptyset$ and $\vec{Y}$ is the minimal unifier for $\mathtt{A} \cup \{a\}$, where*

$$Y_l = \begin{cases} X_l \otimes U & \textit{if } l \in S \\ X_l \otimes V & \textit{if } l \in T \\ X_l & \textit{otherwise} \end{cases}$$

Proof. The proof consists of four parts.

1. Show that $\vec{X}$ is the minimal unifier for $\mathtt{N}$ and $\mathtt{A} \cup \{a\}$ if $a \in \mathtt{A}^*$ and $V \otimes U = \mathbf{1}$.

   If $(i, j) \in \mathtt{A}$ then $B_i \otimes X_i = A_j \otimes A_j$ since $\vec{X}$ unifies $\mathtt{N}$ for $\mathtt{A}$.

   If $(i, j) = (m, n)$ then

   $$B_m \otimes X_m = B_m \otimes X_m \otimes U = A_n \otimes X_n \otimes V = A_n \otimes X_n.$$

   Therefore, $\vec{X}$ unifies $\mathtt{N}$ for $\mathtt{A} \cup \{a\}$.

   Suppose that $\vec{Y}$ is minimal unifier for $\mathtt{A} \cup \{a\}$. Then $\vec{Y}$ unifies $\mathtt{A}$, so $\vec{X} \otimes \vec{Z} = \vec{Y}$ for some $\vec{Z}$ by the minimality of $\vec{X}$ for $\mathtt{A}$. But $\vec{Y}$ is minimal for $\mathtt{A} \cup \{a\}$ and $\vec{X}$ unifies $\mathtt{A} \cup \{a\}$, so $\vec{Y} \otimes \vec{W} = \vec{X}$ for some $\vec{W}$. Consequently, $\vec{Z} = \vec{W} = (\mathbf{1}, \ldots, \mathbf{1})$.

   Therefore, $\vec{X}$ is minimal for $\mathtt{N}$ and $\mathtt{A} \cup \{a\}$.

2. Show that $\mathtt{A} \cup \{a\}$ has no unifier if $a \in \mathtt{A}^*$ and $V \neq \mathbf{1}$ or $U \neq \mathbf{1}$.

   Suppose $V \otimes U \neq \mathbf{1}$ and $a \in \mathtt{A}^*$ and $\vec{Y}$ unifies $\mathtt{N}$ for $\mathtt{A} \cup \{a\}$. Then $\vec{Y}$ also unifies $\mathtt{N}$ for $\mathtt{A}$, so $\vec{X} \otimes \vec{Z} = \vec{Y}$ for some $\vec{Z}$. By Lemma 1(1) we have $Z_m = Z_n$ since $a = (m, n) \in \mathtt{A}^*$. Consequently,

   $$Z_m \otimes B_m \otimes X_m = B_m \otimes Y_m = A_n \otimes Y_n = A_n \otimes X_n \otimes Z_n = A_n \otimes X_n \otimes Z_m$$

   and hence $B_m \otimes X_m = A_n \otimes X_n$, in contradiction to $V \otimes U \neq \mathbf{1}$.

3. Show that for $a \notin \mathtt{A}^*$ the vector $\vec{Y}$ as given in the theorem unifies $\mathtt{N}$ for $\mathtt{A} \cup \{a\}$.

   If $(i, j) \in \mathtt{A}$, then $(i, j) \in \mathtt{A}^*$, so one of the following three cases applies.

   - $\{i, j\} \subset S$. We have $Y_i = X_i \otimes U$ and $Y_j = X_j \otimes U$, so

     $$B_i \otimes Y_i = B_i \otimes X_i \otimes U = A_j \otimes X_j \otimes U = A_j \otimes Y_j$$

     because $\vec{X}$ unifies $\mathtt{A}$.

- $\{i,j\} \subset T$. Similar to the first case, we obtain $B_i \otimes Y_i = A_j \otimes Y_j$ due to $Y_i = X_i \otimes V$ and $Y_j = X_j \otimes V$.
- $\{i,j\} \cap (S \cup T) = \emptyset$. We have $B_i \otimes Y_i = B_i \otimes X_i = A_j \otimes X_j = A_j \otimes Y_j$.

If $(i,j) = (m,n)$, then $m \in S$ and $n \in T$ and thus $Y_m = X_m \otimes U$ and $Y_n = X_n \otimes V$. Consequently,

$$B_m \otimes Y_m = B_m \otimes X_m \otimes U = A_n \otimes X_n \otimes V = A_n \otimes Y_n.$$

4. Show that for $a \notin \mathtt{A}^* \ \vec{Y}$ is minimal for $\mathtt{A} \cup \{a\}$.

   Assume that $\vec{K} \neq \vec{Y}$ is minimal unifier for $\mathtt{A} \cup \{a\}$. Then there is a $\vec{Z}$ such that $\vec{Y} = \vec{K} \otimes \vec{Z}$, and $\vec{K} \neq \vec{Y}$ implies that $Z_\alpha \neq \mathbf{1}$ holds for some $\alpha \leq k$, where k is the length of $\vec{Z}$.

   One of the following three cases applies.

   (a) $\alpha \in S$. We have $Z_l = Z_\alpha$ for all $l \in S$ by Lemma 1(1) because $\vec{Y}$ and $\vec{K}$ both unify $\mathtt{A}$. Therefore,

   $$\forall\, l \in S.\ K_l \otimes Z_\alpha = Y_l = X_l \otimes U \tag{3.4}$$

   In particular, $m \in S$, and thus

   $$A_n \otimes K_n \otimes Z_\alpha = B_m \otimes K_m \otimes Z_\alpha = B_m \otimes X_m \otimes U = A_n \otimes X_n \otimes V$$

   and thus

   $$K_n \otimes Z_\alpha = X_n \otimes V. \tag{3.5}$$

   For $l \in T$ we have $Y_l = X_l \otimes V$, so $X_l \otimes V = K_l \otimes Z_l$. In particular, $n \in T$ and thus

   $$X_n \otimes V = K_n \otimes Z_n. \tag{3.6}$$

   Combining (3.6) and (3.5), we obtain $Z_n = Z_\alpha$. Applying Lemma 1(1) again, we obtain $Z_l = Z_\alpha$ for all $l \in T$.

Let $U/Z_\alpha =_M A/B$ and $V/Z_\alpha =_N C/D$. Using equation (3.4) we obtain

$$X_l \otimes M \otimes A = X_l \otimes U = K_l \otimes Z_\alpha = K_l \otimes M \otimes B$$

and hence $X_l \otimes A = K_l \otimes B$ for $l \in S$. Since A and B are prime this implies that there are W_l with $K_l = A \otimes W_l$ and hence $X_l = W_l \otimes B$ for all $l \in S$. Similarly for $l \in T$, we have

$$X_l \otimes N \otimes C = X_l \otimes V = Y_l = K_l \otimes Z_l = K_l \otimes Z_\alpha = K_l \otimes N \otimes D$$

and thus $X_l \otimes C = K_l \otimes D$. Therefore, for $l \in T$ there are W_l such that $K_l = C \otimes W_l$ and $X_l = W_l \otimes D$.
For $l \notin (S \cup T)$ take $W_l = X_l$.

For showing that $\vec{W}$ unifies $\mathtt{A}$, let $(i, j) \in \mathtt{A}$. There are three cases.

i. $\{i, j\} \subset S$. Then $B_i \otimes W_i \otimes B = B_i \otimes X_i = A_j \otimes X_j = A_j \otimes W_j \otimes B$.

ii. $\{i, j\} \subset T$. Then $B_i \otimes W_i \otimes D = B_i \otimes X_i = A_j \otimes X_j = A_j \otimes W_j \otimes D$.

iii. $\{i, j\} \cap (S \cup T) = \emptyset$. Then $B_i \otimes W_i = B_i \otimes X_i = A_j \otimes X_j = A_j \otimes W_j$.

In all three cases we obtain $B_i \otimes W_i = A_j \otimes W_j$.
Also, $\vec{W}$ occurs in $\vec{X}$ via $\vec{X} = \vec{W} \otimes \vec{L}$ where

$$L_l = \begin{cases} B & \text{if } l \in S \\ D & \text{if } l \in T \\ \mathbf{1} & \text{otherwise} \end{cases}$$

In order to obtain a contradiction to the minimality of $\vec{X}$ for $\mathtt{A}$, suppose $B = D = \mathbf{1}$. Then $N = Z_\alpha = M$ holds, and $U = M \otimes A$ and $V = N \otimes C = M \otimes C$ imply $M = \mathbf{1}$ because U and V are prime. Consequently, $Z_\alpha = M = \mathbf{1}$, in contradiction to the initial assumption $Z_\alpha \neq \mathbf{1}$. Therefore $B \neq \mathbf{1}$ or $D \neq \mathbf{1}$ and so $L_l \neq \mathbf{1}$ for $l = m$ or $l = n$ which means that $\vec{X}$ is not minimal.

(b) $\alpha \in T$. Similar to case (4a)

(c) $\alpha \notin S \cup T$. Let $R = \{l \mid (l, \alpha) \in \mathtt{A}^*\}$ and $Q = \{l \mid l \notin S \cup T \cup R\}$. Then S, T, R and Q are mutually disjoint and by Lemma 1(1) $Z_i = Z_\alpha$ holds for all $l \in R$. Take $W_l = K_l$ for $l \in R$ and $W_l = X_l$ for $l \notin R$.

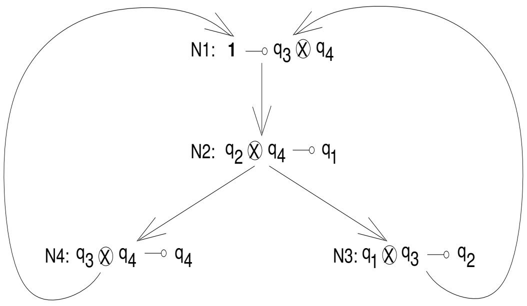


Figure 3.11: Example program for type inference via unification

Then $\vec{W}$ unifies A: For $\{i, j\} \subset R$ we have

$$\begin{aligned} Z_\alpha \otimes B_i \otimes W_i &= B_i \otimes K_i \otimes Z_i = B_i \otimes Y_i \\ &= A_j \otimes Y_j = A_j \otimes K_j \otimes Z_j = Z_\alpha \otimes A_j \otimes W_j \end{aligned}$$

and thus $B_i \otimes W_i = A_j \otimes W_j$. For $\{i, j\} \cap R = \emptyset$, $B_i \otimes W_i = A_j \otimes W_j$ holds trivially.

Again, we obtain a contradiction to the minimality of $\vec{X}$ for A because $\vec{X} = \vec{W} \otimes \vec{L}$ where $L_l = Z_\alpha \neq \mathbf{1}$ for $l \in R$ and $L_l = \mathbf{1}$ for $l \notin R$ and $R \neq \emptyset$.

□

Consequently, type inference for $\mathtt{P} = (\mathtt{N}, \mathtt{A})$ can proceed by typing the bodies of all basic blocks separately and then visiting the arcs A in any order. An algorithm based on Theorem 2 either successively calculates minimal unifiers for arcs $\emptyset \subset \mathtt{A}_1 \subset \ldots \subset \mathtt{A}_n = \mathtt{A}$ or rejects the program as being ill-formed.

Example. Consider the program P with basic blocks N1, ... , N4 and arrows and initial typings as shown in Figure 3.11.

Visiting the arrow (N1, N2) first, we find that $X_1 = \mathsf{q}_2$ and $X_2 = \mathsf{q}_3$ are needed for unifying B_1 and A_2. Visiting (N2, N3) next, the extended product $B_2 \otimes X_2 = \mathsf{q}_1 \otimes \mathsf{q}_3$ is equal to $A_3 \otimes \mathbf{1}$, so no further weakening is needed for unifying N_2 and N_3 and we define $X_3 = \mathbf{1}$. Closing the right loop is possible when visiting (N3, N1) next as $B_3 \otimes X_3 = \mathsf{q}_2 = A_1 \otimes X_1$ holds.

Visiting $(\mathtt{N2},\mathtt{N4})$ next, we see that N2 has to be further weakened by $\mathtt{q}_4$ in order to unify with A_4. N4 in turn has to be weakened by $\mathtt{q}_1$, so we define $Y_4 = \mathtt{q}_1$ and $Y_2 = X_2 \otimes \mathtt{q}_4$. In order to keep the earlier equations valid, we have to promote the new weakening to all blocks already connected to N2, i.e. define $Y_1 = X_1 \otimes \mathtt{q}_4$ and $Y_3 = X_3 \otimes \mathtt{q}_4$. Notice that by construction, the right loop stays unified.

Finally, when we visit the last arrow $(\mathtt{N4},\mathtt{N1})$ we find that the blocks are not unifiable. The blocks N4 and N1 are already transitively related and the identity $B_4 \otimes Y_4 = A_1 \otimes Y_1$ does not hold. Any weakening we apply to $A_1 \otimes Y_1$ is promoted via N2 back to N4 and does hence not lead to unification. Consequently, the program is rejected as non-unifiable. This is correct as each iteration through the left loop produces an item in $\mathtt{q}_1$ and consumes an item in $\mathtt{q}_2$. Since the number of iterations can in general not be determined statically, we should not admit the program for execution. $\diamond$

3.6 Discussion

This chapter introduced an assembly language where operand queue names appear in the position of register identifiers. The language generalised the simple compounding approach as instructions can receive several arguments through forwarding, send their result to several operand queues and can forward across basic block boundaries. The language was given a dynamic semantics corresponding to an instruction set architecture of a processor, where only operand queues can be used. A static semantics complemented the dynamic semantics, ruling out characteristic error situations similar to those present in SCALP. Loops were required to be of neutral net-effect, i.e. each iteration could consume only as many items as it produced, and vice versa.

For the patterns of forwarding expressible in ALEF the notation based on linear logics could have been replaced by other formalisms, such as formal monomials over the set of queue names. The linear formalism was motivated by the fact that linear logics has already been successfully applied to type systems for programming languages and concurrency calculi [Laf88], [TWM95], [BS94], [EW90]. Furthermore, more generous forwarding schemes than the one realised by ALEF might need additional operators for combining types, for which some of the hitherto unused connectives of linear logics may prove useful.

We briefly discuss two aspects of our design decision to capture operand queue names explicitly in the syntax of assembly programs.

Firstly, the computational model of ALEF results in a less flexible forwarding mechanism than hardware-based forwarding. For example, a translation of program

```
for (int i=1; i<n; i++){j = j*k;}
j = j-1;
```
(3.7)

into ALEF needs to fix the destination queue for the result of the multiplication statically. The two consuming instructions, however, execute on different functional units, and the exclusive binding of operand queues to functional units forbids a common destination. Consequently, a translation is only possible at the expense of an additional instruction, such as in

$$
\begin{aligned}
&[3]\dots\\
&[4]\mathtt{mul}\ \mathtt{q}_1\ \mathtt{q}_2\ \mathtt{q}_1\\
&[5]\mathtt{if}\dots 3\ 6\\
&[6]\mathtt{id}^{\mathtt{MUL}}\ \mathtt{q}_1\ \mathtt{q}_3\\
&[7]\mathtt{dec}\ \mathtt{q}_3\ \mathtt{q}_4
\end{aligned}
$$

where we assume that $\mathtt{q}_1$ and $\mathtt{q}_2$ are bound to MUL and $\mathtt{q}_3$ to ALU. In contrast, a hardware based forwarding mechanism might generate different forwarding requests in each iteration, matching the effect of the additional instruction.

It is expected that more generous forwarding schemes can be permitted for an extended instruction set and more sophisticated type systems. For example, the destination of the result of the multiplication coincides with the outcome of the branch instruction. The destination should be $\mathtt{q}_1$ whenever the branch condition is fulfilled, and $\mathtt{q}_3$ otherwise. This relationship can be discovered statically, and a more flexible language would contain a corresponding conditional forwarding mechanism. It should thus be possible to match most of the flexibility provided by hardware-based forwarding in a static semantics. Weighting the costs and limitations of static typing against dynamic error recovery then amounts to a design decision where the complexity of the language and the compiler is traded off against the hardware complexity and the costs of error recovery at runtime.

Secondly, exposing the architectural structure at the assembly level means that the legality of programs is affected by architectural modifications. In particular, alterations to the binding of queues to functional units are not shielded from the programmer by the ISA boundary and directly influence the behaviour of programs. Traditionally, system designers would consider this visibility a disadvantage of our computational model as programs need to be type-checked whenever the architecture is modified. However, for reasoning about architectural modifications and their effect on program compilation, explicitness of architectural information is beneficial as formulae and propositions may directly refer to architectural entities. Furthermore, the usage of specific operand queues and the scheduling of operands to communication paths represents an important aspect of program optimisation, which may only be performed statically if the compiler has sufficient knowledge of the executing hardware. The ALEF model supports this optimisation phase by enabling the compiler to refer to architectural entities directly. Finally, future developments of our techniques will show that not all architectural information needs to be revealed, as long as program and processor fulfil some key requirements. Our computational model thus helps clarifying which parts of an architecture should be explicitly exposed to the programmer, with benefits for future instruction set designers and compiler writers.

3.6.1 Summary and outlook

The introduction of ALEF delivered the first technical contribution promised in Section 1.3. We also discussed how ALEF's model of computation arises from, and relates to, core aspects of SCALP. Dynamic semantics (second item in Section 1.3) and static semantics (third item) were given for AQM's with sequential execution, and their compatibility was demonstrated by proofs of soundness and completeness.

In the next chapter, we will consider AQM's with concurrent execution and execution under operand queues of finite length. This will motivate modifications to the calculi for both dynamic and static semantics, but their general shape will remain unchanged.

Chapter 4

Asynchronous queue machines

Program execution in modern processors does not follow the strict sequentiality suggested by the ISA definition. Instead, instructions are partitioned into micro-operations corresponding to stages of a processing pipeline. Super-scalar architectures contain several instruction pipelines and process instructions concurrently. Branch prediction and speculative execution lead to additional complexity.

This chapter presents an alternative dynamic semantics for ALEF which takes the concurrency of multiple pipelines into account and is thus more closely related to the execution in modern processors. Studying the interaction between this distributed dynamic semantics and forwarding, we discover that interleaved execution may experience additional error conditions. This makes program execution *unsafe* as a distributed execution does not honour the ISA specification. We characterise the source of these hazards and study various alternatives for eliminating them. This is supported by our programming language based view as it enables us to characterise programs which are unsafe.

Having obtained safety of distributed execution, we modify the architecture further by limiting the length of operand queues. Additional hazards arise both with respect to the sequential as well as the distributed dynamic semantics as programs might deadlock due to operand queue overflows. We show how a static semantics can derive the minimal queue capacities for executing a program safely and consider the effect of extending the queue lengths. It is shown that absence of queue overflows is only preserved for those programs which were safe in the unrestricted distributed model.

Synopsis The chapter is structured similarly to the previous chapter. Section 4.1 introduces the dynamic semantics for distributed execution of straight-line code. The operational model interleaves reductions of the sequential model on the instruction level. Observing that the static semantics from Chapter 3 guarantees executability but not determinacy, we discuss the relationship between determinism, safety and typability in Section 4.2. This analysis leads to a static semantics which separates the detection of race conditions from the task of eliminating them. An extended type system is given in Section 4.3.1 where judgements contain constraints on the relative execution order of instructions. These constraints are shown to imply determinacy and hence safety of execution: distributed and sequential execution yield the same result. Sections 4.3.2 to 4.3.4 discuss how the constraints may be discharged by exploiting instruction orderings arising from the dynamic semantics or the program structure. Hence, safety of distributed execution may be obtained by (design time) analysis of the dynamic semantics, (compile time) analysis of the program and runtime mechanisms.

Following this, we reintroduce branching instructions in Section 4.4 and adapt the analysis from Section 3.5 correspondingly. Models with operand queues of finite length are treated in Section 4.5. In Section 4.6, we discuss how distrubuted AQM's relate to Tomasulo's algorithm. This demonstrates the expressiveness of our computational model and provides a novel exposition of Tomasulo's algorithm itself. Finally, we discuss technical improvements and comment on some design decisions in Section 4.7.

As in Chapter 3, all operands in this chapter are communicated using the forwarding mechanism and registers are ignored in the discussion.

4.1 Syntax and dynamic semantics

The dynamic semantics for distributed execution models the instruction-level interleaving in super-scalar asynchronous processors (see Figure 4.1). Instructions are loaded from memory and inserted into waiting stations in front of functional units. These are implemented as instruction queues, and progress in different pipelines is decoupled: once an instruction has entered its waiting station, no central control is exercised and execution of head instructions is purely triggered by the availability of operands.

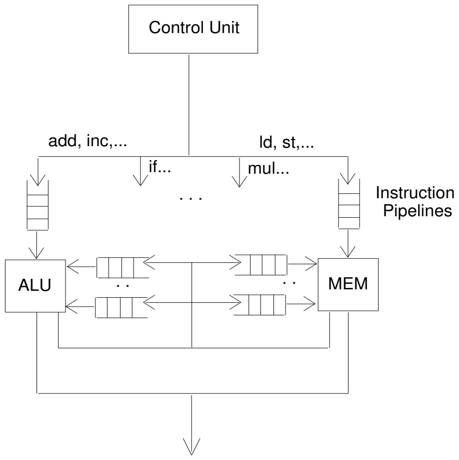


Figure 4.1: Architectural model for distributed dynamic semantics

Syntactically, we model execution by parallel programs with as many components as there are instruction pipelines. For simplicity, the pipelines and the functional units are in one-to-one correspondence, and each pipeline can hence be represented by an instruction sequence of instructions mapped to the same functional unit. For straight-line code, the syntax of distributed programs is given by

$$\pi \in P ::= t_1 \parallel t_2 \parallel t_3$$

where $\parallel$ is non-commutative. For $\pi_1 = t_1 \parallel t_2 \parallel t_3$ and $\pi_2 = s_1 \parallel s_2 \parallel s_3$ we define $\pi_1 \pi_2 = t_1 s_1 \parallel t_2 s_2 \parallel t_3 s_3$.

We denote the parallel program resulting from inserting all instructions of a sequential program t into the correct pipelines by π_t, given inductively by

$$\begin{array}{rcl} \pi_\varepsilon & = & \varepsilon \parallel \varepsilon \parallel \varepsilon \\ \pi_{ins} & = & \begin{cases} ins \parallel \varepsilon \parallel \varepsilon & \text{if} \quad FU(ins) = \mathtt{ALU} \\ \varepsilon \parallel ins \parallel \varepsilon & \text{if} \quad FU(ins) = \mathtt{MUL} \\ \varepsilon \parallel \varepsilon \parallel ins & \text{if} \quad FU(ins) = \mathtt{MEM} \end{cases} \\ \pi_{st} & = & \pi_s \pi_t \end{array}$$

The dynamic semantics is obtained by embedding sequential operation into the distributed model. We define $\mathsf{C} \xrightarrow{t}_{fu} \mathsf{D}$ to hold if $\mathsf{C} \xrightarrow{t} \mathsf{D}$ is derivable in the sequential dynamic semantics from Section 3.2 and all instructions $[n]code$ in t fulfil $FU(code) = fu$.

Formally, the relation $\mathsf{C} \xrightarrow{t}_{fu} \mathsf{D}$ is obtained by the rules

$$\mathrm{FU}_1\ \frac{\mathsf{C} \xrightarrow{[n]code} \mathsf{D}}{\mathsf{C} \xrightarrow{[n]code}_{fu} \mathsf{D}}\ FU(code) = fu \quad \text{and} \quad \mathrm{FU}_2\ \frac{\mathsf{C} \xrightarrow{s}_{fu} \mathsf{E} \quad \mathsf{E} \xrightarrow{t}_{fu} \mathsf{D}}{\mathsf{C} \xrightarrow{st}_{fu} \mathsf{D}}$$

where the premise in rule FU_1 refers to the calculus for binary configurations in Figure 3.5. We define distributed (interleaved) execution $\mathsf{C} \xrightarrow{\pi} \mathsf{D}$ by

$$\mathrm{DIS}_1\ \frac{\mathsf{C} \xrightarrow{t}_{fu} \mathsf{D}}{\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}} \quad \text{and} \quad \mathrm{DIS}_2\ \frac{\mathsf{C} \xrightarrow{\pi_1} \mathsf{E} \quad \mathsf{E} \xrightarrow{\pi_2} \mathsf{D}}{\mathsf{C} \xrightarrow{\pi_1 \pi_2} \mathsf{D}}$$

The relationship between sequential and distributed execution is as follows.

Proposition 7. *1. If* $\mathsf{C} \xrightarrow{t} \mathsf{D}$ *then* $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$.

2. *If* $\mathsf{C} \xrightarrow{\pi} \mathsf{D}$ *then there is a* t *such that* $\pi_t = \pi$ *and* $\mathsf{C} \xrightarrow{t} \mathsf{D}$.

3. *If* $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$, $t = ins_1 \ldots ins_n$ *and* $ins_i = [n_i]code_i$ *then* $(\otimes_{i=1}^{n} A_{code_i}) \otimes shape(\mathsf{D}) = (\otimes_{i=1}^{n} B_{code_i}) \otimes shape(\mathsf{C})$.

Proof. All three parts are shown separately.

1. Induction on the structure of t.

   Case $t = \varepsilon$. As there is no derivation for $\mathsf{C} \xrightarrow{t} \mathsf{D}$, the claim is trivially fulfilled.

   Case $t = [n]\mathtt{add}\ op_1\ op_2\ op_3$. From $\mathsf{C} \xrightarrow{t} \mathsf{D}$ and $FU(\mathtt{add}\ op_1\ op_2\ op_3) = \mathtt{ALU}$ we obtain $\mathsf{C} \xrightarrow{t}_{\mathtt{ALU}} \mathsf{D}$, hence $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ follows by rule DIS_1.

   Other base cases similar.

   Case $t = s\,u$. For $\mathsf{C} \xrightarrow{s\,u} \mathsf{D}$, rule COMP guarantees that there is an E with $\mathsf{C} \xrightarrow{s} \mathsf{E}$ and $\mathsf{E} \xrightarrow{u} \mathsf{D}$. Applying the induction hypothesis to $\mathsf{C} \xrightarrow{s} \mathsf{E}$ and $\mathsf{E} \xrightarrow{u} \mathsf{D}$ yields $\mathsf{C} \xrightarrow{\pi_s} \mathsf{E}$ and $\mathsf{E} \xrightarrow{\pi_u} \mathsf{D}$, so the claim follows by rule DIS_2 and $\pi_s \pi_u = \pi_{s\,u} = \pi_t$.

2. If $\mathsf{C} \xrightarrow{\pi} \mathsf{D}$ holds via $\mathrm{DIS}_1\ \dfrac{\mathsf{C} \xrightarrow{s}_{fu} \mathsf{D}}{\mathsf{C} \xrightarrow{\pi} \mathsf{D}}$ then $\pi = \pi_s$ and the claim is fulfilled for $t = s$.

   If $\mathsf{C} \xrightarrow{\pi} \mathsf{D}$ holds via $\mathrm{DIS}_2\ \dfrac{\mathsf{C} \xrightarrow{\pi_1} \mathsf{E} \quad \mathsf{E} \xrightarrow{\pi_2} \mathsf{D}}{\mathsf{C} \xrightarrow{\pi} \mathsf{D}}$ with $\pi = \pi_1 \pi_2$, then we have $\mathsf{C} \xrightarrow{s} \mathsf{E}$ and $\mathsf{E} \xrightarrow{u} \mathsf{D}$ for some s and u with $\pi_s = \pi_1$ and $\pi_u = \pi_2$ by induction hypothesis. For $t = s\,u$ we consequently obtain $\pi_t = \pi_{s\,u} = \pi_s \pi_u = \pi_1 \pi_2 = \pi$ and $\mathrm{COMP}\dfrac{\mathsf{C} \xrightarrow{s} \mathsf{E} \quad \mathsf{E} \xrightarrow{u} \mathsf{D}}{\mathsf{C} \xrightarrow{t} \mathsf{D}}$.

3. By part (2), there is an s with $\pi_s = \pi_t$ and $\mathsf{C} \xrightarrow{s} \mathsf{D}$, so s consists of the same instructions as t. We may therefore prove the claim by showing that $(\otimes_{i=1}^{n} A_{code_i}) \otimes shape(\mathsf{D}) = (\otimes_{i=1}^{n} B_{code_i}) \otimes shape(\mathsf{C})$ holds for an instruction sequence s which consists of instructions ins_i and fulfils $\mathsf{C} \xrightarrow{s} \mathsf{D}$.

   Case INSTR. For $\mathsf{C} \xrightarrow{ins_1} \mathsf{D}$, Theorem 1 implies $ins_1 : shape(\mathsf{C}) \multimap shape(\mathsf{D})$, and rule AX yields $shape(\mathsf{C}) = A_{code_1} \otimes X$ and $shape(\mathsf{D}) = B_{code_1} \otimes X$ for some X. Combining these two facts yields

$$A_{code_1} \otimes shape(\mathsf{D}) = A_{code_1} \otimes B_{code_1} \otimes X = B_{code_1} \otimes shape(\mathsf{C}).$$

   Case COMP. For $s = r\,u$ there must be an E such that $\mathsf{C} \xrightarrow{r} \mathsf{E}$ and $\mathsf{E} \xrightarrow{u} \mathsf{D}$, so by induction hypothesis we obtain

$$(\otimes_{j=1}^{m} A_{code_{i_j}}) \otimes shape(\mathsf{E}) = (\otimes_{j=1}^{m} B_{code_{i_j}}) \otimes shape(\mathsf{C})$$

   and

$$(\otimes_{j=m+1}^{n} A_{code_{i_j}}) \otimes shape(\mathsf{D}) = (\otimes_{j=m+1}^{n} B_{code_{i_j}}) \otimes shape(\mathsf{E})$$

   where $r = ins_{i_1} \ldots ins_{i_m}$ and $u = ins_{i_{m+1}} \ldots ins_{i_n}$. Hence,

$$\begin{aligned}
(\otimes_{i=1}^{n} A_{code_i}) \otimes shape(\mathsf{D}) &= (\otimes_{j=1}^{m} A_{code_{i_j}}) \otimes (\otimes_{j=m+1}^{n} A_{code_{i_j}}) \\
&\quad \otimes shape(\mathsf{D}) \\
&= (\otimes_{j=1}^{m} A_{code_{i_j}}) \otimes (\otimes_{j=m+1}^{n} B_{code_{i_j}}) \\
&\quad \otimes shape(\mathsf{E}) \\
&= (\otimes_{j=1}^{m} B_{code_{i_j}}) \otimes (\otimes_{j=m+1}^{n} B_{code_{i_j}}) \\
&\quad \otimes shape(\mathsf{C}) \\
&= (\otimes_{i=1}^{n} B_{code_i}) \otimes shape(\mathsf{C}).
\end{aligned}$$

□

In particular, part (3) of Proposition 7 guarantees $shape(\mathsf{D}) = shape(\mathsf{E})$ whenever $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ and $\mathsf{C} \xrightarrow{\pi_t} \mathsf{E}$.

As a further consequence, the type system presented in Section 3.3 is sound for distributed execution in the sense that typability entails executability.

Proposition 8. *If $t : A \multimap B$ and shape$(\mathsf{C}) = A$ then there is a* D *such that* $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ *and shape*$(\mathsf{D}) = B$.

Proof. For $t : A \multimap B$ and *shape*$(\mathsf{C}) = A$ there is by Theorem 1 a (unique) D such that *shape*$(\mathsf{D}) = B$ and $\mathsf{C} \xrightarrow{t} \mathsf{D}$. By Proposition 7 we have $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$. □

It is no longer complete, i.e. $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ does not imply $t : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.

Example. Let $t = [1]\mathtt{add}\ \mathtt{q}_1\ \mathtt{q}_2\ \mathtt{q}_3\ [2]\mathtt{ldc}\ 2\ \mathtt{q}_2$, M arbitrary, $\mathsf{C} = ([\mathtt{q}_1 \mapsto 3], \mathsf{M})$ and $\mathsf{D} = ([\mathtt{q}_3 \mapsto 5], \mathsf{M})$. Then *shape*$(\mathsf{C}) = \mathtt{q}_1$, *shape*$(\mathsf{D}) = \mathtt{q}_3$ and $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$, but t cannot be given the type $\mathtt{q}_1 \multimap \mathtt{q}_3$. Indeed, the principal typing for t is $t :: \mathtt{q}_1 \otimes \mathtt{q}_2 \multimap \mathtt{q}_3 \otimes \mathtt{q}_2$ and a sequential execution of t starting in C deadlocks. ◇

In fact, distributed programs do in general not execute deterministically.

Definition 6. *A distributed program* π *is* deterministic *if* $\mathsf{C} \xrightarrow{\pi} \mathsf{D}$ *and* $\mathsf{C} \xrightarrow{\pi} \mathsf{E}$ *implies* $\mathsf{D} = \mathsf{E}$.

Example. Let $\pi = [1]\mathtt{add}\ \mathtt{q}_1\ \mathtt{q}_2\ \mathtt{q}_3 \parallel \varepsilon \parallel [2]\mathtt{ldc}\ 1\ \mathtt{q}_3$ and $\mathsf{C} = ([\mathtt{q}_1 \mapsto 2,\ \mathtt{q}_2 \mapsto 3], \mathsf{M})$. Then there are $\mathsf{D} \neq \mathsf{E}$ with $\mathsf{C} \xrightarrow{\pi} \mathsf{D}$ and $\mathsf{C} \xrightarrow{\pi} \mathsf{E}$, as shown in Figure 4.2. ◇

Non-determinism arises from race hazards between instructions which execute on different functional units but forward to the same operand queue.

The type system's adequacy for parallel execution is thus limited, in theoretical (incompleteness) and practical (non-determinism) respects. Even a well-typed program may yield different results as the configuration D in Proposition 8 need not be unique. The shape of all final configurations are identical by Proposition 7, but the values inside the operand queues may differ.

Rather than studying distributed programs and their non-determinism directly, we are more interested in characterising instruction sequences for which distributed execution coincides with sequential execution, and with typability. The following section therefore discusses the relationship between distributed execution, sequential execution, race hazards and typing for individual instruction sequences.

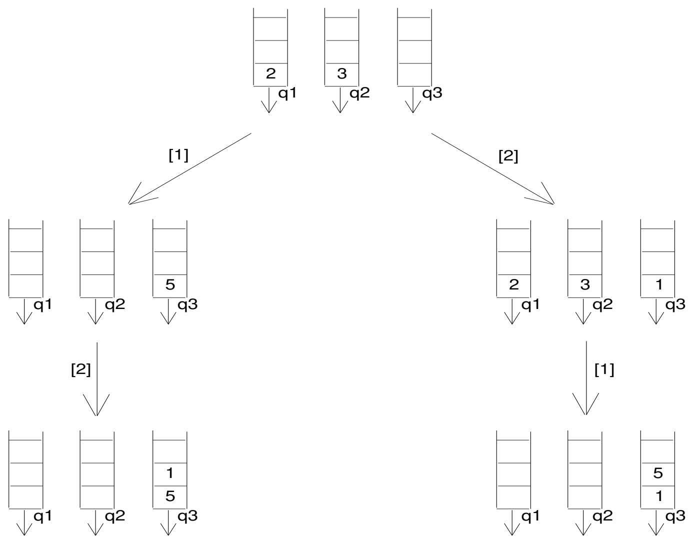


Figure 4.2: Execution of a non-deterministic program

4.2 Determinism, safety and completeness

We first introduce some terminology.

Definition 7. *An instruction sequence t is called*

- deterministic *if* $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ *and* $\mathsf{C} \xrightarrow{\pi_t} \mathsf{E}$ *implies* $\mathsf{D} = \mathsf{E}$
- safe *if* $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ *implies* $\mathsf{C} \xrightarrow{t} \mathsf{D}$
- complete *if* $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ *implies* $t : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.

For practical purposes, determinacy of t is important for studying the processor's behaviour as modelled in the distributed dynamic semantics, without referring to the ISA definition. For establishing correctness of processor behaviour in relation to the ISA specification, safety is most appropriate as for safe programs, sequential and distributed dynamic semantics coincide. Proposition 7 showed that the processor *may* deliver the same result as the ISA semantics promises, and safety guarantees that *any* result delivered by the processor is indeed the one promised. Completeness finally allows us to reason about distributed execution in the compiler. It guarantees that whatever interleaving the processor chooses (possibly influenced by the outcome of race conditions),

the shape of final configurations is determined by the shape of the initial configuration, based on Proposition 2.
The three notions relate to each other as follows.

Proposition 9. *For an instruction sequence t, the following (non-)implications hold.*

- *t safe $\Leftrightarrow$ t complete and t deterministic*
- *t deterministic $\not\Rightarrow$ t safe*
- *t deterministic $\not\Rightarrow$ t complete*
- *t complete $\not\Rightarrow$ t safe*
- *t complete $\not\Rightarrow$ t deterministic*

Before proving Proposition 9, we introduce relativised variants of the above notions.

Definition 8. *An instruction sequence t is called*

- deterministic for A *if for all* C *with* $shape(\mathsf{C}) = A$, $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ *and* $\mathsf{C} \xrightarrow{\pi_t} \mathsf{E}$ *implies* $\mathsf{D} = \mathsf{E}$
- safe for A *if for all* C *with* $shape(\mathsf{C}) = A$, $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ *implies* $\mathsf{C} \xrightarrow{t} \mathsf{D}$
- complete for A *if for all* C *with* $shape(\mathsf{C}) = A$, $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ *implies* $t : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.

By definition, the unrelativised notions are obtained from the relativised notions by quantifying over all products A.

Lemma 2. *The following (non-)implications hold for instruction sequence t and product A.*

- *t safe for A $\Leftrightarrow$ t complete for A and t deterministic for A*
- *t deterministic for A $\not\Rightarrow$ t safe for A*
- *t deterministic for A $\not\Rightarrow$ t complete for A*

- *t complete for $A \not\Rightarrow t$ safe for A*
- *t complete for $A \not\Rightarrow t$ deterministic for A*

Proof. The first claim consists of two parts.

$\Rightarrow$. We have to show that for $shape(\mathsf{C}) = A$, $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ and $\mathsf{C} \xrightarrow{\pi_t} \mathsf{E}$ implies $\mathsf{D} = \mathsf{E}$ and $t : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.

By the definition of safety with respect to A, $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ implies $\mathsf{C} \xrightarrow{t} \mathsf{D}$ and $\mathsf{C} \xrightarrow{\pi_t} \mathsf{E}$ implies $\mathsf{C} \xrightarrow{t} \mathsf{E}$. By Proposition 1, $\mathsf{D} = \mathsf{E}$ follows. Also, $\mathsf{C} \xrightarrow{t} \mathsf{D}$ implies $t : shape(\mathsf{C}) \multimap shape(\mathsf{D})$ by Theorem 1.

$\Leftarrow$. We have to show that $shape(\mathsf{C}) = A$ and $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ implies $\mathsf{C} \xrightarrow{t} \mathsf{D}$.

By the definition of completeness with respect to A, $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ yields $t : shape(\mathsf{C}) \multimap shape(\mathsf{D})$, so by Theorem 1 there is an E with $\mathsf{C} \xrightarrow{t} \mathsf{E}$, and $shape(\mathsf{E}) = shape(\mathsf{D})$ holds. By applying Proposition 7(part (1)), we obtain $\mathsf{C} \xrightarrow{\pi_t} \mathsf{E}$, so determinism with respect to A implies $\mathsf{D} = \mathsf{E}$, hence $\mathsf{C} \xrightarrow{t} \mathsf{D}$.

For the non-implications, consider the programs s and t

$$\begin{aligned} s &= [1]\mathtt{dec}\ \mathsf{q}_1\ \mathsf{q}_2\ [2]\mathtt{ldc}\ 5\ \mathsf{q}_1 \\ t &= [1]\mathtt{ldc}\ 3\ \mathsf{q}_1\ [2]\mathtt{ldc}\ 5\ \mathsf{q}_2\ [3]\mathtt{dec}\ \mathsf{q}_1\ \mathsf{q}_2 \end{aligned}$$

whose execution traces for initial configurations of shape $\mathbf{1}$ are shown in Figure 4.3. Program s is deterministic for $\mathbf{1}$, since for any configuration C with empty queue q_1, the distributed program π_s can only follow the interleaving $[2][1]$, so $\mathsf{C} \xrightarrow{\pi_s} \mathsf{D}$ and $\mathsf{C} \xrightarrow{\pi_s} \mathsf{E}$ implies $\mathsf{D} = \mathsf{E}$. However, s is neither safe for $\mathbf{1}$ nor complete for $\mathbf{1}$. It is not safe because for any C with empty q_1 there is no D with $\mathsf{C} \xrightarrow{s} \mathsf{D}$. It is not complete because s cannot be given type $\mathbf{1} \multimap B$ for any B. Hence, program s proves the second and the third claim.
Program t is complete for $\mathbf{1}$, because $t : \mathbf{1} \multimap \mathsf{q}_2 \otimes \mathsf{q}_2$ holds, and for any C and D with $shape(\mathsf{C}) = \mathbf{1}$ and $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ we have $shape(\mathsf{D}) = \mathsf{q}_2 \otimes \mathsf{q}_2$. On the other hand, t is neither safe for $\mathbf{1}$ nor deterministic for $\mathbf{1}$. It is not safe because for $\mathsf{C} = ([\,], \mathsf{M})$ and $\mathsf{D} = ([\mathsf{q}_2 \mapsto 2\,5])$ we have $shape(\mathsf{C}) = \mathbf{1}$ and $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ via the interleaving $[1][3][2]$, but $\mathsf{C} \not\xrightarrow{t} \mathsf{D}$. It

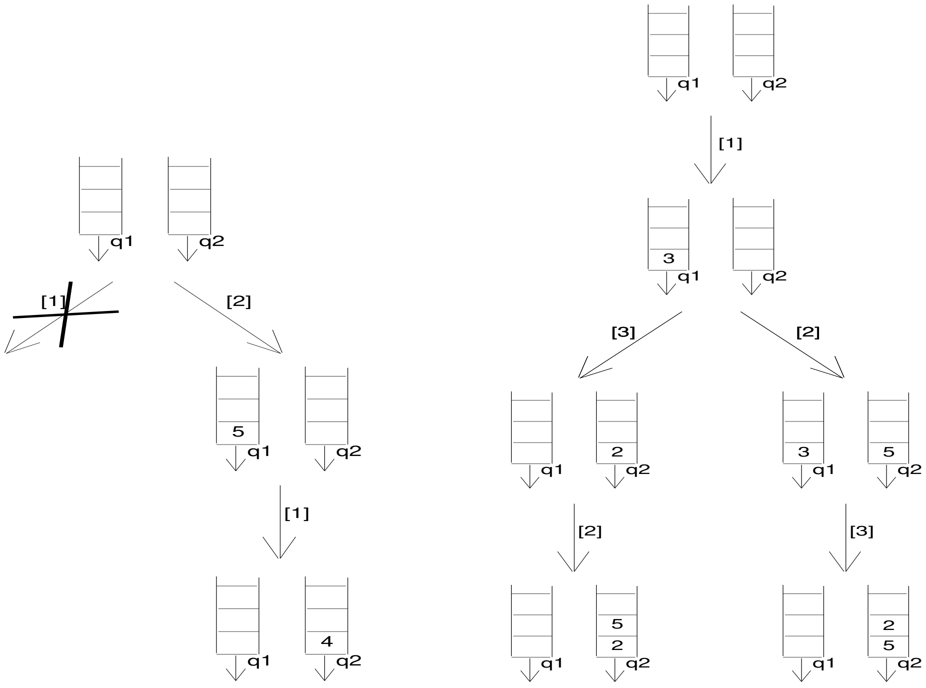


Figure 4.3: Execution traces for π_s (left) and π_t (right).

is not deterministic for **1** because for the same C, configuration $\mathsf{E} = (\mathsf{q}_2 \mapsto 5\,2)$ fulfils $\mathsf{C} \xrightarrow{\pi_t} \mathsf{E}$ via the interleaving $[1][2][3]$, but $\mathsf{D} \neq \mathsf{E}$ holds. Hence, program t proves the forth and fifth claim. $\square$

The results in Proposition 9 are now obtained as corollaries of the corresponding statements in Lemma 2. Safety of t is equivalent to completeness and determinacy because for each product A this equivalence holds relative to A. Likewise, the examples in the proof of Lemma 2 also prove the non-implications in Proposition 9. In each case, we can exhibit a program t and (an A and) a configuration C (of shape A) demonstrating that the implication does not hold.

Proposition 10. *If $t : A \multimap B$ then t complete for A.*

Proof. For $t : A \multimap B$, $shape(\mathsf{C}) = A$ and $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ we have to show $t : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.

Since $t : A \multimap B$ and $shape(\mathsf{C}) = A$ hold, Theorem 1 implies $\mathsf{C} \xrightarrow{t} \mathsf{E}$ for some E with $shape(\mathsf{E}) = B$. By applying Proposition 7(part (1)) we obtain $\mathsf{C} \xrightarrow{\pi_t} \mathsf{E}$, so Proposition

7(part (3)) yields $shape(\mathsf{D}) = shape(\mathsf{E})$, hence $shape(\mathsf{D}) = B$ as claimed. □

In the next section, we will extend the type system by constraints which guarantee (relativised) determinism. Fulfilling the constraints will hence result in (relativised) safety, based on the following consequences of Propositions 9 and 10.

Corollary 1. *Let t be an instruction sequence and $t : A \multimap B$.*

- *t is deterministic for $A \Leftrightarrow t$ is safe for A.*
- *If $A = \mathbf{1}$ then t is complete.*
- *If $A = \mathbf{1}$ then t is deterministic iff t is safe.*

Proof.

- $t : A \multimap B$ implies that t is complete for A by Proposition 10, so the first claim of Lemma 2 implies the equivalence.
- By applying the weakening rule WK we may type $t : C \multimap C \otimes B$ for any C, and for each choice Proposition 10 implies that t is complete for C. Hence t is complete for all C and thus complete.
- Combine the previous item and Proposition 9.

□

None of the relativised notions are monotone with respect to A.

Example. Consider the programs

$$t = [1]\mathtt{add}\ \mathsf{q}_1\ \mathsf{q}_2\ \mathsf{q}_1\ [2]\mathtt{ldc}\ 3\ \mathsf{q}_1 \quad \text{and} \quad s = [1]\mathtt{add}\ \mathsf{q}_1\ \mathsf{q}_2\ \mathsf{q}_3\ [2]\mathtt{ldc}\ 3\ \mathsf{q}_1$$

with the respective principal typings $t : \mathsf{q}_1 \otimes \mathsf{q}_2 \multimap \mathsf{q}_1 \otimes \mathsf{q}_1$ and $s : \mathsf{q}_1 \otimes \mathsf{q}_2 \multimap \mathsf{q}_1 \otimes \mathsf{q}_3$. Then t is complete for $\mathbf{1}$ and $\mathsf{q}_2 \otimes \mathsf{q}_1$ but not for q_2, and it is deterministic for q_2 but not for $\mathsf{q}_2 \otimes \mathsf{q}_1$. Program s is safe for $\mathbf{1}$ and $\mathsf{q}_2 \otimes \mathsf{q}_1$ but not for q_2. ◇

4.3 Dealing with non-determinism

Race hazards leading to non-determinism can be detected at compile-time by analysing the forwarding behaviour, and several choices exist for handling them.

In some applications, non-determinism might be desirable. For example, the latency in globally asynchronous systems depends on environmental conditions such as the temperature. In applications for security, the temperature might hence be (a part of) the secret key necessary for decoding a message correctly, and attempts to decode a message under incorrect temperature settings experience different interleavings and thus fail.

For applications where non-determinism cannot be tolerated, race hazards can be eliminated either in hardware or in software. A simple hardware-based approach would aim at modifying the processor implementation to block the progress in one pipeline until other pipelines have reached a certain state. This corresponds to changing the dynamic semantics by imposing side-conditions on the reduction rules. If implemented naively, such a solution depends on a notion of global state, is thus likely to result in central control and therefore undesirable in an asynchronous setting as the operand-driven mode of operation is compromised.

The programming language view allows us to understand race-hazards and their elimination as two separate issues. First, there are requirements on the relative order of some instructions which *need to be fulfilled* for determinism to hold. These constraints are specific to each input program. Second, there are mechanisms for proving that requirements *are fulfilled*. These discharging techniques exploit existent ordering relations between instructions which originate from the dynamic semantics or are specific to the input program or the initial configuration. They may be of arbitrary complexity as long as they can be implemented as static or dynamic program analysis.

This section therefore presents an extended type system which derives constraints necessary for determinism. Following that, we examine two techniques for discharging constraints, based on the dynamic semantics and on program analysis. The first technique fulfils constraints irrespectively of the shape of initial configurations, and can therefore be used for obtaining determinism. The second technique depends on the shape of an initial configuration and can thus yield at most relativised determinism.

As the extended type system contains the rules for deriving $t : A \multimap B$, the respective determinacy property immediately yields the corresponding safety property, and thus guarantees the predictability of program execution according to the distributed dynamic semantics.

4.3.1 A static semantics for determinism

The type system from Section 3.3 is extended such that it derives sequentiality constraints which guarantee deterministic execution: whenever two instructions forward their results to the same operand queue their execution order must agree in all interleavings. These constraints will later be discharged using instruction dependencies inherent in the dynamic semantics or the structure of the program.

For the dynamic semantics $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$, the only source of non-determinism in a well-typed program is the presence of a race condition between instructions which execute independently but forward to the same operand queue. In order to reason about these output conflicts, we extend sequents $t : A \multimap B$ by annotations Γ and $<$. This yields sequents of the form $(\Gamma, <) \vdash t : A \multimap B$. The set Γ contains entries of the form $op_1 : (l_1, k_1), \ldots, op_n : (l_n, k_n)$ such that the op_i are distinct and the $l_i, k_i \in \mathbf{N}$ are labels of instructions in t. An entry $op : (l, k)$ in Γ represents the fact that instructions l and k forward their result to op, and l is the first such instruction in t and k the last one. The second component, $<$, is a partial order over $\mathbf{N}$ which relates instructions in t by $[n] < [m]$ whenever they forward to the same destination such that $[n]$ precedes $[m]$ in t. Soundness of our calculus consists of showing that t is deterministic in the sense of Section 4.2 provided that all executions of t contain $[n]$ before $[m]$. By embedding the calculus $t : A \multimap B$, we restrict our attention to programs which are sequentially executable and aim for safety rather than pure determinism.

The typing rules for deriving $(\Gamma, <) \vdash t : A \multimap B$ are shown in Figure 4.4 and are defined in terms of the typing relation $t : A \multimap B$ from Section 3.3. They use the following definitions.

Definition 9. *For $B_{code} = op_1^{k_1} \otimes \ldots \otimes op_m^{k_m}$ we set*

$$\Gamma_{[n]code} = \{op_i : (n, n) \mid k_i > 0\}.$$

$$\text{AX}^{<}\,\frac{ins : A \multimap B}{(\Gamma_{ins}, \emptyset) \vdash ins : A \multimap B} \qquad\qquad \text{CUT}^{<}\,\frac{\begin{array}{c}(\Gamma, \subset) \vdash s : A \multimap C \\ (\Delta, \prec) \vdash t : C \multimap B\end{array}}{(\Gamma \# \Delta, <) \vdash st : A \multimap B}$$

Figure 4.4: Type system for deterministic execution

For contexts $(\Gamma, \subset)$ *and* $(\Delta, \prec)$*, the context* $(\Gamma\#\Delta, <)$ *is given by*

$$\begin{aligned} \Gamma\#\Delta \;=\;& \{op : (l,h) \,|\, op : (l,k) \in \Gamma \text{ and } op : (g,h) \in \Delta\} \\ & \cup \{op : (l,k) \,|\, op : (l,k) \in \Gamma \text{ and } op \notin dom\,\Delta\} \\ & \cup \{op : (g,h) \,|\, op : (g,h) \in \Delta \text{ and } op \notin dom\,\Gamma\} \end{aligned}$$

and

$$< \;=\; (\subset \,\cup\, \prec \,\cup\, \{(k,g) \,|\, op : (l,k) \in \Gamma \text{ and } op : (g,h) \in \Delta\})^{+}.$$

The type system is faithful to the calculus from Section 3.3.

Lemma 3. *1.* $t : A \multimap B$ *holds exactly if there are* Γ *and* $<$ *such that* $(\Gamma, <) \vdash t : A \multimap B$ *holds.*

2. *If* $(\Gamma, <) \vdash t : A \multimap B$ *and* $(\Delta, \prec) \vdash t : C \multimap D$ *then* $\Gamma = \Delta$ *and* $< = \prec$.

3. *If* $(\Gamma, <) \vdash t : A \multimap B$*, then*

   - *all labels* n *occuring in* Γ *or* $<$ *are labels of instructions in* t.
   - *if* $n < m$ *then* t *can be written as* $s\,[n]code_n\,r\,[m]code_m\,u$ *and* B_{code_n} *and* B_{code_m} *have a common factor op.*
   - $op : (n, \_) \in \Gamma$ *iff* $[n]$ *is the first instruction* $[m]code$ *in* t *for which op is a factor of* B_{code}.
   - $op : (\_, n) \in \Gamma$ *iff* $[n]$ *is the last instruction* $[m]code$ *in* t *for which op is a factor of* B_{code}.
   - $op : (n,n) \in \Gamma$ *iff* $[n]$ *is the only instruction* $[m]code$ *in* t *for which op is a factor of* B_{code}.

4. *If* $(\Gamma,<)\vdash t : A \multimap B$*, t has the form* $s\,[n]code_1\,r\,[m]code_2\,u$ *and* op *is a factor of both* B_{code_1} *and* B_{code_2}*, then* $n<m$ *holds.*

Proof. We prove all four parts simultaneously by induction on the structure of t.

Case $t=\varepsilon$. Neither $t : A \multimap B$ nor $(\Gamma,<)\vdash t : A \multimap B$ are derivable, so all claims are trivially fulfilled.

Case $t = ins$.

1. If $ins : A \multimap B$ then $(\Gamma,<)\vdash t : A \multimap B$ holds for $\Gamma=\Gamma_{ins}$ and $<=\emptyset$ by rule $\mathrm{AX}^<$.

   If $(\Gamma,<)\vdash ins : A \multimap B$ is derivable then $ins : A \multimap B$ is derivable because it occurs as the premise of $\mathrm{AX}^<$.

2. For $(\Gamma,<)\vdash ins : A \multimap B$ and $(\Delta,\prec)\vdash ins : C \multimap D$ we have $\Gamma=\Gamma_{ins}=\Delta$ and $<=\emptyset=\prec$ by rule $\mathrm{AX}^<$.

3. For $(\Gamma,<)\vdash ins : A \multimap B$ we have $\Gamma=\Gamma_{ins}$ and $<=\emptyset$, so the first two claims hold trivially, and $op : (m,l)$ occurs in Γ exactly if $m=l$ is the label of ins and for $B_{code}=op_1^{k_1}\otimes\ldots\otimes op_n^{k_n}$ there is an i with $op=op_i$ and $k_i>0$, i.e. op is a factor of B_{code}.

4. ins does not have the form $s\,[n]code_1\,r\,[m]code_2\,u$, so the claim is trivially fulfilled.

Case $t=s\,r$.

1. The last step in the derivation for $t : A \multimap B$ must be

$$\mathrm{CUT}\,\frac{s : A \multimap C \quad r : C \multimap B}{t : A \multimap B}$$

   for some C. By induction hypothesis, there are hence Λ, $\subset$, Δ and $\prec$ such that $(\Lambda,\subset)\vdash s : A \multimap C$ and $(\Delta,\prec)\vdash r : C \multimap B$ hold. For $\Gamma=\Lambda\#\Delta$ and $<$ defined by

$$(\subset\ \cup\ \prec\ \cup\{(k,g)\,|\,op:(l,k)\in\Lambda \text{ and } op:(g,h)\in\Delta\})^+$$

   we obtain $(\Gamma,<)\vdash t : A \multimap B$ by rule $\mathrm{CUT}^<$.

   Conversely, the last step in a derivation for $(\Gamma,<)\vdash t : A \multimap B$ must be of the form

$$\mathrm{CUT}^<\,\frac{(\Lambda,\subset)\vdash s : A \multimap C \quad (\Delta,\prec)\vdash r : C \multimap B}{(\Gamma,<)\vdash t : A \multimap B}$$

where $\Gamma = \Lambda\#\Delta$ and $<$ equals

$$(\subset \,\cup\, \prec \,\cup\, \{(k,g)\,|\,op:(l,k)\in\Lambda \text{ and } op:(g,h)\in\Delta\})^{+}.$$

By induction hypothesis, $s : A \multimap C$ and $r : C \multimap B$ hold, hence $s\,r : A \multimap B$ follows by rule CUT.

2. The last rule in the derivations for $(\Gamma,<)\vdash t : A \multimap B$ and $(\Delta,\prec)\vdash t : C \multimap D$ is CUT$^{<}$, so there are $\Lambda_i, \Sigma_i, \subset_i$ and $\prec_i$ for $i\in\{1,2\}$ such that

$$\mathrm{CUT}^{<}\frac{\begin{array}{c}(\Lambda_1,\subset_1)\vdash s : A \multimap E\\ (\Sigma_1,\prec_1)\vdash r : E \multimap B\end{array}}{(\Gamma,<)\vdash s\,r : A \multimap B}\quad \text{and}\,\mathrm{CUT}^{<}\frac{\begin{array}{c}(\Lambda_2,\subset_2)\vdash s : C \multimap F\\ (\Sigma_2,\prec_2)\vdash r : F \multimap D\end{array}}{(\Delta,\prec)\vdash s\,r : C \multimap D}$$

In particular, $\Gamma = \Lambda_1\#\Sigma_1$, $\Delta = \Lambda_2\#\Sigma_2$ and

$$\begin{array}{rcl} < & = & (\subset_1 \,\cup\, \prec_1 \,\cup\, \{(k,g)\,|\,op:(l,k)\in\Lambda_1 \text{ and } op:(g,h)\in\Sigma_1\})^{+}\\ \prec & = & (\subset_2 \,\cup\, \prec_2 \,\cup\, \{(k,g)\,|\,op:(l,k)\in\Lambda_2 \text{ and } op:(g,h)\in\Sigma_2\})^{+}.\end{array}$$

By induction hypothesis, $\Lambda_1 = \Lambda_2$, $\Sigma_1 = \Sigma_2$, $\subset_1 = \subset_2$ and $\prec_1 = \prec_2$ hold, hence

$$\Gamma = \Lambda_1\#\Sigma_1 = \Lambda_2\#\Sigma_2 = \Delta$$

and

$$\begin{array}{rcl} < & = & (\subset_1 \,\cup\, \prec_1 \,\cup\{(k,g)\,|\,op:(l,k)\in\Lambda_1 \text{ and } op:(g,h)\in\Sigma_1\})^{+}\\ & = & (\subset_2 \,\cup\, \prec_2 \,\cup\{(k,g)\,|\,op:(l,k)\in\Lambda_2 \text{ and } op:(g,h)\in\Sigma_2\})^{+}\\ & = & \prec\,. \end{array}$$

3. The last step in the derivation for $(\Gamma,<)\vdash s\,r : A \multimap B$ must have been

$$\mathrm{CUT}^{<}\frac{\begin{array}{c}(\Lambda,\subset)\vdash s : A \multimap C\\ (\Delta,\prec)\vdash r : C \multimap B\end{array}}{(\Gamma,<)\vdash s\,r : A \multimap B}$$

for some Λ, Δ, $\subset$, $\prec$ and C. By induction hypothesis, all labels occuring in Λ or $\subset$ are labels of instructions in s, and all labels occuring in Δ or $\prec$ are labels of instructions in r, so all labels occurring in Γ or $<$ are labels of instructions in $t = s\,r$ by the definition of # and $<$. This proves the first item.

For the third item, observe that for $op : (n, \_) \in \Gamma$, either $op : (n, \_) \in \Lambda$ or $op \notin dom\, \Lambda$ and $op : (n, \_) \in \Delta$ hold. In the first case, $[n]$ is the first instruction for which op occurs in B_{code} in s by induction hypothesis, hence n is the first such instruction in $s\,r = t$. In the second case, there is no instruction in s for which op occurs in B_{code} by induction hypothesis (reverse direction), but $[n]$ is the first instruction for which op occurs in B_{code} in r, also by induction hypothesis. Hence $[n]$ the first such instruction in $s\,r = t$.

Conversely, if n is the first instruction in s for which op occurs in B_{code} in $s\,r$ then either n is the first such instruction or s does not have any instruction for which op occurs in B_{code} and $[n]$ is the first such instruction in r. In the first case, the induction hypothesis implies $op : (n, \_) \in \Lambda$, so $op : (n, \_) \in \Gamma$. In the second case, the induction hypothesis implies $op \notin dom\, \Lambda$, and $op : (n, \_) \in \Delta$ holds, so $op : (n, \_) \in \Gamma$.

The forth and fifth item are proven similarly to the third one.

Finally, for proving the second item, note that by the definition of $<$, any n and m with $n < m$ fulfil either $n \subset m$ or $n \prec m$ or there are l, h and op such that $op : (l, n) \in \Lambda$ and $op : (m, h) \in \Delta$. In the first case, the induction hypothesis guarantees that s can be written as $s_1\,[n]\,s_2\,[m]\,s_3$ and there is a common factor of B_{code_n} and B_{code_m}, so $t = s\,r$ can be written as $s_1\,[n]\,s_2\,[m]\,s_3\,r$ and there is a common factor of B_{code_n} and B_{code_m}. Similarly, the second case results in t having the form $s\,r_1\,[n]\,r_2\,[m]\,r_3$ with a common factor of B_{code_n} and B_{code_m}. In the third case, the first item guarantees that s is of the form $s_1\,[n]\,s_2$ and that r is of the form $r_1\,[m]\,r_2$, so t is of the form $t_1\,[n]\,t_2\,[m]\,t_3$ for $t_1 = s_1$, $t_2 = s_2\,r_1$ and $t_3 = r_2$. Items four and five guarantee that op is a factor of both B_{code_n} and B_{code_m}.

4. For $(\Gamma, <) \vdash s\,t : A \multimap B$ and $s\,r$ is of the form

$$u\,[n] code_1\, v\,[m] code_2\, w$$

such that op is a factor of B_{code_1} and B_{code_2}, the rule $\text{CUT}^{<}$ implies that there are Λ, $\subset$, Δ and $\prec$ such that $(\Lambda,\subset) \vdash s : A \multimap C$ and $(\Delta,\prec) \vdash r : C \multimap B$ hold, where $\Gamma = \Lambda \# \Delta$ and $<$ is equal to

$$(\subset \,\cup\, \prec \,\cup\, \{(k,g) \mid op : (l,k) \in \Lambda \text{ and } op : (g,h) \in \Delta\})^+.$$

There are three cases.

(a) Instructions $[n]$ and $[m]$ both occur in s. Then s can be written in the form $s_1\,[n]code_1\,s_2\,[m]code_2\,s_3$, and the induction hypothesis yields $[n] \subset [m]$, so from the definition of $<$ we obtain $[n] < [m]$.

(b) Instructions $[n]$ and $[m]$ both occur in r. Similarly to the previous case, we obtain $[n] \prec [m]$ by induction hypothesis and thus $[n] < [m]$ from the definition of $<$.

(c) Instruction $[n]$ occurs in s and instruction $[m]$ occurs in r. Let $[l]$ be the last instruction in s such that op is a factor of B_{code} and $[k]$ be the first such instruction in r. By induction hypothesis, $l \neq n$ implies $n \subset l$ and $m \neq k$ implies $k \prec m$. It therefore suffices to show $l < k$, which holds by the definition of $<$ as item (3) yields $op(\_,l) \in \Lambda$ and $op(k,\_) \in \Delta$.

□

For $(\Gamma,<) \vdash t : A \multimap B$, instructions in t are thus related by $<$ whenever they forward to the same operand queue. The order of such instructions for each queue agrees with the order in which they appear in t.

Definition 10. *Let t be an instruction sequence and $\sqsubset$ a partial order over* $\mathbf{N}$. *Then t* respects $\sqsubset$ *if t can be written as $t = s\,[n]code_n\,r\,[m]code_m\,u$ whenever $[n] \sqsubset [m]$ holds.*

For proving soundness, we show that $(\Gamma,<) \vdash t : A \multimap B$ yields determinism relative to A, provided that all constraints $<$ are met: for $shape(\mathsf{C}) = A$, we suppose that all distributed executions of t respect $<$, i.e. that for $\mathsf{C} \xrightarrow{\pi_t} \mathsf{E}$, all s which fulfil $\mathsf{C} \xrightarrow{s} \mathsf{E}$ and $\pi_s = \pi_t$ respect $<$. We show that for the unique D with $\mathsf{C} \xrightarrow{t} \mathsf{D}$ (existent by Theorem 1) the equality $\mathsf{D} = \mathsf{E}$ holds.

Definition 11. *Program s is an* interleaving for t and A *if $\pi_s = \pi_t$ holds and for any C with $shape(\mathsf{C}) = A$ there is a D with $\mathsf{C} \xrightarrow{s} \mathsf{D}$.*

For a partial order $\sqsubset$ *over* $\mathbf{N}$*, program* t *is called* sequential for $\sqsubset$ and A *if all interleavings* s *for* t *and* A *respect* $\sqsubset$*.*

Lemma 4. *Let* $(\Gamma,<) \vdash t : A \multimap B$*,* $shape(\mathsf{C}) = A$*, and* $u = ins\, u'$ *an interleaving of* t *and* A*. Let* t *be sequential for* $\sqsubset$ *and* A*, let* $<$ *be contained in* $\sqsubset$*, and let* $\mathsf{C} \xrightarrow{ins} \mathsf{D}_1$ *and* $t = t_1\, ins\, t_2$*. Then there are* Δ *and* $\prec$ *such that for* $C = shape(\mathsf{D}_1)$ *and* $s = t_1\, t_2$ *the following derivation exists*

$$\frac{(\Gamma_{ins},\emptyset) \vdash ins : A \multimap C \quad (\Delta,\prec) \vdash s : C \multimap B}{(\Gamma_{ins}\#\Delta,<) \vdash ins\, s : A \multimap B}$$

In particular, $\prec$ *is contained in* $<$*.*

Proof. Since u is an interleaving for t and A, t is sequential for $\sqsubset$ and A and $shape(\mathsf{C}) = A$ holds, there is a (unique) D such that $\mathsf{C} \xrightarrow{ins} \mathsf{D}_1 \xrightarrow{u'} \mathsf{D}$. Theorem 1 implies $ins : A \multimap C$, hence $(\Gamma_{ins},\emptyset) \vdash ins : A \multimap C$ by the rule $\mathrm{AX}^{<}$.
Since $(\Gamma,<) \vdash t : A \multimap B$ holds, Lemma 3 implies $t : A \multimap B$, so there is by Theorem 1 a (unique) E with $\mathsf{C} \xrightarrow{t} \mathsf{E}$, and $shape(\mathsf{E}) = B$ holds. The typing $t : A \multimap B$ also implies that there are D and E such that

$$\frac{\dfrac{t_1 : A \multimap D \quad ins : D \multimap E}{t_1\, ins : A \multimap E} \qquad t_2 : E \multimap B}{t : A \multimap B}$$

is a valid derivation. Since $ins : D \multimap E$ holds, we know that there is an X with

$$D = A_{code} \otimes X \text{ and } E = B_{code} \otimes X$$

where $ins = [n]code$.
On the other hand, $ins : A \multimap C$ implies

$$A = A_{code} \otimes Y \text{ and } C = B_{code} \otimes Y$$

for some Y. Consequently,

$$t_1 : A_{code} \otimes Y \multimap A_{code} \otimes X \text{ and } t_2 : B_{code} \otimes X \multimap B.$$

Since ins is the head instruction of its pipeline, instructions in t_1 do not access elements in A_{code}, so A_{code} is used as a weakening in the typing of t_1, i.e. we have $t_1 : Y \multimap X$. By applying the alternative weakening B_{code}, we obtain $t_1 : Y \otimes B_{code} \multimap X \otimes B_{code}$, so

$$s : Y \otimes B_{code} \multimap B.$$

Since $Y \otimes B_{code} = C$ holds, we have $s : C \multimap B$, and hence

$$(\Delta, \prec) \vdash s : C \multimap B \text{ for some } \Delta \text{ and } \prec$$

by Lemma 3. Consequently,

$$(\Gamma_{ins}\#\Delta, \subset) \vdash ins\, s : A \multimap B$$

for $\subset = (\prec \cup \{([n],[m]) \mid op : ([m], \_) \in \Delta,\ op \text{ factor of } B_{code}\})^+$. Instruction sequences u and t are both interleavings for t and $A - u$ by assumption and t due to $\pi_t = \pi_t$ and $t : A \multimap B$. Since t is sequential for $\sqsubset$ and A, u and t both respect $\sqsubset$.
Therefore, $[n]$ must be minimal in $\sqsubset$. For suppose there is an $[m]$ with $[m] \sqsubset [n]$, then u must have the form $u_1\,[m]\,u_2\,[n]\,u_3$, in contradiction to $u = ins\, u'$.
As $[m] < [m]$ implies $[n] \sqsubset [m]$, instruction $[n]$ is also minimal for $<$, and $[n]$ does not occur in Δ or $\prec$ since it does not occur in s (see the previous Lemma).
Let $[l] \prec [k]$. By Lemma 3, $s = t_1 t_2$ has the form $v_1\,[l]code_l\,v_2\,[k]code_k\,v_3$ and there is a common factor op of B_{code_l} and B_{code_k}. Consequently, $[l]$ occurs in front of $[k]$ also in $t = t_1\, ins\, t_2$, so $[l] < [k]$ holds by the last item of Lemma 3.

$\square$

Proposition 11. *Let $(\Gamma, <) \vdash t : A \multimap B$ and $<$ be contained in the partial order $\sqsubset$. If t is sequential for $\sqsubset$ and A then t is deterministic for A.*

Proof. Since $(\Gamma, <) \vdash t : A \multimap B$ implies $t : A \multimap B$, it suffices by Corollary 1 to prove that for all C with $shape(\mathsf{C}) = A$, $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ implies $\mathsf{C} \xrightarrow{t} \mathsf{D}$. So let $shape(\mathsf{C}) = A$ and $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$. By Proposition 7(part (2)) there is a u with $\pi_u = \pi_t$ and $\mathsf{C} \xrightarrow{u} \mathsf{D}$, so u is an interleaving for t and A. As t is sequential for $\sqsubset$ and A, u respects $\sqsubset$, and thus $<$. We write $t = t_1 \ldots t_n$ and $u = u_1 \ldots u_n$ and show $\mathsf{C} \xrightarrow{t} \mathsf{D}$ by induction on n.

- For $n = 1$, $\pi_u = \pi_t$ implies $u = u_1 = t_1 = t$, so $\mathsf{C} \xrightarrow{u} \mathsf{D}$ implies $\mathsf{C} \xrightarrow{t} \mathsf{D}$.
- For $n > 1$, let and $u = u_1 u'$ and $t = t_1 t'$. $\mathsf{C} \xrightarrow{u} \mathsf{D}$ implies the existence of a D_1 with $\mathsf{C} \xrightarrow{u_1} \mathsf{D}_1$ and $\mathsf{D}_1 \xrightarrow{u'} \mathsf{D}$. Since $(\Gamma, <) \vdash t : A \multimap B$ holds, Lemma 3(part (1)) implies $t : A \multimap B$, so by Theorem 1 and $shape(\mathsf{C}) = A$ there is a (unique) E such that $\mathsf{C} \xrightarrow{t} \mathsf{E}$ and $shape(\mathsf{E}) = B$. Consequently, t is an interleaving for t and A, so it respects $\sqsubset$ and $<$, and there is an E_1 such that $\mathsf{C} \xrightarrow{t_1} \mathsf{E}_1$ and $\mathsf{E}_1 \xrightarrow{t'} \mathsf{E}$. We perform a case distinction on whether $FU(u_1) = FU(t_1)$ holds.

Case $FU(u_1) = FU(t_1)$**.** Instructions u_1 and t_1 both execute in configuration C, and they are in the same pipeline, so they must be equal, and $\mathsf{D}_1 = \mathsf{E}_1$ and $\pi_{u'} = \pi_{t'}$ follow. By Lemma 4, there are Δ and $\prec$ such that for $C = \mathit{shape}(\mathsf{E}_1) = \mathit{shape}(\mathsf{D}_1)$ there is a derivation for $(\Delta, \prec) \vdash t' : C \multimap B$, and $\prec$ is contained in $<$ and hence in $\sqsubset$. Furthermore, t' is sequential for $\sqsubset$ and C as for any interleaving s for t' and C, the program $t_1\, s$ is an interleaving for t and A and thus respects $\sqsubset$, so s must respect $\sqsubset$. We can hence apply the induction hypothesis to obtain that t' is deterministic for C. Therefore, for any F with $\mathit{shape}(\mathsf{F}) = C$, $\mathsf{F} \xrightarrow{\pi_{t'}} \mathsf{G}$ and $\mathsf{F} \xrightarrow{\pi_{t'}} \mathsf{H}$ implies $\mathsf{G} = \mathsf{H}$. We choose $\mathsf{F} = \mathsf{D}_1 = \mathsf{E}_1$, $\mathsf{G} = \mathsf{D}$, $\mathsf{F} \xrightarrow{\pi_{t'}} \mathsf{D}$ via interleaving u', and $\mathsf{H} = \mathsf{E}$ via interleaving t'. Consequently, $\mathsf{E} = \mathsf{D}$ holds.

Case $FU(u_1) \neq FU(t_1)$**.** Instructions u_1 and t_1 are completely independent: for their principal typings $u_1 :: A_{u_1} \multimap B_{u_1}$ and $t_1 :: A_{t_1} \multimap B_{t_1}$, $\mathit{prime}(A_{u_1}, A_{t_1})$ and $\mathit{prime}(B_{u_1}, B_{t_1})$ hold, proven as follows.

Suppose q is a joint factor of A_{u_1} and A_{t_1}. Then q fulfils $\gamma(q) = FU(u_1) \neq FU(t_1) = \gamma(q)$, a contradiction. Suppose q is a joint factor of B_{u_1} and B_{t_1}, then Lemma 3(part (4)) implies $t_1 < u_1$ because $u_1 = t_l$ holds for some $l > 1$ due to $\pi_u = \pi_t$. As $<$ is contained in $\sqsubset$, this contradicts the fact that u respects $\sqsubset$, i.e. no instruction may precede u_1 in $\sqsubset$.

Consequently, $\mathit{shape}(\mathsf{C})$ contains both A_{u_1} and A_{t_1} and can thus be written as $\mathit{shape}(\mathsf{C}) = A_{u_1} \otimes A_{t_1} \otimes X$ for some X. Theorem 1 now implies

$$\mathit{shape}(\mathsf{D}_1) = B_{u_1} \otimes A_{t_1} \otimes X \text{ and } \mathit{shape}(\mathsf{E}_1) = A_{u_1} \otimes B_{t_1} \otimes X.$$

Hence t_1 can be executed in D_1 and u_1 in E_1, and the above properties on the disjointness of the A_i and B_i imply that there is a common C_1 with

$$\mathsf{E}_1 \xrightarrow{u_1} \mathsf{C}_1,\ \mathsf{D}_1 \xrightarrow{t_1} \mathsf{C}_1 \text{ and } \mathit{shape}(\mathsf{C}_1) = B_{u_1} \otimes B_{t_1} \otimes X$$

as visualised in Figure 4.5. As $u_1 = t_l$ occurs in t' and $t_1 = u_k$ in u' for some k, there are s', s'', r' and r'' such that

$$t' = s'\, u_1\, s'' \text{ and } u' = r'\, t_1\, r''$$

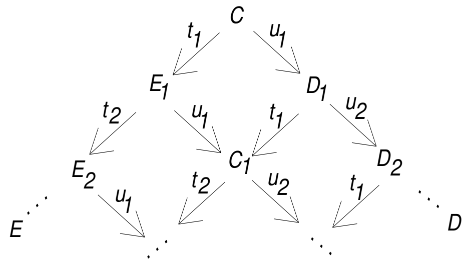


Figure 4.5: Confluence of u and t.

Furthermore, u' and t' are sequential for $shape(\mathsf{D}_1)$ and $shape(\mathsf{E}_1)$ as they are postfixes for u and t. We can thus apply Lemma 4 to the outer spines of the execution tree in Figure 4.5, so there are Δ, Λ, $\prec$ and $\subset$ such that $(\Delta,\prec) \vdash s'\,s'' : shape(\mathsf{C}_1) \multimap B$ and $(\Lambda,\subset) \vdash r'\,r'' : shape(\mathsf{C}_1) \multimap B$ where $\prec$ and $\subset$ are contained in $<$ and thus in $\sqsubset$.

By applying the induction hypothesis we obtain determinacy of t' and u' for $shape(\mathsf{E}_1)$ and $shape(\mathsf{D}_1)$, respectively. Code sequences $u_1\,s'\,s''$ and $t_1\,r'\,r''$ are interleavings of t' and s' for $shape(\mathsf{E}_1)$ and $shape(\mathsf{D}_1)$, respectively, as u_1 and t_1 are at the head positions in E_1 and D_1 and $\pi_{u_1\,s'\,s''} = \pi_{s'\,u_1\,s''} = \pi_{t'}$ and $\pi_{t_1\,r'\,r''} = \pi_{r'\,t_1\,r''} = \pi_{u'}$ hold. Thus, $\mathsf{C}_1 \xrightarrow{s'\,s''} \mathsf{E}$ and $\mathsf{C}_1 \xrightarrow{r'\,r''} \mathsf{D}$ hold. By construction, for $s = s'\,s''$ and $r = r'r''$ the equality $\pi_r = \pi_s$ holds as both sides result from π_t by executing t_1 and u_1. Again, s and r are sequential for $shape(\mathsf{C})$ as they are interleavings of a postfix of t, hence the claim follows from the induction hypothesis.

□

The derived order $<$ thus represents the constraints necessary for (relativised) determinism. Arbitrary means may be used for establishing the existence of a relation $\sqsubset$ which contains $<$.

4.3.2 Pipeline-dependencies

Many programs can be proven deterministic by exploiting the order of instructions in the instruction pipelines. As the dynamic semantics inserts instructions into waiting stations in program order and functional units remove only head instructions, the relative order of instructions $[n]code_n$ and $[m]code_m$ which are mapped to the same functional unit is preserved.

Definition 12. *For instruction sequence*

$$s\,[n]code_n\,r\,[m]code_m\,u,$$

$[m]$ *is* pipeline-dependent *on* $[n]$*, written* $[n] \lessdot [m]$*, if* $FU(code_n) = FU(code_m)$ *holds.*

Members of the same pipeline can never show a race hazard because no two instructions can be in operation concurrently.

Proposition 12. *Let t be a code sequence, A arbitrary and let $\lessdot$ be the order of pipeline-dependencies for t. Then t is sequential for $\lessdot$ and A.*

Proof. For interleaving s for t and A we have to show that s respects $\lessdot$, i.e. that $[n] \lessdot [m]$ implies that s can be written as $s'\,[n]code_n\,s''\,[m]code_m\,s'''$.
As $[n] \lessdot [m]$ is the pipeline-order with respect to t, there are t', t'' and t''' such that $t = t'\,[n]code_n\,t''\,[m]code_m\,t'''$ and $FU(code_n) = FU(code_m)$ holds. For the component $i \in \{1,2,3\}$ corresponding to $FU(code_n)$ and $\pi_t = t_1 \parallel t_2 \parallel t_3$ we thus obtain $t_i = u'\,[n]code_n\,u''\,[m]code_m\,u'''$ for some u', u'' and u'''. As s is an interleaving of t, $\pi_s = \pi_t$ holds, so for $\pi_s = s_1 \parallel s_2 \parallel s_3$ we have $s_i = t_i$. By the definition of π_s the claim follows. $\square$

Consequently, programs are safe if all constraints $<$ can be discharged using pipeline-dependencies:

Corollary 2. *If* $(\Gamma, <) \vdash t : A \multimap B$ *and* $<$ *is contained in the order* $\lessdot$ *of pipeline-dependencies in t, then t is safe for A. If additionally $A = \mathbf{1}$ holds, then t is safe.*

Proof. Combine Propositions 12 and 11 and Corollary 1. $\square$

As $(\Gamma, <) \vdash t : A \multimap B$ implies $(\Gamma, <) \vdash t : A \otimes X \multimap B \otimes X$, safety with respect to A implies safety with respect to $A \otimes X$, provided that only pipeline-dependencies are used for discharging the constraints.

Often, serialisation by pipeline-dependencies suffices to ensure determinism. For example, consider the program (4.1)

$$\begin{array}{llll} [1]\mathtt{ldc}\ 1\ \mathsf{q}_1 & [3]\mathtt{ldc}\ 3\ \mathsf{q}_2 & [5]\mathtt{mul}\ \mathsf{q}_2\ \mathsf{q}_2\ \mathsf{q}_1 & [7]\mathtt{st}\ \mathsf{q}_3\ \mathsf{q}_4 \\ [2]\mathtt{ldc}\ 2\ \mathsf{q}_2 & [4]\mathtt{ldc}\ 4\ \mathsf{q}_4 & [6]\mathtt{add}\ \mathsf{q}_1\ \mathsf{q}_1\ \mathsf{q}_3 & \end{array} \tag{4.1}$$

Provided that $\gamma(\mathsf{q}_1) = \mathtt{ALU}$, $\gamma(\mathsf{q}_2) = \mathtt{MUL}$ and $\gamma(\mathsf{q}_3) = \gamma(\mathsf{q}_4) = \mathtt{MEM}$ hold, this program can be given the type $(\Gamma, <) \vdash t : \mathbf{1} \multimap \mathbf{1}$ where $[2] < [3]$, $[1] < [5]$ and $\Gamma = \{\mathsf{q}_1 : ([1],[5]), \mathsf{q}_2 : ([2],[3]), \mathsf{q}_3 : ([6],[6]), \mathsf{q}_4 : ([4],[4])\}$. If an architecture has sufficiently many queues such that one queue can be dedicated to each forwarding path $fu_1 \to fu_2$, program (4.1) can be transformed to

$$\begin{array}{llll} [1']\mathtt{ldc}\ 1\ \mathsf{q}_5 & [3']\mathtt{ldc}\ 3\ \mathsf{q}_2 & [5']\mathtt{mul}\ \mathsf{q}_2\ \mathsf{q}_2\ \mathsf{q}_1 & [7']\mathtt{st}\ \mathsf{q}_3\ \mathsf{q}_4 \\ [2']\mathtt{ldc}\ 2\ \mathsf{q}_2 & [4']\mathtt{ldc}\ 4\ \mathsf{q}_4 & [6']\mathtt{add}\ \mathsf{q}_5\ \mathsf{q}_1\ \mathsf{q}_3 & \end{array} \tag{4.2}$$

where the new queue q_5 is dedicated to forwardings $\mathtt{MEM} \to \mathtt{ALU}$ and the other queues are assigned to forwardings

$$\mathsf{q}_1 : \mathtt{MUL} \to \mathtt{ALU},\ \mathsf{q}_2 : \mathtt{MEM} \to \mathtt{MUL},\ \mathsf{q}_3 : \mathtt{ALU} \to \mathtt{MEM} \text{ and } \mathsf{q}_4 : \mathtt{MEM} \to \mathtt{MEM}.$$

The typing for the transformed program reads $(\Delta, \prec) \vdash t' : \mathbf{1} \multimap \mathbf{1}$ where $[2'] \prec [3']$ is the only conflict. This constraint is easily seen to be fulfilled by $\lessdot$ as instructions $[2']$ and $[3']$ both execute on $\mathtt{MEM}$. The program is deterministic for all A as the shape of initial configurations has no influence on the execution order of instructions in the same pipeline. Functionally, distributed and sequential executions of program (4.2) also agree with the configuration resulting from (sequential execution of) program (4.1).

Not every architecture has enough operand queues for the above transformation to be applied – the number of operand queues appears to grow quadratically in the number of functional units. Furthermore, some programs need more than one queue for each forwarding path in order to guarantee functional correctness. For example, suppose $\gamma(\mathsf{q}_1) = \mathtt{ALU}$ and $\gamma(\mathsf{q}_2) = \gamma(\mathsf{q}_3) = \mathtt{MEM}$. The program

$$\begin{array}{lll} [1]\mathtt{ldc}\ 4\ \mathsf{q}_1 & [3]\mathtt{ldc}\ 6\ \mathsf{q}_1 & [5]\mathtt{add}\ \mathsf{q}_1\ \mathsf{q}_1\ \mathsf{q}_3 \\ [2]\mathtt{dec}\ \mathsf{q}_1\ \mathsf{q}_2 & [4]\mathtt{ldc}\ 8\ \mathsf{q}_1 & [6]\mathtt{st}\ \mathsf{q}_3\ \mathsf{q}_2 \end{array} \tag{4.3}$$

cannot be blindly converted into the form where only one queue of connectivity ALU → MEM is employed. The resulting program

$$\begin{array}{lll} [1]\texttt{ldc}\ 4\ \texttt{q}_1 & [3]\texttt{ldc}\ 6\ \texttt{q}_1 & [5]\texttt{add}\ \texttt{q}_1\ \texttt{q}_1\ \texttt{q}_2 \\ [2]\texttt{dec}\ \texttt{q}_1\ \texttt{q}_2 & [4]\texttt{ldc}\ 8\ \texttt{q}_1 & [6]\texttt{st}\ \texttt{q}_2\ \texttt{q}_2 \end{array} \tag{4.4}$$

would be functionally different as the values inside queue $\texttt{q}_2$ are swapped, and the unsymmetric instruction [6] would hence produce an incorrect result. Occasionally, the compiler might be able to reorder instructions to recover the correct behaviour, but as a general transformation rule, the above translation is incorrect.

However, an architecture might have a mechanism for *varying* the connectivity pattern, and for these architectures the above transformation might be appropriate. Consequently, the number of queues does not necessarily have to grow quadratically in the number of functional units, and at certain times more than one queue may be available for a particular forwarding path. For example, in reconfigurable architectures such as field-programmable gate arrays, the communication pattern may be modified at predefined moments in time. The above descriptions such as $\texttt{q}_1 : \texttt{MUL} \rightarrow \texttt{ALU}$ can thus be interpreted as generalisations of the association function γ which describe the connectivity needed for a particular region of code. This approach may be extensible to verify ordinary code and reconfiguration instructions in combination, using a type-based calculus where connectivity descriptions are part of the typing context. In architectures where dynamic reconfiguration is not possible, such a calculus for *regions of connectivity* may still be useful. In this context, connectivity descriptions capture an aspect of the *scheduling policy*: for the code region in question, operand queues are *used* as described by the context, and at the boundaries between regions a *virtual* reconfiguration in performed, i.e. a change of usage policy.

4.3.3 Data-dependencies

In addition to serialisation by pipeline-dependencies, a compiler may fulfil serialisation constraints by exploiting *data-dependencies*. Operand queues serialise the execution of the producer of a value and its consumer. Each item needs to be inserted into an operand queue before it can be consumed, and consumption removes an item from its queue. For example, program

$$t = [1]\texttt{ldc}\;4\;\mathtt{q}_1\;[2]\texttt{dec}\;\mathtt{q}_1\;\mathtt{q}_1 \tag{4.5}$$

can be typed $(\Gamma, [1] < [2]) \vdash t : \mathbf{1} \multimap \mathtt{q}_1$. For an initial configuration of shape **1**, no interleaving of t will execute [2] before [1], so the race hazard in fact does not occur and the constraint can be discharged. Serialisation using data-dependencies can be combined with pipeline-dependencies. Program

$$[1]\texttt{ldc}\;7\;\mathtt{q}_2\;[2]\texttt{ldc}\;4\;\mathtt{q}_1\;[3]\texttt{dec}\;\mathtt{q}_1\;\mathtt{q}_2 \tag{4.6}$$

carries the constraint $[1] < [3]$ which can be fulfilled for any configuration of shape **1** by observing that [2] is always pipeline-dependent on [1], and that [3] is data-dependent on [2] relative to that shape, so [1] always precedes [3] and the serialisation constraint is fulfilled.

Serialisation by operand queues concerns the *observed* data-dependencies, and the compiler should ensure that these coincide with the true data-dependencies in the original program. At a low level, analysing data-dependencies is more complex than analysing pipeline-dependencies because the former ones are influenced by contextual instructions and the shape of initial configurations. For example, prefixing program (4.5) by code of type $A \multimap \mathtt{q}_1$ where A is arbitrary destroys the dependency between [1] and [2] and results in a race hazard. In particular, data-dependencies are not preserved by the weakening rule.

Definition 13. *Let $t = ins_1 \dots ins_m \dots ins_n$ where $m \geq 1$ and $ins_i = [n_i]code_i$. Then ins_m is called the i-*th writer to *q* in *t if there are l, X, Y and $k > 0$ such that $\otimes_{j=1}^{m-1} B_{code_j} = q^l \otimes X$, $B_{code_m} = q^k \otimes Y$, $prime(q, X \otimes Y)$ and $l < i \leq k + l$.*

*Likewise, ins_m is called the i-*th reader from *q* in *t if there are l, X, Y and $k > 0$ such that $\otimes_{j=1}^{m-1} A_{code_j} = q^l \otimes X$, $A_{code_m} = q^k \otimes Y$, $prime(q, X \otimes Y)$ and $l < i \leq k + l$.*

For $q = q_1 = q_2$, an instruction like `add` $q_1\ q_2\ q_3$ is the i-th as well as the $(i+1)$-th reader from q in t, and a `dupl` instruction may similarly be a multiple writer.

Definition 14. *For A with $t : A \multimap B$, $[m]$ is* data-dependent on *$[n]$* for *A, written $[n] <_A [m]$, if there are q, $i > 0$ and X such that $[n]$ is the i-th writer to q in t and $[m]$ is the $(i+k)$-th reader, where $A = q^k \otimes X$ and $prime(q, X)$.*

Proposition 13. *Let $t : A \multimap B$. Then $<_A$ respects the order of instructions in t.*

Proof. We have to show that for $[n] <_A [m]$ there are r, s and u with

$$t = r\,[n]code_n\,s\,[m]code_m\,u.$$

For $[n] <_A [m]$ there are by definition q, $i > 0$ and X such that $[n]$ is the i-th writer to q in t and $[m]$ is the $(i+k)$-th reader, where $A = q^k \otimes X$ and $prime(q,X)$. In particular, instructions $[n]$ and $[m]$ occur in t, and $B_{code_n} = q^{k_n} \otimes Y$ and $A_{code_m} = q^{k_m} \otimes Z$ hold with $k_n > 0$ and $k_m > 0$ and $prime(q, Y \otimes Z)$.
Suppose $[m]$ occurs before $[n]$, i.e. there are r', s' and u' with

$$t = r'\,[m]code_m\,s'\,[n]code_n\,u'.$$

Since $t : A \multimap B$ holds, there are $C, \ldots, F$ with

$$\frac{r' : A \multimap C \quad [m] : C \multimap D \quad s' : D \multimap E \quad [n] : E \multimap F \quad u' : F \multimap B}{r'\,[m]code_m\,s'\,[n]code_n\,u' : A \multimap B}$$

This typing yields

$$k_1 = k + k_2 - k_3$$

whenever $C = q^{k_1} \otimes X_1$, $\otimes_{[\_]code \in r'} B_{code} = q^{k_2} \otimes X_2$, $\otimes_{[\_]code \in r'} A_{code} = q^{k_3} \otimes X_3$ and $prime(q, X_1 \otimes X_2 \otimes X_3)$ hold.
We now perform an induction on i, aiming at a contradiction to the typability of t.

Case $i = 1$. As $[n]$ is the first writer to q in t, there are no writers to q in $r'\,[m]\,s'$. In particular, there are no writers to q in r', so all instructions $[\_]code$ in r' fulfil $prime(q, B_{code})$ and $k_2 = 0$ follows.

On the other hand, $[m]$ is the $(k+1)$-th reader from q in t, so r' contains k readers, hence $k_3 = k$. As A contains q^k and there are k readers in r' and no writers, $prime(q,C)$ holds:

$$k_1 = k + k_2 - k_3 = 0.$$

Consequently, no composition of r' with $[m]$ is possible as $[m]$ is a reader from q, so requires $C = A_{code_m} \otimes Z = q \otimes Z' \otimes Z$ for some Z' and Z.

Case $i > 1$. Then r' includes at most $i-1$ writers to q as exactly $i-1$ writers precede $[n]$ in t. Hence $k_2 \leq i-1$. As $k+i-1$ readers precede $[m]$ in t, there must be $k+i-1$ readers in r', i.e. $k_3 = k+i-1$. Hence,

$$k_1 = k + k_2 - k_3 \leq k + i - 1 - k - i + 1 = 0.$$

Again, $prime(q, C)$ follows and no composition with $[m] : q \otimes C' \multimap D$ is possible.

In both cases we obtain a contradiction to the well-typedness of t. □

Program (4.5) suggests that $<_A$ can be used for discharging constraints, as $[1] <_{\mathbf{1}} [2]$ holds. Likewise, program (4.6) motivates us to use pipeline-dependencies and data-dependencies in combination, as $[2] <_{\mathbf{1}} [3]$ and $[1] \lessdot [2]$ holds, so the constraint $[1] < [3]$ can be fulfilled. However, some care is needed, since data-dependencies do *not* entail sequentiality, and neither does their union with pipeline-dependencies.

Example. Let $\gamma(\mathsf{q}_1) = \mathtt{ALU}$ and $\gamma(\mathsf{q}_2) = \mathtt{MUL}$ and

$$t \quad = \quad [1]\mathtt{ldc}\ 3\ \mathsf{q}_1\ [2]\mathtt{id}^{\mathtt{MUL}}\ \mathsf{q}_2\ \mathsf{q}_1\ [3]\mathtt{dec}\ \mathsf{q}_1\ \mathsf{q}_3. \tag{4.7}$$

For $A = \mathsf{q}_2$ and $B = \mathsf{q}_1 \otimes \mathsf{q}_3$ there is a derivation for $(\Gamma, [1] < [2]) \vdash t : A \multimap B$, and $[1] <_A [3]$ holds since $[1]$ is the first writer to q_1, $[3]$ is the first reader from q_1 and $prime(\mathsf{q}_1, A)$. There are no pipeline-dependencies, so $<_A$ and $\lessdot \cup <_A$ consist of the pair $([1], [3])$.
Take $u = [2][3][1]$. Then $u : A \multimap B$ is an interleaving for t and A as $\pi_u = [3] \parallel [2] \parallel [1] = \pi_t$ holds and for any C with $shape(\mathsf{C}) = A$ there is a D with $\mathsf{C} \xrightarrow{u} \mathsf{D}$. However, u does not respect $<_A$, hence t is not sequential for $<_A$ or $\lessdot \cup <_A$ and A. ◇

Consequently, an analogue to Proposition 12 for data-dependencies would fail. The data-dependency in the program (4.7) is only fulfilled *provided* that $[3]$ is bound never to consume the result of $[2]$. Of course, in order to prove this fact one needs to show that any interleaving executes $[1]$ before $[2]$ – which is what we set out to prove in the first place when aiming at discharging $[1] < [2]$. This appears to imply that data-dependencies cannot be used for discharging constraints as they *need* sequentiality before they can be used for *proving* sequentiality. However, pipeline-dependencies and data-dependencies can be combined in a different way. Instead of taking their union, we extend the set of pipeline-dependencies by a *single* data-dependency at a time. An entry $[n] <_A [m]$ is added provided that all instructions which are in an output conflict with $[n]$ are already related to $[n]$. In fact, this inductive process succeeds if $\lessdot$ is replaced by any order $\sqsubset$ for which t is sequential.

Proposition 14. *Let t be sequential for $\sqsubset$ and A and let $[n] <_A [m]$. For all instructions $[l]code_l$ in t with $n \neq l \neq m$, let*

- *$[l] \sqsubset [n]$; or $[n] \sqsubset [l]$ hold if there is a q which occurs in B_{code_l} and B_{code_n}*
- *$[l] \sqsubset [m]$; or $[m] \sqsubset [l]$ hold if there is a q which occurs in A_{code_l} and A_{code_m}.*

If $\sqsubset \cup \{([n],[m])\}$ is a partial order, then t is sequential for $\sqsubset \cup \{([n],[m])\}$ and A.

Proof. Let s be an interleaving for t and A, i.e. $\pi_s = \pi_t$ and for any C with $shape(\mathsf{C})$ there is a D such that $\mathsf{C} \xrightarrow{s} \mathsf{D}$. We have to show that s respects $\sqsubset \cup \{([n],[m])\}$ where $\sqsubset$ and $[n] <_A [m]$ are as given in the proposition and the side condition on competitors $[l]$ is fulfilled.
Any pair $([k_1],[k_2])$ in $\sqsubset \cup \{([n],[m])\}$ with $([k_1],[k_2]) \neq ([n],[m])$ is contained in $\sqsubset$ (modulo the transitive closure) and hence respected by s by assumption, i.e. $[k_1]$ occurs before $[k_2]$ in s. Hence, we only have to show that s respects $([n],[m])$.
Suppose $[m]$ occurs before $[n]$ in s. Then s can be written as

$$s = r\,[m]\,u\,[n]\,v$$

for some code sequences r, u and v. For C with $shape(\mathsf{C}) = A$ we have $\mathsf{C} \xrightarrow{s} \mathsf{D}$ for some D, so there are D_1 and D_2 such that

$$\mathsf{C} \xrightarrow{r} \mathsf{D}_1 \xrightarrow{[m]} \mathsf{D}_2 \xrightarrow{u\,[n]\,v} \mathsf{D}.$$

Since s is an interleaving of t for A which respects $\sqsubset$,

- r contains all instructions *ins* with $ins \sqsubset [m]$.
- $u\,[n]\,v$ contains all instructions *ins* with $[m] \sqsubset ins$.
- $r\,[m]\,u$ contains all instructions *ins* with $ins \sqsubset [n]$.

- v contains all instructions ins with $[n] \subset ins$.

Since $[n] <_A [m]$ holds, there are q and $i > 0$ such that $[n]$ is the i-th writer to q in t and $[m]$ the $(k+i)$-th reader, where $A = q^k \otimes X$ and $prime(q,X)$. By the side condition on instructions forwarding to the same queue as $[n]$, instructions $[l]$ which are the j-th writer to q in t fulfil

- $[l] \subset [n]$ (or $l = n$) if $1 \leq j < i$ and
- $[n] \subset [l]$ (or $l = n$) if $j > i$.

In particular, any writer $[l]$ to q which occurs in r cannot fulfil $[n] \subset [l]$ (see above, s obeys $\subset$), so it must fulfil $[l] \subset [n]$ as $l = n$ does not hold. There are at most $i-1$ such writers as other writers fulfil $[n] \subset [l]$.
Similarly, readers $[l]$ from q which occur in t before $[m]$ must occur in r as they fulfil $[l] \subset [m]$ – if they occurred in $u\,[n]\,v$ then s would not respect $\subset$. There are exactly $k+i-1$ such readers since $[m]$ is the $(k+i)$-th reader and other readers fulfil $[m] \subset [l]$.
Consequently, for $shape(\mathsf{D}_1) = q^h \otimes Y$ with $prime(q,Y)$ we obtain

$$h \leq k + (i-1) - (k+i-1) = 0.$$

This contradicts $\mathsf{D}_1 \xrightarrow{[m]} \mathsf{D}_2$ as $[m]$ is a reader from q, so q is a factor of A_{code_m} and D_1 must consequently contain at least one entry in q. ☐

Plugging in pipeline-dependencies $\lessdot$ for the initial $\subset$ is merely a convenient choice for starting the iterative process – an axiom in a formal inductive derivation system for sequentialisation orders in which the process described in the proposition represents the step case. Using pipeline-dependencies ensures that the side condition relating readers $[l]$ with $[m]$ is always fulfilled.

Example. In program (4.5), there is no $[1] \neq [l] \neq [2]$, so the side condition for extending the (empty!) pipeline-dependencies is trivially fulfilled and we can discharge $[1] < [2]$.
In program (4.6), the only pipeline-dependency is $[1] \lessdot [2]$, and no $[l]$ exists which forwards to $[2]$'s destination queue q_1 and fulfils $[2] \neq [l] \neq [3]$. Hence $[2] <_{\mathbf{1}} [3]$ can safely be added to $[1] \lessdot [2]$ and the constraint $[1] < [3]$ can be discharged.

In program (4.7), the pipeline-dependencies do not get the iterative process started. All instructions queue in different pipelines, and the data-dependency $[1] <_A [3]$ can never be used as the condition on competitor $[l] = [2]$ of $[1]$ is not fulfilled. $\diamond$

4.3.4 Other sequentialisation mechanisms

We already discussed *program transformations* as a means for creating pipeline-dependencies, and other transformations may prove useful, too. For example, the constraint $[1] < [2]$ in

$$[1]\mathtt{ld}\ \mathtt{q}_1\ \mathtt{q}_2\ [2]\mathtt{dec}\ \mathtt{q}_3\ \mathtt{q}_2$$

may be resolved by introducing artificial dependencies. A compiler could (for the typing $A \multimap B$ where $A = \mathtt{q}_1 \otimes \mathtt{q}_3$) insert additional instructions to yield

$$[1]\mathtt{ld}\ \mathtt{q}_1\ \mathtt{q}_2\ [1']\mathtt{ldc}\ 0\ \mathtt{q}_4\ [1'']\mathtt{skip}^{\mathtt{ALU}}\ \mathtt{q}_4\ [2]\mathtt{dec}\ \mathtt{q}_3\ \mathtt{q}_2 : A \multimap B$$

and thus discharge $[1] < [2]$ using $[1] \lessdot [1'] <_A [1''] \lessdot [2]$. Little instruction-level parallelism is compromised as the instructions $[1]$ and $[2]$ are not supposed to execute concurrently anyway. On the other hand, performance degrades if this transformation is employed excessively as additional instructions need to be loaded from memory, inserted into pipelines and executed. Therefore, a compiler should first aim to achieve sequentialisation by reordering instructions (aiming at utilising more pipeline-dependencies) before introducing such instructions.

SCALP contains a sequentialisation instruction with behaviour of the shape

$$\frac{\gamma(op_1) = \gamma(op_2)}{\mathtt{seq}\ op_1\ op_2\ op_3\ op_4 : op_1 \otimes op_2 \multimap op_3 \otimes op_4}$$

which awaits operands in op_1 and op_2 and then sends them unchanged to op_3 and op_4. However, this requires operands to be communicated more often than necessary and apparently forces op_1 and op_2 to be distinct.

More sophisticated assembly languages might encode the above communication patterns into the opcodes by allowing instructions to exchange notifications of completion. These signals could be realised by special-purpose operand queues $\mathtt{p}_i$ and encodings of `dupl` into other instructions. In

$$[1]\mathtt{ldp}\ \mathtt{q}_1\ \mathtt{q}_2\ \mathtt{p}_1\ [2]\mathtt{pdec}\ \mathtt{p}_1\ \mathtt{q}_3\ \mathtt{q}_2$$

the (initially empty) queue $\mathtt{p}_1$ achieves the same result as the above code insertion if `ldp` and `pdec` are typed via axioms of the form

$$\text{AX-P}\ \frac{}{\mathtt{ldp}\ op_1\ op_2\ p : X \otimes op_1 \multimap op_2 \otimes p \otimes X}\ \gamma(op_1) = \mathtt{MEM}$$

and

$$\text{P-AX}\ \frac{}{\mathtt{pdec}\ p\ op_1\ op_2 : p \otimes op_1 \otimes X \multimap op_2 \otimes X}\ \gamma(p) = \gamma(op_1) = \mathtt{ALU}$$

and executed by

$$\text{LD-P}\ \frac{\mathsf{C} \xrightarrow{(read(\mathtt{MEM}),a,op_1)} (\mathsf{Q},\mathsf{M}) \quad (\mathsf{Q},\mathsf{M}) \xrightarrow{(write,\mathsf{M}(a),op_2)} \mathsf{E} \quad \mathsf{E} \xrightarrow{(write,\cdot,p)} \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{ldp}\ op_1\ op_2\ p} \mathsf{D}}$$

and

$$\text{P-DEC}\frac{\mathsf{C} \xrightarrow{(read(\mathtt{ALU}),\cdot,p)} \mathsf{F} \quad \mathsf{F} \xrightarrow{(read(\mathtt{ALU}),a,op_1)} \mathsf{E} \quad \mathsf{E} \xrightarrow{(write,a-1,op_2)} \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{pdec}\ p\ op_1\ op_2} \mathsf{D}}$$

where $\cdot$ denotes an anonymous value. Queues $\mathtt{p}_i$ need only be wide enough to carry a single signal (no data) and may thus be not too expensive in hardware, although some runtime costs are involved. A high-performance compiler would replace ordinary instructions with instructions of the new form only when other sequentialisation techniques fail.

Our calculus thus has the necessary ingredients for exploring the trade-offs between program analysis, code transformation techniques and hardware modifications. If the dynamic semantics is equipped with performance annotations a system designer can study design alternatives by simulation.

4.4 Program execution

Programs including jump instructions are treated similarly to Section 3.5. Basic blocks consist of instruction sequences where at most the last instruction may be a jump, and programs $\mathtt{P}$ are represented as pairs $\mathtt{P} = (\mathtt{N},\mathtt{A})$ of basic blocks and control flow arrows. Modern processor implementations employ a variety of means how execution of instructions inside basic blocks and scheduling of new blocks interact, Our dynamic semantics models a scheme where the loading of instructions from memory may interleave arbitrarily with the execution of instructions. Whenever a branch target becomes known, the issue unit may load instructions from memory and insert them into the pipelines.

4.4.1 Dynamic semantics

The dynamic semantics generalises the ternary parallel programs of the previous sections to programs with four components $t_1 \parallel t_2 \parallel t_3 \parallel t_4$ where the rightmost component represents the $\mathtt{BU}$ pipeline. Composition $\pi_1\pi_2$ and conversion of a sequential program t into its parallel program π_t are defined similarly to Section 4.1. Formally, reduction $\mathsf{C} \xrightarrow{t}_{fu} \mathsf{D}$ is defined by the rules

$$\text{3FU}_1 \frac{\mathsf{C} \xrightarrow{[n]code} \mathsf{D}}{\mathsf{C} \xrightarrow{[n]code}_{fu} \mathsf{D}} \begin{cases} FU(code) = fu \\ \mathsf{C} = (\mathsf{Q},\mathsf{M},\widetilde{n}) \end{cases}$$

and

$$\text{3FU}_2 \frac{\mathsf{C} \xrightarrow{s}_{fu} \mathsf{E} \quad \mathsf{E} \xrightarrow{t}_{fu} \mathsf{D}}{\mathsf{C} \xrightarrow{st}_{fu} \mathsf{D}}\ \mathsf{C} = (\mathsf{Q},\mathsf{M},\widetilde{n})$$

where the side conditions $\mathsf{C} = (\mathsf{Q},\mathsf{M},\widetilde{n})$ indicate that configurations are of the ternary shape from Section 3.5.1 (Figure 3.8), and similarly for

$$\text{3DIS}_1 \frac{\mathsf{C} \xrightarrow{t}_{fu} \mathsf{D}}{\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}}\ \mathsf{C} = (\mathsf{Q},\mathsf{M},\widetilde{n}) \quad \text{and}\ \text{3DIS}_2 \frac{\mathsf{C} \xrightarrow{\pi_1} \mathsf{E} \quad \mathsf{E} \xrightarrow{\pi_2} \mathsf{D}}{\mathsf{C} \xrightarrow{\pi_1\pi_2} \mathsf{D}}\ \mathsf{C} = (\mathsf{Q},\mathsf{M},\widetilde{n})$$

The propositions and statements relating distributed and sequential dynamic execution, can be transferred to ternary configurations and the extended instruction set.

Interleaving the execution of instructions with the loading of basic blocks depending on the outcome of branch conditions is achieved through a relation $\mathsf{C} \xrightarrow{w} \mathsf{D}$, similar to the rules EXECUTE and COMPOSE in Section 3.5.1. We present a semantics where the issue unit is represented as a forth component of configurations, and consists of a parallel program π. Loading the basic block labelled n from the memory inserts the instructions $\mathtt{N}(n)$ into the issue unit. Arbitrary prefixes of π may then enter the pipelines and execute completely unsupervised. Consequently, the reloading and instruction execution may interleave freely. Formally, the dynamic semantics for distributed program execution is given by the rules EXEC, LOAD and COMPOSE.

$$\text{LOAD}\,\frac{}{(\mathsf{Q},\mathsf{M},n,\pi)\xrightarrow{n}(\mathsf{Q},\mathsf{M},nil,\pi\pi_t)}\ \mathtt{N}(n)=t$$

$$\text{EXEC}\,\frac{(\mathsf{Q},\mathsf{M},\widetilde{n})\xrightarrow{\pi_1}(\mathsf{P},\mathsf{N},\widetilde{m})}{(\mathsf{Q},\mathsf{M},\widetilde{n},\pi_1\pi_2)\xrightarrow{\lambda}(\mathsf{P},\mathsf{N},\widetilde{m},\pi_2)}$$

$$\text{COMPOSE}\,\frac{\mathsf{C}\xrightarrow{v}\mathsf{E}\quad \mathsf{E}\xrightarrow{w}\mathsf{D}}{\mathsf{C}\xrightarrow{vw}\mathsf{D}}$$

Rule COMPOSE formally agrees with the identically named rule in Section 3.5.1, but now relates configurations with four components.

4.4.2 Static semantics

Instantiating the typing rules from Section 4.3.1 (Figure 4.4) to `if` and `jmp` amounts to the explicit rules

$$\frac{[n]\mathtt{if}\ op\ n_1\ n_2 : A \multimap B}{(\emptyset,\emptyset)\vdash [n]\mathtt{if}\ op\ n_1\ n_2 : A \multimap B}\quad\text{and}\quad\frac{[n]\mathtt{jmp}\ m : A \multimap B}{(\emptyset,\emptyset)\vdash [n]\mathtt{jmp}\ m : A \multimap B}$$

and race conditions within basic blocks can be dealt with as discussed before. It should be noted that the analysis of data-dependencies $<_A$ has to be performed with respect to the initial shape A arising from matching a block with its neighbours, since applying the weakening rule changes the observed data-dependencies. Consequently, determinism and safety can be achieved for the bodies of basic blocks as discussed in the previous sections.

For typing programs $\mathtt{P}=(\mathtt{N},\mathtt{A})$, we insert a judgement $(\Gamma,<)\vdash n : A\multimap B$ into the type environment Σ provided that $n\in dom\,\mathtt{N}$ and $(\Gamma,<)\vdash \mathtt{N}(n) : A\multimap B$ holds such that t is safe for A. Again Σ contains exactly one such entry for each $n\in dom\,\mathtt{N}$. Given that pipelines might include instructions of several basic blocks we also have to consider race hazards between instructions of different basic blocks. We therefore introduce additional constraints $[l] < [k]$ between instructions labelled $[l]$ and $[k]$ whenever there is a queue q such that $[l]$ is the last writer to q in its block $\mathtt{N}(n)$, $[k]$ is the first writer to q in block $\mathtt{N}(m)$ and there is a path from n to m in $\mathtt{A}$. Since different copies of a basic block execute identical sets of instructions, it suffices to consider simple paths $n, \ldots, m$ where at most the two outermost nodes n and m are equal and no intermediate block mentions q. We denote such a path by $SP(n, op, m)$. In the typing rule

$$\text{DisPROG}\ \frac{\begin{array}{c}\Sigma = \cup_{n \in dom\, \mathtt{N}} \{(\Gamma_n, <_n) \vdash n : A_n \multimap B_n\} \\ (n,m) \in \mathtt{A} \text{ implies } B_n = A_m \\ \left.\begin{array}{c} SP(n, op, m) \text{ and} \\ op : (g, l) \in \Gamma_n \text{ and} \\ op : (k, h) \in \Gamma_m \end{array}\right\} \text{implies } [l] < [k] \end{array}}{(\Sigma, <) \vdash \mathtt{P} : \Box}\ \mathtt{P} = (\mathtt{N}, \mathtt{A})$$

the constraints $\{<_n \mid (\Gamma_n, <_n) \vdash n : A_n \multimap B_n \in \Sigma\} \cup <$ are interpreted modulo instantiations due to loops: in a constraint $[l] < [l]$ the right $[l]$ refers to the instance of $[l]$ in the next iteration.

These constraints suffice for making any program slice deterministic:

Proposition 15. *Let* $(\Sigma, <) \vdash \mathtt{P} : \Box$ *and* $(n_1, \ldots, n_k)$ *be a path in* $\mathtt{A}$*. Then there are* Γ *and* $\prec$ *such that* $(\Gamma, \prec) \vdash \mathtt{N}(n_1) \ldots \mathtt{N}(n_k) : A_{n_1} \multimap B_{n_k}$ *holds and* $\prec$ *is contained in* $\{<_n \mid (\Gamma_n, <_n) \vdash n : A_n \multimap B_n \in \Sigma\} \cup <$.

Proof. **Case** $k = 1$**.** Since $n_1 \in dom\ \mathtt{N}$, we have $(\Gamma_{n_1}, <_{n_1}) \vdash n_1 : A_{n_1} \multimap B_{n_1} \in \Sigma$ for some Γ_{n_1}, $<_{n_1}$, A_{n_1} and B_{n_1}. By the definition of Σ, $(\Gamma_{n_1}, <_{n_1}) \vdash \mathtt{N}(n_1) : A_{n_1} \multimap B_{n_1}$ holds. Take $\prec = <_1$ and $\Gamma = \Gamma_{n_1}$.

Case $k > 1$**.** For a path $(n_1, \ldots, n_k, n_{k+1})$ in $\mathtt{A}$ and $i \leq k+1$, there are $\Gamma_{n_i}, <_{n_i}, A_{n_i}$ and B_{n_i} with $(\Gamma_{n_i}, <_{n_i}) \vdash \mathtt{N}(n_i) : A_{n_i} \multimap B_{n_i} \in \Sigma$. By the definition of Σ, this implies

$$(\Gamma_{n_i}, <_{n_i}) \vdash \mathtt{N}(n_i) : A_{n_i} \multimap B_{n_i} \in \Sigma \ \text{ for all } i \leq k+1 \tag{4.8}$$

For $1 \leq i \leq k$ it is $(n_i, n_{i+1}) \in \mathtt{A}$ by the definition of a path, so $B_{n_i} = A_{n_{i+1}}$ by the side-condition for $\mathtt{P} : \Box$. In particular, $B_{n_k} = A_{n_{k+1}}$.

By induction hypothesis, there are Δ and $\sqsubset$ such that

$$(\Delta, \sqsubset) \vdash \mathtt{N}(n_1)\ldots\mathtt{N}(n_k) : A_{n_1} \multimap B_{n_k}$$

and $\sqsubset$ is contained in $\cup_{n \in dom\,\mathtt{N}}\{<_n \mid (\Gamma_n, <_n) \vdash \mathtt{N}(n) : A_n \multimap B_n \in \Sigma\} \cup <$. Also, specialising property (4.8) for $i = k+1$ yields

$$(\Gamma_{n_{k+1}}, <_{n_{k+1}}) \vdash \mathtt{N}(n_{k+1}) : A_{n_{k+1}} \multimap B_{n_{k+1}}.$$

The cut rule (see Figure 4.4), thus implies

$$(\Gamma, \prec) \vdash \mathtt{N}(n_1)\ldots\mathtt{N}(n_{k+1}) : A_{n_1} \multimap B_{n_{k+1}}$$

for $\Gamma = \Delta\#\Gamma_{n_{k+1}}$ and $\prec$ equal to

$$\sqsubset \cup <_{n_{k+1}} \cup \{(m,g) \mid op : (l,m) \in \Delta \text{ and } op : (g,h) \in \Gamma_{n_{k+1}}\}.$$

As remarked above, $\sqsubset$ is contained in $\cup_{n \in dom\,\mathtt{N}}\{<_n \mid (\Gamma_n, <_n) \vdash \mathtt{N}(n) : A_n \multimap B_n \in \Sigma\} \cup <$, and so is (obviously) $<_{n_{k+1}}$. For showing that all of $\prec$ is contained in this relation, it remains to show that pairs (m,g) where $op : (l,m) \in \Delta$ and $op : (g,h) \in \Gamma_{n_{k+1}}$ are contained in this relation.

For any such pair (m,g), there is a $j < k+1$ such that instruction m is the last writer to op in $\mathtt{N}(n_j)$ and g is the first writer to op in $\mathtt{N}(n_{k+1})$, and no instruction in $\mathtt{N}(n_{j+1}), \ldots, \mathtt{N}(n_k)$ writes to op. In particular, all blocks $\mathtt{N}(n_{j+1}), \ldots, \mathtt{N}(n_k)$ are different from $\mathtt{N}(n_j)$. By removing loops in $\mathtt{N}(n_{j+1}), \ldots, \mathtt{N}(n_k)$, the path $(\mathtt{N}(n_j), \ldots, \mathtt{N}(n_k), \mathtt{N}(n_{k+1}))$ contains a simple path $SP(\mathtt{N}(n_j), op, \mathtt{N}(n_{k+1}))$. By the side condition on $(\Sigma, <) \vdash \mathtt{P} : \Box$, the pair (m,g) is hence related by $[m] < [g]$. $\Box$

Consequently, distributed executions are safe if we can fulfil the constraints in $<$. For inserting the entries $(\Gamma_n, <_n) \vdash n : A_n \multimap B_n$ into Σ we had to prove them safe, but no further work needs to be spent on their bodies.
The following result relates distributed program execution as defined in this section to the dynamic semantics for sequential execution in Section 3.5.

Proposition 16. *Let* $(\Sigma, <) \vdash \mathtt{P} : \Box$, $n \in dom\,\mathtt{N}$ *and* $(\Gamma_n, <_n) \vdash \mathtt{N}(n) : A_n \multimap B_n \in \Sigma$.

1. *If* $(\mathsf{Q},\mathsf{M},n) \xrightarrow{w} (\mathsf{P},\mathsf{L},\widetilde{m})$ *then* $(\mathsf{Q},\mathsf{M},n,\pi_\varepsilon) \xrightarrow{w} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)$.

2. *Let* $\mathsf{C} = (\mathsf{Q},\mathsf{M},n,\pi_\varepsilon)$, *shape*$(\mathsf{C}) = A_n$ *and* $\mathsf{C} \xrightarrow{w} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)$. *Let* $\sqsubset$ *be a partial order containing* $<$ *and for all simple paths* $SP(\mathtt{N}(n_i),op,\mathtt{N}(n_j))$ *let the instruction sequence* $\mathtt{N}(n_i)\ldots\mathtt{N}(n_j) : A_{n_i} \multimap B_{n_j}$ *be sequential for* $\sqsubset$ *and* A_{n_i}. *Then* $(\mathsf{Q},\mathsf{M},n) \xrightarrow{w} (\mathsf{P},\mathsf{L},\widetilde{m})$.

Proof. 1. Rule induction for $(\mathsf{Q},\mathsf{M},n) \xrightarrow{w} (\mathsf{P},\mathsf{L},\widetilde{m})$ (see Section 3.5.1).

Case EXECUTE. By the definition of the rule EXECUTE, w has the form n and $(\mathsf{Q},\mathsf{M},nil) \xrightarrow{t} (\mathsf{P},\mathsf{L},\widetilde{m})$ holds where $t = \mathtt{N}(n)$. Proposition 7 implies $(\mathsf{Q},\mathsf{M},nil) \xrightarrow{\pi_t} (\mathsf{P},\mathsf{L},\widetilde{m})$, so

$$\dfrac{\dfrac{}{(\mathsf{Q},\mathsf{M},n,\pi_\varepsilon) \xrightarrow{n} \mathsf{C}}\ \mathtt{N}(n)=t \qquad \dfrac{(\mathsf{Q},\mathsf{M},nil) \xrightarrow{\pi_t} (\mathsf{P},\mathsf{L},\widetilde{m})}{\mathsf{C} \xrightarrow{\lambda} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)}}{(\mathsf{Q},\mathsf{M},n,\pi_\varepsilon) \xrightarrow{n} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)}$$

by rule COMPOSE, where $\mathsf{C} = (\mathsf{Q},\mathsf{M},nil,\pi_t)$.

Case COMPOSE. By the definition of the rule, there are v, w and E such that $(\mathsf{Q},\mathsf{M},n) \xrightarrow{v} \mathsf{E}$ and $\mathsf{E} \xrightarrow{w} (\mathsf{P},\mathsf{L},\widetilde{m})$, so for $\mathsf{E} = (\mathsf{Q}',\mathsf{M}',\widetilde{l})$ it must be $\widetilde{l} \neq nil$, hence $(\mathsf{Q},\mathsf{M},n) \xrightarrow{v} (\mathsf{Q}',\mathsf{M}',l)$. The induction hypothesis consequently yields

$$(\mathsf{Q},\mathsf{M},n,\pi_\varepsilon) \xrightarrow{v} (\mathsf{Q}',\mathsf{M}',l,\pi_\varepsilon)$$

and

$$(\mathsf{Q}',\mathsf{M}',l,\pi_\varepsilon) \xrightarrow{w} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon).$$

Now $(\mathsf{Q},\mathsf{M},n,\pi_\varepsilon) \xrightarrow{vw} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)$ follows by rule COMPOSE.

2. Induction on the length of w.

For $|w| = 0$ we have $w = \lambda$, so the only rule applicable is EXEC, but no reduction is possible as the forth component of C is π_ε.

For $w = 1$, the derivation of $\mathsf{C} \xrightarrow{w} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)$ must have the form

$$\text{COMPOSE}\,\dfrac{\text{COMPOSE}\,\dfrac{\text{LOAD}\,\dfrac{}{\mathsf{C} \xrightarrow{n} \mathsf{D}} \qquad \text{EXEC}\,\dfrac{}{\mathsf{D} \xrightarrow{\lambda} \mathsf{E}}}{\mathsf{C} \xrightarrow{n} \mathsf{E}} \qquad \dfrac{\vdots}{\mathsf{E} \xrightarrow{\lambda} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)}}{\mathsf{C} \xrightarrow{n} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)}$$

or

$$\mathrm{COMPOSE}\,\dfrac{\mathrm{LOAD}\,\dfrac{}{\mathsf{C} \xrightarrow{n} \mathsf{D}} \qquad \mathrm{COMPOSE}\,\dfrac{\mathrm{EXEC}\,\dfrac{}{\mathsf{D} \xrightarrow{\lambda} \mathsf{E}} \qquad \dfrac{\vdots}{\mathsf{E} \xrightarrow{\lambda} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)}}{\mathsf{D} \xrightarrow{\lambda} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)}}{\mathsf{C} \xrightarrow{n} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)}$$

because the only reduction possible from C is LOAD. By LOAD's definition, D is of the form $(\mathsf{Q},\mathsf{M},nil,\pi_t)$, and both derivation trees yield $(\mathsf{Q},\mathsf{M},nil,\pi_t) \xrightarrow{\lambda} (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)$. The instructions involved in this reduction are exactly $\mathtt{N}(n)$, and all single steps are applications of the rule EXEC. For each step, $\mathsf{D} = \mathsf{D}_0 \xrightarrow{\lambda} \mathsf{D}_1 \xrightarrow{\lambda} \ldots \xrightarrow{\lambda} \mathsf{D}_\mathsf{m} = \mathsf{E}$, there are by rule EXEC prefixes π_i such that for the ternary configurations D'_i corresponding to the configurations D_i the reductions $\mathsf{D}'_{\mathsf{i}-1} \xrightarrow{\pi_i} \mathsf{D}'_\mathsf{i}$ hold. By Proposition 7 an u_i with $\pi_{u_i} = \pi_i$ and $\mathsf{D}'_{\mathsf{i}-1} \xrightarrow{u_i} \mathsf{D}'_\mathsf{i}$, so $u_1 \ldots u_m$ is an interleaving for $\mathtt{N}(n)$ and A. As $\mathtt{N}(n)$ is safe for A due to the formation condition of Σ and the shape of D, $\mathsf{D}'_0 \xrightarrow{\mathtt{N}(n)} (\mathsf{P},\mathsf{L},\widetilde{m})$ holds, i.e. $(\mathsf{Q},\mathsf{M},nil) \xrightarrow{\mathtt{N}(n)} (\mathsf{P},\mathsf{L},\widetilde{m})$. By rule EXECUTE we obtain $(\mathsf{Q},\mathsf{M},n) \xrightarrow{n} (\mathsf{P},\mathsf{L},\widetilde{m})$.

For $|w| > 1$, we have $w = vu$ for some v and u, and there is an E such that $\mathsf{C} \xrightarrow{v} \mathsf{E}$ and $\mathsf{E} \xrightarrow{u} \mathsf{G}$ where $\mathsf{G} = (\mathsf{P},\mathsf{L},\widetilde{m},\pi_\varepsilon)$. For $\mathsf{E} = (\mathsf{R},\mathsf{N},\widetilde{l},\pi)$ we have two cases.

Case $\pi = \pi_\varepsilon$. Then $\widetilde{l} \neq nil$ must hold – otherwise no further reductions would be possible. The induction hypothesis yields

$$(\mathsf{Q},\mathsf{M},n) \xrightarrow{v} (\mathsf{R},\mathsf{N},l) \textit{ and } (\mathsf{R},\mathsf{N},l) \xrightarrow{u} (\mathsf{P},\mathsf{L},\widetilde{m})$$

so $(\mathsf{Q},\mathsf{M},n) \xrightarrow{vu} (\mathsf{P},\mathsf{L},\widetilde{m})$ holds by rule COMPOSE.

Case $\pi = \pi_1\pi_2 \neq \pi_\varepsilon$. Since COMPOSE is associative, we can assume that the left branch of $\mathsf{E} \xrightarrow{u} \mathsf{G}$ is either LOAD or EXEC. If it is a LOAD step, its index must be l, i.e.

$$\mathrm{COMPOSE}\,\dfrac{\mathrm{LOAD}\,\dfrac{}{\mathsf{E} \xrightarrow{l} \mathsf{F}} \qquad \mathrm{COMPOSE}\,\dfrac{\mathrm{X}\,\dfrac{}{\mathsf{F} \xrightarrow{u_1} \mathsf{D}} \qquad \dfrac{\vdots}{\mathsf{D} \xrightarrow{u_2} \mathsf{G}}}{\mathsf{F} \xrightarrow{u_1u_2} \mathsf{G}}}{\mathsf{E} \xrightarrow{u} \mathsf{G}}$$

and the step X cannot be a LOAD because $\mathsf{F} = (\mathsf{Q}',\mathsf{M}',nil,\pi\pi_{\mathtt{N}(l)})$ (again, step X is w.l.o.g. a single step). Hence, X must be EXEC, so $u_1 = \lambda$. If the reduction $\mathsf{F} \xrightarrow{\lambda} \mathsf{D}$ involves instructions from $\mathtt{N}(l)$, we can delay them because $<$ is contained in $\sqsubset$ and by Proposition 15 no race conditions exist in the composition $\mathtt{N}(n_1)\ldots\mathtt{N}(l)$ of the basic blocks $n_1,\ldots l$, since $n_1,\ldots l$ is a prefix of w and thus a path in $\mathtt{A}$. We can thus reorder and obtain $\mathsf{E} \xrightarrow{\lambda} \mathsf{H} \xrightarrow{l} \mathsf{D}$ for some H. Repeated application leads to H with empty issue unit, i.e. $\mathsf{H} = (\_,\_,l,\pi_\varepsilon)$. Now the first case applies.

$\square$

Again, for satisfying $[l] < [g]$ several natural serialisations can be used. In practice, analysis of data-dependencies along (simple) paths might be too costly. On the other hand, the condition on simple paths may be strengthened and thus be made more easy to verify. As pipeline dependencies can be analysed more efficiently, one can for example require that $FU(code_l) = FU(code_g)$ holds whenever $op : (k,l) \in \Gamma_n$ and $op : (g,h) \in \Gamma_m$ and $(n,m) \in \mathtt{A}^+$. Additionally, *control-dependencies* may be used for discharging constraints along one particular path: any conditional jump between l and g which is dependent on l guarantees that l will have retired when g is issued.

Consequently, typing $\Sigma \vdash \mathtt{P} : \square$ is sound for distributed execution. Provided that the constraints are met, an execution is either infinite or terminates in the same configuration as sequential execution (see Proposition 5).

4.5 Finite operand queues

Processor implementations of our computational models restrict operand queues to be of finite length. In contrast, the sequential and distributed dynamic semantics considered so far admitted queues with arbitrarily many entries. In this section, the dynamic semantics are modified such that for each operand queue a maximal number of elements is specified. No further execution of instructions is possible if all successor configurations violate the length restrictions. We modify the static semantics to determine length restrictions necessary for deadlock-free execution and examine the effect of relaxing length restrictions.

The dynamic semantics for execution under finite operand queues are obtained by replacing the rule WR by the rule WR-LIM(C)

$$\text{WR-LIM}(C)\frac{}{w \xrightarrow{(write,a,q)}_C wa}\ \exists X.q^{|wa|} \otimes X = C$$

where C is a $\otimes$-product. We write $\mathsf{C} \xrightarrow{t}_C \mathsf{D}$ and $\mathsf{C} \xrightarrow{\pi_t}_C \mathsf{D}$ if $shape(\mathsf{C}) \otimes X = C$ holds for some X and $\mathsf{C} \xrightarrow{t} \mathsf{D}$ and $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ can be derived using WR-LIM(C) instead of WR. Using structural induction one can show that $shape(\mathsf{D}) \otimes Y = C$ holds for some Y whenever $\mathsf{C} \xrightarrow{t}_C \mathsf{D}$ or $\mathsf{C} \xrightarrow{\pi_t}_C \mathsf{D}$.

Example. For $\mathsf{C} = ([\,], \mathsf{M})$, $\mathsf{D} = ([\mathtt{q}_1 \mapsto 3], \mathsf{M})$ and any M, program

$$t = [1]\mathtt{ldc}\ 4\ \mathtt{q}_1\ [2]\mathtt{dec}\ \mathtt{q}_1\ \mathtt{q}_1$$

fulfils $\mathsf{C} \xrightarrow{t}_{\mathtt{q}_1 \otimes \mathtt{q}_2} \mathsf{D}$ but not $\mathsf{C} \xrightarrow{t}_{\mathtt{q}_2} \mathsf{D}$. ◇

Sequential execution under finite operand queues is deterministic as $\mathsf{C} \xrightarrow{t}_C \mathsf{D}$ implies $\mathsf{C} \xrightarrow{t} \mathsf{D}$. Determinism is preserved when length limitations are relaxed, and distributed executability under identical limitations is guaranteed.

Proposition 17. *If* $\mathsf{C} \xrightarrow{t}_C \mathsf{D}$ *and X arbitrary then* $\mathsf{C} \xrightarrow{\pi_t}_C \mathsf{D}$ *and* $\mathsf{C} \xrightarrow{t}_{C \otimes X} \mathsf{D}$.

Proof. For $\mathsf{C} \xrightarrow{\pi_t}_C \mathsf{D}$ choose the interleaving t of π_t. For $\mathsf{C} \xrightarrow{t}_{C \otimes X} \mathsf{D}$ observe that $shape(\mathsf{E}) \otimes Y = C$ implies $shape(\mathsf{E}) \otimes Y \otimes X = C \otimes X$ and apply this to $\mathsf{E} = \mathsf{C}$ and $\mathsf{E} = \mathsf{D}$. □

For eliminating deadlock due to operand queue overflow, Figure 4.6 shows a modified type system for sequential execution. Judgements $\vdash_C t : A \multimap B$ capture operand queue lengths necessary for executing t in an initial configuration of shape A. In fact, $\vdash_C t : A \multimap B$ gives the tightest length limitations necessary for executing a program successfully.

Proposition 18. *Let* $\vdash_C t : A \multimap B$ *and* $shape(\mathsf{C}) = A$. *Then*

1. $A \otimes X = C = B \otimes Y$ *for some X and Y.*

2. $\mathsf{C} \xrightarrow{t}_C \mathsf{D}$ *for some* D.

$$\text{AX-LIM}\ \frac{}{\vdash_{X\otimes A_{code}\otimes V} [n]code : X\otimes A_{code} \multimap X\otimes B_{code}} \begin{cases} SC(code) \\ B_{code}/A_{code} = V/U \end{cases}$$

$$\text{CUT-LIM}\ \frac{\vdash_{C_1} s : A \multimap D \quad \vdash_{C_2} t : D \multimap B}{\vdash_{C_1\otimes V} st : A \multimap B}\ C_2/C_1 = V/U$$

Figure 4.6: Type system for finite operand queues

3. $\mathsf{C} \xrightarrow{t}_D \mathsf{E}$ *implies* $D = C\otimes W$ *for some* W.

Proof. All three parts are proven separately.

1. Induction on the rules for $\vdash_C t : A \multimap B$. In the rule for single instructions, $A = X\otimes A_{code}$ occurs explicitly in C and $B = X\otimes B_{code}$ occurs in C due to $C = X\otimes A_{code}\otimes V = X\otimes B_{code}\otimes U$. In the cut rule, we may assume $A\otimes X_1 = C_1 = D\otimes Y_1$ and $D\otimes X_2 = C_2 = B\otimes Y_2$ by induction hypothesis. Hence $C = C_1\otimes V = A\otimes X_1\otimes V$ and $C = C_1\otimes V = C_2\otimes U = B\otimes Y_2\otimes U$.

2. We first prove auxiliary claims regarding rules WR-LIM(C), RD, μ and CODE.

   RD **Claim 1:** If $w = av$ and $\gamma(q) = fu$ then there are unique u and b with $w \xrightarrow{(read(fu),b,q)}_C u$. Furthermore, $|u| = |w| - 1$ holds.

   Proof For $w = av$ and $\gamma(q) = fu$ both side conditions of rule RD are fulfilled and the claim holds for the unique $a = b$ and $u = v$.

   WR-LIM(C) **Claim 2:** For $C = q^k\otimes X$ with $prime(q,X)$, $|w| < k$ and arbitrary a there is a unique u such that $w \xrightarrow{(write,a,q)}_C u$. Furthermore, $|u| = |w| + 1$ holds.

   Proof Rule WR-LIM(C) carries exactly the assumptions as side conditions, so the claim follows for $u = wa$.

   μ There are two claims.

   Claim 3: If $shape(\mathsf{C}) = q\otimes X$, $\gamma(q) = fu$ and C arbitrary, then there are unique D and a with $\mathsf{C} \xrightarrow{(read(fu),a,q)}_C \mathsf{D}$. Furthermore, $shape(\mathsf{D}) = X$ holds.

   Proof For $\mathsf{C} = (\mathsf{Q},\mathsf{M})$ and $shape(\mathsf{C}) = q\otimes X$ we have $|\mathsf{Q}(q)| \geq 1$, so $\mathsf{Q}(q) = aw$ for some unique a and w. By claim 1, there are unique u and b with

$\mathsf{Q}(q) \xrightarrow{(read(fu),b,q)}_C u$, and $|u| = |\mathsf{Q}(q)| - 1$ holds. Hence, $\mathsf{C} \xrightarrow{(read(fu),a,q)}_C \mathsf{D}$ is derivable for $\mathsf{D} = (\mathsf{Q}[q \mapsto u], \mathsf{M})$, and uniqueness of D and a follow as in Proposition 1.

Furthermore, $|\mathsf{Q}[q \mapsto u](q)| = |u| = |\mathsf{Q}(q)| - 1$ holds and $|\mathsf{Q}[q \mapsto u](q')| = |\mathsf{Q}(q')|$ for $q' \neq q$, hence $shape(\mathsf{D}) \otimes q = shape(\mathsf{C}) = q \otimes X$ and therefore $shape(\mathsf{D}) = X$.

Claim 4: If $shape(\mathsf{C}) = A$, $A \otimes q \otimes X = C$ for some X and a arbitrary, then there is a unique D with $\mathsf{C} \xrightarrow{(write,a,q)}_C \mathsf{D}$. Furthermore, $shape(\mathsf{D}) = A \otimes q$ holds.

Proof The definition of $\otimes$ implies that for $A \otimes q \otimes X = C$, $C = q^k \otimes Y$, $A = q^l \otimes Z$ and $prime(q, Y \otimes Z)$, $l < k$ holds. Hence, for $\mathsf{C} = (\mathsf{Q}, \mathsf{M})$ with $shape(\mathsf{C}) = A$ we obtain $|\mathsf{Q}(q)| = l < k$. By claim 2, there is consequently a unique u with $\mathsf{Q}(q) \xrightarrow{(write,a,q)}_C u$, and $u = |\mathsf{Q}(q)| + 1$ holds. Hence, $\mathsf{C} \xrightarrow{(write,a,q)}_C \mathsf{D}$ is derivable for $\mathsf{D} = (\mathsf{Q}[q \mapsto u], \mathsf{M})$ and uniqueness of D follows as in Proposition 1.

Furthermore, $|\mathsf{Q}[q \mapsto u](q)| = |u| = |\mathsf{Q}(q)| + 1$ holds and $|\mathsf{Q}[q \mapsto u](q')| = |\mathsf{Q}(q')|$ for $q' \neq q$, hence $shape(\mathsf{D}) = shape(\mathsf{C}) \otimes q = A \otimes q$.

CODE **Claim 5:** If $\vdash_C [n]code : A \multimap B$ and $shape(\mathsf{C}) = A$ and then there is a unique D such that $\mathsf{C} \xrightarrow{code}_C \mathsf{D}$. Furthermore, $shape(\mathsf{D}) = B$ holds.

Proof We treat the representative case ADD explicitly.

Case $code = \texttt{add}\ op_1\ op_2\ op_3$**.** For $\vdash_C [n]\texttt{add}\ op_1\ op_2\ op_3 : A \multimap B$, the axiom AX-LIM and Table 3.1 guarantee

- $\gamma(op_1) = \gamma(op_2) = \texttt{ALU}$
- $A = op_1 \otimes op_2 \otimes Y$ and $B = op_3 \otimes Y$ for some Y
- $C = op_1 \otimes op_2 \otimes Y \otimes V$ where $op_3 / op_1 \otimes op_2 = V/U$.

Therefore, $Y \otimes op_3 \otimes U = C$. By the assumption that all operand identifiers op are operand queue identifiers q, there are q_1, q_2 and q_3 such that $q_i = op_i$ holds for $i \in \{1,2,3\}$. Therefore, $shape(\mathsf{C}) = A = q_1 \otimes q_2 \otimes Y$ and $\gamma(q_1) = \gamma(q_2) = \texttt{ALU}$, so by claim 3 there are unique E

and a with $\mathsf{C} \xrightarrow{(read(\mathtt{ALU}),a,q_1)}_C \mathsf{E}$, and $shape(\mathsf{E}) = q_2 \otimes Y$ holds. Applying claim 3 again yields unique F and b with $\mathsf{E} \xrightarrow{(read(\mathtt{ALU}),b,q_2)}_C \mathsf{F}$, and $shape(\mathsf{F}) = Y$ holds.

The above $Y \otimes op_3 \otimes U = C$ implies $shape(\mathsf{F}) \otimes q_3 \otimes U = C$, so by applying claim 4, there is a unique D with $\mathsf{F} \xrightarrow{(write,a+b,q_3)}_C \mathsf{D}$, and $shape(\mathsf{D}) = shape(\mathsf{F}) \otimes q_3$ holds. Consequently, the following derivation is possible

$$\text{ADD}\,\frac{\mathsf{C} \xrightarrow{(read(\mathtt{ALU}),a,q_1)}_C \mathsf{E} \quad \mathsf{E} \xrightarrow{(read(\mathtt{ALU}),b,q_2)}_C \mathsf{F} \quad \mathsf{F} \xrightarrow{(write,a+b,q_3)}_C \mathsf{D}}{\mathsf{C} \xrightarrow{\mathtt{add}\ q_1\ q_2\ q_3}_C \mathsf{D}}$$

Using $q_i = op_i$ for $i \in \{1,2,3\}$ we thus obtain a derivation for $\mathsf{C} \xrightarrow{code}_C \mathsf{D}$, and $shape(\mathsf{D}) = shape(\mathsf{F}) \otimes q_3 = Y \otimes op_3 = B$ holds. Uniqueness of D with respect to $\mathsf{C} \xrightarrow{code}_C \mathsf{D}$ follows as in Proposition 1.

Cases $code \neq \mathtt{add}\ op_1\ op_2\ op_3$. Similar.

For proving the main claim, we perform a structural induction on t.

If $t = \varepsilon$, no typing judgement $\vdash_C t : A \multimap B$ is derivable, so the claim is trivially fulfilled.

If $t = ins$, then claim 5 guarantees that there is a unique D with $\mathsf{C} \xrightarrow{code}_C \mathsf{D}$, where $ins = [n]code$. By rule INSTR we obtain $\mathsf{C} \xrightarrow{ins}_C \mathsf{D}$. Furthermore, claim 5 guarantees $shape(\mathsf{D}) = B$, and uniqueness of D with respect to $\mathsf{C} \xrightarrow{ins}_C \mathsf{D}$ follows from Proposition 1.

If $t = s\,r$, then the typing rule CUT-LIM guarantees that there are D, C_1 and C_2 with $\vdash_{C_1} s : A \multimap D$, $\vdash_{C_2} r : D \multimap B$ and $C = C_1 \otimes V$, where $C_2/C_1 = V/U$. A simple structural induction shows that $\vdash_{C_1 \otimes V} s : A \multimap D$ and $\vdash_{C_2 \otimes U} r : D \multimap B$ are derivable, i.e. $\vdash_C s : A \multimap D$ and $\vdash_C r : D \multimap B$. As $shape(\mathsf{C}) = A$ holds, we can thus apply the induction hypothesis to s to obtain a unique E with $\mathsf{C} \xrightarrow{s}_C \mathsf{E}$ and $shape(\mathsf{E}) = D$. Applying the induction hypothesis again yields the existence of a D with $\mathsf{E} \xrightarrow{r}_C \mathsf{D}$, and $shape(\mathsf{D}) = B$ hold. Consequently, $\mathsf{C} \xrightarrow{s\,r}_C \mathsf{D}$. Uniqueness of D with respect to $\mathsf{C} \xrightarrow{s\,r}_C \mathsf{D}$ follows from Proposition 1.

3. Again, the claim follows by induction on t.

If $t = \varepsilon$**,** no judgement $\mathsf{C} \xrightarrow{t}_D \mathsf{D}$ is derivable, so the claim is trivially fulfilled.

If $t = ins = [n]code$**,** the rule AX-LIM implies that there is a (unique) X with $A = X \otimes A_{code}$ and $B = X \otimes B_{code}$, and that for $B_{code}/A_{code} = V/U$, $C = A \otimes V$ holds. The assumption $\mathsf{C} \xrightarrow{[n]code}_D \mathsf{E}$ yields $\mathsf{C} \xrightarrow{[n]code} \mathsf{E}$, so for $shape(\mathsf{C}) = A$, $shape(\mathsf{E}) = B$ follows, and the definition of $\rightarrow_D$ guarantees the existence of some Y and Z with

$$\begin{aligned} X \otimes A_{code} \otimes Y &= A \otimes Y = shape(\mathsf{C}) \otimes Y = D \\ &= shape(\mathsf{E}) \otimes Z = B \otimes Z = X \otimes B_{code} \otimes Z \end{aligned}$$

Consequently, $A_{code} \otimes Y = B_{code} \otimes Z$ follows, so the minimality of V and U implies that there is a W such that

$$Y = V \otimes W \text{ and } Z = U \otimes W,$$

from which the claim follows due to $D = A \otimes Y = A \otimes V \otimes W = C \otimes W$.

If $t = s\,r$**,** then the last rule in the derivation for $\mathsf{C} \xrightarrow{t}_D \mathsf{E}$ must have been COMP, so there is a unique F with $\mathsf{C} \xrightarrow{s}_D \mathsf{F}$ and $\mathsf{F} \xrightarrow{r}_D \mathsf{E}$. Furthermore, for $\vdash_C s\,r : A \multimap B$ to hold there must by rule CUT-LIM be C_1, C_2 and E with

$$\vdash_{C_1} s : A \multimap E,\ \vdash_{C_2} t : E \multimap B \text{ and } C = C_1 \otimes V = C_2 \otimes U$$

where $C_2/C_1 = V/U$, and it can easily be seen that the entities C_1, C_2 and E are uniquely determined. By induction hypothesis there are thus W_1 and W_2 such that

$$C_1 \otimes W_1 = D = C_2 \otimes W_2.$$

Since V and U are prime and fulfil $C_1 \otimes V = C_2 \otimes U$, there must hence be a W such that $W_1 = V \otimes W$ and $W_2 = U \otimes W$, and for this W we obtain $D = C_1 \otimes W_1 = C_1 \otimes V \otimes W = C \otimes X$.

$\square$

4.5.1 Distributed execution

For distributed execution, a deadlock occurs if progress in all pipelines is blocked, due either to a lack of operands or to an overflow in a queue used as the destination for a forwarding operation. Such a situation occurs if an interleaving of t enters a configuration E but is unable to reach a final configuration from E under the same restrictions C.

Definition 15. *A program t is* deadlock-free *for the pair (A, C) if $A \otimes X = C$ for some X and for all C, E, π_1 and π_2 with shape$(\mathsf{C}) = A$, $\pi_1\pi_2 = \pi_t$ and $\mathsf{C} \xrightarrow{\pi_1}_C \mathsf{E}$ there is a D such that $\mathsf{E} \xrightarrow{\pi_2}_C \mathsf{D}$.*

Absence of deadlock is a priori not preserved when operand queues are extended.

Example. For $\gamma(\mathsf{q}_2) = \gamma(\mathsf{q}_3) = \mathtt{MUL}$ and $\gamma(\mathsf{q}_1) = \mathtt{ALU}$ consider the program t in (4.9)

$$\begin{array}{ll} [1]\mathtt{ldc}\ 3\ \mathsf{q}_1 & [5]\mathtt{dec}\ \mathsf{q}_1\ \mathsf{q}_3 \\ [2]\mathtt{dec}\ \mathsf{q}_1\ \mathsf{q}_2 & [6]\mathtt{mul}\ \mathsf{q}_3\ \mathsf{q}_2\ \mathsf{q}_1 \\ [3]\mathtt{ldc}\ 4\ \mathsf{q}_1 & [7]\mathtt{ldc}\ 5\ \mathsf{q}_2 \\ [4]\mathtt{ldc}\ 3\ \mathsf{q}_2 & \end{array} \tag{4.9}$$

The distributed program π_t is $[2][5] \parallel [6] \parallel [1][3][4][7]$. For $C = \mathsf{q}_1 \otimes \mathsf{q}_2 \otimes \mathsf{q}_2 \otimes \mathsf{q}_3$ we obtain $\vdash_C t : \mathbf{1} \multimap \mathsf{q}_1 \otimes \mathsf{q}_2 \otimes \mathsf{q}_2$, and for $\mathsf{C} = ([\,], \mathsf{M})$ and $\mathsf{D} = ([\mathsf{q}_1 \mapsto [6], \mathsf{q}_2 \mapsto [3,5]], \mathsf{M})$, $\mathsf{C} \xrightarrow{t}_C \mathsf{D}$ follows. Indeed, t is deadlock-free for (A, C), as shown in Figure 4.7 (left), where the interleavings starting in C under restrictions C are depicted. The states are given as triples (i, j, k) such that ${\mathsf{q}_1}^i \otimes {\mathsf{q}_2}^j \otimes {\mathsf{q}_3}^k$ is the shape of the corresponding configuration. States violating the length restrictions are marked, and execution paths are cut off at these points. Absence of deadlock can be detected by observing that each non-marked configuration has at least one successor which is also non-marked. The in-order execution is seen to obey restrictions C as the interleaving $[1] \ldots [7]$ is a successful path.

Relaxing length restrictions to $C \otimes X$ with $X = \mathsf{q}_1$ enables more interleavings – see the graph in Figure 4.7 (right). Again, states violating the (extended) length restrictions are marked. Some new interleavings are harmless, such as any interleaving starting with $[1][3][2]$. However, the configuration reached by the prefix $[1][3][4][7]$ is a deadlock -

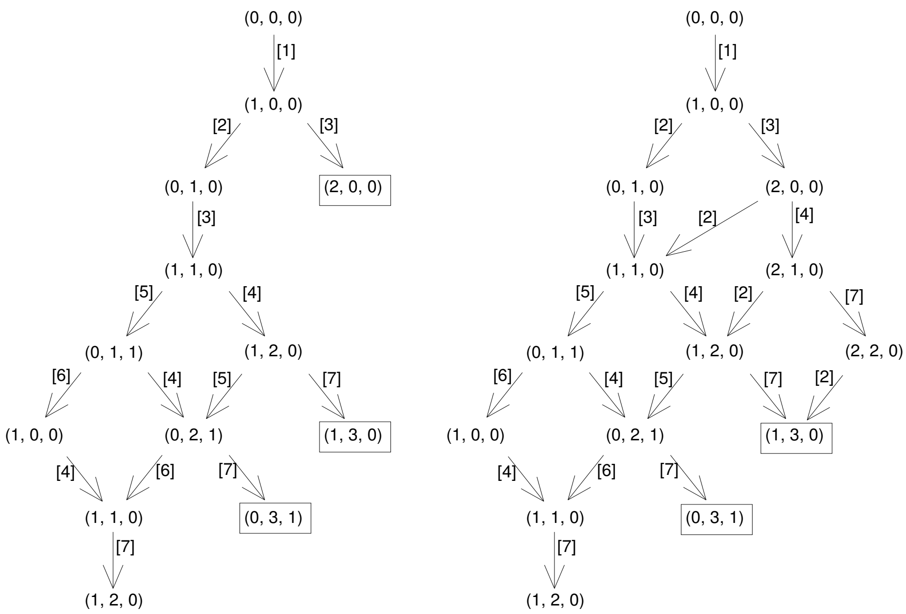


Figure 4.7: Interleavings of program (4.9) for various length restrictions.

its only successor violates the extended length restrictions. This path could have been completed, had we extended C by $\mathsf{q}_1 \otimes \mathsf{q}_2$ instead of q_1. $\diamond$

Failure of this monotonicity property is practically relevant. Extending some operand queues supports the exploitation of the instruction-level parallelism inherent in a program. The performance gain should not be paid for by new deadlocks in programs which were previously deadlock-free. Of course, reexamination of programs should also be avoided. For programs without race hazards this is possible – and the above example program is indeed non-deterministic for shape **1**.

Proposition 19. *Let* $(\Gamma, <) \vdash t : A \multimap B$, $<$ *be contained in* $\sqsubset$ *and* t *sequential for* $\sqsubset$ *and* A*. If* t *is deadlock-free for* (A, C) *then* t *is deadlock-free for* $(A, C \otimes X)$.

Proof. Let $shape(\mathsf{C}) = A$, $\pi_1\pi_2 = \pi_t$ where both π_i are non-empty and $\mathsf{C} \xrightarrow{\pi_1}_{C\otimes X} \mathsf{E}$. We have to show that $\mathsf{E} \xrightarrow{\pi_2}_{C\otimes X} \mathsf{D}$ holds for some D.
Let n be the length of t and u be the interleaving of π_1, i.e. $\mathsf{C} = \mathsf{C}_0 \xrightarrow{u_1}_{C\otimes X} \mathsf{C}_1 \xrightarrow{u_2}_{C\otimes X} \ldots \xrightarrow{u_l}_{C\otimes X} \mathsf{C}_l = \mathsf{E}$ for some $l \geq 0$. Let $0 \leq m \leq l$ be smallest such that there is no X_m

with $shape(\mathsf{C_m}) \otimes X_m = C$,

$$\mathsf{C}_0 \xrightarrow{u_1}_C \mathsf{C}_1 \xrightarrow{u_2}_C \dots \xrightarrow{u_{m-1}}_C \mathsf{C}_{m-1} \xrightarrow{u_m}_{C\otimes X} \mathsf{C}_m \xrightarrow{u_{m+1}}_{C\otimes X} \dots \xrightarrow{u_l}_{C\otimes X} \mathsf{C}_l$$

and suppose there are no w and C_{l+1} with $\mathsf{C}_l \xrightarrow{w}_{C\otimes X} \mathsf{C}_{l+1}$.
Since t is deadlock-free for (A,C) there is a Y such that $shape(\mathsf{C}) \otimes Y = A \otimes Y = C$, hence $m > 0$ must hold. Furthermore, absence of deadlock for (A,C) implies that there is an interleaving v of $\pi_{u_m \dots u_l}\pi_2$ such that $\mathsf{C}_{m-1} \xrightarrow{v}_C \mathsf{D}$. Let $v = v_1 \dots v_k$.

1. If v_1 occurs in $u_m \dots u_l$, say $v_1 = u_i$, then instructions u_m to u_{i-1} are completely independent from v_1, i.e. they queue in different instruction queues and forward to different operand queues. This follows from the sequentiality of t for $\sqsubset$ and A: since v_1 is executable in C_{m-1} it must be at the head of its instruction queue. Since v_1 is not in $u_m \dots u_{i-1}$, it is still at the head position in C_{i-1}. If an instruction within $u_m \dots u_{i-1}$ forwarded to the same queue as v_1 we had a race hazard in the unrestricted model, in contradiction to Proposition 11. Consequently, we can reorder the instructions and obtain

$$\mathsf{C}_{m-1} \xrightarrow{v_1}_C \mathsf{D}_1 \xrightarrow{u_m}_{C\otimes X} \dots \xrightarrow{u_{i-1}}_{C\otimes X} \mathsf{D}_{i-1} \xrightarrow{u_{i+1}}_{C\otimes X} \dots \xrightarrow{u_l}_{C\otimes X} \mathsf{D}_k = \mathsf{C}_l$$

   We have thus decreased the distance between the last configuration of shape C and C_l by one instruction, hence by induction on $l-m$ we obtain $\mathsf{C}_0 \xrightarrow{\pi_u}_C \mathsf{C}_l$. Consequently $\mathsf{C}_l \xrightarrow{v}_C \mathsf{D}$ and $\pi_v = \pi_2$ follow, hence $\mathsf{C}_l \xrightarrow{v}_{C\otimes X} \mathsf{D}$ and $\mathsf{C}_l \xrightarrow{\pi_2}_{C\otimes X} \mathsf{D}$ by Proposition 17.

2. If v_1 does not occur in $u_m \dots u_l$ then it must still be at the head of its instruction queue in configuration C_l. Since all its operands were available in C_{m-1}, they are still available in C_l (no intermediate instruction can have removed them because they queue in different queues) and no instruction in $u_m \dots u_l$ can have forwarded to a queue to which v_1 forwards (otherwise we had a race condition). Consequently, we have $\mathsf{C}_l \xrightarrow{v_1}_{C\otimes X} \mathsf{D}_1$ for some D_1, in contradiction to the assumption that no w and C_{l+1} exist with $\mathsf{C}_l \xrightarrow{w}_{C\otimes X} \mathsf{C}_{l+1}$. Consequently, the case $v_1 \notin \{u_m \dots u_l\}$ does not occur.

□

The calculi $\vdash_C t : A \multimap B$ and $(\Gamma, <) \vdash t : A \multimap B$ can be combined to a calculus for sequential and deterministic distributed execution with finite queues where judgements are of the form $(\Gamma, <) \vdash_C t : A \multimap B$. For this calculus, we obtain the following result.

Theorem 3. *Let $(\Gamma, <) \vdash_C t : A \multimap B$ and t sequential for $\sqsubset$ and A where $<$ is contained in $\sqsubset$. Let $shape(\mathsf{C}) = A$. Then*

1. $\mathsf{C} \xrightarrow{t}_{C \otimes X} \mathsf{D}$ *for some unique* D *and any* X.
2. *t is deterministic for A and deadlock-free with respect to $(A, C \otimes X)$ for any X.*
3. $\mathsf{C} \xrightarrow{\pi_t}_{C \otimes X} \mathsf{D}$ *for any* X.

Proof.

1. Combine Propositions 18 and 17 to obtain $\mathsf{C} \xrightarrow{t}_{C \otimes X} \mathsf{D}$ for some D and any X. Uniqueness of D follows from Proposition 1.
2. Given Propositions 11 and 19, it remains to show that t is deadlock-free with respect to (A, C).

   Let $\pi_t = \pi_1 \pi_2$ and $\mathsf{C} \xrightarrow{\pi_1}_C \mathsf{C}_l$ such that $u = u_1 \ldots u_l$ is the interleaving of π_1 and

   $$\mathsf{C} \xrightarrow{u_1}_C \mathsf{C}_1 \xrightarrow{u_2}_C \ldots \xrightarrow{u_l}_C \mathsf{C}_l.$$

   Furthermore, part (1) implies

   $$\mathsf{C} \xrightarrow{t_1}_C \mathsf{D}_1 \xrightarrow{t_2}_C \ldots \xrightarrow{t_n}_C \mathsf{D}$$

   for some D_i and $t = t_1 \ldots t_n$. Let k be smallest such that $t_k \notin \{u_1, \ldots, u_l\}$. Then $\mathsf{C}_l \xrightarrow{t_k}_C \mathsf{E}$ for some E because

   (a) t_k is at the head of its instruction queue: since $t_1, \ldots, t_{k-1}$ are in $\{u_1, \ldots, u_l\}$ there is an $i \leq l$ such that t_k is at the head of its instruction queue in C_i. Since distributed execution respects $\lessdot$, t_k is still at the head of its instruction queue in C_l.

   (b) all operands for t_k are available in C_l: they are available in C_i, and no instruction in $u_{i+1}, \ldots, u_l$ consumed them.

(c) there is a Y such that $shape(\mathsf{E}) \otimes Y = C$: configuration D_k fits into C and no instruction in $u_{i+1}, \ldots, u_l$ has entered items into a queue q to which t_k writes (otherwise we had a race condition and t could not be sequential for $\sqsubset$ and A). Also, instructions $u_1, \ldots, u_i$ have only entered as many items into q as $t_1 \ldots t_{k-1}$ (again because t is sequential for $\sqsubset$ and A), and $u_1, \ldots, u_l$ removed at least as many items from q as $t_1, \ldots, t_{k-1}$ since t_k is the first instruction not in $u_1, \ldots, u_l$.

By induction on l the claim follows.

3. Combine part (1) and Proposition 17.

□

4.6 AQM's and Tomasulo's algorithm

In order to demonstrate how AQM's with distributed execution relate to hardware implementations of forwarding we briefly discuss Tomasulo's algorithm [Tom67] [HP96]. As the material presented here will not be needed in later chapters, a reader who is more interested in the technical development may skip this section and proceed directly to the discussion (Section 4.7).

4.6.1 Tomasulo's algorithm

In Tomasulo's algorithm (see Figure 4.8), functional units are equipped with reservation stations. At issue time, instructions are assigned a symbolic tag and sent to a reservation station associated to a suitable functional unit. If an operand is not available at this time, the corresponding operand field is filled with the tag of the instruction producing the operand. Instructions inside the reservation stations whose operand fields contain valid operands may execute. The functional units attach the instruction's tags to the results and then arbitrate for access to a common data bus (CDB). The CDB delivers the result to the register bank, possibly mediated by a reorder buffer. Results are also made available to instructions inside the reservation stations. These match the tags of outstanding operands against the tag of a result provided by the CDB and substitute

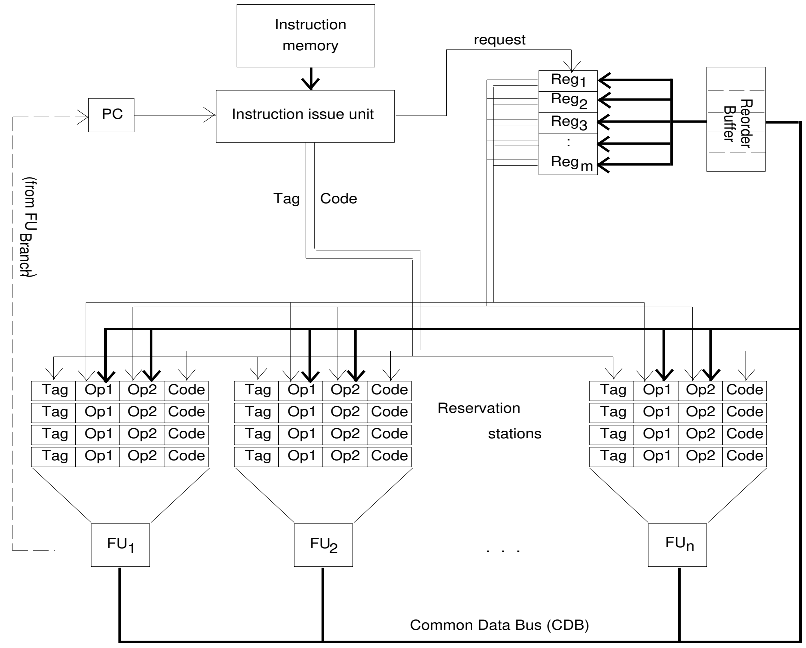


Figure 4.8: Tomasulo's algorithm

the result into the operand field whenever the match succeeds. Correctness of execution under Tomasulo's algorithm relies on the allocation of tags. In most implementations, tags represent either the index of the reservation station holding the instruction which will produce the result [HP96], or its index in the reorder buffer [KMP99]. Due to the complexity of out-of-order operation in the functional units, concurrent tag matching in reservation stations and tag allocation in the issue unit, Tomasulo's algorithm has been a popular case study for formal hardware verification [HGS99] [McM98] [SH98] [AP99] [McM00] [JM01].

Example. We consider an architecture using [HP96]'s interpretation of tags, but without a reorder buffer. For the code sequence

$$\begin{array}{ll} [1]\mathtt{ld}\ \mathtt{r}_2\ \mathtt{r}_{16} & [4]\mathtt{sub}\ \mathtt{r}_{16}\ \mathtt{r}_3\ \mathtt{r}_{18} \\ [2]\mathtt{ld}\ \mathtt{r}_3\ \mathtt{r}_{12} & [5]\mathtt{div}\ \mathtt{r}_{14}\ \mathtt{r}_{10}\ \mathtt{r}_{20} \\ [3]\mathtt{mul}\ \mathtt{r}_{12}\ \mathtt{r}_{14}\ \mathtt{r}_{10} & [6]\mathtt{add}\ \mathtt{r}_{18}\ \mathtt{r}_{10}\ \mathtt{r}_{16} \end{array} \tag{4.10}$$

Figure 4.9 shows a possible situation after issuing instruction $[1]$. Reservation station `Mem1` contains $[1]$'s code and the operand obtained from register $\mathtt{r}_2$. Also, the tag

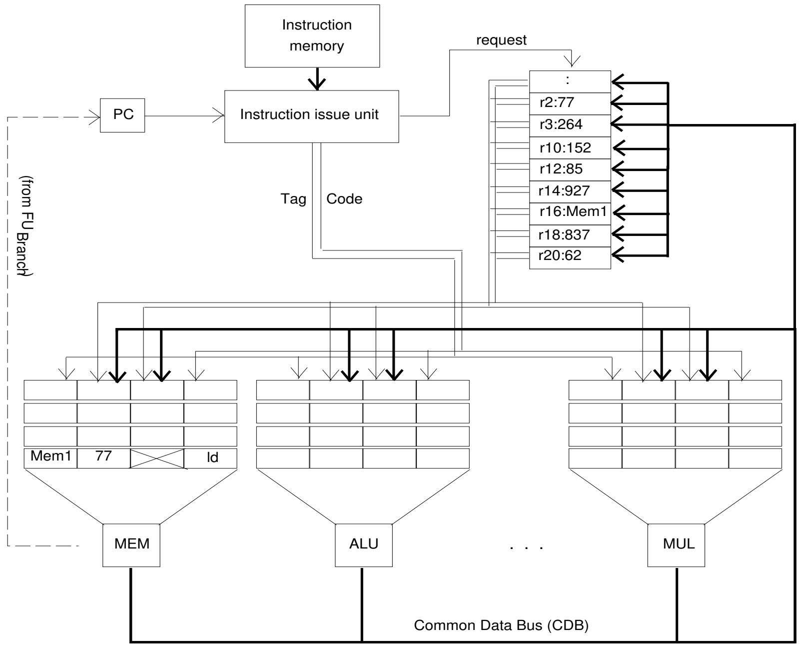


Figure 4.9: Snapshot after issuing instruction [1].

Mem1 has been written to the target register of [1], r_{16}. Thus, when instructions [2] to [5] are issued (Figure 4.10), any read access to r_{16} yields the tag Mem1 instead of any former value. For example, when instruction [4] is allocated to reservation station Alu1, its first operand field is set to Mem1. Notice that reservation stations for a particular functional unit are not allocated in any specific order. Finally, when instruction [6] is issued, the content of r_{16} is overwritten – see Figure 4.11 where [6] is allocated to Alu3. Any read access from r_{16} by instructions [7] and later will yield the tag Alu3.

When an instruction is executed, its result is written into the target register containing the producer's reservation station index: if instruction [1] executes and retires prior to the issuing of [6], its result will overwrite the tag Mem1 - further instructions reading from r_{16} would thus obtain the operand directly. If instruction [1] retires after instruction [6] has issued, no register contains the tag Mem1 any longer and the result of [1] will not be written back. Independently of instruction [6]'s issue status at the time of [1]'s retirement, the result of [1] will be substituted for the placeholder tag in the first

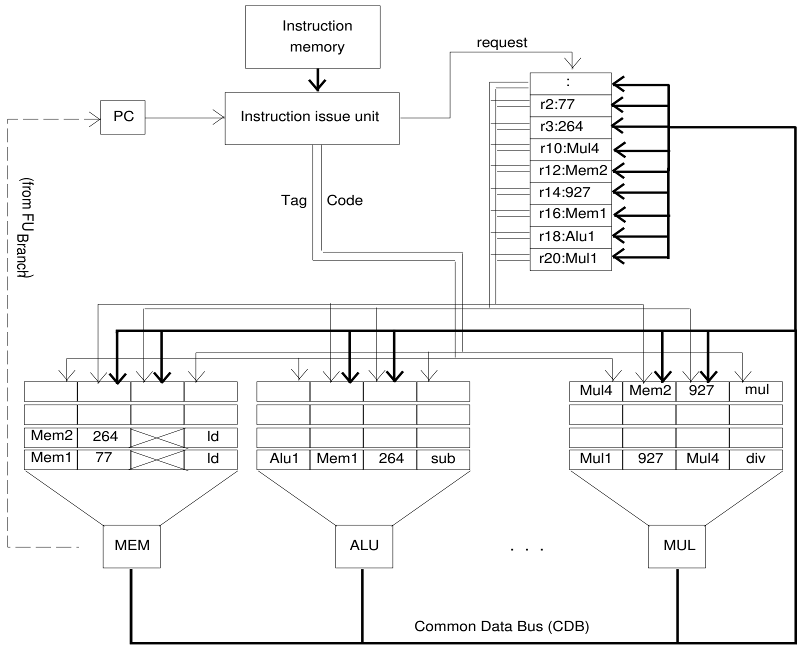


Figure 4.10: Snapshot after issuing instructions [1] to [5].

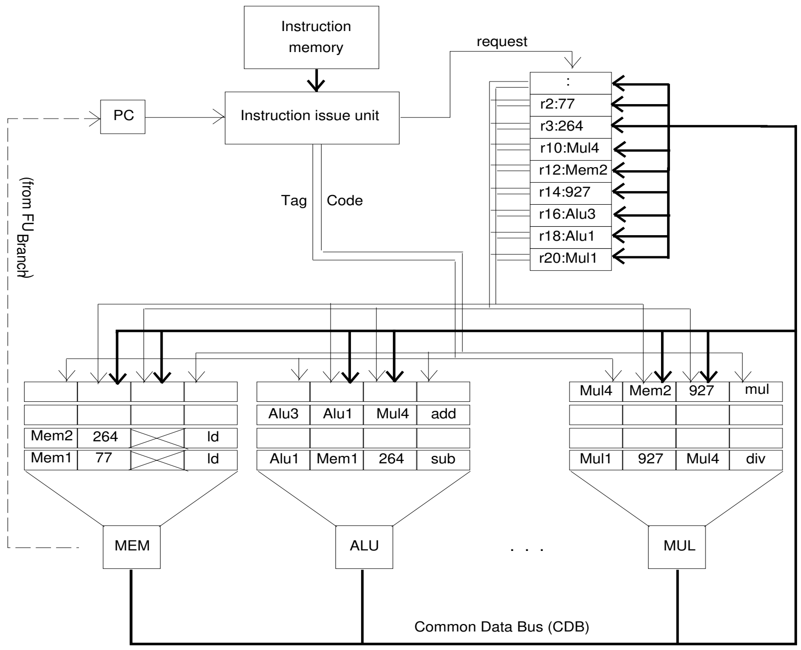


Figure 4.11: Snapshot after issuing all instructions of program (4.10).

operand of instruction [4] in reservation station `Alu1`. At the same time, the reservation station `Mem1` will be freed, again triggered by the availability of the current entry's result on the CDB. ◇

4.6.2 Mapping to AQM's

For mapping Tomasulo's architecture to the AQM model, we replace the unspecific broadcasting of results via the CDB by directed forwarding to the consuming instructions. As forwarding in AQM's is initiated by the producer of the result and no assumptions are made regarding execution delays inside functional units, the order in which operands arrive at a particular reservation station is undetermined. In contrast to the concurrent matching of result tags against placeholder tags in reservation station entries, correct behaviour is achieved by having several operand queues delivering values to each functional unit (see Figure 4.1). Verifying the allocation of operand queues to operands is thus equally important as verifying the allocation of tags in Tomasulo's algorithm, but is in ALEF under the control of the compiler.

Example. Suppose an architecture with $\gamma(\mathtt{q}_1) = \mathtt{ALU}$ and $\gamma(\mathtt{q}_2) = \mathtt{MUL}$ is given, and an initial configuration of shape **1**. For program (4.10), registers $\mathtt{r}_{16}$ and $\mathtt{r}_{18}$ may both be replaced by forwarding to $\mathtt{q}_1$ as no race condition can develop. If additionally, the usage of $\mathtt{r}_{12}$ is replaced by forwarding to $\mathtt{q}_2$ we obtain

$$
\begin{array}{ll}
[1]\mathtt{ld}\ \mathtt{r}_2\ \mathtt{q}_1 & [4]\mathtt{sub}\ \mathtt{q}_1\ \mathtt{r}_3\ \mathtt{q}_1 \\
[2]\mathtt{ld}\ \mathtt{r}_3\ \mathtt{q}_2 & [5]\mathtt{div}\ \mathtt{r}_{14}\ \mathtt{r}_{10}\ \mathtt{r}_{20} \\
[3]\mathtt{mul}\ \mathtt{q}_2\ \mathtt{r}_{14}\ \mathtt{r}_{10} & [6]\mathtt{add}\ \mathtt{q}_1\ \mathtt{r}_{10}\ \mathtt{r}_{16}
\end{array}
$$

◇

4.6.3 Limitations

While the general AQM model is powerful enough to simulate Tomasulo's algorithm, the restrictions of ALEF make the translation cumbersome.

Firstly, in ALEF a result can only be forwarded to a small (and fixed) number of operand queues. Hence, the effect of the broadcast to substitute a result for many tags at the same time can only be matched non-atomically, using duplication instructions.

Example. As the result of instruction [3] is consumed by two instructions, $\mathtt{r}_{10}$ can only be eliminated at the cost of a duplication instruction.

$$\begin{array}{ll} [1]\mathtt{ld}\ \mathtt{r}_2\ \mathtt{q}_1 & \\ [2]\mathtt{ld}\ \mathtt{r}_3\ \mathtt{q}_2 & [4]\mathtt{sub}\ \mathtt{q}_1\ \mathtt{r}_3\ \mathtt{q}_1 \\ [3]\mathtt{mul}\ \mathtt{q}_2\ \mathtt{r}_{14}\ \mathtt{q}_2 & [5]\mathtt{div}\ \mathtt{r}_{14}\ \mathtt{q}_2\ \mathtt{r}_{20} \\ [3']\mathtt{dupl}^{\mathtt{MUL}}\ \mathtt{q}_2\ \mathtt{q}_2\ \mathtt{q}_3 & [6]\mathtt{add}\ \mathtt{q}_1\ \mathtt{q}_3\ \mathtt{r}_{16} \end{array}$$

Provided that we chose the duplication on MUL, the forwarding from [3] to [3′] as well as the forwarding from [3′] to [5] may reuse $\mathtt{q}_2$. For the forwarding to instruction [6], a race condition with the result of [4] is avoided by introducing a new operand queue $\mathtt{q}_3$ with $\gamma(\mathtt{q}_3) = \mathtt{ALU}$. The remaining registers lack either instructions writing to them (registers $\mathtt{r}_2$, $\mathtt{r}_3$ and $\mathtt{r}_{14}$) or instructions reading from them (registers $\mathtt{r}_{16}$ and $\mathtt{r}_{20}$) and can thus not be eliminated – see the following chapter for a treatment of programs which use operand queues and registers. ◇

A similar situation occurs if a result in Tomasulo's algorithm updates a register but is also consumed by a reservation station entry. Again, a duplication instruction is needed in ALEF for achieving a similar effect.

Another aspect related to duplication is [Tom67]'s variation where some instructions effectively execute in zero time. For example, consider the instruction sequence

$$[1]\mathtt{add}\ \mathtt{r}_1\ \mathtt{r}_2\ \mathtt{r}_3\ [2]\mathtt{id}^{\mathtt{MUL}}\ \mathtt{r}_3\ \mathtt{r}_4,$$

for which issuing under the standard scheme may lead to the situation shown in Figure 4.12 on the left. Under Tomasulo's variation, the issue unit would not insert instruction [2] into a reservation station but instead copy the tag found in $\mathtt{r}_3$ into $\mathtt{r}_4$ – see Figure 4.12 on the right. Consequently, the execution of instruction [1] would automatically update both registers and thus achieve the same effect as the original program. In ALEF, this feature cannot be simulated, but future extensions of ALEF might well

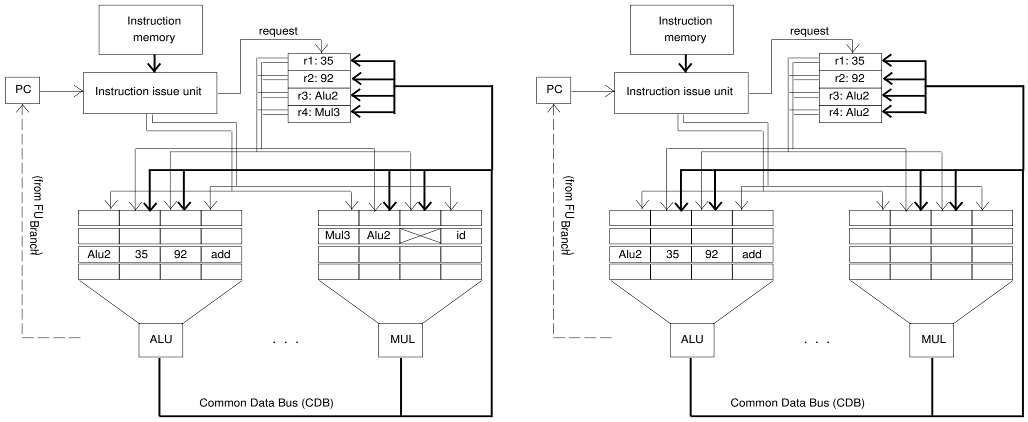


Figure 4.12: Example for instruction execution in zero time.

combine duplication with ordinary instructions. This would not only allow the elimination of the identity instruction [2] but also lead to a more efficient simulation of broadcasting to several locations.

Secondly, Tomasulo's algorithm and AQM's as treated in this chapter differ in the relative order of instructions which are mapped to the same functional unit. In contrast to the in-order execution resulting from AQM's instruction *queues*, reservation stations may release instructions in an arbitrary order. This difference represents a particularity of the operational semantics rather than a conceptual limitation of the AQM model. In fact, by defining alternative operational semantics for ALEF the consequences of such architectural changes may easily be studied.

Thirdly, modern micro-processors combine Tomasulo's algorithm with mechanisms for exception handling and speculative execution, often relying on the reorder buffer which rearranges the instructions into program order before retirement. It is beyond the scope of this thesis to study the interaction between forwarding and these techniques, and we have not evaluated possibilities for extending AQM's by a reorder buffer. Notice, however, that extended opcodes such as `addp` $op_1\ op_2\ op_3\ p : op_1 \otimes op_2 \multimap op_3 \otimes p$ may in principle signal the completion of instructions to data structures similar to the reorder buffer.

On the other hand, Tomasulo's synchronous broadcasting and the CDB are not expected to scale well as the number of functional units and the size of reservation stations grow. In contrast, forwarding paths in AQM's may be implemented distributedly where the availability of individual connections is indicated in architecture descriptions. We will briefly discuss in Chapter 9 how type systems could be made parametric in abstract descriptions of processor configurations.

4.6.4 Conclusion

The preceding discussion indicates that AQM's are powerful enough to simulate Tomasulo's algorithm. However, several features of super-scalar processors remain unsupported at the time of writing. We hope that the methodology proposed in this thesis will be helpful for overcoming these deficiencies. As the outlined mapping provides a novel explanation of Tomasulo's algorithm which complements existing informal expositions as well as formal verifications of individual implementations, AQM's may also be useful for studying and comparing future variations of Tomasulo's architecture.

4.7 Discussion

This chapter considered program execution in processors with independent instruction-pipelines and operand queues of infinite and finite length. We modified the dynamic semantics of ALEF appropriately and updated the static semantics for guaranteeing desirable properties. In the distributed model, we were concerned with determinism of execution. We showed that the programming language approach allows one to tackle non-determinism using a variety of means, based on the relationship between determinism, safety and completeness. In the model for finite queues, we ensured absence of queue overflow for sequential and distributed execution. In the latter case, we observed that extending queues is only safe for deterministic programs. We believe it would have been difficult to guarantee robustness of program execution with respect to this architectural change using a purely hardware-based verification approach. Finally, the comparison to Tomasulo's algorithm demonstrated that AQM's can simulate existent hardware implementations of forwarding and highlight some of their technical aspects.

We briefly comment on two design decisions, and how the corresponding limitations may be overcome.

First, we considered interleaving at the instruction level as each instruction was executed atomically. An obvious dynamic semantics for more fine-grained interleaving is an an SOS for the underlying micro-instructions *read* and *write*. For sequential execution, this merely amounts to a switch of granularity in the small-step semantics. For distributed execution, changing the level of granularity exposes more asynchrony as *read*- and *write*-operations of different instructions may interleave. In particular, non-atomic execution of a `dupl` q_1 q_2 q_3 instruction allows an instruction which is data-dependent on the value sent through q_2 to start (and finish) execution before the forwarding to q_3 has been performed. Consequently, a new type of race conditions is observed, like in program

$$[\mathtt{1}]\mathtt{dupl}^{fu}\ \mathtt{q}_1\ \mathtt{q}_2\ \mathtt{q}_3\ [\mathtt{2}]\mathtt{dec}\ \mathtt{q}_2\ \mathtt{q}_3.$$

Although a race occurs for writing to $\mathtt{q}_3$, the program is admitted by our system. The typing calculus correctly discovers the race condition and inserts the constraint $[\mathtt{1}] < [\mathtt{2}]$. For $fu \neq \mathtt{ALU}$, this constraint cannot be eliminated using a pipeline-dependency, but for initial configurations of shape $\mathtt{q}_1$ a data-dependency exists. A compiler using data-dependencies for discharging constraints will thus (falsely) discharge $[\mathtt{1}] < [\mathtt{2}]$.

Hardware-based solutions for making `dupl` appear atomic may be difficult to implement under an asynchronous regime. However, a small modification of the definition of data-dependencies solves the problem: when discharging serialisation constraints, only data-dependencies carried through the second destination queue of `dupl` may be used. This expresses the fact that the `dupl` q_1 q_2 q_3 can only be considered completed once the second forwarding has been carried out. Consequently, the above race-condition $[\mathtt{1}] < [\mathtt{2}]$ cannot be discharged and the program is rejected as being non-deterministic. In contrast, a program such as

$$[\mathtt{1}]\mathtt{dupl}^{fu}\ \mathtt{q}_1\ \mathtt{q}_2\ \mathtt{q}_3\ [\mathtt{2}]\mathtt{dec}\ \mathtt{q}_3\ \mathtt{q}_2$$

is (rightly) accepted.

If the forwardings for `dupl` are initiated in a different order, other data-dependencies must be considered – if the order is unspecified, no data-dependencies involving results of `dupl`-instructions may be used for discharging constraints. As the processor's behaviour is expressed as a dynamic semantics for ALEF, our approach is adaptable to all these situations – one has to change the definition of data-dependencies appropriately and prove a result similar to Proposition 14.

We thus expect our approach can handle fine-grained models of execution, and supports the implementation of instructions such as `dupl` in an asynchronous environment.

The second design decision concerns the one-to-one correspondence of functional units to instruction pipelines. In a realistic processor, several functional units of a particular functionality may be present, and different association patterns between functional units and instruction queues are possible. For example, several functional units of type `ALU` could be served jointly by a single queue or jointly by several queues, or each functional might have its own private queue, or several private queues. Also the mapping of instructions to functional units and/or to instruction queues can be statically fixed or be determined dynamically. Finally, operand queues can be associated to single functional units, blocks of functional units, single instruction queues or blocks of instruction queues which are connected to functional units in any of the above ways.

For some of these scenarios even functional correctness is difficult to achieve as the mapping of instruction to pipelines needs to be consistent with the operand queues used by earlier instructions for communicating operands. Clearly, the exploration of various design alternatives needs a systematic framework. The static semantics presented in this chapter provides a first step towards such a framework. This is particular so for the distinction between the derivation of constraints and their discharge, as many of the above design alternatives result either in additional constraints or in restrictions on the techniques for discharging them. For example, hazards for *reading* from an operand queue occur if operand queues serve several functional units or instruction pipelines. Extending the static semantics by constraints $[\mathtt{n}] < [\mathtt{m}]$ whenever $[\mathtt{n}]$ and $[\mathtt{m}]$ access the same queue for reading, we can retain determinism by the same approach as before. If several pipelines serve a single functional unit, we use only those instructions for discharging a constraint which inhabit the same pipeline - it suffices to modify the definition of pipeline-dependencies such that two instruction's instruction queues are

compared rather than the functional units on which they execute.
The ability to treat various architectural alternatives using similar techniques highlights the generality of the programming language approach and should make AQM's applicable to a range of target processors.

Finally, we note that the technical requirement in Proposition 14 that a competitor $[l]$ be related to $[n]$ by $[l] \subset [n]$; or $[n] \subset [l]$ is a little too restrictive. For example, program

$$[1]\mathtt{ldc}\ 5\ \mathtt{q}_1\ [2]\mathtt{dupl}^{\mathtt{MUL}}\ \mathtt{q}_1\ \mathtt{q}_1\ \mathtt{q}_2\ [3]\mathtt{dec}\ \mathtt{q}_2\ \mathtt{q}_1$$

is (wrongly) rejected for shape **1**. Although instruction $[3]$ is data-dependent via $\mathtt{q}_2$ on $[2]$ and the (empty) order of pipeline-dependencies can extended by $[2] <_{\mathbf{1}} [3]$, this cannot be used when the $\mathtt{q}_1$-carried data-dependency between $[1]$ and $[2]$ is examined as the competitor $[3]$ is *not yet* related to $[1]$. Further work is needed in Proposition 14 to see whether the condition may be relaxed such that competitor $[l]$ only fulfils $[l] \subset [n]$; or $[m] \subset [l]$.

4.7.1 Summary and outlook

We extended the dynamic and static semantics of ALEF to cover concurrent execution and finite operand queues. As in Chapter 3, soundness of the static semantics with respect to the dynamic semantics was shown. In the static semantics, we separated deriving constraints from discharging them and discussed several techniques for the latter task. Our analysis paid off when the same criteria could be used to preserve absence of deadlock as queue lengths were extended. We also discussed how Tomasulo's algorithm relates to AQM's model of computation. This demonstrates the expressiveness of the AQM model, but also provides an alternative exposition of Tomasulo's algorithm itself. Assessing the status of our main contributions (Section 1.3), the second and third items have thus been provided for AQM's which realise some features of modern processors.
In the next chapter, we will introduce registers and thus cover the full language ALEF. Revisiting the computational models considered so far, we will extend the configurations of the dynamic semantics by a register bank. This will trigger modifications in the

static semantics, but the changes will leave the essential characteristics untouched. In particular, the distinction between deriving constraints for eliminating race conditions and means for discharging such constraints will remain valid and will be extended to new classes of hazards. The robustness of this distinction highlights an advantage of our methodology as orthogonal issues may be analysed separately.

Chapter 5

Queue machines with registers

The previous chapters introduced computational models in which all operands are communicated via forwarding. These models are well suited for studying the fundamental properties of explicit forwarding, but they are not satisfactory models of practical computation. The necessity to enter values into operand queues repeatedly if they are used more than once leads to congestion in the forwarding network and an increased static and dynamic instruction count. For example, variable `k` in the Java excerpt

```
...
for (int i=1; i<n; i++){
  j = j * k;
}
j = j + k;
```
(5.1)

is accessed `n` times where `n` might only be known at runtime. If the computation of `k` is costly and can not be moved inside the loop, its value must be duplicated at the start of each iteration using a `dupl` instruction. One copy is sent to the multiplication and the other copy is forwarded to the next iteration or – in the last iteration – to the instruction following the loop. If the $\texttt{dupl}^{\texttt{ALU}}$ instruction is used, the value of `k` will be available at the correct functional unit in both cases. Similar duplication schemes are necessary for variables `n` and `j`, and one can easily obtain programs where each iteration contains several uses of a value. Furthermore, an `id` instruction is needed directly after the loop since the loop-carried forwarding for `j` has destination `MUL` whereas the instruction following the loop executes on `ALU`. A translation of program (5.1) into ALEF where no sharing of operand queues is implemented thus results in

$$
\begin{array}{ll}
\multicolumn{2}{l}{\quad \texttt{//n in } \mathsf{q}_5, \texttt{k in } \mathsf{q}_4, \texttt{j in } \mathsf{q}_1} \\
[1]\texttt{ldc}\ 1\ \mathsf{q}_3 & \texttt{// i=1} \\
[2]\texttt{jmp}\ 3 & \texttt{// jmp 3} \\
[3]\texttt{dupl}^{\texttt{ALU}}\ \mathsf{q}_4\ \mathsf{q}_2\ \mathsf{q}_4 & \texttt{// dupl k} \\
[4]\texttt{mul}\ \mathsf{q}_1\ \mathsf{q}_2\ \mathsf{q}_1 & \texttt{// j = j * k1} \\
[5]\texttt{inc}\ \mathsf{q}_3\ \mathsf{q}_3 & \texttt{// i++} \\
[6]\texttt{dupl}^{\texttt{ALU}}\ \mathsf{q}_3\ \mathsf{q}_3\ \mathsf{q}_3 & \texttt{// dupl i} \\
[7]\texttt{dupl}^{\texttt{ALU}}\ \mathsf{q}_5\ \mathsf{q}_5\ \mathsf{q}_5 & \texttt{// dupl n} \\
[8]\texttt{sub}\ \mathsf{q}_5\ \mathsf{q}_3\ \mathsf{q}_7 & \texttt{// x = n1 - i1} \\
[8]\texttt{if}\ \mathsf{q}_7\ 9\ 3 & \texttt{// if x=0 l3 l2} \\
[9]\texttt{id}^{\texttt{MUL}}\ \mathsf{q}_1\ \mathsf{q}_6 & \texttt{// move j} \\
[10]\texttt{add}\ \mathsf{q}_6\ \mathsf{q}_4\ \mathsf{q}_6 & \texttt{// j = j* k2}
\end{array}
\tag{5.2}
$$

where we assume $\gamma(\mathsf{q}_1) = \gamma(\mathsf{q}_2) = \texttt{MUL}$, $\gamma(\mathsf{q}_3) = \gamma(\mathsf{q}_4) = \gamma(\mathsf{q}_5) = \gamma(\mathsf{q}_6) = \texttt{ALU}$ and $\gamma(\mathsf{q}_7) = \texttt{BU}$. Indeed, the result is well-typed in the calculus of Chapter 3, with typings $N_1 : \mathbf{1} \multimap A$, $N_3 : A \multimap A$ and $N_9 : A \multimap \mathsf{q}_6 \otimes \mathsf{q}_3 \otimes \mathsf{q}_5$ for $A = \mathsf{q}_3 \otimes \mathsf{q}_4 \otimes \mathsf{q}_5 \otimes \mathsf{q}_1$.

Even a better allocation of operand queues cannot avoid that values such as `k` need to be inserted into the queues repeatedly. As a further drawback, queues which are used inside a loop become essentially unavailable for holding other values as `i`, `j`, `k` and `n` must be at the head of their queue(s) in each iteration. Consequently, any finite number of operand queues effectively limits the nesting depth of programs[1].

It is hence to be expected that architectures with explicit forwarding will in addition contain a small register bank. In most cases, registers will contain values which are repeatedly accessed or whose range of liveness spans across many instructions or several loops. This fits well to the motivation for forwarding in conventional architectures: to provide a fast but less costly mechanism for communicating operands between instructions executed in close succession.

[1] By inserting a chain of $\texttt{id}^{\gamma(q)}\ q\ q$ instructions one can let a value reach the head of its queue while retaining the relative order of other entries, but this scheme has to be considered extremely inefficient.

This chapter consequently considers dynamic semantics for the full language, including registers. The earlier models for sequential, distributed and finite-queue execution and the type systems for eliminating runtime failure are reexamined. In most cases, the calculi for static and dynamic semantics arise as variations or extensions of the calculi presented in the previous chapters. As this correspondence carries over to many proofs of the core statements, the compositional nature of structured programming language formalisms and machine models is demonstrated, supporting our argument for programming language based system development.

5.1 Sequential execution

Motivated by similar reasons as ours, the explicit-forwarding architecture SCALP also contains a register bank. This takes the form of another functional unit, but the performance results given in [End96] indicate that this solution is unsatisfactory. Explicit instructions are needed for transferring values between the register bank and operand queues, thus adding to the pressure on the instruction memory and the decoding unit to provide the processor with instructions. Hence, increased latency and energy consumption are observed as well as low code density.
Consequently, we employ a more traditional register bank which is directly accessible by ordinary instructions. The architectural configuration is shown in Figure 5.1 where a global register bank is connected to the functional units by an operand bus.
This architectural choice is reflected in the structure of ALEF instructions where registers and operand queue names appear in the same position.
A program can use only a finite number of registers, and this set can be statically determined as indirect register addressing is not possible. Should the number of used registers (or operand queues) exceed the number of available registers, remaining values have to be spilled into memory using conventional techniques [App98a].

5.1.1 Dynamic semantics

The dynamic semantics is given by extending the operational model from Chapter 3. Configurations C are equipped with a register file $\mathsf{R} : R \rightharpoonup \mathit{Val}$, a finite map from reg-

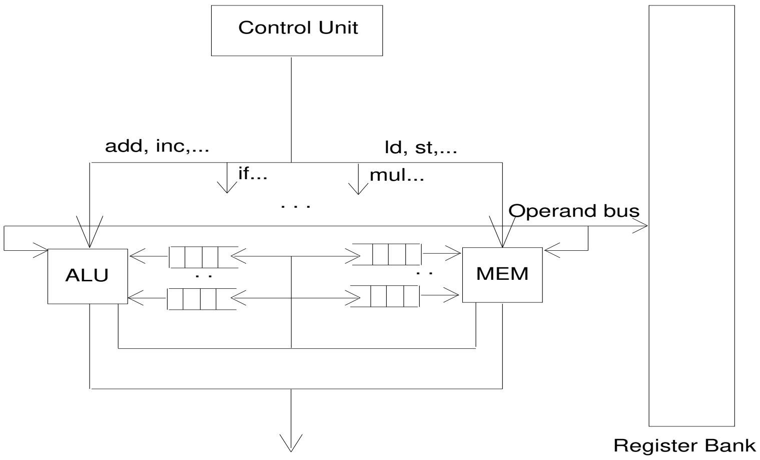


Figure 5.1: Architecture for sequential dynamic semantics with registers

isters r to values a. For straight-line code, this results in configurations $\mathsf{C} = (\mathsf{Q}, \mathsf{R}, \mathsf{M})$. Programs involving jump instructions use configurations of the form $\mathsf{C} = (\mathsf{Q}, \mathsf{R}, \mathsf{M}, \widetilde{n})$. The operational model (Figures 3.4 and 3.5 in Section 3.2) is extended such that relation $\mathsf{C} \xrightarrow{t} \mathsf{D}$ relates the above ternary configurations. This relation is obtained by

- replacing each rule of the form

$$\text{CODE}\frac{\mathsf{C_1} \xrightarrow{(\mu_1,a_1,op_1)} \mathsf{C_2} \quad \ldots \quad \mathsf{C_{n-1}} \xrightarrow{(\mu_{n-1},a_n,op_{n-1})} \mathsf{C_n}}{\mathsf{C_1} \xrightarrow{code} \mathsf{C_n}}$$

  of Figure 3.4 by the rule

$$\text{CODE}\frac{\mathsf{D_1} \xrightarrow{(\mu_1,a_1,op_1)} \mathsf{D_2} \quad \ldots \quad \mathsf{D_{n-1}} \xrightarrow{(\mu_{n-1},a_n,op_{n-1})} \mathsf{D_n}}{\mathsf{D_1} \xrightarrow{code} \mathsf{D_n}}$$

  where $\mathsf{D_i} = (\mathsf{Q_i}, \mathsf{R_i}, \mathsf{M_i})$ for $\mathsf{C_i} = (\mathsf{Q_i}, \mathsf{M_i})$.

- replacing the rules μ, INSTR and COMP by the rules μ-Q, INSTR and COMP given in Figure 5.2. Although the latter two rules appear identical to their earlier counterparts, they are different as configurations contain a third component.

- renaming WR and RD to WR-Q and RD-Q, respectively, without changing their definition.

$$\text{RD-R}\,\frac{}{a \xrightarrow{(read(fu),a,r)} a} \qquad \text{WR-R}\,\frac{}{w \xrightarrow{(write,a,r)} a}$$

$$\mu\text{-R1}\,\frac{\mathsf{R}(r) \xrightarrow{(\mu,a,r)} b}{(\mathsf{Q},\mathsf{R},\mathsf{M}) \xrightarrow{(\mu,a,r)} (\mathsf{Q},\mathsf{R}[r \mapsto b],\mathsf{M})}\; r \in dom\,\mathsf{R}$$

$$\mu\text{-R2}\,\frac{\varepsilon \xrightarrow{(\mu,a,r)} b}{(\mathsf{Q},\mathsf{R},\mathsf{M}) \xrightarrow{(\mu,a,r)} (\mathsf{Q},\mathsf{R}[r \mapsto b],\mathsf{M})}\; r \notin dom\,\mathsf{R}$$

$$\mu\text{-Q}\,\frac{\mathsf{Q}(q) \xrightarrow{(\mu,a,q)} w}{(\mathsf{Q},\mathsf{R},\mathsf{M}) \xrightarrow{(\mu,a,q)} (\mathsf{Q}[q \mapsto w],\mathsf{R},\mathsf{M})}$$

$$\text{INSTR}\frac{\mathsf{C} \xrightarrow{code} \mathsf{D}}{\mathsf{C} \xrightarrow{[n]code} \mathsf{D}} \qquad \text{COMP}\frac{\mathsf{C} \xrightarrow{s} \mathsf{E} \quad \mathsf{E} \xrightarrow{t} \mathsf{D}}{\mathsf{C} \xrightarrow{st} \mathsf{D}}$$

Figure 5.2: Sequential execution with registers: dynamic semantics

- adding the four new rules RD-R, WR-R, μ-R1 and μ-R2 given in Figure 5.2.

Determinacy of execution is preserved as Proposition 1 carries over from Chapter 3.

Proposition 20. *If* $\mathsf{C} \xrightarrow{t} \mathsf{D}$ *and* $\mathsf{C} \xrightarrow{t} \mathsf{E}$ *then* $\mathsf{D} = \mathsf{E}$.

Proof. The rules RD-R and WR-R are easily seen to be deterministic, and determinacy of μ-R1 and μ-R2 follows similarly as the one for μ (now called μ-Q) in the proof of Proposition 1, and so do the claims for the rules CODE and the INSTR and the subsequent induction on the structure of t. □

In addition to deadlocks due to empty operand queues, execution also stalls if a register is accessed for which no entry exists in the register bank. In contrast, operand queue mismatches have no corresponding failure scenario for registers since registers are globally accessible.

5.1.2 Static semantics

Similar to Section 3.3, a type system can eliminate programs which would otherwise deadlock. For the same grammar of types as before, we treat products modulo the identity

$$r \otimes r = r$$

for any register r. This is motivated by the fact that registers can be repeatedly queried for their content (including zero times) but contain only one value. Commutativity and associativity of $\otimes$ remains.

For $A = r_1 \otimes \ldots \otimes r_m \otimes q_1 \otimes \ldots \otimes q_n$ we define $rg(A)$ to be the register part $r_1 \otimes \ldots \otimes r_m$ and $q(A)$ to be the operand queue part $q_1 \otimes \ldots \otimes q_n$. The above identity yields $A \otimes rg(A) = A$ and $rg(A) \otimes rg(A) = rg(rg(A)) = rg(A)$ for any A.

The static semantics is given by axioms

$$\text{R-AX}\,\frac{}{[n]code : X \otimes A_{code} \multimap X \otimes B_{code} \otimes rg(A_{code})}\,SC(code)$$

where side conditions $\gamma(r) = fu$ are always fulfilled and products A_{code} and B_{code} are as given in Table 3.1. Program fragments are combined using the same cut rule CUT as in Section 3.3.

The usage of $rg(A_{code})$ to the right of $\multimap$ in R-AX prevents a register from being deleted. For example, in the program

$$[1]\mathtt{ldc}\ 2\ \mathtt{r}_1\ [2]\mathtt{dec}\ \mathtt{r}_1\ \mathtt{q}_1$$

the value in $\mathtt{r}_1$ is still available after the execution of the second instruction. This is captured in the derivation

$$\frac{[1]\mathtt{ldc}\ 2\ \mathtt{r}_1 : \mathbf{1} \multimap \mathtt{r}_1 \quad [2]\mathtt{dec}\ \mathtt{r}_1\ \mathtt{q}_1 : \mathtt{r}_1 \multimap \mathtt{r}_1 \otimes \mathtt{q}_1}{[1]\mathtt{ldc}\ 2\ \mathtt{r}_1\ [2]\mathtt{dec}\ \mathtt{r}_1\ \mathtt{q}_1 : \mathbf{1} \multimap \mathtt{r}_1 \otimes \mathtt{q}_1}$$

The same program can also be typed if a value is already present in $\mathtt{r}_1$

$$\frac{\dfrac{[1]\mathtt{ldc}\ 2\ \mathtt{r}_1 : \mathtt{r}_1 \multimap \mathtt{r}_1 \otimes \mathtt{r}_1}{[1]\mathtt{ldc}\ 2\ \mathtt{r}_1 : \mathtt{r}_1 \multimap \mathtt{r}_1} \quad [2]\mathtt{dec}\ \mathtt{r}_1\ \mathtt{q}_1 : \mathtt{r}_1 \multimap \mathtt{r}_1 \otimes \mathtt{q}_1}{[1]\mathtt{ldc}\ 2\ \mathtt{r}_1\ [2]\mathtt{dec}\ \mathtt{r}_1\ \mathtt{q}_1 : \mathtt{r}_1 \multimap \mathtt{r}_1 \otimes \mathtt{q}_1}$$

using the identity $\mathtt{r}_1 \otimes \mathtt{r}_1 = \mathtt{r}_1$. No typing is possible where the value in $\mathtt{r}_1$ is deleted as any r appearing to the left of $\multimap$ also appears to its right.

Lemma 5. *If $t : A \multimap B$ then $B = rg(A) \otimes B$.*

Proof. Induction on the structure of t. For all base cases, the rule R-AX guarantees that there is an X such that $A = A_{code} \otimes X$ and $B = B_{code} \otimes rg(A_{code}) \otimes X$, so $rg(A) = rg(A_{code}) \otimes rg(X)$ and thus

$$B \otimes rg(A) = B_{code} \otimes rg(A_{code}) \otimes X \otimes rg(A_{code}) \otimes rg(X) = B_{code} \otimes rg(A_{code}) \otimes X.$$

For $\dfrac{s : A \multimap C \quad t : C \multimap B}{st : A \multimap B}$, $C = C \otimes rg(A)$ and $B = B \otimes rg(C)$ follow from the induction hypothesis. Consequently, $rg(C) = rg(C) \otimes rg(rg(A)) = rg(C) \otimes rg(A)$, so $B = B \otimes rg(C) = B \otimes rg(C) \otimes rg(A) = B \otimes rg(A)$. □

For relating static and dynamic semantics, the definition of shape is adapted to reflect the modified notion of configurations.

Definition 16. *The* shape *of a configuration* $\mathsf{C} = (\mathsf{Q}, \mathsf{R}, \mathsf{M})$ *is*

$$shape(\mathsf{C}) = \otimes_{q \in Q} q^{|\mathsf{Q}(q)|} \otimes \otimes_{r \in dom\, \mathsf{R}} r.$$

We obtain soundness and completeness for sequential execution as in Theorem 1.

Theorem 4. *Let t be an instruction sequence.*

1. *If $t : A \multimap B$ and $shape(\mathsf{C}) = A$ then there is a unique D such that $\mathsf{C} \xrightarrow{t} \mathsf{D}$, and $shape(\mathsf{D}) = B$ holds.*
2. *If $\mathsf{C} \xrightarrow{t} \mathsf{D}$ then $t : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.*

Proof. 1. The proof proceeds similarly to the proof of Theorem 1. The sub-cases for rules WR-Q, RD-Q and μ-Q transfer directly. For rules RD-R, WR-R, μ-R1 and μ-R2 they are as follows.

RD-R **Claim 1:** For arbitrary a, fu and r, there are b and c with $a \xrightarrow{(read(fu),b,r)} c$. Furthermore, $a = b = c$ holds.

Proof Follows directly from the definition of rule RD-R.

WR-R **Claim 2:** For arbitrary w, a and r there is a b such that $w \xrightarrow{(write,a,r)} b$. Furthermore, $a = b$ holds.

Proof Follows directly from the definition of rule WR-R.

μ-R1 There are two claims.

Claim 3: If $shape(\mathsf{C}) = r \otimes X$ and fu arbitrary, then there are unique D and a with $\mathsf{C} \xrightarrow{(read(fu),a,r)} \mathsf{D}$. Furthermore, $shape(\mathsf{D}) = shape(\mathsf{C})$ holds.

Proof For $\mathsf{C} = (\mathsf{Q}, \mathsf{R}, \mathsf{M})$ and $shape(\mathsf{C}) = r \otimes X$ we have $r \in dom\ \mathsf{R}$, so $\mathsf{R}(r) = a$ holds for some unique a. By claim 1, there are b and c with $\mathsf{R}(r) \xrightarrow{(read(fu),b,r)} c$, and $a = b = c$ holds. Hence, $\mathsf{C} \xrightarrow{(read(fu),a,q)} \mathsf{D}$ is derivable for $\mathsf{D} = (\mathsf{Q}, \mathsf{R}[r \mapsto a], \mathsf{M})$ i.e. $\mathsf{D} = \mathsf{C}$ by the uniqueness of D (Proposition 20). Consequently, $shape(\mathsf{D}) = shape(\mathsf{C})$ follows.

Claim 4: If $shape(\mathsf{C}) = r \otimes X$ and a arbitrary, then there is a unique D with $\mathsf{C} \xrightarrow{(write,a,r)} \mathsf{D}$. Furthermore, $shape(\mathsf{D}) = shape(\mathsf{C})$ holds.

Proof For $\mathsf{C} = (\mathsf{Q}, \mathsf{R}, \mathsf{M})$ and $shape(\mathsf{C}) = r \otimes X$ we have $r \in dom\ \mathsf{R}$, so $\mathsf{R}(r) = b$ for some unique b. Applying claim 2 yields a unique c with $b \xrightarrow{(write,a,r)} c$, and $c = a$ holds, hence $\mathsf{R}(r) \xrightarrow{(write,a,r)} a$ and $\mathsf{C} \xrightarrow{(write,a,r)} \mathsf{D}$ follow for $\mathsf{D} = (\mathsf{Q}, \mathsf{R}[r \mapsto a], \mathsf{M})$, and uniqueness of D follows from Proposition 20.

Furthermore, $shape(\mathsf{D}) = shape(\mathsf{C}) \otimes r = r \otimes X \otimes r = r \otimes X = shape(\mathsf{C})$ follows by applying the definition of $shape(\mathsf{D})$ and the identity $r \otimes r = r$.

μ-R2 **Claim 5:** If $prime(r, shape(\mathsf{C}))$ and a arbitrary, then there is a unique D with $\mathsf{C} \xrightarrow{(write,a,r)} \mathsf{D}$. Furthermore, $shape(\mathsf{D}) = r \otimes shape(\mathsf{C})$ holds.

Proof By Definition 16, $prime(r, shape(\mathsf{C}))$ implies $r \notin dom\ \mathsf{R}$. Applying claim 2 to $w = \varepsilon$ yields a unique b such that $\varepsilon \xrightarrow{(write,a,r)} b$ holds, with $a = b$. Consequently, rule μ-R2 yields $\mathsf{C} \xrightarrow{(write,a,r)} \mathsf{D}$ for $\mathsf{D} = (\mathsf{Q}, \mathsf{R}[r \mapsto a], \mathsf{M})$ where $\mathsf{C} = (\mathsf{Q}, \mathsf{R}, \mathsf{M})$. Uniqueness of D follows as in Proposition 20, and Definition 16 implies $shape(\mathsf{D}) = shape(\mathsf{C}) \otimes r$ as required.

For the rules CODE, we prove that if $[n]code : A \multimap B$ and $shape(\mathsf{C}) = A$ hold, then there is a unique D with $\mathsf{C} \xrightarrow{t} \mathsf{D}$, and additionally $shape(\mathsf{D}) = B$. We treat the case `dec` $op_1\ op_2$ explicitly.

Case $code =$ `dec` $op_1\ op_2$**.** By the typing rule R-AX, there is an X such that $shape(\mathsf{C}) = A = op_1 \otimes X$ and $B = op_2 \otimes X$, and X is unique modulo identities $r \otimes r = r$. For deriving $\mathsf{C} \xrightarrow{\mathtt{dec}\ op_1\ op_2} \mathsf{D}$ for some D, we have to show that there are E and a such that

$$\mathsf{C} \xrightarrow{(read(\mathtt{ALU}),a,op_1)} \mathsf{E} \text{ and } \mathsf{E} \xrightarrow{(write,a-1,op_2)} \mathsf{D}.$$

In the case of $op_1 = q$ for some q, R-AX implies $\gamma(q) = FU(\mathtt{dec}) = \mathtt{ALU}$, so the claim for rule μ-Q (claim 3 in the proof of Theorem 1) implies the (unique) existence of E and a with $\mathsf{C} \xrightarrow{(read(\mathtt{ALU}),a,op_1)} \mathsf{E}$, and $shape(\mathsf{E}) = X$. If $op_1 = r$ for some r, the property $A = op_1 \otimes X$, together with claim 3 above, implies the same existence of E and a, with $shape(\mathsf{E}) = shape(\mathsf{C}) = r \otimes X$.

For showing $\mathsf{E} \xrightarrow{(write,a-1,op_2)} \mathsf{D}$, we apply claim 4 of the proof of Theorem 1 in the case of $op_2 = q'$ for some q', and apply the above claims 4 or 5 in the case of $op_2 = r'$ for some r', depending on whether $prime(r', shape(\mathsf{E}))$ holds of not. In the former case, we obtain $shape(\mathsf{D}) = shape(\mathsf{E}) \otimes q' = B$, and in the latter case we obtain $shape(\mathsf{D}) = shape(\mathsf{E}) \otimes r' = B$. Uniqueness of D follows in both cases from Proposition 20.

Cases $code \neq \mathtt{dec}\ op_1\ op_2$**.** Similar.

Finally, the induction on the structure of t proceeds exactly as in the proof of Theorem 1.

2. We perform a structural induction similar to the proof of the second part of Theorem 1. The sub-cases for rules WR-Q, RD-Q and μ-Q transfer directly. For rules RD-R, WR-R, μ-R1 and μ-R2 they are as follows.

RD-R **Claim 1:** If $w \xrightarrow{(read(fu),a,r)} v$ then $w = v = a$.

Proof Follows directly from the definition of rule RD-R.

WR-R **Claim 2:** If $w \xrightarrow{(write,a,r)} v$ then $v = a$.

Proof Follows directly from the definition of rule WR-R.

μ-R1 and μ-R2 **Claim 3:** If $\mathsf{C} \xrightarrow{(read(fu),a,r)} \mathsf{D}$ then $shape(\mathsf{C}) = r \otimes shape(\mathsf{D})$.

Proof If the last rule in the derivation of $\mathsf{C} \xrightarrow{(read(fu),a,r)} \mathsf{D}$ was μ-R1, then $\mathsf{D} = (\mathsf{Q}, \mathsf{R}[r \mapsto b], \mathsf{M})$ must hold for some b with $\mathsf{R}(r) \xrightarrow{(read(fu),a,r)} b$ and

$\mathsf{C} = (\mathsf{Q},\mathsf{R},\mathsf{M})$, so $r \in dom\ \mathsf{R}$. By claim 1, $\mathsf{R}(r) = b = a$ follows, and the definition of *shape* implies

$$\begin{aligned} r \otimes shape(\mathsf{D}) &= r \otimes \otimes_{q\in Q} q^{|\mathsf{Q}(q)|} \otimes \otimes_{r'\in dom\ \mathsf{R}[r\mapsto b]} r' \\ &= r \otimes \otimes_{q\in Q} q^{|\mathsf{Q}(q)|} \otimes \otimes_{r'\in dom\ \mathsf{R}} r' \\ &= \otimes \otimes_{q\in Q} q^{|\mathsf{Q}(q)|} \otimes \otimes_{r'\in dom\ \mathsf{R}} r' = shape(\mathsf{C}) \end{aligned}$$

If the last rule in the derivation for $\mathsf{C} \xrightarrow{(read(fu),a,r)} \mathsf{D}$ was μ-R2, then $r \notin dom\ \mathsf{R}$, then $\varepsilon \xrightarrow{(read(fu),a,r)} b$ and $\mathsf{D} = (\mathsf{Q},\mathsf{R}[r \mapsto b],\mathsf{M})$ must hold for some b where $\mathsf{C} = (\mathsf{Q},\mathsf{R},\mathsf{M})$. By claim 1, however, the second condition cannot be fulfilled, so the last rule for deriving $\mathsf{C} \xrightarrow{(read(fu),a,r)} \mathsf{D}$ cannot have been μ-R2.

In particular, $prime(r, shape(\mathsf{C}))$ implies that there are no D and a with $\mathsf{C} \xrightarrow{(read(fu),a,r)} \mathsf{D}$.

Claim 4: If $\mathsf{C} \xrightarrow{(write,a,r)} \mathsf{D}$ then $shape(\mathsf{D}) = r \otimes shape(\mathsf{C})$.

Proof No matter which of the two rules was used, a derivation with final sequent $\mathsf{C} \xrightarrow{(write,a,r)} \mathsf{D}$ where $\mathsf{C} = (\mathsf{Q},\mathsf{R},\mathsf{M})$ forces $\mathsf{D} = (\mathsf{Q},\mathsf{R}[r \mapsto b],\mathsf{M})$ for some b, so the definition of *shape* implies

$$\begin{aligned} shape(\mathsf{D}) &= \otimes \otimes_{q\in Q} q^{|\mathsf{Q}(q)|} \otimes \otimes_{r'\in dom\ \mathsf{R}[r\mapsto b]} r' \\ &= \otimes_{q\in Q} q^{|\mathsf{Q}(q)|} \otimes r \otimes \otimes_{r'\in dom\ \mathsf{R}} r' \\ &= r \otimes_{q\in Q} q^{|\mathsf{Q}(q)|} \otimes \otimes_{r'\in dom\ \mathsf{R}} r' = r \otimes shape(\mathsf{C}). \end{aligned}$$

For the rules CODE, we prove that $\mathsf{C} \xrightarrow{code} \mathsf{D}$ implies $[n]code : shape(\mathsf{C}) \multimap shape(\mathsf{D})$. Again, we treat only one case explicitly.

Case $code = \mathtt{dec}\ op_1\ op_2$**.** For $\mathsf{C} \xrightarrow{\mathtt{dec}\ op_1\ op_2} \mathsf{D}$ to hold, there must be E and a with

$$\mathsf{C} \xrightarrow{(read(\mathtt{ALU}),a,op_1)} \mathsf{E} \text{ and } \mathsf{E} \xrightarrow{(write,a-1,op_2)} \mathsf{D}$$

If $op_1 = q$, the claim for rule μ-Q (claim 3 in the proof of Theorem 1) implies $shape(\mathsf{C}) = shape(\mathsf{E}) \otimes q = shape(\mathsf{E}) \otimes op_1$ and $\gamma(op_1) = \gamma(q) = \mathtt{ALU}$. If $op_1 = r$, claim 3 above implies $shape(\mathsf{C}) = r \otimes shape(\mathsf{E}) = shape(\mathsf{E}) \otimes op_1$, and $\gamma(op_1) = \gamma(r) = \mathtt{ALU}$ is vacuously true.

Furthermore, for $op_2 = q'$, the claim for *write* and rule μ-Q (claim 4 in the proof of Theorem 1) implies $shape(\mathsf{D}) = shape(\mathsf{E}) \otimes q' = shape(\mathsf{E}) \otimes op_2$, and for $op_2 = r'$, the above claim 4 implies $shape(\mathsf{D}) = shape(\mathsf{E}) \otimes r' = shape(\mathsf{E}) \otimes op_2$.

With $X = shape(\mathsf{E})$, the typing axiom R-AX indeed derives $[n]\mathtt{dec}\ op_1\ op_2 : op_1 \otimes X \multimap op_2 \otimes X$, i.e. $[n]\mathtt{dec}\ op_1\ op_2 : shape(\mathsf{C}) \multimap shape(\mathsf{D})$.

Cases $code \neq \mathtt{dec}\ op_1\ op_2$**.** Similar.

The induction on the structure of t now proceeds exactly as in the proof of Theorem 1 (second part).

□

Consequently, well-typed programs cannot run into deadlocks (operand queue starvation or empty registers) or operand queue mismatch. In particular, every register is guaranteed to be initialised when it is accessed for the first time.

Similar to Section 3.4, the type system admits the derived rule

$$\text{R-WK}\frac{t : A \multimap B}{t : A \otimes X \multimap B \otimes X}$$

and has principal typings. These can be obtained by the typing rules

$$\text{P-R-AX}\ \frac{}{[n]code :: A_{code} \multimap B_{code} \otimes rg(A_{code})}\ SC(code)$$

and

$$\text{P-R-CUT}\ \frac{s :: A \multimap B \qquad t :: C \multimap D}{st :: A \otimes U \multimap V \otimes D}\ B/C = V/U$$

where a modified definition of cancellation ensures that the identity $r \otimes r = r$ does not introduce additional products.

Definition 17. *The pair* (V,U) *is called the* cancellation *of* A *and* B*, written* $A/B = V/U$*, if* $prime(V,U)$ *and* $prime(rg(V), rg(B))$ *hold and also* $prime(rg(U), rg(A))$ *and* $U \otimes A = V \otimes B$*.*

The definition of principal types transfers literally from Chapter 3.

Definition 18. *A typing* $t : A \multimap B$ *is* principal *if for every typing* $t : C \multimap D$ *there is an* X *such that* $C = A \otimes X$ *and* $D = B \otimes X$*.*

Example. For $[1]\texttt{ldc}\ 4\ \texttt{r}_1$ and $[2]\texttt{add}\ \texttt{r}_1\ \texttt{r}_2\ \texttt{r}_3$, rule P-R-AX yields

$$[1] :: \mathbf{1} \multimap \texttt{r}_1 \text{ and } [2] :: \texttt{r}_1 \otimes \texttt{r}_2 \multimap \texttt{r}_1 \otimes \texttt{r}_2 \otimes \texttt{r}_3.$$

We have $\texttt{r}_1/(\texttt{r}_1 \otimes \texttt{r}_2) = \mathbf{1}/\texttt{r}_2$, hence

$$[1]\,[2] :: \texttt{r}_2 \multimap \texttt{r}_1 \otimes \texttt{r}_2 \otimes \texttt{r}_3.$$

If the condition $prime(rg(U), rg(A))$ was dropped in Definition 17, the identity $\texttt{r}_1 = \texttt{r}_1 \otimes \texttt{r}_1$ would yield the alternative cancellation $\texttt{r}_1/(\texttt{r}_1 \otimes \texttt{r}_2) = \mathbf{1}/\texttt{r}_1 \otimes \texttt{r}_2$ and thus the typing $[1]\,[2] :: \texttt{r}_2 \otimes \texttt{r}_1 \multimap \texttt{r}_1 \otimes \texttt{r}_2 \otimes \texttt{r}_3$. While this is indeed a valid typing for $[1]\,[2]$, it is not the principal typing. $\diamond$

Proposition 21. *$t :: A \multimap B$ iff $t : A \multimap B$ is the principal typing for t.*

Proof. There are three parts.

$t :: A \multimap B$ **implies** $t : A \multimap B$**.** Structural induction on t. For single instructions, the claim is trivial as the weakening $X = \mathbf{1}$ can be used in the axioms for $t : A \multimap B$. For the cut rule where $s :: A \multimap B$, $t :: C \multimap D$ and $s\,t :: A \otimes U \multimap V \otimes D$ with $B/C = V/U$, we have $s : A \multimap B$ and $t : C \multimap D$ by induction hypothesis, and the weakening rule implies $s : A \otimes U \multimap B \otimes U$. The definition of cancellation yields $B \otimes U = V \otimes C$, and by applying the weakening V to the typing for t we obtain $t : V \otimes C \multimap V \otimes D$. Applying the cut rule yields $s\,t : A \otimes U \multimap V \otimes D$.

$t :: A \multimap B$ **implies** $t : A \multimap B$ **is principal typing for** t**.** We show that for $t :: A \multimap B$ and $t : C \multimap D$ there is an X such that $C = A \otimes X$ and $D = B \otimes X$, again by structural induction on t.

For a single instruction, $A = A_{code}$ and $B = B_{code} \otimes rg(A_{code})$ hold, and the typing rule for $t : C \multimap D$ implies $C = A_{code} \otimes Z$ and $D = B_{code} \otimes rg(A_{code}) \otimes Z$ for some Z. Take $X = Z$.

For $s\,t :: A \multimap B$ and $s\,t : C \multimap D$ there are by the typing rules products such that

$$\frac{s :: E \multimap F \quad t :: G \multimap H}{s\,t :: E \otimes U \multimap V \otimes H}\; F/G = V/U$$

and

$$\frac{s : C \multimap I \quad t : I \multimap D}{st : C \multimap D}$$

where $A = E \otimes U$ and $B = V \otimes H$. Since s is part of st and $s :: E \multimap F$ holds, the induction hypothesis guarantees that $s : E \multimap F$ is the principal typing for s, so for $s : C \multimap I$ there is a Z with

$$E \otimes Z = C \text{ and } F \otimes Z = I.$$

Likewise, $t : G \multimap H$ is principal, so for $t : I \multimap D$ there is a Y with

$$G \otimes Y = I \text{ and } H \otimes Y = D.$$

Consequently, $F \otimes Z = I = G \otimes Y$ and by the definition of cancellation $F/G = V/U$ there is an M such that $U \otimes M = Z$ and $V \otimes M = Y$. Therefore,

$$C = E \otimes Z = E \otimes U \otimes M = A \otimes M$$

and

$$D = H \otimes Y = H \otimes V \otimes M = B \otimes M.$$

Take $X = M$.

$t : A \multimap B$ **principal typing for** t **implies** $t :: A \multimap B$**.** If t is a single instruction, then $A = A_{code}$ and $B = B_{code} \otimes rg(A_{code})$ and the typing $t :: A_{code} \multimap B_{code} \otimes rg(A_{code})$ is derivable.

For $st : A \multimap B$ principal, the last rule was CUT, so there is a C with $s : A \multimap C$ and $t : C \multimap B$. Let $s : F \multimap G$ and $t : H \multimap I$ be the principal typings for s and t, respectively. Since s and t occur inside st we can apply the induction hypothesis, so $s :: F \multimap G$ and $t :: H \multimap I$. By the principality of $s : F \multimap G$ and $t : H \multimap I$ there are Z and Y with

$$A = F \otimes Z, G \otimes Z = C = H \otimes Y \text{ and } B = I \otimes Y.$$

Hence, for $G/H = V/U$ we obtain both

$$st :: F \otimes U \multimap I \otimes V$$

and $U \otimes L = Z$ and $V \otimes L = Y$ for some L.

We have to show $F \otimes U = A$ and $I \otimes V = B$. By the first part of this proof, $st :: F \otimes U \multimap I \otimes V$ implies $st : F \otimes U \multimap I \otimes V$, so the principality of $st : A \multimap B$ guarantees that there is an X such that

$$A \otimes X = F \otimes U \text{ and } B \otimes X = I \otimes V.$$

On the other hand,

$$A = F \otimes Z = F \otimes U \otimes L \text{ and } B = I \otimes Y = I \otimes V \otimes L$$

holds. By combining the equations, we have

$$\begin{aligned} A &= L \otimes F \otimes U = L \otimes A \otimes X \\ B &= I \otimes V \otimes L = L \otimes B \otimes X \end{aligned}$$

so X and L cannot contain any operand queue names as factors, i.e. it it $X = rg(X)$ and $L = rg(L)$. Hence

$$\begin{aligned} F \otimes U &= A \otimes X = L \otimes A \otimes X \otimes rg(X) = L \otimes A \otimes X = A \\ I \otimes V &= B \otimes X = L \otimes B \otimes X \otimes rg(X) = L \otimes B \otimes X = B \end{aligned}$$

as claimed.

□

5.1.3 Program execution

Similarly to Section 3.5, execution of programs with registers and jump instructions is defined in terms of basic blocks. A program $\mathtt{P} = (\mathtt{N}, \mathtt{A})$ consists of a partial map $\mathtt{N} : \mathbf{N} \rightharpoonup \mathit{Iseq}$, together with an association $\mathtt{A} \subset \mathit{dom}\,\mathtt{N} \times \mathit{dom}\,\mathtt{N}$ such that

- the labelling of instructions is unique
- the first instruction of $\mathtt{N}(n)$ has label $[n]$
- at most the last instruction of $\mathtt{N}(n)$ is a jump instruction
- $\mathtt{A}$ contains exactly those pairs (n, m) where the opcode of last instruction in $\mathtt{N}(n)$ is either `jmp` m or `if` $op\ n_1\ n_2$ with $m \in \{n_1, n_2\}$.

$$\text{R-IF-T}\frac{(\mathsf{Q},\mathsf{R},\mathsf{M}) \xrightarrow{(read(\mathtt{BU}),0,op)} (\mathsf{P},\mathsf{S},\mathsf{N})}{(\mathsf{Q},\mathsf{R},\mathsf{M},nil) \xrightarrow{\mathtt{if}\ op\ n_1\ n_2} (\mathsf{P},\mathsf{S},\mathsf{N},n_1)}$$

$$\text{R-IF-F}\frac{(\mathsf{Q},\mathsf{R},\mathsf{M}) \xrightarrow{(read(\mathtt{BU}),a,op)} (\mathsf{P},\mathsf{S},\mathsf{N})}{(\mathsf{Q},\mathsf{R},\mathsf{M},nil) \xrightarrow{\mathtt{if}\ op\ n_1\ n_2} (\mathsf{P},\mathsf{S},\mathsf{N},n_2)}\ a \neq 0$$

$$\text{R-JMP}\frac{--}{(\mathsf{Q},\mathsf{R},\mathsf{M},nil) \xrightarrow{\mathtt{jmp}\ n} (\mathsf{Q},\mathsf{R},\mathsf{M},n)}$$

$$\text{R-INJ}\ \frac{(\mathsf{Q},\mathsf{R},\mathsf{M}) \xrightarrow{code} (\mathsf{P},\mathsf{S},\mathsf{N})}{(\mathsf{Q},\mathsf{R},\mathsf{M},\widetilde{n}) \xrightarrow{code} (\mathsf{P},\mathsf{S},\mathsf{N},\widetilde{n})}\ FU(code) \neq \mathtt{BU}$$

$$\text{R-INSTR}\ \frac{\mathsf{C} \xrightarrow{code} \mathsf{D}}{\mathsf{C} \xrightarrow{[n]code} \mathsf{D}} \qquad \text{R-COMP}\ \frac{\mathsf{C} \xrightarrow{s} \mathsf{E} \quad \mathsf{E} \xrightarrow{t} \mathsf{D}}{\mathsf{C} \xrightarrow{st} \mathsf{D}}$$

$$\text{R-EXECUTE}\ \frac{(\mathsf{Q},\mathsf{R},\mathsf{M},nil) \xrightarrow{\mathtt{N}(n)} \mathsf{D}}{(\mathsf{Q},\mathsf{R},\mathsf{M},n) \xrightarrow{n} \mathsf{D}} \qquad \text{R-COMPOSE}\ \frac{\mathsf{C} \xrightarrow{v} \mathsf{E} \quad \mathsf{E} \xrightarrow{w} \mathsf{D}}{\mathsf{C} \xrightarrow{vw} \mathsf{D}}$$

Figure 5.3: Sequential dynamic semantics including jumps and registers

5.1.3.1 Dynamic semantics

The dynamic semantics is obtained by similar rules as in Section 3.5, modulo the updated notion of configurations $(\mathsf{Q},\mathsf{R},\mathsf{M},\widetilde{n})$ (see Figure 5.3) and the additional rules RD-R, WR-R, μ-R1 and μ-R2. Consequently, determinism of execution may be proven in a similar way as in Proposition 4.

5.1.3.2 Static semantics

Instantiating the axiom R-AX with the data in Table 3.2 yields the typing rules

$$\text{R-AX-IF}\ \frac{}{[n]\mathtt{if}\ op\ n_1\ n_2 : X \otimes op \multimap X \otimes rg(op)}\ \gamma(op) = \mathtt{BU}$$

and

$$\text{R-AX-JMP}\ \frac{}{[n]\,\texttt{jmp}\ m : X \multimap X}$$

where any side condition $\gamma(r) = \texttt{BU}$ is considered true.

The rule for composing basic blocks is generalised to

$$\text{R-PROG}\ \frac{\Sigma = \cup_{n \in dom\,\texttt{N}}\{n : A_n \multimap B_n\} \quad (n,m) \in \texttt{A} \text{ implies } \exists X \text{ s.t. } B_n = A_m \otimes X \text{ and } B_m \otimes X = B_m}{\Sigma \vdash (\texttt{N},\texttt{A}) : \circ}$$

such that $\texttt{N}(n)$ delivers *at least* all register-carried operands needed by (possibly weakened) $\texttt{N}(m)$, and additional registers must be amongst those promised by B_m to its successors. In particular, the additional items X fulfil $X = rg(X)$ and occur in $rg(B_m)$. For operand queues, the rule agrees with the rule from Section 3.5.2, but registers written to in $\texttt{N}(m)$ may or may not contain values before $\texttt{N}(m)$ is called. For example,

$$\begin{aligned} \texttt{N}(1) &= [1]\texttt{ldc}\ 7\ \texttt{q}_1\ [2]\texttt{jmp}\ 3 \\ \texttt{N}(3) &= [3]\texttt{ldc}\ 9\ \texttt{r}_1\ [4]\texttt{add}\ \texttt{q}_1\ \texttt{r}_1\ \texttt{q}_1\ [5]\texttt{jmp}\ 3 \end{aligned}$$

can be typed in context $\Sigma = [1 \mapsto \mathbf{1} \multimap \texttt{q}_1,\quad 3 \mapsto \texttt{q}_1 \multimap \texttt{r}_1 \otimes \texttt{q}_1]$, where the arrow $(\texttt{N}(1),\texttt{N}(3))$ uses $X = \mathbf{1}$ and the arrow $(\texttt{N}(3),\texttt{N}(3))$ uses $X = \texttt{r}_1$. The earlier rule

$$\frac{\Sigma = \cup_{n \in dom\,\texttt{N}}\{n : A_n \multimap B_n\} \quad (n,m) \in \texttt{A} \text{ implies } B_n = A_m}{\Sigma \vdash (\texttt{N},\texttt{A}) : \circ}$$

would have required to weaken the typings to

$$\Sigma' = [1 \mapsto \texttt{r}_1 \multimap \texttt{r}_1 \otimes \texttt{q}_1,\quad 3 \mapsto \texttt{r}_1 \otimes \texttt{q}_1 \multimap \texttt{r}_1 \otimes \texttt{q}_1],$$

forcing an initial configuration to contain an element in $\texttt{r}_1$.

As before, the A_i and B_i are obtained from typings of the bodies by weakenings.

Proposition 5 carries over from Chapter 3 with similar modifications regarding the shapes of configurations as in the previous section:

Proposition 22. *Let $\Sigma \vdash \texttt{P} : \circ$, and for $\mathsf{C} = (\mathsf{Q},\mathsf{R},\mathsf{M},n)$ let $n : A \multimap B$ be the entry for n in Σ. Let $shape(\mathsf{C}) = A \otimes X$ with $B \otimes X = B$ for some X. Then there is a unique D such that $\mathsf{C} \xrightarrow{n} \mathsf{D}$, and $shape(\mathsf{D}) = B$ holds. Furthermore, if $\mathsf{D} = (\mathsf{P},\mathsf{S},\mathsf{L},m)$, $m \neq nil$ and $m : C \multimap D$ is the entry for m in Σ, then there is a Y such that $B = C \otimes Y$ and $D \otimes Y = D$.*

Proof. The definition of Σ implies $\mathtt{N}(n) : A \multimap B$, so the weakening rule yields $\mathtt{N}(n) : A \otimes X \multimap B \otimes X$, hence $\mathtt{N}(n) : \mathit{shape}(\mathsf{C}) \multimap B$. By Theorem 4, there is thus a unique D such that $(\mathsf{Q},\mathsf{R},\mathsf{M},\mathit{nil}) \xrightarrow{\mathtt{N}(n)} \mathsf{D}$, and $\mathit{shape}(\mathsf{D}) = B$ holds. By rule R-EXECUTE, we thus obtain $\mathsf{C} \xrightarrow{n} \mathsf{D}$.
If D has the form $(\mathsf{P},\mathsf{S},\mathsf{L},m)$ with $m \neq \mathit{nil}$, then $\mathtt{N}(n)$ contains an instruction $\mathtt{jmp}\ m$ or $\mathtt{if}\ \mathit{op}\ m_1\ m_2$ with $m \in \{m_1,m_2\}$. By the definition of basic blocks there is at most one such instruction in $\mathtt{N}(n)$, and it must be the last instruction, so $(n,m) \in \mathtt{A}$ holds. By the definition of program $\mathtt{P}$ and context Σ, there is a unique entry $m : C \multimap D$ in Σ, and the typing rule for $\Sigma \vdash \mathtt{P} : \circ$ implies $\mathit{shape}(\mathsf{D}) = B = C \otimes Y$ for some Y with $D \otimes Y = D$ due to $(n,m) \in \mathtt{A}$. $\square$

5.1.3.3 Type inference

Type inference for programs with registers aims at finding weakenings X_i such that the condition in the typing rule for $\Sigma \vdash \mathtt{P} : \circ$ is fulfilled. Similarly to Section 3.5.3, the task can be seen as a unification problem.

Definition 19. *Let* $\mathtt{N} : \mathbf{N} \rightharpoonup \mathit{Iseq}$ *be a partial map,* $\mathit{dom}\,\mathtt{N} = \{n_1,\ldots,n_k\}$*,* $\mathtt{A} \subset \mathit{dom}\,\mathtt{N} \times \mathit{dom}\,\mathtt{N}$ *and for* $i \leq k$ *let* $\mathtt{N}(n_i) : A_i \multimap B_i$*. A tuple* $\vec{X} = (X_1,\ldots,X_k)$ unifies $\mathtt{N}$ for $\mathtt{A}$ *if* $(n_i,n_j) \in \mathtt{A}$ *implies* $B_i \otimes X_i = A_j \otimes X_j \otimes Z_i^j$ *for some* Z_i^j *with* $B_j \otimes X_j \otimes Z_i^j = B_j \otimes X_j$*.*

The equivalent to Proposition 6 is the following.

Proposition 23.

1. *If* $\Sigma = \cup_{n_i \in \mathit{dom}\,\mathtt{N}} \mathtt{N}(n_i) : A_i \otimes X_i \multimap B_i \otimes X_i$ *and* $\vec{X}$ *is a unifier for* $\mathtt{N}$ *and* $\mathtt{A}$ *then* $\Sigma \vdash (\mathtt{N},\mathtt{A}) : \circ$*.*

2. *If* $\Sigma \vdash (\mathtt{N},\mathtt{A}) : \circ$ *and* $\Sigma = \cup_{n_i \in \mathit{dom}\,\mathtt{N}} \mathtt{N}(n_i) : A_i \multimap B_i$ *then* $\vec{\mathbf{1}}$ *unifies* $\mathtt{N}$ *for* $\mathtt{A}$*.*

Proof.

1. For $\Sigma = \cup_{n_i \in \mathit{dom}\,\mathtt{N}} \mathtt{N}(n_i) : A_i \otimes X_i \multimap B_i \otimes X_i$ and $(n_i,n_j) \in \mathtt{A}$ take $X = Z_i^j$. Then $B_i \otimes X_i = A_j \otimes X_j \otimes Z_i^j = A_j \otimes X_j \otimes X$ and $B_j \otimes X_j \otimes X = B_j \otimes X_j \otimes Z_i^j = B_j \otimes X_j$ since $\vec{X}$ unifies $\mathtt{N}$ for $\mathtt{A}$. Consequently, all conditions in the rule for $\Sigma \vdash (\mathtt{N},\mathtt{A}) : \circ$ are fulfilled.

2. The rule for $\Sigma \vdash (\mathtt{N},\mathtt{A}) : \circ$ implies $\Sigma = \cup_{n_i \in \mathit{dom}\,\mathtt{N}} \mathtt{N}(n_i) : A_i \multimap B_i$, and $B_i = A_j \otimes X$ holds for some X with $B_j \otimes X = B_j$ whenever $(n_i,n_j) \in \mathtt{A}$. Consequently for

$X_i = X_j = \mathbf{1}$ and $Z_i^j = X$, $B_i \otimes X_i = B_i = A_j \otimes X = A_j \otimes Z_i^j = A_j \otimes X_j \otimes Z_i^j$ and $B_j \otimes Z_i^j = B_j \otimes X = B_j$ follow.

□

Again, *minimal* unifiers are of particular interest.

Definition 20. *A unifier $\vec{X}$ for* $\mathtt{N}$ *and* $\mathtt{A}$ *is* minimal *if for any unifier $\vec{Y}$ for* $\mathtt{N}$ *and* $\mathtt{A}$ *there is a $\vec{Z}$ such that $\vec{X} \otimes \vec{Z} = \vec{Y}$.*

In particular, the register part of the minimal unifier $\vec{X}$ occurs in any unifier $\vec{Y}$, i.e. $Y_l \otimes rg(X_l) = Y_l$ holds in every component l.

In contrast to the case for operand queues, a unification is always possible, based on the following iterative approach. For ease of notation, we concentrate on exclusive register usage, i.e. require that all typings $t : A \multimap B$ fulfil $A = rg(A)$ and $B = rg(B)$.

Theorem 5. *Let $\vec{X}$ be the minimal unifier for* $\mathtt{N}$ *and* $\mathtt{A}$*, and for all $n \in dom\, \mathtt{N}$ let $\mathtt{N}(n) : A_n \multimap B_n$ with $A_n = rg(A_n)$ and $B_n = rg(B_n)$. Let $m, n \in \mathtt{N}$ and $a = (m,n) \notin \mathtt{A}$. Define*

$$S = \{l \,|\, (l,m) \in \mathtt{A}^*\} \text{ and } T = \{l \,|\, (n,l) \in \mathtt{A}^*\}$$

and let $(X_m \otimes B_m)/(A_n \otimes X_n) = V/U$ and $V/B_n = W/Y$. For each factor r of U let S_r be the least subset of S with

- $m \in S_r$
- *if $k \in S_r$, $(l,k) \in \mathtt{A}$ and $prime(r, B_l \otimes X_l)$ then $l \in S_r$*
- *if $k \in S_r$, $(k,l) \in \mathtt{A}$ and $prime(r, B_l \otimes X_l)$ then $l \in S_r$*

and for each factor r of W let T_r be the least subset of T with

- $n \in T_r$
- *if $k \in T_r$, $(l,k) \in \mathtt{A}$ and $prime(r, B_l \otimes X_l)$ then $l \in T_r$*
- *if $k \in T_r$, $(k,l) \in \mathtt{A}$ and $prime(r, B_l \otimes X_l)$ then $l \in T_r$*

Then for $Y_l = X_l \otimes \bigotimes_{l \in S_r} r \otimes \bigotimes_{l \in T_r} r$, the vector $\vec{Y}$ is the minimal unifier for $\mathtt{N}$ *and* $\mathtt{A} \cup \{a\}$.

Proof. **Show that $\vec{Y}$ is unifier for $\mathtt{N}$ and $\mathtt{A} \cup \{a\}$.** Let $(i,j) \in \mathtt{A} \cup \{a\}$. We show that there is a Z such that $B_i \otimes Y_i = A_j \otimes Y_j \otimes Z$ and $B_j \otimes Y_j \otimes Z = B_j \otimes Y_j$.

Case $(i,j) = a = (m,n)$ and $n \neq m$. By the definitions in the theorem, $m \in S_r$ holds for each factor r of U and $n \in T_r$ holds for each factor r of W. Therefore,

$$Y_m = X_m \otimes U \otimes W' \text{ and } Y_n = X_n \otimes W \otimes U'$$

holds for some factors $U' = \otimes_{n \in S_r} r$ of U and $W' = \otimes_{m \in T_r} r$ of W.

We first show $W' = \mathbf{1}$. Each factor r of W' fulfils $m \in T_r$, so by $n \neq m$, there must be a $k \in T_r$ such that $prime(r, B_m \otimes X_m)$ and $(k,m) \in \mathtt{A}$ or $(m,k) \in \mathtt{A}$. On the other hand, since r is a factor of W', r is a factor of W, thus of V and then of $B_m \otimes X_m$. Contradiction, so W' has no factors r, hence $Y_m = X_m \otimes U$. Similarly, each factor r of U' fulfils $n \in S_r$, so there must be a $k \in S_r$ with $(k,n) \in \mathtt{A}$ or $(n,k) \in \mathtt{A}$ and $prime(r, B_n \otimes X_n)$. On the other hand, r is a factor of U', hence of U. Since U occurs in $A_n \otimes X_n$ and A_n in B_n by Lemma 5, r occurs in $B_n \otimes X_n$. Contradiction, so U' has no factors r, hence $Y_n = X_n \otimes W$. Let Z be the common part of V and B_n, i.e. $V/B_n =_Z W/Y$. Then

$$B_m \otimes Y_m = B_m \otimes X_m \otimes U = A_n \otimes X_n \otimes V = A_n \otimes X_n \otimes Z \otimes W = A_n \otimes Y_n \otimes Z$$

and observing $Z = rg(Z)$ as Z occurs in B_n yields $B_n \otimes Y_n = B_n \otimes Y_n \otimes Z$.

Case $(i,j) = a = (m,n)$ and $n = m$. Since $X_n = X_m$ occurs both in $X_m \otimes B_m$ and $X_n \otimes A_n$ and $A_n = rg(A_n)$ occurs in $rg(B_n) = rg(B_m)$ Lemma 5 implies $U = \mathbf{1}$. For the same reason, V is a factor of B_m, hence $W = \mathbf{1}$ and $Y_n = Y_m = X_n$. Consequently,

$$B_m \otimes Y_m = B_m \otimes X_m = B_m \otimes X_m \otimes U = A_n \otimes X_n \otimes V = A_n \otimes Y_n \otimes V$$

so we take $Z = V$. Then $B_m \otimes Y_m \otimes Z = B_m \otimes Y_m \otimes V = B_m \otimes Y_m$ as $V = rg(V)$ occurs in B_n.

Case $n = j \neq i \neq m$. With the same argument as in the fist case, $U' = \mathbf{1}$ follows, so $Y_n = X_n \otimes W$. Since $\vec{X}$ unifies N for $\mathtt{A}$ and $(i,j) \neq a$ we have

$$B_i \otimes Y_i = B_i \otimes X_i \otimes \otimes_{i \in S_r} r \otimes \otimes_{i \in T_r} r = A_j \otimes X_j \otimes \overline{Z} \otimes \otimes_{i \in S_r} r \otimes \otimes_{i \in T_r} r$$

where $B_j \otimes X_j \otimes \overline{Z} = B_j \otimes X_j$.

For any r with $i \in S_r$, $prime(r, B_i \otimes X_i)$ holds due to $m \neq i$. Also such an r is a factor of U, so of $A_n \otimes X_j = A_j \otimes X_j$, hence

$$A_j \otimes X_j \otimes \otimes_{i \in S_r} r = A_j \otimes X_j.$$

Any r with $i \in T_r$ is a factor of W and $i \neq j = n$ implies $prime(r, B_i \otimes X_i)$, so W can be written as $W = W_a \otimes W_b$ where

$$W_a = \otimes_{i \in T_r, r \text{ factor of } W} r \text{ and } W_b = \otimes_{i \notin T_r, r \text{ factor of } W} r.$$

A factor r of W with $i \notin T_r$ occurs in $B_i \otimes X_i$ since $n \in T_r$ and $(i, j) \in \mathtt{A}$ holds due to $i \neq m$. Hence

$$B_i \otimes X_i = B_i \otimes X_i \otimes W_b.$$

Combining the equations and taking $Z = \overline{Z}$ we obtain

$$\begin{aligned}
B_i \otimes Y_i &= B_i \otimes X_i \otimes \otimes_{i \in S_r} r \otimes \otimes_{i \in T_r} r = B_i \otimes X_i \otimes \otimes_{i \in S_r} r \otimes W_a \\
&= B_i \otimes X_i \otimes W_b \otimes \otimes_{i \in S_r} r \otimes W_a = A_j \otimes X_j \otimes \overline{Z} \otimes W \otimes \otimes_{i \in S_r} r \\
&= A_j \otimes X_j \otimes \overline{Z} \otimes W = A_j \otimes Y_j \otimes \overline{Z} \\
&= A_j \otimes Y_j \otimes Z
\end{aligned}$$

and $B_j \otimes Y_j \otimes Z = B_j \otimes X_j \otimes W \otimes \overline{Z} = B_j \otimes X_j \otimes W = B_j \otimes Y_j$.

Case $i = j$. For $B_i \otimes X_i = A_j \otimes X_j \otimes \overline{Z}$ with $B_j \otimes X_j \otimes \overline{Z} = B_j \otimes X_j$ we obtain

$$\begin{aligned}
B_i \otimes Y_i &= B_i \otimes X_i \otimes \otimes_{i \in S_r} r \otimes \otimes_{i \in T_r} r = A_j \otimes X_j \otimes \overline{Z} \otimes_{i \in S_r} r \otimes \otimes_{i \in T_r} r \\
&= A_j \otimes X_j \otimes \overline{Z} \otimes_{j \in S_r} r \otimes \otimes_{j \in T_r} r = A_j \otimes Y_j \otimes \overline{Z}
\end{aligned}$$

so take $Z = \overline{Z}$. Then

$$\begin{aligned}
B_j \otimes Y_j \otimes Z &= B_j \otimes X_j \otimes \otimes_{j \in S_r} r \otimes \otimes_{j \in T_r} r \otimes \overline{Z} \\
&= B_j \otimes X_j \otimes \otimes_{j \in S_r} r \otimes \otimes_{j \in T_r} r = B_j \otimes Y_j
\end{aligned}$$

Case $i \neq j \neq n$. Any r with $j \in T_r$ fulfils

- r is a factor of W

- $prime(r, B_j \otimes X_j)$ since $j \neq n$, hence $prime(r, A_j \otimes X_j)$
- $i \in T_r$ or r a factor of $B_i \otimes X_i$ since $(i, j) \in \mathtt{A}$.

and any $i \in T_r$ fulfils

- $j \in T_r$ or r a factor of $B_j \otimes X_j$ and
- r occurs in W.

Consequently, any r with $i \in T_r$ but $j \notin T_r$ occurs in $B_j \otimes X_j$ and any r with $j \in T_r$ but $i \notin T_r$ occurs in $B_i \otimes X_i$.

Hence,

$$\begin{aligned} B_j \otimes X_j &= B_j \otimes X_j \otimes \bigotimes_{i \in T_r, j \notin T_r} r && (5.3) \\ B_i \otimes X_i &= B_i \otimes X_i \otimes \bigotimes_{j \in T_r, i \notin T_r} r && (5.4) \end{aligned}$$

Similarly, any r with $i \in S_r$ occurs in U and either fulfils $j \in S_r$ or occurs in $B_j \otimes X_j$ and any r with $j \in S_r$ is in U and either fulfils $i \in S_r$ or occurs in $B_i \otimes X_i$. Hence,

$$\begin{aligned} B_i \otimes X_i &= B_i \otimes X_i \otimes \bigotimes_{j \in S_r, i \notin S_r} r && (5.5) \\ B_j \otimes X_j &= B_j \otimes X_j \otimes \bigotimes_{i \in S_r, j \notin S_r} r && (5.6) \end{aligned}$$

Hence, for $B_i \otimes X_i = A_j \otimes X_j \otimes \overline{Z}$ and $B_j \otimes X_j \otimes \overline{Z} = B_j \otimes X_j$ we obtain

$$\begin{aligned} B_i \otimes Y_i &= B_i \otimes X_i \otimes \bigotimes_{i \in S_r} r \otimes \bigotimes_{i \in T_r} r \\ &= B_i \otimes X_i \otimes \bigotimes_{i \in S_r, j \in S_r} r \otimes \bigotimes_{i \in S_r, j \notin S_r} r \otimes \bigotimes_{i \in T_r, j \in T_r} r \otimes \bigotimes_{i \in T_r, j \notin T_r} r \\ &= B_i \otimes X_i \otimes \bigotimes_{j \in S_r, i \notin S_r} r \otimes \bigotimes_{i \in S_r, j \in S_r} r \otimes \bigotimes_{i \in S_r, j \notin S_r} r \\ &\quad \otimes \bigotimes_{i \in T_r, j \in T_r} r \otimes \bigotimes_{i \in T_r, j \notin T_r} r \\ &= B_i \otimes X_i \otimes \bigotimes_{j \in T_r, i \notin T_r} r \otimes \bigotimes_{j \in S_r, i \notin S_r} r \otimes \bigotimes_{i \in S_r, j \in S_r} r \otimes \bigotimes_{i \in S_r, j \notin S_r} r \\ &\quad \otimes \bigotimes_{i \in T_r, j \in T_r} r \otimes \bigotimes_{i \in T_r, j \notin T_r} r \\ &= A_j \otimes X_j \otimes \overline{Z} \otimes_{j \in T_r, i \notin T_r} r \otimes \bigotimes_{j \in S_r, i \notin S_r} r \\ &\quad \otimes \bigotimes_{i \in S_r, j \in S_r} r \otimes \bigotimes_{i \in S_r, j \notin S_r} r \otimes \bigotimes_{i \in T_r, j \in T_r} r \otimes \bigotimes_{i \in T_r, j \notin T_r} r \\ &= A_j \otimes X_j \otimes \overline{Z} \otimes \bigotimes_{j \in T_r} r \otimes \bigotimes_{j \in S_r} r \otimes \bigotimes_{i \in S_r, j \notin S_r} r \otimes \bigotimes_{i \in T_r, j \notin T_r} r \\ &= A_j \otimes Y_j \otimes \overline{Z} \otimes \bigotimes_{i \in S_r, j \notin S_r} r \otimes \bigotimes_{i \in T_r, j \notin T_r} r \end{aligned}$$

where the third equality holds by (5.5), the fourth by (5.4) and the other equalities result from reordering indices and applying the definitions of Y_i, Y_j and $\overline{Z}$. For $Z = \overline{Z} \otimes \otimes_{i \in S_r, j \notin S_r} r \otimes \otimes_{i \in T_r, j \notin T_r} r$ we consequently obtain $B_i \otimes Y_i = A_j \otimes Y_j \otimes Z$ and

$$\begin{aligned}
B_j \otimes Y_j \otimes Z &= B_j \otimes X_j \otimes \otimes_{j \in T_r} r \otimes \otimes_{j \in S_r} r \otimes \overline{Z} \otimes \otimes_{i \in S_r, j \notin S_r} r \\
&\quad \otimes \otimes_{i \in T_r, j \notin T_r} r \\
&= B_j \otimes X_j \otimes \otimes_{j \in T_r} r \otimes \otimes_{j \in S_r} r \otimes \overline{Z} \\
&= B_j \otimes X_j \otimes \otimes_{j \in T_r} r \otimes \otimes_{j \in S_r} r \\
&= B_j \otimes Y_j
\end{aligned}$$

where the second equality holds due to (5.3) and (5.6) and the other equalities follow from the definition of Y_j and $\overline{Z}$.

Show that $\vec{Y}$ is minimal. We show that for the minimal unifier $\vec{Z}$ of $\mathtt{A} \cup \{a\}$ the equation

$$\vec{Z} \otimes rg(\vec{Y}) = \vec{Z}$$

holds, i.e. that all l fulfil $Z_l \otimes rg(Y_l) = Z_l$. The claim follows from proving that each factor r of Y_l occurs in Z_l, so let r occur in Y_l. By the definition of Y_l, r occurs in X_l or $l \in S_r$ or $l \in T_r$ holds.

In the first case, we know that $\vec{Z}$ unifies $\mathtt{N}$ also for $\mathtt{A}$, but $\vec{X}$ is the minimal unifier for $\mathtt{N}$ and $\mathtt{A}$, so $Z_l \otimes rg(X_l) = Z_l$ and r consequently occurs in Z_l.

The other two cases are proven inductively, according to the definition of S_r and T_r.

Claim 1: Let $l \in S_r$ and r occur in Y_l. Then r is a factor of Z_l. The base case is $m = l$. For $m \in S_r$, r is a factor of U, hence of $A_n \otimes X_n$, and the definition of cancellation (Definition 17) implies $prime(r, B_m \otimes X_m)$. As $\vec{Z}$ unifies $\mathtt{A} \cup \{a\}$,

$$B_m \otimes Z_m = A_n \otimes Z_n \otimes Z \text{ for some } Z \text{ with } B_n \otimes Z_n \otimes Z = B_n \otimes Z_n$$

holds. Since $\vec{Z}$ unifies `A`, $Z_m \otimes rg(X_m) = Z_m$ and $Z_n \otimes rg(X_n) = Z_n$ hold, hence

$$B_m \otimes Z_m \otimes rg(X_m) = A_n \otimes Z_n \otimes rg(X_n) \otimes Z.$$

Since r is a factor of $A_n \otimes X_n$, r is also a factor of $B_m \otimes Z_m \otimes rg(X_m)$, so $prime(r, B_m \otimes X_m)$ implies that r occurs in Z_m.

Any $l \neq m$ with $l \in S_r$ due to $(l,k) \in A$ for some $k \in S_r$ and $prime(r, B_l \otimes X_l)$ fulfils

$$B_l \otimes Z_l = A_k \otimes Z_k \otimes Z \text{ for some } Z \text{ with } B_k \otimes Z_k \otimes Z = B_k \otimes Z_k$$

since $\vec{Z}$ unifies `N` for `A` $\cup \{a\}$. The definition of $\vec{Y}$ implies that r occurs in Y_k since $k \in S_r$. By induction hypothesis, r is thus a factor of Z_k, hence of $B_l \otimes Z_l$. Now $prime(r, B_l \otimes X_l)$ implies $prime(r, B_l)$, so r must occur in Z_l.

Any $l \neq m$, $l \in S_r$ due to $(k,l) \in A$ for some $k \in S_r$ and $prime(r, B_l \otimes X_l)$ fulfils

$$B_k \otimes Z_k = A_l \otimes Z_l \otimes Z \text{ for some } Z \text{ with } B_l \otimes Z_l \otimes Z = B_l \otimes Z_l$$

since $\vec{Z}$ unifies `N` for `A` $\cup \{a\}$. Again, $Z_l = Z_l \otimes rg(X_l)$ holds as $\vec{Z}$ unifies `A`, and $Z_k = Z_k \otimes rg(Y_k)$ holds by induction hypothesis, and $k \in S_r$ implies that r occurs in $rg(Y_k)$ by the definition of $\vec{Y}$. Thus, r is a factor of

$$B_k \otimes Z_k \otimes rg(Y_k) = B_k \otimes Z_k = A_l \otimes Z_l \otimes Z = A_l \otimes Z_l \otimes rg(X_l) \otimes Z.$$

On the other hand, r cannot occur in $A_l \otimes X_l$ as $B_l = B_l \otimes rg(A_l)$ holds by Lemma 5 and $prime(r, B_l \otimes X_l)$. So r must be a factor of $Z \otimes Z_l$ and hence of Z_l as $B_l \otimes Z_l \otimes Z = B_l \otimes Z_l$ holds and $prime(r, B_l)$.

Claim 2: Let $l \in T_r$ and r occur in Y_l. Then r is a factor of Z_l. The base case is $n = l$. For $n \in T_r$, r is a factor of W, hence of V and $B_m \otimes X_m$, and the definition of cancellation (Definition 17) implies $prime(r, A_n \otimes X_n)$ and $prime(r, B_n)$. As $\vec{Z}$ unifies `A` $\cup \{a\}$,

$$B_m \otimes Z_m = A_n \otimes Z_n \otimes Z \text{ for some } Z \text{ with } B_n \otimes Z_n \otimes Z = B_n \otimes Z_n$$

holds. Since $\vec{Z}$ unifies `A`, $Z_m \otimes rg(X_m) = Z_m$ and $Z_n \otimes rg(X_n) = Z_n$ follow, hence

$$B_m \otimes Z_m \otimes rg(X_m) = A_n \otimes Z_n \otimes rg(X_n) \otimes Z.$$

Since r is a factor of $B_m \otimes X_m$, r is thus a factor of $A_n \otimes Z_n \otimes rg(X_n) \otimes Z$. Now $prime(r, B_n)$ and $prime(r, A_n \otimes X_n)$ imply that r must occur in $Z_n \otimes Z$ and hence in Z_n since $B_n \otimes Z_n \otimes Z = B_n \otimes Z_n$ and $prime(r, B_n)$.

The two step cases are proven similarly to the first claim.

□

Type inference for programs involving registers and operand queues proceeds by combining the approaches described in Theorems 2 and 5. First, type inference is attempted with respect to the operand queues, but might fail. In case of success, type inference is carried out for the register components, and will always succeed.

5.2 Distributed execution

5.2.1 Dynamic semantics

The computational model for distributed execution involving registers mirrors the approach described in Chapter 4. Execution in the processor is highly unsupervised and a type system ensures deterministic execution. Distributed programs π are parallelly composed sequential programs, and the operational model is defined as interleaved execution by embedding the sequential rules from the previous section via the rules

$$\text{R-FU}_1 \frac{\mathsf{C} \xrightarrow{[n]code} \mathsf{D}}{\mathsf{C} \xrightarrow{[n]code}_{fu} \mathsf{D}} \; FU(code) = fu \qquad \text{R-FU}_2 \frac{\mathsf{C} \xrightarrow{s}_{fu} \mathsf{E} \quad \mathsf{E} \xrightarrow{t}_{fu} \mathsf{D}}{\mathsf{C} \xrightarrow{st}_{fu} \mathsf{D}}$$

$$\text{R-DIS}_1 \frac{\mathsf{C} \xrightarrow{t}_{fu} \mathsf{D}}{\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}} \qquad \text{R-DIS}_2 \frac{\mathsf{C} \xrightarrow{\pi_1} \mathsf{E} \quad \mathsf{E} \xrightarrow{\pi_2} \mathsf{D}}{\mathsf{C} \xrightarrow{\pi_1 \pi_2} \mathsf{D}}$$

As before, instructions are inserted into instruction queues according to the functional unit they execute in. Consequently, the relationship between sequential and distributed execution is as follows.

Proposition 24. *1. If* $\mathsf{C} \xrightarrow{t} \mathsf{D}$ *then* $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$.

2. *If* $\mathsf{C} \xrightarrow{\pi} \mathsf{D}$ *then there is a* t *such that* $\pi_t = \pi$ *and* $\mathsf{C} \xrightarrow{t} \mathsf{D}$.

3. *If* $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$, $t = ins_1 \ldots ins_n$ *and* $ins_i = [n_i]code_i$ *then* $(\otimes_{i=1}^{n} A_{code_i}) \otimes shape(\mathsf{D}) = (\otimes_{i=1}^{n} B_{code_i}) \otimes shape(\mathsf{C})$.

Proof. The first two parts are proven exactly as those of Proposition 7. The third part proceeds similarly to the proof of Proposition 7, too, but in the base case of the induction (rule INSTR), $ins_1 : shape(\mathsf{C}) \multimap shape(\mathsf{D})$ holds by Theorem 4, and rule R-AX yields $shape(\mathsf{C}) = A_{code_1} \otimes X$ and $shape(\mathsf{D}) = B_{code_1} \otimes X \otimes rg(A_{code_1})$ for some X. We obtain

$$\begin{aligned} A_{code_1} \otimes shape(\mathsf{D}) &= A_{code_1} \otimes B_{code_1} \otimes X \otimes rg(A_{code_1}) \\ &= A_{code_1} \otimes B_{code_1} \otimes X = B_{code_1} \otimes shape(\mathsf{C}) \end{aligned}$$

as claimed, using the identity $r \otimes r = r$. The step case (rule COMP) is again proven as before. □

Likewise, soundness of the sequential static semantics with respect to distributed execution follows as in Chapter 4.

Proposition 25. *If* $t : A \multimap B$ *and* $shape(\mathsf{C}) = A$ *then there is a* D *such that* $\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}$ *and* $shape(\mathsf{D}) = B$.

Proof. Exactly as the proof of Proposition 8. □

Again, completeness is lost, with additional non-determinism resulting from registers. In fact, the following three types of race conditions may occur for instructions $[n]code_n$ and $[m]code_m$ if $[n]$ occurs before $[m]$ in t but $FU(code_n) \neq FU(code_m)$.

output dependence: also known under the name *write-after-write*, this conflict occurs if $[n]$ and $[m]$ compete for writing to the same location. In addition to the analysis in Chapter 4 we now have to include registers, as the program $t_{\texttt{WAW}} = [1]\texttt{inc}\ \texttt{q}_1\ \texttt{r}_1\ [2]\texttt{mul}\ \texttt{q}_2\ \texttt{q}_3\ \texttt{r}_1$ shows.

true dependence: (*read-after-write*) occurs if $[m]$ might read from a location before $[n]$ has written the intended operand to that location. For example, the program $t_{\texttt{RAW}} = [1]\texttt{ldc}\ 2\ \texttt{r}_1\ [2]\texttt{inc}\ \texttt{r}_1\ \texttt{r}_1\ [3]\texttt{mul}\ \texttt{r}_1\ \texttt{r}_1\ \texttt{q}_1$ contains a race condition between the second and the third instruction.

anti-dependence: (*write-after-read*) occurs if $[m]$ might write to a location before $[n]$ has read the previous content. In $t_{\mathtt{WAR}} = [1]\mathtt{ldc}\ 2\ \mathtt{r}_1\ [2]\mathtt{inc}\ \mathtt{r}_1\ \mathtt{q}_1\ [3]\mathtt{ldc}\ 3\ \mathtt{r}_1$ for example, a race occurs involving the second and the third instruction.

The fourth possibility where $[n]$ and $[m]$ read from the same register in the order $[m]$ *before* $[n]$ is uncritical as they access the same value. Remember that such *read-after-read* conflicts cannot occur for operand queues due to the sequential order of instructions in instruction queues.

All example programs are well-typed, with principal types

$$\begin{aligned}
t_{\mathtt{WAW}} &: \mathtt{q}_1 \otimes \mathtt{q}_2 \otimes \mathtt{q}_3 \multimap \mathtt{r}_1 \\
t_{\mathtt{RAW}} &: \mathbf{1} \multimap \mathtt{r}_1 \otimes \mathtt{q}_2 \\
t_{\mathtt{WAR}} &: \mathbf{1} \multimap \mathtt{r}_1 \otimes \mathtt{q}_2
\end{aligned}$$

Again, hardware solutions can be introduced for discovering race hazards at runtime and delay the execution of particular instructions. In fact, modern processors achieve similar tasks by register renaming and register locking [HP96].

Definition 21. *A distributed program* π *is* deterministic *if* $\mathsf{C} \xrightarrow{\pi} \mathsf{D}$ *and* $\mathsf{C} \xrightarrow{\pi} \mathsf{E}$ *implies* $\mathsf{D} = \mathsf{E}$.

With the same motivation as before, we first explore the sources of non-determinism before choosing a solution for eliminating race hazards.

5.2.2 Determinism, safety and completeness

For the same definitions of determinism, safety and completeness of programs t, the results of Section 4.2 transfer literally. In the proofs, the references to Propositions 7 and Theorem 1 are replaced by invoking Proposition 24 and Theorem 4 respectively.

5.2.3 A static semantics for determinism

Non-determinism may be dealt with by various means, similarly to the discussion in Chapter 4. We again outline how the static semantics can be extended to detect race conditions, extending the approach from Section 4.3. We equip judgements $t : A \multimap B$ with contexts

$$\Gamma = (\Gamma^W, \Gamma^R)$$

and sequentiality constraints $<$. Both components of Γ contain entries of the form $op : (l,k)$, where the l and $k \in \mathbf{N}$ denote (respectively) the first and last *writer* to op in case of Γ^W and the first and last *reader* from op in case of Γ^R. For Γ^R it suffices to consider only entries for op of the form r. As before, $<$ is a partial order over $\mathbf{N}$.
The typing axiom and the cut rule are modified to

$$\text{R-AX}^{<}\ \frac{ins : A \multimap B}{(\Gamma_{ins}, \emptyset) \vdash ins : A \multimap B}$$

$$\text{R-CUT}^{<}\ \frac{\begin{array}{c}(\Gamma, \subset) \vdash s : A \multimap B \\ (\Delta, \prec) \vdash t : B \multimap C\end{array}}{(\Gamma \# \Delta, <) \vdash st : A \multimap C}$$

where # is defined by applying # from Section 4.3.1 in both components and $\Gamma_{ins} = (\Gamma^W_{ins}, \Gamma^R_{ins})$ and $<$ are given by

$$\begin{array}{rcl}
\Gamma^W_{[n]code} & = & \{op : (n,n) \mid \exists B. B_{code} = op \otimes B\} \\
\Gamma^R_{[n]code} & = & \{r : (n,n) \mid rg(A_{code}) = r \otimes rg(A_{code})\} \\
< & = & (\subset \cup \prec \\
& & \cup \{(k,g) \mid op : (l,k) \in \Gamma^W, op : (g,h) \in \Delta^W\} \\
& & \cup \{(k,g) \mid op : (l,k) \in \Gamma^W, op : (g,h) \in \Delta^R\} \\
& & \cup \{(k,g) \mid op : (l,k) \in \Gamma^R, op : (g,h) \in \Delta^W\})^+
\end{array}$$

In the definition of $<$ each of the three sets corresponds to one type of conflict involving registers and the first set contains additionally the constraints for operand queues.
We obtain similar results as in Section 4.3.1.

Lemma 6. *1. $t : A \multimap B$ holds exactly if there are Γ and $<$ such that $(\Gamma, <) \vdash t : A \multimap B$.*

2. If $(\Gamma, <) \vdash t : A \multimap B$ and $(\Delta, \prec) \vdash t : C \multimap D$ then $\Gamma = \Delta$ and $< = \prec$.

3. *If* $((\Gamma^W, \Gamma^R), <) \vdash t : A \multimap B$, *then*

   - *all labels* n *occuring in* $\Gamma = (\Gamma^W, \Gamma^R)$ *or* $<$ *are labels of instructions in* t.
   - *if* $n < m$ *then* t *can be written as* $s\,[n]code_n\, r\,[m]code_m\, u$*, there a common factor op of both* $B_{code_n} \otimes rg(A_{code_n})$ *and* $B_{code_m} \otimes rg(A_{code_m})$.
   - $op : (n, \_) \in \Gamma^W$ (Γ^R) *iff* $[n]$ *is the first instruction* $[m]code$ *in* t *for which op is a factor of* B_{code} $(rg(A_{code}))$.
   - $op : (\_, n) \in \Gamma^W$ (Γ^R) *iff* $[n]$ *is the last instruction* $[m]code$ *in* t *for which op is a factor of* B_{code} $(rg(A_{code}))$.
   - $op : (n, n) \in \Gamma^W$ (Γ^R) *iff* $[n]$ *is the only instruction* $[m]code$ *in* t *for which op is a factor of* B_{code} $(rg(A_{code}))$.

4. *If* $(\Gamma, <) \vdash t : A \multimap B$, t *has the form* $s\,[n]code_1\, r\,[m]code_2\, u$ *and op is a factor of*

   - B_{code_1} *and* B_{code_2} *or*
   - $rg(A_{code_1})$ *and* B_{code_2} *or*
   - B_{code_1} *and* $rg(A_{code_2})$

   then $n < m$ *holds.*

Proof. Proceeds similarly to the Proof of Lemma 3 by an induction on t, proving all four claims simultaneously. □

Lemma 7. *Let* $(\Gamma, <) \vdash t : A \multimap B$, $shape(\mathsf{C}) = A$, *and* $u = ins\, u'$ *an interleaving of* t *and* A*. Let* t *be sequential for* $\sqsubset$ *and* A*, let* $<$ *be contained in* $\sqsubset$*, and let* $\mathsf{C} \xrightarrow{ins} \mathsf{D}_1$ *and* $t = t_1\, ins\, t_2$*. Then there are* Δ *and* $\prec$ *such that for* $C = shape(\mathsf{D}_1)$ *and* $s = t_1 t_2$ *the derivation*

$$\frac{(\Gamma_{ins}, \emptyset) \vdash ins : A \multimap C \quad (\Delta, \prec) \vdash s : C \multimap B}{(\Gamma_{ins} \# \Delta, <) \vdash ins\, s : A \multimap B}$$

exists. In particular, $\prec$ *is contained in* $<$.

Proof. Since u is an interleaving for t and A, t is sequential for $\sqsubset$ and A and $shape(\mathsf{C}) = A$ holds, there is a (unique) D such that $\mathsf{C} \xrightarrow{ins} \mathsf{D}_1 \xrightarrow{u'} \mathsf{D}$. Theorem 4 implies $ins : A \multimap C$, hence

$$(\Gamma_{ins}, \emptyset) \vdash ins : A \multimap C$$

holds by the rule R-AX$^<$.

By Lemma 6, $(\Gamma,<) \vdash t : A \multimap B$ implies $t : A \multimap B$, so there is by Theorem 4 a unique E with $\mathsf{C} \xrightarrow{t} \mathsf{E}$, and $shape(\mathsf{E}) = B$ holds. The typing $t : A \multimap B$ also implies that there are D and E such that

$$\dfrac{\dfrac{t_1 : A \multimap D \quad ins : D \multimap E}{t_1\, ins : A \multimap E} \qquad t_2 : E \multimap B}{t : A \multimap B}$$

is a valid derivation. Since $ins : D \multimap E$ holds, we know that there is an X with

$$D = A_{code} \otimes X \text{ and } E = B_{code} \otimes rg(A_{code}) \otimes X$$

where A_{code} and B_{code} are those from Table 3.1, and $ins = [n]code$. On the other hand, $ins : A \multimap C$ implies $A = A_{code} \otimes Y$ and $C = B_{code} \otimes rg(A_{code}) \otimes Y$ for some Y, hence

$$t_1 : A_{code} \otimes Y \multimap A_{code} \otimes X \text{ and } t_2 : B_{code} \otimes rg(A_{code}) \otimes X \multimap B.$$

Since ins is the head instruction of its pipeline, instructions in t_1 don't access elements in the queue component of A_{code}, so $q(A_{code})$ is used as a weakening in the typing of t_1, i.e. we have $t_1 : rg(A_{code}) \otimes Y \multimap X \otimes rg(A_{code})$. By applying the alternative weakening B_{code}, we obtain

$$\dfrac{\dfrac{t_1 : Y \otimes B_{code} \otimes rg(A_{code}) \multimap X \otimes B_{code} \otimes rg(A_{code})}{t_1 : C \multimap E} \qquad t_2 : E \multimap B}{s : C \multimap B}$$

so $(\Delta,\subset) \vdash s : C \multimap B$ for some Δ and $\subset$ by Lemma 6. Consequently, $(\Gamma_{ins}\#\Delta,\prec) \vdash ins\, s : A \multimap B$ for

$$\begin{aligned} \prec \quad = \quad & (\subset \cup \{(n,m) \,|\, op \text{ factor of } B_{code}, op : (m,\_) \in \Delta^W\} \\ & \cup \{(n,m) \,|\, r \text{ factor of } B_{code}, r : (m,\_) \in \Delta^R\} \\ & \cup \{(n,m) \,|\, r \text{ factor of } A_{code}, r : (m,\_) \in \Delta^W\})^+ \end{aligned}$$

Instruction sequences u and t are both interleavings for t and $A - u$ by assumption and t due to $\pi_t = \pi_t$ and $t : A \multimap B$. Since t is sequential for $\sqsubset$ and A, u and t thus both respect $\sqsubset$, and the claim follows from the minimality of $[n]$ with respect to $\sqsubset$ as in the proof of Lemma 4. $\square$

Proposition 26. *Let $(\Gamma,<) \vdash t : A \multimap B$ and $<$ be contained in $\sqsubset$. If t is sequential for $\sqsubset$ and A then t is deterministic for A.*

Proof. The proof proceeds similarly to the one of Proposition 11. In the case $FU(u_1) \neq FU(t_1)$, the instructions u_1 and t_1 with the principal typings $t_1 :: A_{t_1} \multimap B_{t_1}$ and $u_1 :: A_{u_1} \multimap B_{u_1}$ are not completely independent but $prime(rg(A_{u_1}), B_{t_1})$, $prime(B_{u_1}, rg(A_{t_1}))$ and $prime(B_{u_1}, B_{t_1})$ hold: for suppose any of these claims is violated, then $t_1 < u_1$ holds by Lemma 6(part (4)) (as t_1 is the first instruction of t), in contradiction to the fact that u respects $<$.
Any joint factor of A_{u_1} and A_{t_1} is a register r (because of $FU(u_1) \neq FU(t_1)$), and the values in r in configurations D_1 and E_1 agree. Formally, such an r occurs in $shape(\mathsf{D}_1)$ and $shape(\mathsf{E}_1)$ as $B_{t_1} = B_{t_1} \otimes rg(A_{t_1})$ and $B_{u_1} = B_{u_1} \otimes rg(A_{u_1})$ hold. Hence executability of u_1 in D_1 and t_1 in E_1 is preserved, and $prime(B_{u_1}, B_{t_1})$ implies the existence of a unique C_1 such that $\mathsf{D}_1 \xrightarrow{t_1} \mathsf{C}_1$ and $\mathsf{E}_1 \xrightarrow{u_1} \mathsf{C}_1$. $\square$

The increased number of race hazards resulting from the introduction of registers is not matched by the existence of additional tools for discharging constraints. Pipeline-dependencies still result in serialisation, and so do data-dependencies for values which are communicated through operand queues. No serialisation is achieved by register-communicated values, exemplified by the existence of RAW hazards.

5.2.4 Program execution

The distributed dynamic semantics of programs with registers and jumps combines the SOS systems for distributed program execution (Section 4.4.1) and program execution involving registers (Section 5.1.3.1). Configurations are equipped with a register component and reduction is defined by the rules in Figure 5.4.
Likewise, the static semantics is given by composing the approaches in Sections 4.4.2 and 5.1.3.2. Instructions `if` and `jmp` are typed according to the rules

$$\frac{[n]\mathtt{if}\ op\ n_1\ n_2 : A \multimap B}{(\Gamma_{\mathtt{if}\ op\ n_1\ n_2}, \emptyset) \vdash [n]\mathtt{if}\ op\ n_1\ n_2 : A \multimap B} \quad \text{and} \quad \frac{}{(\emptyset,\emptyset) \vdash [n]\mathtt{jmp}\ m : X \multimap X}$$

At boundaries of basic blocks, additional constraints are introduced according to the three race conditions. Judgements $(\Gamma_n, <_n) \vdash n : A \multimap B$ are inserted into Σ provided

$$\text{4FU}_1 \frac{\mathsf{C} \xrightarrow{ins} \mathsf{D}}{\mathsf{C} \xrightarrow{ins}_{fu} \mathsf{D}} \begin{cases} ins = [n]code \\ FU(code) = fu \end{cases} \qquad \text{4FU}_2 \frac{\mathsf{C} \xrightarrow{s}_{fu} \mathsf{E} \quad \mathsf{E} \xrightarrow{t}_{fu} \mathsf{D}}{\mathsf{C} \xrightarrow{st}_{fu} \mathsf{D}}$$

$$\text{4DIS}_1 \frac{\mathsf{C} \xrightarrow{t}_{fu} \mathsf{D}}{\mathsf{C} \xrightarrow{\pi_t} \mathsf{D}} \qquad \text{4DIS}_2 \frac{\mathsf{C} \xrightarrow{\pi_1} \mathsf{E} \quad \mathsf{E} \xrightarrow{\pi_2} \mathsf{D}}{\mathsf{C} \xrightarrow{\pi_1\pi_2} \mathsf{D}}$$

$$\text{LOAD} \frac{}{(\mathsf{Q},\mathsf{R},\mathsf{M},n,\pi) \xrightarrow{n} (\mathsf{Q},\mathsf{R},\mathsf{M},nil,\pi\pi_t)} \ \mathtt{N}(n) = t$$

$$\text{EXEC} \frac{(\mathsf{Q},\mathsf{R},\mathsf{M},\widetilde{n}) \xrightarrow{\pi_1} (\mathsf{P},\mathsf{S},\mathsf{L},\widetilde{m})}{(\mathsf{Q},\mathsf{R},\mathsf{M},\widetilde{n},\pi_1\pi_2) \xrightarrow{\lambda} (\mathsf{P},\mathsf{S},\mathsf{L},\widetilde{m},\pi_2)}$$

$$\text{COMPOSE} \frac{\mathsf{C} \xrightarrow{v} \mathsf{E} \quad \mathsf{E} \xrightarrow{w} \mathsf{D}}{\mathsf{C} \xrightarrow{vw} \mathsf{D}}$$

Figure 5.4: Dynamic semantics for programs involving registers

that for the typing $(\Gamma_n, <_n) \vdash \mathtt{N}(n) : A \multimap B$ holds, and constraints $[l] < [g]$ are inserted whenever one of the following three conditions holds where $n, \ldots, m$ are related by a path in $\mathtt{A}$

- $op : (k,l) \in \Gamma_n^W, op : (g,h) \in \Gamma_m^W$
- $r : (k,l) \in \Gamma_n^R, r : (g,h) \in \Gamma_m^W$
- $r : (k,l) \in \Gamma_n^W, r : (g,h) \in \Gamma_m^R$

Again, it suffices to consider loop-free paths where at most the endpoint are equal and no intermediate block mentions op. As before, constraints $[n] < [n]$ in the rule

$$\text{R-DisPROG} \frac{\begin{array}{c} \Sigma = \cup_{n \in dom\,\mathtt{N}} \{(\Gamma_n, <_n) \vdash n : A_n \multimap B_n\} \\ (n,m) \in \mathtt{A} \text{ implies } \exists X \text{ s.t. } B_n = A_m \otimes X \text{ and } B_m \otimes X = B_m \\ SP(n,op,m), op : (k,l) \in \Gamma_n^W, op : (g,h) \in \Gamma_m^W \text{ implies } [l] < [g] \\ SP(n,op,m), op : (k,l) \in \Gamma_n^W, op : (g,h) \in \Gamma_m^R \text{ implies } [l] < [g] \\ SP(n,op,m), op : (k,l) \in \Gamma_n^R, op : (g,h) \in \Gamma_m^W \text{ implies } [l] < [g] \end{array}}{(\Sigma, <) \vdash (\mathtt{N}, \mathtt{A}) : \square}$$

are interpreted modulo loop instantiations. Consequently, similar properties hold as in Section 4.4.2.

Proposition 27. *Let* $(\Sigma,<) \vdash \mathtt{P} : \Box$ *and* $(n_1,\ldots,n_k)$ *be a path in* $\mathtt{A}$. *Then there are* Γ *and* $\prec$ *such that* $(\Gamma,\prec) \vdash \mathtt{N}(n_1)\ldots\mathtt{N}(n_k) : A_{n_1} \multimap B_{n_k}$ *holds and* $\prec$ *is contained in* $\{<_n \mid (\Gamma_n,<_n) \vdash n : A_n \multimap B_n \in \Sigma\} \cup <$.

Proof. Similar to the proof of Proposition 15. In the case $k > 1$, $B_i = A_{i+1} \otimes X_i$ holds for some X_i with $B_{i+1} \otimes X_i = B_{i+1}$ and all $1 \le i \le k$, and $\prec$ has the form

$$\begin{aligned} \prec \;=\; & \sqsubset \cup <_{k+1} \cup \{(m,g) \mid op : (l,m) \in \Delta^W \text{ and } op : (g,h) \in \Gamma^W_{k+1}\} \\ & \cup \{(m,g) \mid r : (l,m) \in \Delta^W \text{ and } r : (g,h) \in \Gamma^R_{k+1}\} \\ & \cup \{(m,g) \mid r : (l,m) \in \Delta^R \text{ and } r : (g,h) \in \Gamma^W_{k+1}\}. \end{aligned}$$

□

Proposition 28. *Let* $(\Sigma,<) \vdash \mathtt{P} : \Box$, $n \in dom\,\mathtt{N}$ *and* $(\Gamma_n,<_n) \vdash \mathtt{N}(n) : A_n \multimap B_n \in \Sigma$.

1. *If* $(\mathsf{Q},\mathsf{R},\mathsf{M},n) \xrightarrow{w} (\mathsf{P},\mathsf{S},\mathsf{L},\widetilde{m})$ *then* $(\mathsf{Q},\mathsf{R},\mathsf{M},n,\pi_\varepsilon) \xrightarrow{w} (\mathsf{P},\mathsf{S},\mathsf{L},\widetilde{m},\pi_\varepsilon)$.
2. *Let* $\mathsf{C} = (\mathsf{Q},\mathsf{R},\mathsf{M},n,\pi_\varepsilon)$, *shape*$(\mathsf{C}) = A_n$ *and* $\mathsf{C} \xrightarrow{w} (\mathsf{P},\mathsf{S},\mathsf{L},\widetilde{m},\pi_\varepsilon)$. *Let* $\sqsubset$ *be a partial order containing* $<$ *and for all* $SP(\mathtt{N}(n_i),op,\mathtt{N}(n_j))$ *let the instruction sequence* $\mathtt{N}(N_i)\ldots\mathtt{N}(n_j) : A_{n_i} \multimap B_{n_j}$ *be sequential for* $\sqsubset$ *and* A_{n_i}. *Then* $(\mathsf{Q},\mathsf{R},\mathsf{M},n) \xrightarrow{w} (\mathsf{P},\mathsf{S},\mathsf{L},\widetilde{m})$.

Proof. Similar to the proof of Proposition 16, using configurations extended by the register bank and the modified rules EXEC, LOAD, EXECUTE and COMPOSE. □

5.3 Discussion

This section gave dynamic and static semantics for the full language ALEF, again for sequential as well as distributed execution. Most properties were obtained by transferring and combining the results from earlier chapters, and the ease with which this was possible demonstrates the robustness of our approach. In the type system, registers were modelled as products fulfilling the axiom $r \otimes r = r$ which turns them into exponentials in the setting of linear logic [Gir86]. Indeed, accessing a register corresponds to the familiar axiom $!r \multimap r \otimes !r$, and for $A = op_1 \otimes \ldots \otimes op_n$, our notation $rg(A)$ represents the weakest formula $A^?$ such that $A \otimes A^? = !A$ holds, where $!A = op_1^! \otimes \ldots op_n^!$ and $op^! = \begin{cases} !r & \text{if } op = r \\ op & \text{otherwise} \end{cases}$

Our approach employs only a tiny fragment of linear logic as types are (non-curried) first-order, no connectives other than $\otimes$ and $\multimap$ appear and the exponential is only applied to atoms op. We have not investigated further connections to linear proof theory, but doing so might be useful for modelling more complex forwarding behaviour.

5.3.1 AQM's and compounding

The static and dynamic semantics presented in this chapter allows us to examine the compounding approach more closely. The description given in Chapter 2 suggests that communication within a compound should coincide with operand queue usage while inter-compound communication should correspond to register usage. This motivates us to require that compounds have type $rg(A) \multimap rg(B)$ for some A and B. However, this property does not characterise compounds uniquely. The program

$$[1]\mathtt{dupl}^{\mathtt{MEM}}\,\mathtt{r}_1\,\mathtt{q}_1\,\mathtt{q}_2\;[2]\mathtt{dec}\,\mathtt{q}_1\,\mathtt{q}_3\;[3]\mathtt{add}\,\mathtt{q}_2\,\mathtt{q}_3\,\mathtt{r}_2 :: \mathtt{r}_1 \multimap \mathtt{r}_1 \otimes \mathtt{r}_2$$

is not a compound in the sense of [Mul01] as there are two forwarding paths between [1] and [3]. ALEF notation highlights the difference between linear chains of instructions in the instruction dependence graph and a chain of data-dependencies.

A second issue concerns the underlying dynamic semantics. The results of this chapter show that differences in the dynamic semantics influence what notions of compounding should be admitted. The following examples demonstrate the problems arising from defining compounds as *units of forwarding* but understanding and using them also as *units of scheduling*. The program

$$t = [1]\mathtt{ldc}\,5\,\mathtt{q}_1\;[2]\mathtt{dupl}^{\mathtt{BU}}\,\mathtt{q}_1\,\mathtt{q}_2\,\mathtt{r}_1\;[3]\mathtt{dec}\,\mathtt{r}_1\,\mathtt{r}_2\;[4]\mathtt{mul}\,\mathtt{q}_2\,\mathtt{r}_2\,\mathtt{r}_3 \tag{5.7}$$

consists of two compounds, $C_1 = [1]\,[2]\,[4]$ and $C_2 = [3]$. Both schedules $C_1 C_2$ and $C_2 C_1$ execute deterministically for an initial configuration of shape **1** under the distributed model of execution, resulting in the same final configuration. In fact,

$$\pi_{C_1 C_2} = \pi_t = \pi_{C_2 C_1} = [3] \parallel [4] \parallel [1] \parallel [2]$$

holds and *any* issue order of the instructions of t leads to this distributed program. However, under the sequential model, *both* schedules deadlock. The order $C_1 C_2$ results in the interleaving [1] [2] [4] [3] and deadlocks after executing [2]. The order $C_2 C_1$ results in the interleaving [3] [1] [2] [4] and deadlocks immediately. On the other hand, the *in-order* execution of t succeeds. The above compounding was perfectly fine for distributed execution but not for sequential one.

For sequential execution, additional constraints on the formation of compounds have been proposed. For example, *acyclic* compoundings require the graph of compounds and and inter-compound dependencies to be loop-free. This eliminates program (5.7) where C_2 and C_1 are mutually dependent on each other. Of course, this requirement is conservative as there are cyclic compoundings which may be admitted. On the other hand, for a distributed underlying semantics in the sense of this Chapter, even acyclic compounding does not guarantee determinism. For example, the program

$$\begin{aligned} t \quad = \quad & [1]\mathtt{ldc}\ 5\ \mathtt{q}_1\ [2]\mathtt{dupl}^{\mathtt{ALU}}\ \mathtt{q}_1\ \mathtt{r}_1\ \mathtt{r}_2\ [3]\mathtt{dupl}^{\mathtt{MUL}}\ \mathtt{r}_2\ \mathtt{r}_4\ \mathtt{q}_2 \\ & [4]\mathtt{ld}\ \mathtt{q}_2\ \mathtt{r}_3\ [5]\mathtt{ld}\ \mathtt{r}_1\ \mathtt{q}_3\ [6]\mathtt{dec}\ \mathtt{q}_3\ \mathtt{r}_5\ [7]\mathtt{add}\ \mathtt{r}_5\ \mathtt{r}_4\ \mathtt{r}_3 \end{aligned} \tag{5.8}$$

is compounded acyclicly, with $C_1 = [1]\,[2]$, $C_2 = [3]\,[4]$, $C_3 = [5]\,[6]$ and $C_4 = [7]$. The two schedules respecting the inter-compound dependencies are $C_1 C_2 C_3 C_4$ and $C_1 C_3 C_2 C_4$. For initial configurations of shape $\mathbf{1}$, both schedules execute successfully under the sequential model of execution, and indeed lead to the same result. However, for distributed execution only the schedule $C_1 C_2 C_3 C_4$ is deterministic, while $C_1 C_3 C_2 C_4$ shows a race condition between instructions [4] and [7]. In contrast, the original program t is deterministic for $\mathbf{1}$ as any interleaving of $\pi_t = [2]\,[6]\,[7] \parallel [3] \parallel [1]\,[4]\,[5] \parallel \varepsilon$ agrees with the sequential execution. The typing of t is

$$(\Gamma, [4] < [7]) \vdash t : \mathbf{1} \multimap \mathtt{r}_1 \otimes \mathtt{r}_2 \otimes \mathtt{r}_3 \otimes \mathtt{r}_4 \otimes \mathtt{r}_5,$$

and the constraint $[4] < [7]$ may be discharged using the $\mathtt{q}_3$-carried data-dependence between [5] and [6] and the pipeline-dependencies $[4] \lessdot [5]$ and $[6] \lessdot [7]$. Thanks to the SOS- and type-based formalism, we discover that the race hazard is introduced through understanding compounds as units of scheduling. Type-checking the schedule $C_1 C_3 C_2 C_4$ fails as the constraint $[4] < [7]$ cannot be fulfilled any longer: $[4] \lessdot [5]$ does not hold.

We hence argue that AQM's are well suited for reasoning about compoundings as they capture the interaction between forwarding, model of execution and scheduling clearly.

5.3.2 Conclusion and outlook

The preceding chapters presented a programming language based analysis of forwarding. We introduced AQM's as a novel machine model, realised as an assembly language in which operand queue names have the same status as registers. Explicitness of forwarding was shown to be beneficial for defining processor behaviour at various levels of abstraction, ranging from ISA-style sequential execution with operand queues of unbounded length and exclusive forwarding of operands to distributed execution with registers and operand queues of bounded length. We discussed how AQM's and ALEF relate to three previous models of operand communication (SCALP, compounding, and Tomasulo's algorithm), demonstrating that AQM's unify various concrete realisations of forwarding. The benefits of our approach became apparent as the dynamic semantics were complemented by static semantics eliminating characteristic runtime hazards. The core of these type systems remained static as we proceeded from simple towards more complex models, demonstrating the robustness of the formalisms. The separation into the derivation of serialisation constraints and their discharge was introduced in Chapter 4, but could be reused for AQM's with registers and might also be transferrable to other dynamic semantics of ALEF.

This completes the first part of this thesis. Starting with the following chapter, we substantiate the second advocated benefit of a programming language based approach as we link forwarding in ALEF to intermediate program analysis and compilation.

Chapter 6

Intermediate program analysis

This chapter together with Chapter 7 presents the core of a compiler-backend with target language ALEF. As operands in ALEF may either be forwarded or be passed through registers, compilation needs to decide which mechanism to use for each intermediate result, respecting the fundamental properties of operand queues:

1. a value in an operand queue can be consumed only once
2. the order in which values are inserted into an operand queue must agree with the order in which they are consumed

In this chapter, we concentrate on the first property and analyse how intermediate values are used by a program. Our method consists of a dataflow analysis for an intermediate language - variables for which the analysis can prove that each assignment corresponds to a single future usage are candidates for forwarding while variables with different usage patterns must be communicated through registers.

The second property restricts the allocation of operand queues to those variables which are forwardable, and will be dealt with in Chapter 7. Together, these two chapters provide a translation from the intermediate language into ALEF where the allocation of operand queues and registers is based on solutions to the dataflow equations.

Both the analysis and the translation are proven correct with respect to static and dynamic semantics. In case of the dataflow equations, we show that variables identified as being used linearly may indeed be deleted during their first usage. This result is obtained by defining a non-standard semantics which is sound with respect to the standard semantics and also related to the dataflow solutions. In case of the translation, we prove functional correctness of the resulting ALEF program with respect to the original program in the intermediate language, and well-typedness with respect to the calculus of Chapter 5, where the types relate to the dataflow solutions.

Synopsis of Chapter 6 Our intermediate language is a subset of languages found in modern compilers for imperative or object-oriented programming languages [App98a]. It contains assignments $x = e$ where x is a variable and e a simple expression, conditional and unconditional jumps, and pseudo-assignments of the form $x = \Phi(x_1, \ldots, x_n)$ for expressing programs in *static single assignment (SSA) form*. We provide the program analysis necessary for converting SSA and non-SSA programs into ALEF code, based on a usage analysis for IL variables.

We start by an informal introduction to the SSA discipline in Section 6.1. Section 6.2 then defines the syntax and dynamic semantics of IL programs and formalises the implicit conventions of SSA. We also summarise how non-SSA programs may be transformed into SSA form and define more restrictive SSA disciplines. Subsequently, the analysis of the dynamic number of accesses to a variable using dataflow equations is presented in Section 6.3. As usage analysis arises as a generalisation of liveness analysis, our discussion starts with a brief summary on the dataflow equations for liveness, before introducing usage analysis for non-SSA and SSA programs. Finally, Section 6.4 shows the soundness of the analysis by first defining a non-standard dynamic semantics where the number of read operations for a variable may be restricted and then proving that restrictions arising from solutions to the equations execute successfully.

6.1 Static single assignment

Static single assignment (SSA) is a compiler-intermediate program representation introduced in [AWZ88] for detecting equality of variables. By restricting the way in which variables are named, SSA forces all reading accesses to a variable to refer to the same assigning instruction. Consequently, the structure of a program is more explicitly available, with benefits for various program analysis techniques. In the past, SSA and its generalisations [JP93] [Ana99] have successfully served as a common representation for dead code elimination, constant propagation [SWG94], value numbering [BCS97] and partial redundancy elimination [KCL+99].

The main principle of SSA requires that each variable x has only one point of *definition*, i.e. that a program can have at most one instruction assigning to x. For straight-line code, this is achieved by introducing a new variable x_i whenever a previously assigned variable x is assigned to again. The *uses* (occurrences of x in the right hand side of an assignment or in a conditional branch) are renamed such that they always refer to the most recently introduced copy of x. For example, in code 6.1 (left), the two assignments to `x` may be renamed to `x1` and `x2` and the three assignments to `y` to `y1`, `y2` and `y3`.

$$\begin{array}{ll} [1]\mathtt{y} = \mathtt{5} & [1]\mathtt{y1} = \mathtt{5} \\ [2]\mathtt{x} = \mathtt{7} & [2]\mathtt{x1} = \mathtt{7} \\ [3]\mathtt{y} = \mathtt{x} + \mathtt{y} & [3]\mathtt{y2} = \mathtt{x1} + \mathtt{y1} \\ [4]\mathtt{x} = \mathtt{x} + \mathtt{y} & [4]\mathtt{x2} = \mathtt{x1} + \mathtt{y2} \\ [5]\mathtt{y} = \mathtt{x} * \mathtt{y} & [5]\mathtt{y3} = \mathtt{x2} * \mathtt{y2} \end{array} \tag{6.1}$$

Updating the uses of `x` and `y` accordingly results in code 6.1 (right). Data-dependencies between instructions are explicit and each site of use refers to the correct assignment. Furthermore, the life span of variables x_i resulting from a variable x do not overlap: for example, `y1` is *dead* after instruction $[3]$ as it will never be used again.

At points where control flow paths merge, the requirement that each variable have a unique static defining instruction is maintained by pseudo-instructions

$$x = \Phi(x_1, \ldots, x_n)$$

where $1, \ldots, n$ is an arbitrary but fixed enumeration of the incoming control flow arrows. Similar to other assignments, a Φ-instruction $x = \Phi(x_1, \ldots, x_n)$ represents a point of definition for variable x and a point of use for the x_i. Semantically, the execution of $x = \Phi(x_1, \ldots, x_n)$ assigns the value x_i to x whenever the the Φ-instruction is reached via control flow arrow numbered i. If a program is treated as a graph of basic blocks, Φ-instructions need only occur at the beginning of blocks and a group of Φ-instructions is assumed to be executed concurrently, i.e. as a single instruction which first extracts the correct values from the tuples on the right-hand side and then assigns them atomically to the variables on the left-hand side. As an example, consider

$$\begin{array}{ll}
[1]\mathtt{if\ v\ 2\ 5} & [1]\ \mathtt{if\ v\ 2\ 5} \\
[2]\mathtt{x} = 5 & [2]\mathtt{x1} = 5 \\
[3]\mathtt{y} = 6 & [3]\mathtt{y1} = 6 \\
[4]\mathtt{jmp\ 8} & [4]\mathtt{jmp\ 8a} \\
[5]\mathtt{x} = 2 & [5]\mathtt{x2} = 2 \\
[6]\mathtt{y} = 9 + \mathtt{v} & [6]\mathtt{y2} = 9 + \mathtt{v} \\
[7]\mathtt{jmp\ 8} & [7]\mathtt{jmp\ 8a} \\
[8]\mathtt{z} = \mathtt{x} * \mathtt{y} & [8\mathtt{a}]\mathtt{x3} = \Phi(\mathtt{x1}, \mathtt{x2}) \\
[9]\mathtt{z} = \mathtt{z} + \mathtt{v} & [8\mathtt{b}]\mathtt{y3} = \Phi(\mathtt{y1}, \mathtt{y2}) \\
 & [8]\mathtt{z1} = \mathtt{x3} * \mathtt{y3} \\
 & [9]\mathtt{z2} = \mathtt{z1} + \mathtt{v}
\end{array} \tag{6.2}$$

the program 6.2 (left) where (depending on the value of v) instruction [8] is reached either through the control flow edge $[4] \rightarrow [8]$ or through the edge $[7] \rightarrow [8]$. When the blocks [2]..[4] and [5]..[7] are transformed into SSA, different names are used for the assignments to x and y in order to ensure that each variable has a single point of definition. Inserting Φ-instructions [8a] and [8b] guarantees that instruction [8] accesses the correct values no matter which path was used. In the code 6.2 (right), the first component in both Φ-instructions is implicitly associated with the control flow arrow $[4] \rightarrow [8]$ and the second component with the arrow $[7] \rightarrow [8]$. The order of instructions [8a] and [8b] is semantically irrelevant. No Φ-instruction is needed for v since neither of the two branches assigns to v and the earlier value is still valid at point [8].

6.2 An intermediate language IL

This section introduces an imperative intermediate language IL as a formal basis for the translation of programs into ALEF. The language is similar to quadruple-based intermediate languages [App98a], but additionally contains Φ-instructions. It can thus express general (i.e. non-SSA) programs as well as programs in SSA form.

$$
\begin{array}{rcl}
bop \in BinOp & ::= & + \mid - \mid * \\
e \in Expr & ::= & a \mid x \mid x\; bop\; a \mid x\; bop\; x \\
\mathbf{x}(k) \in Var^k & ::= & \begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix} \\
\mathbf{n}(m) \in \mathbf{N}^m & ::= & \begin{pmatrix} n_1 \\ \vdots \\ n_m \end{pmatrix} \\
\mathbf{M}(k,m) \in Var^{k\times m} & ::= & \begin{pmatrix} x_{1,1} & \dots & x_{1,m} \\ & \vdots & \\ x_{k,1} & \dots & x_{k,m} \end{pmatrix} \\
ass \in Assignment & ::= & [n]x = e \\
\phi \in \Phi\text{-}Block & ::= & [n]\mathbf{x}(k) = \mathbf{M}(k,m)\mathbf{n}(m) \\
jump \in Jump & ::= & [n]\texttt{if}\; x\; m\; l \mid [n]\texttt{jmp}\; m
\end{array}
$$

Figure 6.1: Intermediate language IL: syntax

We first give the syntax of IL and collect some basic definitions regarding the usage of variables in programs. We then summarise an approach from the literature for transforming IL programs into SSA form and a more restrictive form called edge-split SSA. Finally, we define the dynamic semantics of IL, including the execution of Φ-instructions.

6.2.1 Syntax and regular programs

For a set $Var = \{\texttt{x}, \texttt{y}, \dots\}$ of program variables and values $a \in Val$ and numbers $n \in \mathbf{N}$ as before, the syntax of IL is given by the grammar in Figure 6.1. We let I range over the set *Instrs* of instructions (assignments, Φ-blocks and jumps) and $\vec{I}$ over sequences of instructions. In accordance with the general definitions in Chapter 2, the length n of $\vec{I} = I_1 \dots I_n$ is denoted by $|\vec{I}|$, and for $n > 0$ we write $fst(\vec{I})$ for I_1 and $lst(\vec{I})$ for I_n.

For simplicity, IL does not contain memory operations. These can however be added without further complications at the cost of extending the operational models by a memory component.

Example. The abstract syntax for Φ-Blocks ensures that all pseudo-assignments $x = \Phi(x_1,\dots,x_k)$ in a Φ-Block are of the same arity and use the same numbering of incoming control flow arrows. For example, the basic block $[\mathtt{8a}][\mathtt{8b}][\mathtt{8}][\mathtt{9}]$ in program 6.2 (right) takes the form

$$\begin{aligned}
&[\mathtt{8a}]\begin{pmatrix}\mathtt{x3}\\ \mathtt{y3}\end{pmatrix} = \begin{pmatrix}\mathtt{x1} & \mathtt{x2}\\ \mathtt{y1} & \mathtt{y2}\end{pmatrix}\begin{pmatrix}4\\ 7\end{pmatrix}\\
&[\mathtt{8}]\mathtt{z1} = \mathtt{x3} * \mathtt{y3}\\
&[\mathtt{9}]\mathtt{z2} = \mathtt{z1} + \mathtt{v}
\end{aligned}$$

The operational behaviour (to be defined shortly) assigns $\mathtt{x1}$ to $\mathtt{x3}$ and $\mathtt{y1}$ to $\mathtt{y3}$ if the control flow reached $[\mathtt{8a}]$ through instruction $[\mathtt{4}]$ while entering $[\mathtt{8a}]$ through the path $[\mathtt{7}] \to [\mathtt{8a}]$ results in the assignments $\mathtt{x3} = \mathtt{x2}$ and $\mathtt{y3} = \mathtt{y2}$. In both cases, the respective pair of assignments is carried out atomically. ◇

Definition 22. *For instruction I the sets defs(I) and uses(I) of variables* defined *and* used *in I are given by*

$$\begin{array}{rcl}
uses(a) & = & \emptyset\\
uses(x) & = & \{x\}\\
uses(x\,bop\,a) & = & \{x\}\\
uses(x\,bop\,y) & = & \{x,y\}\\
uses(\begin{pmatrix}x_{11} & \dots & x_{1m}\\ & \vdots & \\ x_{k1} & \dots & x_{km}\end{pmatrix}) & = & \{x_{ij} \mid 1 \le i \le k, 1 \le j \le m\}\\
uses([n]x = e) & = & uses(e)\\
uses([n]\mathbf{x} = \mathbf{Mn}) & = & uses(\mathbf{M})\\
uses([n]\mathtt{if}\ x\ m\ l) & = & \{x\}\\
uses([n]\mathtt{jmp}\ m) & = & \emptyset
\end{array}$$

and

$$
\begin{array}{rcl}
defs(\begin{pmatrix} x_1 \\ : \\ x_k \end{pmatrix}) & = & \{x_i \mid 1 \leq i \leq k\} \\
defs([n]x = e) & = & \{x\} \\
defs([n]\mathbf{x} = \mathbf{Mn}) & = & defs(\mathbf{x}) \\
defs([n]\mathtt{if}\ x\ m\ l) & = & \emptyset \\
defs([n]\mathtt{jmp}\ m) & = & \emptyset
\end{array}
$$

Definition 23. *A* basic block *is an instruction sequence $\vec{I} = I_1 \ldots I_n$ where $n \geq 0$ holds, at most I_n is a jump instruction and at most I_1 is a Φ-block. We denote the set of basic blocks by Blocks, and let B range over this set.*

Definition 24. *A program $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ consists of a partial function $\mathtt{N} : \mathbf{N} \rightharpoonup Blocks$, a relation $\mathtt{A} \subset dom\, \mathtt{N} \times dom\, \mathtt{N}$, a partial function $\mathtt{succ} : Instrs \rightharpoonup \mathcal{P}(Instrs)$ and a label $\mathtt{entry} \in dom\, \mathtt{N}$ such that*

- *the labelling of instructions is unique*
- *for all $n \in dom\, \mathtt{N}$, $|\mathtt{N}(n)| > 0$ holds and $fst(\mathtt{N}(n))$ has the label $[n]$*
- *the relation $\mathtt{A}$ is given by*

$$\mathtt{A} = \{(n,m)\} \mid lst(\mathtt{N}(n)) \in \{[\_]\mathtt{jmp}\ m, [\_]\mathtt{if}\ x\ m\ l, [\_]\mathtt{if}\ x\ l\ m\}\}$$

- *the function $\mathtt{succ}$ is given by*

$$
\begin{array}{rcl}
\mathtt{succ}(I) & = & \{J \mid \exists n \in dom\, \mathtt{N}\ s.t.\ \mathtt{N}(n) = \vec{K}IJ\vec{L}\} \\
& & \cup \{fst(\mathtt{N}(m)) \mid (n,m) \in \mathtt{A}, I = lst(\mathtt{N}(n))\}
\end{array}
$$

A partial map $\pi : \mathbf{N} \to \mathbf{Instrs}_{\mathtt{P}}$ with $\pi(i+1) \in \mathtt{succ}(\pi(i))$ is called an infinite path in $\mathtt{P}$ if $dom\, \pi = \mathbf{N}$ holds, and is called a finite path (of length n) if $dom\, \pi = \{1, \ldots, n\}$. In the latter case,we also write π as $(\pi(1), \ldots, \pi(n))$ and say that π is a path from $\pi(1)$ to $\pi(n)$

- *for $x \in uses(I_n)$, a path $(I_1, \dots, I_n)$ in* `P` *with $I_1 = fst(\mathtt{N}(\mathtt{entry}))$, and $I_n \notin \Phi$-Block, there is an $i < n$ such that $x \in defs(I_i)$ holds*

where the (finite) sets of instructions, variables and labels used by `P` *are denoted by* $\mathbf{Instrs}_{\mathtt{P}}$, $\mathbf{Var}_{\mathtt{P}} \subset Var$ *and* $\mathbf{N}_{\mathtt{P}} \subset \mathbf{N}$*, respectively, and the set of instruction sequences* $\vec{J}$ *such that there is a* $B \in cod\ \mathtt{N}$ *with* $B = \vec{I}\vec{J}$ *by* $\mathbf{Blocks}_{\mathtt{P}}$.

Definition 25. *Program* $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ *is* regular *if no basic block* $B \in cod\ \mathtt{N}$ *has the form* $\phi\vec{I}$.

6.2.2 Programs in Static Single Assignment form

The informal discussion of SSA in the previous section is imprecise as no *syntactic* conditions on SSA programs are given. For example, the requirement that all Φ-functions of a block have distinct variables on the left-hand side is necessary for proving properties of SSA code, but follows only implicitly from the semantics - all Φ-functions of a block are executed *concurrently*. Treatments of SSA in the literature often leave many properties implicit or consider SSA programs resulting from particular algorithms. In order to avoid later complications, the following definition of SSA programs captures syntactically those conditions which we require. It is based on the dominance notion which stipulates that each use of a variable be preceded by an execution of its (unique) defining instruction.

Definition 26. *For* $I, J \in \mathbf{Instrs}_{\mathtt{P}}$, I dominates J *if each path from* $fst(\mathtt{N}(\mathtt{entry}))$ *to* J *in* `P` *contains* I.

Definition 27. *Program* $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ *is an* (SSA) program *if*

- $fst(\mathtt{N}(\mathtt{entry})) \notin \Phi$*-Block*
- *for each* $x \in \mathbf{Var}_{\mathtt{P}}$ *there is at most one* $I \in \mathbf{Instrs}_{\mathtt{P}}$ *with* $x \in defs(I)$. *We denote this instruction by* I_x.
- *for each* $\phi = [l]\mathbf{x}(k) = \mathbf{M}(k,m)\mathbf{n}(m)$ *in* `P`*, the following conditions are fulfilled*
  - *for all* $1 \leq j \leq m$, $n_j = n$ *holds iff there is an* I *such that* $\phi \in \mathtt{succ}(I)$ *and* n *is the label of* I

– $1 \leq i \neq j \leq m$ *implies* $n_i \neq n_j$

– $1 \leq i \neq j \leq k$ *implies* $x_i \neq x_j$

– *if* n_j *is the label of* I, $\phi \in \mathtt{succ}(I)$ *and* x_{ij} *in occurs in* $\mathbf{M}$ *then* $I_{x_{ij}}$ *dominates* I

Informally,the last item captures the following properties.

- Each control flow edge $I \rightarrow \phi$ corresponds to exactly one entry in $\mathbf{n}$ (and consequently one column of $\mathbf{M}$).

- All variables on the LHS of a ϕ-block are distinct (necessary for concurrent execution of the assignments).

- All variables used in a row j are well-defined when the control flow enters the ϕ-block through the arrow associated to n_j.

6.2.2.1 Translation into SSA form

In order to transform a non-SSA program into an SSA program, variables have to be renamed suitably and Φ-instructions have to be inserted at appropriate points. In order to avoid unnecessary Φ-instructions, several algorithms with varying complexity have been proposed ([CFR$^+$91], [JP93], [BP96]). Our summary follows one of the original algorithms [CFR$^+$91] based on the dominance relation (see also [App98a]). This relation is most often represented in form of the *dominator tree* with instructions as nodes, root $\mathit{fst}(\mathtt{N}(\mathtt{entry}))$ and immediate dominance relations as arrows.

Definition 28. *I is the* immediate dominator *of J if I* dominates *J,* $I \neq J$ *and for all* $K \neq I$ *which dominate J, I does not dominate K. I* strictly dominates *J if I dominates J and* $I \neq J$.

It can be shown that immediate dominators are unique and Lengauer and Tarjan showed how to compute them efficiently using a depth-first traversal of the program graph [LT79].
The *dominance frontier* of I consists of instructions J which are not strictly dominated by I themselves, but one of their predecessors is:

Definition 29. *The* dominance frontier *of I is given by*

$$\begin{aligned} DF(I) &= \{J \mid I \textit{ does not strictly dominate } J\} \\ &\cap \{J \mid \exists K. J \in \mathtt{succ}(K) \textit{ and } I \textit{ dominates } K\}. \end{aligned}$$

An instruction J in $DF(I)$ thus represents a point of convergence, i.e. a common end-point of otherwise disjoint paths $(I,\ldots,J)$ and $(K,\ldots,J)$. Hence, a Φ-instruction is needed just before J for each variable x which fulfils $x \in \mathit{defs}(I)$ and is used in J or a successor of J. One can show that iterating this *dominance frontier criterion* by including the *defs* arising from Φ-instructions inserted in the i-th step during the iteration $i+1$ leads to the insertion of the fewest number of Φ-instructions necessary for satisfying the property that any use of a variable be dominated by its definition.
After all Φ-Blocks have been inserted, the program is transformed into SSA form by traversing the program graph or the dominator tree and renaming variables such that for each x there is only one I with $x \in \mathit{defs}(I)$.

6.2.2.2 Restricted SSA forms

The translation in Chapter 7 requires SSA programs to fulfil additional constraints. The first constraint is known under the name *edge-split* SSA.

Definition 30. *An SSA program* $\mathtt{P} = (\mathtt{N},\mathtt{A},\mathtt{succ},\mathtt{entry})$ *is in* edge-split form *if* $\phi \in \mathtt{succ}(I)$ *implies* $I \neq [n]\mathtt{if}\ x\ m\ l$.

The second constraint concerns the order of Φ-instructions in a block. As the translation needs to fix a particular order of these instructions, functional correctness can only be ensured if programs adhere to the following policy.

Definition 31. *A program is in* standard *edge-split form if it is in edge-split form and for all* ϕ *of the form* $[n]\mathbf{x} = \mathbf{Mn}\ x_{l\,j} \neq x_i$ *holds for all* j *and* i *and all* $l > i$.

Ordinary SSA programs may be transformed into edge-split form in a semantics-preserving way by inserting extra basic blocks whenever the condition in Definition

30 is not fulfilled. For example, program 6.3 (left)

$$
\begin{array}{rl@{\qquad}rl}
\mathtt{N}(1) = & [1]\mathtt{u} = \dots & \mathtt{N}(1) = & [1]\mathtt{u} = \dots \\
& [2]\mathtt{v} = \dots & & [2]\mathtt{v} = \dots \\
& [3]\mathtt{w} = \dots & & [3]\mathtt{w} = \dots \\
& [4]\mathtt{if\ u\ 5\ 8} & & [4]\mathtt{if\ u\ 5a\ 8a} \\
& & \mathtt{N}(5\mathtt{a}) = & [5\mathtt{a}]\mathtt{jmp\ 5} \\
\mathtt{N}(5) = & [5]\begin{pmatrix}\mathtt{x}\\\mathtt{y}\end{pmatrix} = \begin{pmatrix}\mathtt{v} & \mathtt{x}\\\mathtt{w} & \mathtt{x}\end{pmatrix}\begin{pmatrix}4\\7\end{pmatrix} & \mathtt{N}(5) = & [5]\begin{pmatrix}\mathtt{y}\\\mathtt{x}\end{pmatrix} = \begin{pmatrix}\mathtt{w} & \mathtt{x}\\\mathtt{v} & \mathtt{x}\end{pmatrix}\begin{pmatrix}4\\7\end{pmatrix} \\
& [6]\mathtt{z} = \mathtt{x} - 1 & & [6]\mathtt{z} = \mathtt{x} - 1 \\
& [7]\mathtt{if\ z\ 8\ 5} & & [7]\mathtt{if\ z\ 8a\ 5a} \\
& & \mathtt{N}(8\mathtt{a}) = & [8\mathtt{a}]\mathtt{jmp\ 8} \\
\mathtt{N}(8) = & [8]\begin{pmatrix}\mathtt{s}\\\mathtt{t}\end{pmatrix} = \begin{pmatrix}\mathtt{v} & \mathtt{y}\\\mathtt{u} & \mathtt{x}\end{pmatrix}\begin{pmatrix}4\\7\end{pmatrix} & \mathtt{N}(8) = & [8]\begin{pmatrix}\mathtt{s}\\\mathtt{t}\end{pmatrix} = \begin{pmatrix}\mathtt{v} & \mathtt{y}\\\mathtt{u} & \mathtt{x}\end{pmatrix}\begin{pmatrix}4\\7\end{pmatrix} \\
& [9]\dots & & [9]\dots
\end{array}
\tag{6.3}
$$

is transformed into edge-split form by introducing two new basic blocks, containing instructions $[5\mathtt{a}]$ and $[8\mathtt{a}]$, respectively.

In most cases, *standard* edge-split form may be obtained from edge-split form by reordering the Φ-instructions in each basic block – program 6.3 (right) shows the effect of swapping the Φ-instructions for `x` and `y`. If cyclic dependencies exist between variables in a Φ-Block, additional variables need to be inserted. For example, the Φ-Block

$$
\begin{array}{l}
[1]\begin{pmatrix}\mathtt{x}\\\mathtt{y}\end{pmatrix} = \begin{pmatrix}\mathtt{y} & \mathtt{x1}\\\mathtt{y1} & \mathtt{x}\end{pmatrix}\begin{pmatrix}5\\7\end{pmatrix} \\
[2]\mathtt{z} = \mathtt{x} * \mathtt{y}
\end{array}
$$

may be transformed to

$$
\begin{array}{l}
[1]\begin{pmatrix}\mathtt{w}\\\mathtt{y}\end{pmatrix} = \begin{pmatrix}\mathtt{y} & \mathtt{x1}\\\mathtt{y1} & \mathtt{x}\end{pmatrix}\begin{pmatrix}5\\7\end{pmatrix} \\
[2\mathtt{a}]\mathtt{x} = \mathtt{w} \\
[2]\mathtt{z} = \mathtt{x} * \mathtt{y}
\end{array}
$$

where $[2\mathtt{a}]$ is a fresh label and `w` a fresh variable. We expect that in SSA programs resulting from application programs few additional variables are needed – Φ-instructions manage different copies x_i of a program variable x and the situation shown in the example should not arise. However, our program analysis treats SSA programs irrespectively of how they were obtained. Consequently, a syntactic restriction is necessary for the development in this thesis. An embedding of our analysis in a compiler which generates SSA internally may replace the above transformation with a formal proof that the translation into SSA satisfies the condition of standard edge-split SSA.

For the development in this chapter, ordinary SSA discipline suffices. In the remainder of this chapter, we will thus always assume that programs $\mathtt{P}$ are either regular (and not treat cases for Φ-instructions) or in SSA form (and include Φ-instructions in our analysis).

6.2.3 Dynamic semantics

The dynamic semantics of an IL-program $\mathtt{P}$ is given by the transition relation $\mathsf{S} \rightarrow \mathsf{T}$ between ternary configurations of the form

$$(\sigma, n, B) \in (\mathbf{Var}_{\mathtt{P}} \rightharpoonup Val) \times \mathbf{N}_{\mathtt{P}} \times \mathbf{Blocks}_{\mathtt{P}}.$$

The components are interpreted as follows.

- the partial map σ represents the state by mapping variables to their content
- n is the label of the instruction from which the control flow was transferred to the current instruction, and is used for executing Φ-blocks
- B represents the remaining instructions of the current basic block.

The update operation $\sigma[x \mapsto a]$ is defined according to the general definition in Chapter 2 and is generalised to

$$\sigma[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n] = \sigma[x_1 \mapsto a_1] \ldots [x_n \mapsto a_n]$$

provided that the $x_1, \ldots, x_n$ are distinct.

The evaluation of expressions is defined by the relation $\sigma, e \Downarrow a$

$$\frac{}{\sigma, a \Downarrow a} \qquad \frac{\sigma, x \Downarrow a_1}{\sigma, x\, bop\, a \Downarrow BOP(a_1, a)}$$

$$\frac{x \in dom\, \sigma}{\sigma, x \Downarrow \sigma(x)} \qquad \frac{\sigma, x \Downarrow a_1 \quad \sigma, y \Downarrow a_2}{\sigma, x\, bop\, y \Downarrow BOP(a_1, a_2)}$$

where *BOP* is the semantic operation corresponding to the syntactic operator *bop*. For a program $\mathtt{P}$, the (standard) dynamic semantics $\rightarrow$ is given by the rules

$$
\begin{gathered}
\text{Jmp}\,\frac{}{(\sigma,n,[k]\mathtt{jmp}\,m)\rightarrow(\sigma,k,\mathtt{N}(m))} \\[2ex]
\text{If-t}\,\frac{\sigma,x\Downarrow 0}{(\sigma,n,[k]\mathtt{if}\,x\,m_1\,m_2)\rightarrow(\sigma,k,\mathtt{N}(m_1))} \\[2ex]
\text{If-f}\,\frac{\sigma,x\Downarrow a}{(\sigma,n,[k]\mathtt{if}\,x\,m_1\,m_2)\rightarrow(\sigma,k,\mathtt{N}(m_2))}\;a\neq 0 \\[2ex]
\text{Ass}\,\frac{\sigma,e\Downarrow a}{(\sigma,n,[k]x=e\,\vec{I})\rightarrow(\sigma[x\mapsto a],n,\vec{I})} \\[2ex]
\text{Phi}\,\frac{\forall i.\ \sigma,x_{ij}\Downarrow a_i}{(\sigma,n,\phi\vec{I})\rightarrow(\sigma',n,\vec{I})}\;\begin{cases}\phi=[l]\mathbf{x}(k)=\mathbf{M}(m,k)\mathbf{n}(m)\\ n=n_j\\ \sigma'=\sigma[x_i\mapsto a_i]\end{cases}
\end{gathered}
\tag{6.4}
$$

As usual, we denote the transitive closure of $\rightarrow$ by $\rightarrow^+$ and the reflexive transitive closure by $\rightarrow^*$.

Definition 32. *For program* $\mathtt{P}=(\mathtt{N},\mathtt{A},\mathtt{succ},\mathtt{entry})$*, the initial configuration is* $\mathsf{S}_0=([],\mathtt{entry},\mathtt{N}(\mathtt{entry}))$.

Some basic properties of $\rightarrow$ are collected in the following lemmas and propositions.

Lemma 8. *Let* $(\sigma,n,B)\rightarrow(\sigma',n',B')$.

1. *If* $|B'|>0$ *then* $\mathit{fst}(B')\in\mathtt{succ}(\mathit{fst}(B))$.

2. *If* $|B'|=0$ *then* $\mathtt{succ}(\mathit{fst}(B))=\emptyset$.

In both cases, $\mathit{dom}\,\sigma\subset\mathit{dom}\,\sigma'$ *holds, and* $x\in\mathit{defs}(\mathit{fst}(B))$ *implies* $x\in\mathit{dom}\,\sigma'$.

Proof. Induction on the rules for $(\sigma,n,B)\rightarrow(\sigma',n',B')$

1. Both $\mathit{fst}(B')$ and $\mathit{fst}(B)$ are well-defined as $|B'|>0$ holds and no reduction $(\sigma,n,B)\rightarrow(\sigma',n',B')$ exists for $|B|=0$.

Case Jmp. By the definition of the rule, $\sigma = \sigma'$ holds and there is an m with $B = [n']\texttt{jmp}\ m$ and $B' = \texttt{N}(m)$. For the k with $lst(\texttt{N})(k) = [n']\texttt{jmp}\ m$ we obtain $(k,m) \in \texttt{A}$ by Definition 24, hence $fst(\texttt{N}(m)) \in \texttt{succ}(lst(\texttt{N})(k))$ as claimed. Also, $dom\,\sigma \subset dom\,\sigma$ holds.

Cases If-t and If-f. Similar to the case Jmp.

Case Ass. By the definition of the rule, we have $B = [k]x = e\,B'$, $n = n'$ and $\sigma' = \sigma[x \mapsto a]$ for $\sigma, e \Downarrow a$. The definition of $\sigma[x \mapsto a]$ implies $dom\,\sigma \subset dom\,\sigma'$. As $\{B, B'\} \subset \mathbf{Blocks}_{\mathsf{P}}$ holds, there is an $m \in dom\,\texttt{N}$ such that $\texttt{N}(m) = \vec{I}B$ for some $\vec{I}$, so $fst(B') \in \texttt{succ}(fst(B))$ as claimed. Furthermore, we have $defs(fst(B)) = \{x\}$, and $x \in dom\,\sigma'$ holds.

Case Phi. By the definition of the rule, we have $n = n' = n_j$ for some j, $B = [l]\mathbf{x}(k) = \mathbf{M}(k,m)\mathbf{n}(m)\,B'$ and $\sigma' = \sigma[x_i \mapsto a_i]$ where $\sigma, x_{ij} \Downarrow a_i$ for all $1 \leq i \leq k$. The definition of $\sigma[x_i \mapsto a_i]$ implies $dom\,\sigma \subset dom\,\sigma'$, and since $B \in \mathbf{Blocks}_{\mathsf{P}}$ holds, there is an $h \in dom\,\texttt{N}$ such that $\texttt{N}(h) = B$ by the definition of basic blocks. Hence, $fst(B') \in \texttt{succ}(fst(B))$ as claimed. Furthermore, we have $defs(fst(B)) = \{x_i \mid 1 \leq i \leq k\}$, and $x_i \in dom\,\sigma'$ holds for all i.

2. For $|B'| = 0$ and $(\sigma, n, B) \to (\sigma', n', B')$, $|B| > 0$ holds by the definition of $\to$, hence $fst(B)$ is well-defined. Suppose $J \in \texttt{succ}(fst(B))$. Then there is a $k \in dom\,\texttt{N}$ such that $\texttt{N}(k) = \vec{K}fst(B)\vec{L}$ and either $J = fst(\texttt{N}(m))$ where $(k,m) \in \texttt{A}$ and $fst(B) = lst(\texttt{N}(k))$ (i.e. $|\vec{L}| = 0$) or $\texttt{N}(k) = \vec{K}fst(B)J\vec{M}$ (i.e. $\vec{L} = J\vec{M}$). In the first case, $fst(B) \in \{[\_]\texttt{jmp}\ m, [\_]\texttt{if}\ x\,m\,l, [\_]\texttt{if}\ x\,l\,m\}$, $\{m,l\} \subset dom\,N$, $|\texttt{N}(m)| > 0$ (and $|\texttt{N}(l)| > 0$) hold by Definition 24, so the rules Jmp, If-t and If-f imply $|B'| > 0$, in contradiction to $|B'| = 0$. In the second case, $fst(B)$ is not the last instruction of its basic block, hence it is either a Φ-block or an assignment and rules Phi and Ass imply $B' = J\vec{M}$, in contradiction to $|B'| = 0$. The claims regarding $dom\,\sigma'$ follows as in part (1).

□

Lemma 9. *Let* $\mathsf{S}_0 \to^* (\sigma, n, I\vec{I})$.

1. *If* $I \notin \Phi$*-Block then* $x \in uses(I)$ *implies* $x \in dom\,\sigma$.

2. *If I has the form $[l]\mathbf{x}(k) = \mathbf{M}(k,m)\mathbf{n}(m)$ and $\mathbf{n} = (n_1,\dots,n_m)$ then there is a unique j such that $n = n_j$. Furthermore, $x_{ij} \in dom\, \sigma$ holds for all i.*

Proof. 1. If $(\sigma, n, I\vec{I}) = \mathsf{S}_0$ then $uses(I) = \emptyset$ because the last condition in Definition 24 requires for each $x \in uses(I)$ the existence of an $J \neq I$ on the trivial path $(fst(\mathtt{N}(\mathtt{entry}))) = (I)$ with $x \in defs(J)$.

For $(\sigma, n, I\vec{I}) \neq \mathsf{S}_0$ and $\mathsf{S}_0 \to \ldots \to \mathsf{S}_\mathsf{l} = (\sigma, n, I\vec{I})$ and $\mathsf{S}_\mathsf{i} = (\sigma_i, n_i, B_i)$, Lemma 8 implies $fst(B_{i+1}) \in \mathtt{succ}(fst(B_i))$ for $0 \leq i < l$. Consequently, the last condition in Definition 24 guarantees that for each $x \in uses(I)$ there is a $j < l$ such that $x \in defs(fst(B_j))$. By Lemma 8 $x \in dom\, \sigma_{j+1}$ holds and by induction on $l - j$ we obtain $x \in dom\, \sigma'$.

2. For $I = \phi$, $\mathtt{P}$ cannot be a regular program since no Φ-Blocks occur in regular programs.

For an SSA program $\mathtt{P}$ and $I = \phi$, $\mathsf{S}_0 \neq (\sigma, n, I\vec{I})$ holds because $I\vec{I} = \mathtt{N}(\mathtt{entry})$ and Definition 27 requires $fst(\mathtt{N}(\mathtt{entry})) \neq \phi$. Hence there are $h > 0$ and $\mathsf{S}_{\mathsf{h}-1}$ such that $\mathsf{S}_0 \to^* \mathsf{S}_{\mathsf{h}-1} \to \mathsf{S}_\mathsf{h} = (\sigma, n, I\vec{I})$. We write $\mathsf{S}_{\mathsf{h}-1} = (\sigma', n', B)$. Since I is of the form $[l]\mathbf{x} = \mathbf{Mn}$, the step $\mathsf{S}_{\mathsf{h}-1} \to^* (\sigma, n, I\vec{I})$ must have been a Jmp, If-t or If-f step, as the form of component B on the LHS in the rules Ass and Phi implies that the component B in the RHS of the same rule does not start with a Φ-block. In the three rules Jmp, If-t and If-f, the value in the second component of the resulting configuration equals the label of the jump instruction, so Lemma 8 yields $I \in \mathtt{succ}(fst(B))$. By Definition 27 there is thus a j such that $n_j = n$, and j is unique since all components of $\mathbf{n}$ are distinct by Definition 27. By the last clause of that definition, $I_{x_{ij}}$ dominates $fst(B)$, so $x_{ij} \in dom\, \sigma$ follows in a similar way as in part (1).

□

Proposition 29. *(Determinacy of $\to$) If $\mathsf{S} \to \mathsf{T}$ and $\mathsf{S} \to \mathsf{R}$ then $\mathsf{T} = \mathsf{R}$.*

Proof. Follows by induction on the rules for $\to$, using the determinacy of $\Downarrow$ and $\sigma[x \mapsto a]$, and in rule Phi the distinctness of the components of $\mathbf{x}$ (Definition 27). □

Proposition 30. *If* $\mathsf{S}_0 \rightarrow^* (\sigma, n, B)$ *and* $|B| > 0$ *then there is a unique* T *such that* $(\sigma, n, B) \rightarrow \mathsf{T}$.

Proof. For $B = I\vec{I}$ and $I \neq \phi$ Lemma 9 implies that $uses(I) \subset dom\,\sigma$, so for all rules the conditions $\sigma, e \Downarrow a$ succeed since in all rules all variables in the expression e are contained in $uses(I)$. Also, evaluation $\sigma, e \Downarrow a$ is deterministic. In the rules for (unconditional and conditional) jumps, $I \in \{[k]\mathtt{jmp}(m), [k]\mathtt{if}\ x\ m\ l, [k]\mathtt{if}\ x\ l\ m\}$ implies $m \in dom\,\mathtt{N}$ by Definition 24, so the branch target $\mathtt{N}(m)$ is guaranteed to exist.
If I has the form $[l]\mathbf{x}(k) = \mathbf{M}(k,m)\mathbf{n}(m)$ then Lemma 9 implies that there is a unique j such that $n = n_j$ and $x_{ij} \in dom\,\sigma$ for all $1 \leq i \leq k$. Again, this implies that $\sigma, x_{ij} \Downarrow a_i$ is well-defined and for each i a unique such a_i exists, so the definition of σ' in rule Phi is unique. ☐

In particular, targets of a conditional or unconditional jump are always in the domain of $\mathtt{N}$ and the corresponding rules Jmp, If-t and If-t are well-defined.

6.3 Analysis of variable usage

This section presents the program analysis on which the translation from IL into ALEF is based, and we treat both regular and SSA programs. The analysis consists of determining the usage pattern for each variable. In general, a variable whose value is accessed exactly once may be suitable for forwarding whereas a variable whose content is accessed repeatedly has to be communicated through a register.
The usage analysis we are interested in concerns the *dynamic* number of reading accesses rather than the *static* number of accessing instructions. For example, in program

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\mathtt{N}(1) & = & [1]\mathtt{v} = \ldots & \mathtt{N}(6) & = & [6]\mathtt{u} = \mathtt{x} * \mathtt{2} \\
 & & [2]\mathtt{x} = \mathtt{2} & & & [7]\mathtt{jmp}\ \mathtt{8} \\
 & & [3]\mathtt{if}\ \mathtt{v}\ \mathtt{4}\ \mathtt{6} & & & \\
\mathtt{N}(4) & = & [4]\mathtt{u} = \mathtt{x} + \mathtt{3} & \mathtt{N}(8) & = & [8]\mathtt{v} = \mathtt{u} + \mathtt{4} \\
 & & [5]\mathtt{jmp}\ \mathtt{8} & & &
\end{array} \tag{6.5}$$

the variable x is always accessed exactly once although there are two (static) instructions reading from x. Also u is accessed exactly once, although the values might stem from different assigning instructions.

The opposite situation may also occur: a variable may be dynamically accessed multiple times despite the existence of only one accessing instruction. For example, the number of accesses to y in code

$$
\begin{aligned}
\mathtt{N}(1) &= [1]\mathtt{v} = \ldots & \mathtt{N}(4) &= [4]\mathtt{u} = \mathtt{y}+3 & \mathtt{N}(7) &= [7]\mathtt{v} = 7\\
& [2]\mathtt{y} = 2 & & [5]\mathtt{v} = \mathtt{v}-1 & &\\
& [3]\mathtt{if\ v}\ 4\ 7 & & [6]\mathtt{if\ v}\ 4\ 7 & &
\end{aligned}
\tag{6.6}
$$

depends on the initial value of v.

Other variables are accessed an unknown number of times but each access corresponds to one execution of an assignment to that variable. This is for example the case for both w and z in program

$$
\begin{aligned}
\mathtt{N}(1) &= [1]\mathtt{v} = \ldots & \mathtt{N}(4) &= [4]\mathtt{z} = \mathtt{w}*2 & \mathtt{N}(8) &= [8]\mathtt{v} = \mathtt{w}+1\\
& [2]\mathtt{w} = 2 & & [5]\mathtt{w} = \mathtt{z}+3 & &\\
& [3]\mathtt{if\ v}\ 4\ 8 & & [6]\mathtt{v} = \mathtt{v}-1 & &\\
& & & [7]\mathtt{if\ v}\ 4\ 8 & &
\end{aligned}
\tag{6.7}
$$

Each execution of an assignment to w or z corresponds to exactly one reading access, independently from the initial value of v.

A complete analysis would detect all situations where the number of dynamic assignments equals the number of dynamic accesses in all executions. However, this task is in general not computable, as is shown by program

$$
\begin{aligned}
\mathtt{N}(1) &= [1]\mathtt{v} = 0 & \mathtt{N}(6) &= [6]\mathtt{y3} = \mathtt{y1}*\mathtt{y1}\\
& [2]\mathtt{y1} = 2 & & [7]\mathtt{jmp}\ 8\\
& [3]\mathtt{if\ v}\ 4\ 6 & &\\
\mathtt{N}(4) &= [4]\mathtt{y2} = \mathtt{y1}+3 & \mathtt{N}(8) &= [8]\mathtt{v} = 7\\
& [5]\mathtt{jmp}\ 8 & &
\end{aligned}
\tag{6.8}
$$

where y1 is dynamically assigned to exactly once, and is also accessed exactly once provided that the left branch is taken. Indeed, the assignment $\mathtt{v} = 0$ ensures that this is the case. However, the general task to detect what path the control flow takes at runtime is undecidable. Thus, any usage analysis is necessarily conservative in the sense that some opportunities for forwarding are not detected. The analysis we present does not

involve techniques for analysing the values of variables or the control flow and will hence mark variable `y1` in as being non-forwardable. On the other hand, our analysis is powerful enough to detect the linear usage of variables `w`, `x` and `z` in programs (6.5) and (6.7) and to separate it from the possibly non-linear usage of `y` in program (6.6).
Our method consists of a dataflow analysis similar to the task of determining the liveness of a variable. We therefore briefly review liveness analysis before adapting it to usage analysis. Our summary follows the expositions in [App98a] and [NNH99], and the more interested reader is referred to these sources for more in-depth descriptions.

6.3.1 Dataflow analysis for liveness

Liveness analysis aims at detecting whether the value of a variable is needed beyond a particular point in a program.

Definition 33. *Variable* $x \in \mathbf{Var}_{\mathtt{P}}$ *is called* live at instruction $I \in \mathbf{Instr}_{\mathtt{P}}$ *if there is a* $J \in \mathbf{Instr}_{\mathtt{P}}$ *with* $x \in \mathit{uses}(J)$ *and a path from* I *to* J *in* P *which does not contain any* K *with* $x \in \mathit{defs}(K)$.

Like other dataflow problems, liveness can be calculated from the sets of variables defined and used in instructions as follows.
To each instruction I, two sets of variables $\mathit{liveIn}(I)$ and $\mathit{liveOut}(I)$ are associated containing (approximations to) the set of variables live before and after the execution of I, respectively. Formally, this corresponds to defining two functions

$$\mathit{liveIn}, \mathit{liveOut} : \mathbf{Instr}_{\mathtt{P}} \to \mathcal{P}(\mathbf{Var}_{\mathtt{P}})$$

and we denote the function space $\mathbf{Instr}_{\mathtt{P}} \to \mathcal{P}(\mathbf{Var}_{\mathtt{P}})$ by $\mathbf{FSP}_{lv}$. A variable x is live-out at I if it is needed in any of the successors of I. It is live-in at I if it is either live-out but not assigned to in I or it is used in the RHS of I. This motivates us to require that the functions fulfil the pointwise constraints

$$\begin{array}{rcl} \mathit{liveOut}(I) & \supseteq & \bigcup_{J \in \mathtt{succ}(I)} \mathit{liveIn}(J) \\ \mathit{liveIn}(I) & \supseteq & (\mathit{liveOut}(I) \setminus \mathit{defs}(I)) \cup \mathit{uses}(I) \end{array} \tag{6.9}$$

Any pair (f,g) of functions mapping instructions to subsets of $\mathbf{Var}_{\mathtt{P}}$ which fulfils the above inclusions for each I (i.e. can be substituted for *liveOut* and *liveIn*) gives some liveness information. However, the larger the subsets grow, the less useful information they convey – in particular the trivial pair $(\lambda I.\mathbf{Var}_{\mathtt{P}}, \lambda I.\mathbf{Var}_{\mathtt{P}})$ conveys no information at all – it states that all variables are (potentially) always live. The following approach results in non-trivial solutions which are as small as possible.

The endofunction

$$update : (\mathbf{FSP}_{lv} \times \mathbf{FSP}_{lv}) \to (\mathbf{FSP}_{lv} \times \mathbf{FSP}_{lv})$$

corresponding to the above constraints, defined by

$$update(f,g) = (\lambda I. \cup_{J \in \mathtt{succ}(I)} g(J),\ \lambda I.(f(I) \setminus defs(I)) \cup uses(I))$$

is monotone in both arguments where the partial order over functions in $\mathbf{FSP}_{lv}$ is defined by

$$f \sqsubseteq g \text{ iff } \forall I \in \mathbf{Instr}_{\mathtt{P}}.f(I) \subseteq g(I)$$

and $\lambda I.\emptyset$ is the least element. The codomain of $\mathbf{FSP}_{lv}$ is a finite lattice, consisting of the subsets of $\mathbf{Var}_{\mathtt{P}}$ ordered by set inclusion, with $\bot = \emptyset$ and $\top = \mathbf{Var}_{\mathtt{P}}$. Consequently, *update* has a (unique) least fixed point (f^{lfp}, g^{lfp}) by the theorem of Knaster-Tarski [Tar55]: there is a unique pair (f^{lfp}, g^{lfp}) which fulfils the constraints (6.9) with equality and is minimal with respect to that property.

The above motivation suggests that the *least* fixed point represents the preferred solution to the liveness constraints as it results in the smallest sets of live variables at each program point. The calculation of the fixed-point solution proceeds by initialising

$$liveIn(I) = liveOut(I) = \emptyset$$

for all I and iterating the *update* operation – termination is guaranteed. Rapid convergence can be achieved by work-list algorithms and traversing the nodes in reverse postorder in each update iteration, i.e. by updating the sets for successors of I prior to visiting I. Since $liveIn(I)$ is defined in terms of $liveOut(I)$, the flow of information is in the opposite direction of the program (execution) flow, and liveness analysis is called a *backwards* dataflow problem.

Example. For program

$$\begin{array}{rclrclrcl} \mathtt{N}(1) & = & [1]\mathtt{x}=7 & \mathtt{N}(4) & = & [4]\mathtt{x}=\mathtt{x}+1 & \mathtt{N}(7) & = & [7]\mathtt{y}=\mathtt{x}+\mathtt{y} \\ & & [2]\mathtt{y}=2 & & & [5]\mathtt{z}=\mathtt{x}+\mathtt{y} & & & [8]\mathtt{jmp}\ 1 \\ & & [3]\mathtt{if}\ \mathtt{y}\ 4\ 7 & & & [6]\mathtt{jmp}\ 1 & & & \end{array} \tag{6.10}$$

the constraints (6.9) expand to

$$\begin{array}{rcl} \mathit{liveIn}([1]) & \supseteq & (\mathit{liveOut}([1]) \setminus \{\mathtt{x}\}) \cup \emptyset \\ \mathit{liveOut}([1]) & \supseteq & \mathit{liveIn}([2]) \supseteq (\mathit{liveOut}([2]) \setminus \{\mathtt{y}\}) \cup \emptyset = \mathit{liveOut}([2]) \setminus \{\mathtt{y}\} \\ \mathit{liveOut}([2]) & \supseteq & \mathit{liveIn}([3]) \supseteq (\mathit{liveOut}([3]) \setminus \emptyset) \cup \{\mathtt{y}\} = \mathit{liveOut}([3]) \cup \{\mathtt{y}\} \\ \mathit{liveOut}([3]) & \supseteq & \mathit{liveIn}([4]) \cup \mathit{liveIn}([7]) \\ & \supseteq & ((\mathit{liveOut}([4]) \setminus \{\mathtt{x}\}) \cup \{\mathtt{x}\}) \cup ((\mathit{liveOut}([7]) \setminus \{\mathtt{y}\}) \cup \{\mathtt{x},\mathtt{y}\}) \\ & = & (\mathit{liveOut}([4]) \cup \{\mathtt{x}\}) \cup (\mathit{liveOut}([7]) \cup \{\mathtt{x},\mathtt{y}\}) \\ \mathit{liveOut}([4]) & \supseteq & \mathit{liveIn}([5]) \supseteq (\mathit{liveOut}([5]) \setminus \{\mathtt{z}\}) \cup \{\mathtt{x},\mathtt{y}\} \\ \mathit{liveOut}([5]) & \supseteq & \mathit{liveIn}([6]) \supseteq (\mathit{liveOut}([6]) \setminus \emptyset) \cup \emptyset = \mathit{liveOut}([6]) \\ \mathit{liveOut}([6]) & \supseteq & \mathit{liveIn}([1]) \\ \mathit{liveOut}([7]) & \supseteq & \mathit{liveIn}([8]) \supseteq (\mathit{liveOut}([8]) \setminus \emptyset) \cup \emptyset = \mathit{liveOut}([8]) \\ \mathit{liveOut}([8]) & \supseteq & \mathit{liveIn}([1]) \end{array} \tag{6.11}$$

resulting in the recursive relation

$$\begin{array}{rcl} \mathit{liveOut}([1]) & \supseteq & \mathit{liveOut}([2]) \setminus \{\mathtt{y}\} \\ & \supseteq & (\mathit{liveOut}([3]) \cup \{\mathtt{y}\}) \setminus \{\mathtt{y}\} = \mathit{liveOut}([3]) \setminus \{\mathtt{y}\} \\ & \supseteq & ((\mathit{liveOut}([4]) \cup \{\mathtt{x}\}) \cup (\mathit{liveOut}([7]) \cup \{\mathtt{x},\mathtt{y}\})) \setminus \{\mathtt{y}\} \\ & = & (\mathit{liveOut}([4]) \cup \mathit{liveOut}([7]) \cup \{\mathtt{x}\}) \setminus \{\mathtt{y}\} \\ & \supseteq & ((\mathit{liveOut}([5]) \setminus \{\mathtt{z}\}) \cup \{\mathtt{x},\mathtt{y}\} \cup \mathit{liveOut}([8]) \cup \{\mathtt{x}\}) \setminus \{\mathtt{y}\} \\ & = & ((\mathit{liveOut}([5]) \setminus \{\mathtt{z}\}) \cup \mathit{liveOut}([8]) \cup \{\mathtt{x}\}) \setminus \{\mathtt{y}\} \\ & \supseteq & ((\mathit{liveOut}([1]) \setminus \{\mathtt{x},\mathtt{z}\}) \cup (\mathit{liveOut}([1]) \setminus \{\mathtt{x}\}) \cup \{\mathtt{x}\}) \setminus \{\mathtt{y}\} \\ & = & (\mathit{liveOut}([1]) \cup \{\mathtt{x}\}) \setminus \{\mathtt{y}\} \end{array}$$

Its least fixed point is $\mathit{liveOut}([1]) = \{\mathtt{x}\}$, and substituting this into the constraints (6.11) yields the (again minimal) sets

$$\begin{array}{rcl} \mathit{liveOut}([2]) = \mathit{liveOut}([3]) = \mathit{liveOut}([4]) & = & \{\mathtt{x},\mathtt{y}\} \\ \mathit{liveOut}([5]) = \mathit{liveOut}([6]) = \mathit{liveOut}([7]) = \mathit{liveOut}([8]) & = & \emptyset. \end{array}$$

Any solution contains the minimal solution, hence the analysis shows that x and y are jointly live-out at instructions [2], [3] and [4]. During register allocation, this liveness information is used for deducing that x and y must be mapped to different registers. In contrast to non-minimal solutions such as $liveOut = \lambda I.\{\mathtt{x},\mathtt{y},\mathtt{z}\}$ the above solution allows z to share a register with either x or y. $\diamond$

Another application of liveness analysis is the elimination of *useless* variables. Any variable x for which $x \in defs(I)$ implies $x \notin liveOut(I)$ may be eliminated from the program. For example, the above minimal solution for program (6.10) indicates that z is useless as it is not live-out at its defining instruction. In general, deleting defining instructions of a useless variables opens further opportunities for optimisation. Not only may other variables become useless themselves, but also repeating the liveness-analysis may reduce the number of conflicts between variables for register allocation.

For proving soundness, (solutions to) the dataflow equations need to be related to the dynamic semantics. [NNH99] present an elegant indirect approach: they prove that the value to which a variable x is mapped is irrelevant at program points where x is not live. A corresponding equivalence relation equates states which only differ in the content of non-live variables and is shown to be preserved by the dynamic semantics.

6.3.2 Usage analysis for regular programs

Usage analysis arises from liveness analysis by replacing the question *whether* a variable is used by the question *how often* it is accessed. In this section we aim to detect variables with *exactly one* use for each execution of an assigning instruction. The direction of the flow of information agrees with that of liveness: once we know how often a variable is used in the part of the program following an instruction I we can deduce the number of times it is used in the part including I by inspecting the form of I. In addition, usage analysis for variable x can be restricted to the part of the program in which x is live – and algorithms may thus work backwards from the liveness frontier to the instructions defining x.

The two pieces of information associated with each instruction need to be modified.

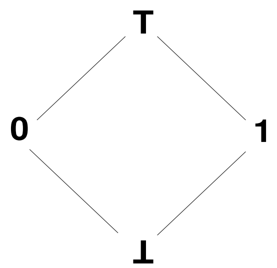
Instead of assigning sets of variables to instructions via functions

$$\mathit{liveIn}, \mathit{liveOut} : \mathbf{Instrs}_{\mathrm{P}} \to \mathcal{P}(\mathbf{Var}_{\mathrm{P}})$$

or equivalently functions $\mathit{liveIn}, \mathit{liveOut} : \mathbf{Instrs}_{\mathrm{P}} \to \mathbf{Var}_{\mathrm{P}} \to \mathcal{B}$ from variables to the lattice of booleans, we employ functions

$$\mathit{fwdIn}, \mathit{fwdOut} : \mathbf{Instrs}_{\mathrm{P}} \to \mathbf{Var}_{\mathrm{P}} \to \mathcal{L}$$

where $\mathcal{L}$ is the lattice



The bottom element represents unknown information, whereas $\top$ indicates contradictory usage in different branches or existence of more than one usage. Elements 0 and 1 represent exactly none and exactly one usage, respectively.

We define the operation $\oplus$ over $\mathcal{L}$ by

$\oplus$	$\bot$	0	1	$\top$
$\bot$	$\bot$	$\bot$	1	$\top$
0	$\bot$	0	1	$\top$
1	1	1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

and use $\sqcup$ to denote least upper bounds of elements in $\mathcal{L}$, with $\sqcup\emptyset = \bot$.

The operation $\oplus$ is commutative and associative and fulfils the following properties.

Lemma 10. *Let $a, b, \ldots \in \mathcal{L}$. Then $a \oplus b \sqsubseteq 0$ holds exactly if $a \sqsubseteq 0$ and $b \sqsubseteq 0$. Furthermore, $\oplus$ is pointwise monotone: if $a \sqsubseteq c$ and $b \sqsubseteq d$ then $a \oplus b \sqsubseteq c \oplus d$.*

Proof. Inspection of the table defining $\oplus$ and the structure of $\mathcal{L}$. $\square$

On the other hand, $\oplus$ is not increasing as $a \sqsubseteq a \oplus b$ does not hold in general.

We define functions $\mathit{uses} : \mathit{Expr} \to \mathbf{Var}_{\mathrm{P}} \to \mathcal{L}$ and $\mathit{uses} : \mathbf{Instrs}_{\mathrm{P}} \to \mathbf{Var}_{\mathrm{P}} \to \mathcal{L}$ by

$$\begin{array}{rcl}
\mathit{uses}(a)(y) & = & 0 \\
\mathit{uses}(x)(y) & = & \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \\
\mathit{uses}(x \; \mathit{bop} \; e)(y) & = & \mathit{uses}(x)(y) \oplus \mathit{uses}(e)(y) \\
\mathit{uses}([n]x = e)(y) & = & \mathit{uses}(e)(y) \\
\mathit{uses}([n]\mathtt{if}\; x \; n_1 \; n_2)(y) & = & \mathit{uses}(x)(y) \\
\mathit{uses}([n]\mathtt{jmp}\; m)(y) & = & 0
\end{array} \tag{6.12}$$

generalising the *uses* predicate of liveness analysis. The functions *fwdIn* and *fwdOut* are given by the mutually recursive equations

$$\begin{array}{rcl}
\mathit{fwdOut}(I)(x) & = & \begin{cases} 0 & \text{if } \mathtt{succ}(I) = \emptyset \\ \sqcup_{J \in \mathtt{succ}(I)} \mathit{fwdIn}(J)(x) & \text{otherwise} \end{cases} \\
\mathit{fwdIn}(I)(x) & = & \begin{cases} \mathit{uses}(I)(x) & \text{if } x \in \mathit{defs}(I) \\ \mathit{uses}(I)(x) \oplus \mathit{fwdOut}(I)(x) & \text{otherwise} \end{cases}
\end{array} \tag{6.13}$$

The *update* function corresponding to equations (6.13) is

$$\begin{array}{rcl}
\mathit{update}(f, g) & = & (\lambda I.\lambda x.\ \text{if } \mathtt{succ}(I) = \emptyset \text{ then } 0 \text{ else } \sqcup_{J \in \mathtt{succ}(I)} g(J)(x), \\
& & \;\lambda I.\lambda x.\ \text{if } x \in \mathit{defs}(I) \text{ then } \mathit{uses}(I)(x) \text{ else } \mathit{uses}(I)(x) \oplus f(I)(x))
\end{array}$$

where f and g are of functionality

$$\mathbf{FSP}_{\mathit{fwd}} = \mathbf{Instr}_{\mathrm{P}} \to (\mathbf{Var}_{\mathrm{P}} \to \mathcal{L}).$$

The lattice structure of $\mathcal{L}$ is inherited by the function space $\mathbf{Var}_{\mathrm{P}} \to \mathcal{L}$ where functions $F, G : \mathbf{Var}_{\mathrm{P}} \to \mathcal{L}$ are ordered pointwise by

$$F \sqsubseteq G \text{ iff } \forall x \in \mathbf{Var}_{\mathrm{P}}.F(x) \sqsubseteq G(x)$$

and minimal and maximal elements are $\lambda x.\bot$ and $\lambda x.\top$. For $f, g : \mathbf{FSP}_{\mathit{fwd}}$ with

$$f \sqsubseteq g \text{ iff } \forall I \in \mathbf{Instr}_{\mathrm{P}}.f(I) \sqsubseteq g(I)$$

the function

$$update : (\mathbf{FSP}_{fwd} \times \mathbf{FSP}_{fwd}) \to (\mathbf{FSP}_{fwd} \times \mathbf{FSP}_{fwd})$$

is thus monotone in both arguments: $f \sqsubseteq f'$ implies by definition $f(I) \sqsubseteq f'(I)$ for all I and thus $f(I)(x) \sqsubseteq f'(I)(x)$ for all I and x, so $l \oplus f(I)(x) \sqsubseteq l \oplus f'(I)(x)$ holds (by Lemma 10) for all I, x and $l \in \mathcal{L}$, hence

$$\begin{aligned}
update(f,g) &= (\lambda I.\lambda x.\text{ if } \mathtt{succ}(I) = \emptyset \text{ then } 0 \text{ else } \sqcup_{J \in \mathtt{succ}(I)}\, g(J)(x),\\
&\qquad \lambda I.\lambda x.\text{ if } x \in \mathit{defs}(I) \text{ then } \mathit{uses}(I)(x) \text{ else } \mathit{uses}(I)(x) \oplus f(I)(x))\\
&\sqsubseteq (\lambda I.\lambda x.\text{ if } \mathtt{succ}(I) = \emptyset \text{ then } 0 \text{ else } \sqcup_{J \in \mathtt{succ}(I)}\, g(J)(x),\\
&\qquad \lambda I.\lambda x.\text{ if } x \in \mathit{defs}(I) \text{ then } \mathit{uses}(I)(x) \text{ else } \mathit{uses}(I)(x) \oplus f'(I)(x))\\
&= update(f',g)
\end{aligned}$$

and similarly one obtains $update(f,g) \sqsubseteq update(f,g')$ for $g \sqsubseteq g'$. Consequently, fixed points exist by the theorem of Knaster-Tarski. In accordance with the general observation of [NNH99] on the relation of fixed points to the direction of the lattice operation $\sqcup$ we observe that the *least* fixed point to the equations (6.13) represents the preferred solution to the usage analysis problem as it detects the most linearly used variables.
The result of the forwardability analysis is represented in the *fwdOut* component of fixed points: we will prove in Section 6.4 that the value assigned to x in instruction I is accessed exactly once if $\mathit{fwdOut}(I)(x) = 1$ holds.

Example. The structure of the lattice $\mathcal{L}$ may be motivated using the two programs

$$\begin{array}{llll}
\mathtt{N}(1) = & [1]\mathtt{x} = \mathtt{2} & \mathtt{N}(1) = & [1]\mathtt{x} = \mathtt{2}\\
 & [2]\mathtt{y} = \mathtt{3} & & [2]\mathtt{jmp}\ \mathtt{3}\\
 & [3]\mathtt{jmp}\ \mathtt{4} & \mathtt{N}(3) = & [3]\mathtt{y} = \mathtt{x} + \mathtt{1}\\
\mathtt{N}(4) = & [4]\mathtt{y} = \mathtt{y} + \mathtt{x} & & [4]\mathtt{if}\ \mathtt{x}\ \mathtt{3}\ \mathtt{5}\\
 & [5]\mathtt{jmp}\ \mathtt{4} & \mathtt{N}(5) = & [5]\mathtt{z} = \mathtt{y} + \mathtt{2}
\end{array} \tag{6.14}$$

In the program on the left, *uses* and *defs* are given by

$$\mathit{uses}([4])(\mathtt{x}) = \mathit{uses}([4])(\mathtt{y}) = 1 \text{ and } \mathit{uses}(I)(\mathtt{x}) = \mathit{uses}(I)(\mathtt{y}) = 0 \text{ for } I \neq [4]$$

and

$$\mathit{defs}([1]) = \{\mathtt{x}\}, \mathit{defs}([2]) = \mathit{defs}([4]) = \{\mathtt{y}\} \text{ and } \mathit{defs}([3]) = \mathit{defs}([5]) = \emptyset.$$

The definition of *fwdIn* and *fwdOut* results in the recursive equations

$$\begin{aligned}
\mathit{fwdOut}([4])(\mathtt{x}) &= 0 \oplus 1 \oplus \mathit{fwdOut}([4])(\mathtt{x}) \\
\mathit{fwdOut}([4])(\mathtt{y}) &= 0 \oplus 1 \\
\mathit{fwdOut}([2])(\mathtt{x}) &= 0 \oplus 1 \oplus \mathit{fwdOut}([4])(\mathtt{x}) \\
\mathit{fwdOut}([2])(\mathtt{y}) &= 0 \oplus 1 \\
\mathit{fwdOut}([1])(\mathtt{x}) &= 0 \oplus 0 \oplus 1 \oplus \mathit{fwdOut}([4])(\mathtt{x}) \\
\mathit{fwdOut}([1])(\mathtt{y}) &= 0.
\end{aligned}$$

We aim at an analysis which detects that $\mathtt{x}$ is possibly used infinitely often but allows $\mathtt{y}$ to be forwarded: each value which is assigned to $\mathtt{y}$ is subsequently accessed exactly once. By the above correspondence between $\mathit{fwdOut}(I)(x)$ and $x \in \mathit{defs}(I)$ we thus aim at a fixed point with

$$\begin{aligned}
\mathit{fwdOut}([1])(\mathtt{x}) &= \top \\
\mathit{fwdOut}([2])(\mathtt{y}) &= \mathit{fwdOut}([4])(\mathtt{y}) = 1
\end{aligned}$$

and the last condition motivates us to define

$$0 \oplus 1 = 1.$$

Monotonicity $1 \oplus \bot \sqsubseteq 1 \oplus 0$ yields $1 \oplus \bot \sqsubseteq 1$, but defining $1 \oplus \bot = \bot$ would lead to the least fixed point for $\mathit{fwdOut}([4])(\mathtt{x})$ being $\bot$, hence to $\mathit{fwdOut}([1])(\mathtt{x}) = \bot \neq \top$, in contradiction to our requirement. We therefore stipulate

$$1 \oplus \bot \neq \bot.$$

For the program on the right, the interesting equations are

$$\begin{aligned}
\mathit{fwdOut}([3])(\mathtt{y}) &= \mathit{fwdIn}([4])(\mathtt{y}) = 0 \oplus (\mathit{fwdIn}([5])(\mathtt{y}) \sqcup \mathit{fwdIn}([3])(\mathtt{y})) \\
\mathit{fwdIn}([5])(\mathtt{y}) &= 1 \oplus 0 \\
\mathit{fwdIn}([3])(\mathtt{y}) &= 0
\end{aligned}$$

which simplify to

$$\mathit{fwdOut}([3])(\mathtt{y}) = 0 \oplus ((1 \oplus 0) \sqcup 0) = 0 \oplus (1 \sqcup 0).$$

The intended value is $\mathit{fwdOut}([3])(\mathtt{y}) = \top$ since y cannot be forwarded – it is not used inside the loop. This motivates us to require $1 \sqcup 0 = \top$ which yields the diamond shape of $\mathcal{L}$.

For defining $\oplus$, the entries for $\top \oplus l$ and the values on the diagonal are clearly motivated by the purpose of the analysis. We observed $0 \oplus 1 = 1$ above, and noted $\bot \neq 1 \oplus \bot \sqsubseteq 1$, so the only option left is to set

$$1 \oplus \bot = 1.$$

Finally, the only choice for $\bot \oplus 0$ is

$$\bot \oplus 0 = \bot$$

since all other choices violate monotonicity – consider $1 \oplus 0 = 1$ for the alternative values 0 and $\top$ and $0 \oplus 0 = 0$ for 1. $\diamond$

Example. The analysis for the example programs (6.5) to (6.8) is as follows. Table 6.1 shows the sets $\mathit{defs}(I)$ and the pointwise functions $\mathit{uses}(I)$ for all instructions. For example, $\{\mathtt{z}\} = \mathit{defs}([4])$ holds in program (6.7) and $\mathit{uses}([6])(\mathtt{y1}) = \top$ in program (6.8).

Applying equations (6.13) to program (6.5) yields Table 6.2 from which

$$\mathit{fwdIn}([6])(\mathtt{x}) = 1 \oplus \mathit{fwdOut}([6])(\mathtt{x})$$

and

$$\mathit{fwdOut}([3])(\mathtt{u}) = \mathit{fwdIn}([4])(\mathtt{u}) \sqcup \mathit{fwdIn}([6])(\mathtt{u})$$

can be deduced. The data in Table 6.2 simplifies to Table 6.3. We can discover the linear usage of x by observing that $\mathit{fwdOut}([2])(\mathtt{x}) = 1$ holds where $[2]$ is the only instruction I with $\mathtt{x} \in \mathit{defs}(I)$. Likewise, both assignments to u are used linearly as for $I \in \{[4],[6]\}$ we have $\mathtt{u} \in \mathit{defs}(I)$ and $\mathit{fwdOut}(I)(\mathtt{u}) = 1$.

Similarly, Table 6.4 shows the result of applying equations (6.13) to program (6.6). For the variable y we obtain

$$\begin{aligned}
\mathit{fwdOut}([2])(\mathtt{y}) &= \ldots = & \mathit{fwdIn}([4])(\mathtt{y}) \sqcup \mathit{fwdIn}([7])(\mathtt{y}) \\
\mathit{fwdIn}([4])(\mathtt{y}) &= \ldots = & 1 \oplus (\mathit{fwdIn}([4])(\mathtt{y}) \sqcup \mathit{fwdIn}([7])(\mathtt{y})) \\
\mathit{fwdIn}([7])(\mathtt{y}) &= \ldots = & 0
\end{aligned}$$

I	(6.5)				(6.6)				(6.7)				(6.8)				
	defs	*uses*			*defs*	*uses*			*defs*	*uses*			*defs*	*uses*			
		x	u	v		y	u	v		w	z	v		y1	y2	y3	v
[1]	{v}	0	0	0	{v}	0	0	0	{v}	0	0	0	{v}	0	0	0	0
[2]	{x}	0	0	0	{y}	0	0	0	{w}	0	0	0	{y1}	0	0	0	0
[3]	$\emptyset$	0	0	1	$\emptyset$	0	0	1	$\emptyset$	0	0	1	$\emptyset$	0	0	0	1
[4]	{u}	1	0	0	{u}	1	0	0	{z}	1	0	0	{y2}	1	0	0	0
[5]	$\emptyset$	0	0	0	{v}	0	0	1	{w}	0	1	0	$\emptyset$	0	0	0	0
[6]	{u}	1	0	0	$\emptyset$	0	0	1	{v}	0	0	1	{y3}	$\top$	0	0	0
[7]	$\emptyset$	0	0	0	{v}	0	0	0	$\emptyset$	0	0	1	$\emptyset$	0	0	0	0
[8]	{v}	0	1	0	–	–	–	–	{v}	1	0	0	{v}	0	0	0	0

Table 6.1: Dataflow information for programs (6.5) to (6.8).

Since the least fixed point of $l = 1 \oplus (l \sqcup 0)$ is $\top$, this equation yields $\mathit{fwdIn}([4])(\mathtt{y}) = \top$, hence $\mathit{fwdOut}([2])(\mathtt{y}) = \top$, and we discover the non-linear usage of y due to $\mathtt{y} \in \mathit{defs}([2])$.

For program (6.7) we similarly obtain the equations

$$\begin{aligned}
\mathit{fwdOut}([2])(\mathtt{w}) &= \mathit{fwdIn}([4])(\mathtt{w}) \sqcup \mathit{fwdIn}([8])(\mathtt{w})\\
\mathit{fwdOut}([5])(\mathtt{w}) &= \mathit{fwdIn}([4])(\mathtt{w}) \sqcup \mathit{fwdIn}([8])(\mathtt{w})\\
\mathit{fwdIn}([4])(\mathtt{w}) &= \mathit{fwdIn}([8])(\mathtt{w}) = 1\\
\mathit{fwdOut}([4])(\mathtt{z}) &= 1 \oplus (\mathit{fwdIn}([4])(\mathtt{z}) \sqcup \mathit{fwdIn}([8])(\mathtt{z}))\\
\mathit{fwdIn}([4])(\mathtt{z}) &= \mathit{fwdIn}([8])(\mathtt{z}) = 0\\
\mathit{fwdOut}([1])(\mathtt{v}) &= 1 \oplus (\mathit{fwdIn}([4])(\mathtt{v}) \sqcup \mathit{fwdIn}([8])(\mathtt{v}))\\
\mathit{fwdIn}([4])(\mathtt{v}) &= 1\\
\mathit{fwdIn}([8])(\mathtt{v}) &= 0
\end{aligned}$$

and thus correctly identify z and w as being used linearly and v as being non-linear. Finally, in program (6.8), $\mathit{uses}([6])(\mathtt{y1}) = \top$ yields $\mathit{fwdOut}([2])(\mathtt{y1}) = \top$, and we correctly identify y1 as being used non-linearly. $\diamond$

I	*fwdIn*		
	x	u	v
[1]	$\mathit{fwdOut}([1])$	$\mathit{fwdOut}([1])$	0
[2]	0	$\mathit{fwdOut}([2])$	$\mathit{fwdOut}([2])$
[3]	$\mathit{fwdOut}([3])$	$\mathit{fwdOut}([3])$	$1 \oplus \mathit{fwdOut}([3])$
[4]	$1 \oplus \mathit{fwdOut}([4])$	0	$\mathit{fwdOut}([4])$
[5]	$\mathit{fwdOut}([5])$	$\mathit{fwdOut}([5])$	$\mathit{fwdOut}([5])$
[6]	$1 \oplus \mathit{fwdOut}([6])$	0	$\mathit{fwdOut}([6])$
[7]	$\mathit{fwdOut}([7])$	$\mathit{fwdOut}([7])$	$\mathit{fwdOut}([7])$
[8]	$\mathit{fwdOut}([8])$	$1 \oplus \mathit{fwdOut}([8])$	$\mathit{fwdOut}([8])$

I	*fwdOut*
	x u v
[1]	$\mathit{fwdIn}([2])$
[2]	$\mathit{fwdIn}([3])$
[3]	$\mathit{fwdIn}([4]) \sqcup \mathit{fwdIn}([6])$
[4]	$\mathit{fwdIn}([5])$
[5]	$\mathit{fwdIn}([8])$
[6]	$\mathit{fwdIn}([7])$
[7]	$\mathit{fwdIn}([8])$
[8]	0

Table 6.2: Equations (6.13) applied to program (6.5).

I	*fwdIn*			*fwdOut*		
	x	u	v	x	u	v
[1]	0	0	0	0	0	1
[2]	0	0	1	1	0	1
[3]	1	0	1	1	0	0
[4]	1	0	0	0	1	0
[5]	0	1	0	0	1	0
[6]	1	0	0	0	1	0
[7]	0	1	0	0	1	0
[8]	0	1	0	0	0	0

Table 6.3: Simplification of Table 6.2

I	*fwdIn*			*fwdOut*
	y	u	v	y u v
[1]	$\mathit{fwdOut}([1])$	$\mathit{fwdOut}([1])$	0	$\mathit{fwdIn}([2])$
[2]	0	$\mathit{fwdOut}([2])$	$\mathit{fwdOut}([2])$	$\mathit{fwdIn}([3])$
[3]	$\mathit{fwdOut}([3])$	$\mathit{fwdOut}([3])$	$1 \oplus \mathit{fwdOut}([3])$	$\mathit{fwdIn}([4]) \sqcup \mathit{fwdIn}([7])$
[4]	$1 \oplus \mathit{fwdOut}([4])$	0	$\mathit{fwdOut}([4])$	$\mathit{fwdIn}([5])$
[5]	$\mathit{fwdOut}([5])$	$\mathit{fwdOut}([5])$	1	$\mathit{fwdIn}([6])$
[6]	$\mathit{fwdOut}([6])$	$\mathit{fwdOut}([6])$	$1 \oplus \mathit{fwdOut}([6])$	$\mathit{fwdIn}([4]) \sqcup \mathit{fwdIn}([7])$
[7]	$\mathit{fwdOut}([7])$	$\mathit{fwdOut}([7])$	0	0

Table 6.4: Equations (6.13) applied to program (6.6).

6.3.3 Usage analysis for SSA

In the presence of Φ-functions, we use modified definitions of *uses*, *fwdIn* and *fwdOut*. As motivation, consider a block $B = \phi\vec{I}$ with ϕ of the form $[k](x_3) = \begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} n \\ m \end{pmatrix}$ and incoming control flow arrows from instructions $lst(B_1)$ and $lst(B_2)$ labelled n and m, respectively, such that each B_i contains an assignment to x_i. When calculating $fwdIn(\phi)$, only the usage of x_1 should be promoted upwards to B_1 (it is not available in B_2) and only the usage of x_2 to B_2.

By providing the control flow successor through an additional argument, the functionalities of *uses*, *fwdIn* and *fwdOut* are generalised to

$$\begin{gathered} uses : \mathbf{Instrs}_{\mathrm{P}} \to \mathbf{N}_{\mathrm{P}} \to \mathbf{Var}_{\mathrm{P}} \to \mathcal{L} \\ fwdIn, fwdOut : \mathbf{Instrs}_{\mathrm{P}} \to \mathbf{N}_{\mathrm{P}} \to \mathbf{Var}_{\mathrm{P}} \to \mathcal{L} \end{gathered} \tag{6.15}$$

For ϕ of the form $[l]\mathbf{x}(k) = \mathbf{M}(k,m)\mathbf{n}(m)$, we define

$$uses(\phi)(n)(y) = \begin{cases} \oplus_{i=1}^{k} uses(x_{ij})(y) & \text{if } n = n_j \text{ for some } j \\ 0 & \text{if } \forall 1 \leq j \leq m.n \neq n_j \end{cases} \tag{6.16}$$

and for non-Phi instructions we define

$$uses(I)(n)(y) = uses'(I)(y) \tag{6.17}$$

where $uses'$ is the definition of *uses* in the previous section. For all instruction forms, functions *fwdIn* and *fwdOut* are defined by

$$fwdIn(I)(n)(y) \quad = \quad \begin{cases} uses(I)(n)(y) & \text{if } y \in defs(I) \\ uses(I)(n)(y) \oplus fwdOut(I)(n)(y) & \text{otherwise} \end{cases} \tag{6.18}$$

and

$$fwdOut(I)(n)(x) \quad = \quad \begin{cases} 0 & \text{if } \mathtt{succ}(I) = \emptyset \\ \sqcup_{J \in \mathtt{succ}(I)} fwdIn(J)(k)(x) & \text{if } \mathtt{succ}(I) \neq \emptyset \end{cases} \tag{6.19}$$

where k is the label of I.

Similarly to the previous subsection, we obtain solutions to the dataflow equations by calculating the (least) fixed point. Functions in $\mathbf{N}_{\mathrm{P}} \to \mathbf{Var}_{\mathrm{P}} \to \mathcal{L}$ may be ordered pointwise by

$$\varphi \sqsubseteq \psi \text{ iff } \forall n \in \mathbf{N}_{\mathrm{P}}.\varphi(n) \sqsubseteq \psi(n)$$

where $\varphi(n) \sqsubseteq \psi(n)$ is the partial order over $\mathbf{Var}_{\mathrm{P}} \rightarrow \mathcal{L}$ defined above. Consequently, the function space

$$\mathbf{FSP}_{SSA} = \mathbf{Instrs}_{\mathrm{P}} \rightarrow \mathbf{N}_{\mathrm{P}} \rightarrow \mathbf{Var}_{\mathrm{P}} \rightarrow \mathcal{L}$$

is ordered by

$$f \sqsubseteq g \text{ iff } \forall I \in \mathbf{Instrs}_{\mathrm{P}}. f(I) \sqsubseteq g(I)$$

where $\lambda I.(\lambda n.(\lambda x.\bot))$ is the least element. Hence,

$$update : (\mathbf{FSP}_{SSA} \times \mathbf{FSP}_{SSA}) \rightarrow (\mathbf{FSP}_{SSA} \times \mathbf{FSP}_{SSA})$$

defined by

$$\begin{array}{rcll} update(f,g) & = & (\;\lambda I.\lambda n.\lambda x. & \text{if } \mathtt{succ}(I) = \emptyset \text{ then } 0 \\ & & & \text{else let } k = label(I) \text{ in } \sqcup_{J \in \mathtt{succ}(I)} g(J)(k)(x), \\ & & \lambda I.\lambda n.\lambda x. & \text{if } x \in defs(I) \text{ then } uses(I)(n)(x) \\ & & & \text{else } uses(I)(n)(x) \oplus f(I)(n)(x)) \end{array}$$

is monotone: $f \sqsubseteq f'$ implies by definition $f(I) \sqsubseteq f'(I)$ for all $I \in \mathbf{Instrs}_{\mathrm{P}}$, hence $f(I)(n) \sqsubseteq f'(I)(n)$ for all $I \in \mathbf{Instrs}_{\mathrm{P}}$ and $n \in \mathbf{N}_{\mathrm{P}}$ and $f(I)(n)(x) \sqsubseteq f'(I)(n)(x)$ for all $I \in \mathbf{Instrs}_{\mathrm{P}}$, $n \in \mathbf{N}_{\mathrm{P}}$ and $x \in \mathbf{Var}_{\mathrm{P}}$, thus $l \oplus f(I)(n)(x) \sqsubseteq l \oplus f'(I)(n)(x)$ for all $I \in \mathbf{Instrs}_{\mathrm{P}}$, $n \in \mathbf{N}_{\mathrm{P}}$, $x \in \mathbf{Var}_{\mathrm{P}}$ and $l \in \mathcal{L}$ and therefore

$$\begin{array}{rcll} update(f,g) & = & (\lambda I.\lambda n.\lambda x. & \text{if } \mathtt{succ}(I) = \emptyset \text{ then } 0 \\ & & & \text{else let } k = label(I) \text{ in } \sqcup_{J \in \mathtt{succ}(I)} g(J)(k)(x), \\ & & \lambda I.\lambda n.\lambda x. & \text{if } x \in defs(I) \text{ then } uses(I)(n)(x) \\ & & & \text{else } uses(I)(n)(x) \oplus f(I)(n)(x)) \\ & \sqsubseteq & (\lambda I.\lambda n.\lambda x. & \text{if } \mathtt{succ}(I) = \emptyset \text{ then } 0 \\ & & & \text{else let } k = label(I) \text{ in } \sqcup_{J \in \mathtt{succ}(I)} g(J)(k)(x), \\ & & \lambda I.\lambda n.\lambda x. & \text{if } x \in defs(I) \text{ then } uses(I)(n)(x) \\ & & & \text{else } uses(I)(n)(x) \oplus f'(I)(n)(x)) \\ & = & update(f',g) & \end{array}$$

Similarly, one may show $update(f,g) \sqsubseteq update(f,g')$ for $g \sqsubseteq g'$, hence the finiteness of the lattice $\mathbf{FSP}_{SSA}$ and the theorem of Knaster-Tarski guarantee the existence of (least) fixed points.

Example. Consider the following program $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, 1)$

$$
\begin{array}{rclrcl}
\mathtt{N}(1) & = & [1]\mathtt{v} = 7 & \mathtt{N}(6) & = & [6]\mathtt{y2} = 7 \\
 & & [2]\mathtt{if\ v}\ 3\ 6 & & & [7]\mathtt{z2} = \mathtt{y2} * 2 \\
\mathtt{N}(3) & = & [3]\mathtt{y1} = 3 & & & [8]\mathtt{jmp}\ 9 \\
 & & [4]\mathtt{z1} = 2 & \mathtt{N}(9) & = & [9]\begin{pmatrix}\mathtt{y}\\ \mathtt{z}\end{pmatrix} = \begin{pmatrix}\mathtt{y1} & \mathtt{y2}\\ \mathtt{z1} & \mathtt{z2}\end{pmatrix}\begin{pmatrix}5\\ 8\end{pmatrix} \\
 & & [5]\mathtt{jmp}\ 9 & & & [10]\mathtt{w} = \mathtt{z} + \mathtt{y}
\end{array}
\tag{6.20}
$$

The definition of *uses* yields

$$
\begin{aligned}
uses([2])(m)(\mathtt{v}) &= uses([7])(m)(\mathtt{y2}) = 1 \text{ for all } m \\
uses([9])(5)(\mathtt{y1}) &= uses([9])(5)(\mathtt{z1}) = 1 \\
uses([9])(8)(\mathtt{y2}) &= uses([9])(8)(\mathtt{z2}) = 1 \\
uses([10])(m)(\mathtt{y}) &= uses([10])(m)(\mathtt{z}) = 1 \text{ for all } m
\end{aligned}
$$

and $uses(I)(m)(x) = 0$ elsewhere. We consequently obtain

$$
\begin{aligned}
fwdOut([10])(m)(x) &= 0 \\
fwdIn([10])(m)(x) &= \begin{cases} uses([10])(m)(x) & \text{if } x = \mathtt{w} \\ uses([10])(m)(x) \oplus fwdOut([10])(m)(x) & \text{otherwise} \end{cases} \\
&= \begin{cases} 1 & \text{if } x \in \{\mathtt{z}, \mathtt{y}\} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
fwdOut([9])(m)(x) &= fwdIn([10])([9])(x) = \begin{cases} 1 & \text{if } x \in \{\mathtt{z}, \mathtt{y}\} \\ 0 & \text{otherwise} \end{cases} \\
fwdIn([9])(m)(x) &= \begin{cases} uses([9])(m)(\mathtt{w}) & \text{if } x \in \{\mathtt{y}, \mathtt{z}\} \\ uses([9])(m)(\mathtt{w}) \oplus fwdOut([9])(m)(\mathtt{w}) & \text{otherwise} \end{cases} \\
&= \begin{cases} 1 & \text{if } x \in \{\mathtt{z1}, \mathtt{y1}\} \text{ and } m = 5 \\ 1 & \text{if } x \in \{\mathtt{z2}, \mathtt{y2}\} \text{ and } m = 8 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

$$\begin{aligned}
\mathit{fwdOut}([8])(m)(x) &= \mathit{fwdIn}([9])([8])(x) \\
&= \begin{cases} 1 & \text{if } x \in \{\mathtt{z2}, \mathtt{y2}\} \\ 0 & \text{otherwise} \end{cases} \\
\mathit{fwdIn}([8])(m)(x) &= \mathit{uses}([8])(m)(x) \oplus \mathit{fwdOut}([8])(m)(x) \\
&= \begin{cases} 1 & \text{if } x \in \{\mathtt{z2}, \mathtt{y2}\} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\mathit{fwdOut}([7])(m)(x) &= \mathit{fwdIn}([8])([7])(x) = \begin{cases} 1 & \text{if } x \in \{\mathtt{z2}, \mathtt{y2}\} \\ 0 & \text{otherwise} \end{cases} \\
\mathit{fwdIn}([7])(m)(x) &= \begin{cases} \mathit{uses}([7])(m)(x) & \text{if } x \in \{\mathtt{z2}\} \\ \mathit{uses}([7])(m)(x) \oplus \mathit{fwdOut}([7])(m)(x) & \text{otherwise} \end{cases} \\
&= \begin{cases} \top & \text{if } x \in \{\mathtt{y2}\} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\mathit{fwdOut}([6])(m)(x) &= \mathit{fwdIn}([7])([6])(x) = \begin{cases} \top & \text{if } x \in \{\mathtt{y2}\} \\ 0 & \text{otherwise} \end{cases} \\
\mathit{fwdIn}([6])(m)(x) &= \begin{cases} \mathit{uses}([6])(m)(x) & \text{if } x \in \{\mathtt{y2}\} \\ \mathit{uses}([6])(m)(x) \oplus \mathit{fwdOut}([6])(m)(x) & \text{otherwise} \end{cases} \\
&= 0
\end{aligned}$$

$$\begin{aligned}
\mathit{fwdOut}([5])(m)(x) &= \mathit{fwdIn}([9])([5])(x) \\
&= \begin{cases} 1 & \text{if } x \in \{\mathtt{z1}, \mathtt{y1}\} \\ 0 & \text{otherwise} \end{cases} \\
\mathit{fwdIn}([5])(m)(x) &= \mathit{uses}([5])(m)(x) \oplus \mathit{fwdOut}([5])(m)(x) \\
&= \begin{cases} 1 & \text{if } x \in \{\mathtt{z1}, \mathtt{y1}\} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

$$
\begin{aligned}
\mathit{fwdOut}([4])(m)(x) &= \mathit{fwdIn}([5])([4])(x) = \begin{cases} 1 & \text{if } x \in \{\mathtt{z1}, \mathtt{y1}\} \\ 0 & \text{otherwise} \end{cases} \\
\mathit{fwdIn}([4])(m)(x) &= \begin{cases} \mathit{uses}([4])(m)(x) & \text{if } x \in \{\mathtt{z1}\} \\ \mathit{uses}([4])(m)(x) \oplus \mathit{fwdOut}([4])(m)(x) & \text{otherwise} \end{cases} \\
&= \begin{cases} 1 & \text{if } x \in \{\mathtt{y1}\} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fwdOut}([3])(m)(x) &= \mathit{fwdIn}([4])([3])(x) = \begin{cases} 1 & \text{if } x \in \{\mathtt{y1}\} \\ 0 & \text{otherwise} \end{cases} \\
\mathit{fwdIn}([3])(m)(x) &= \begin{cases} \mathit{uses}([3])(m)(x) & \text{if } x \in \{\mathtt{y1}\} \\ \mathit{uses}([3])(m)(x) \oplus \mathit{fwdOut}([3])(m)(x) & \text{otherwise} \end{cases} \\
&= 0
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fwdOut}([2])(m)(x) &= \mathit{fwdIn}([3])([2])(x) \sqcup \mathit{fwdIn}([6])([2])(x) = 0 \\
\mathit{fwdIn}([2])(m)(x) &= \mathit{uses}([2])(m)(x) \oplus \mathit{fwdOut}([2])(m)(x) \\
&= \begin{cases} 1 & \text{if } x \in \{\mathtt{v}\} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fwdOut}([1])(m)(x) &= \mathit{fwdIn}([2])([1])(x) = \begin{cases} 1 & \text{if } x \in \{\mathtt{v}\} \\ 0 & \text{otherwise} \end{cases} \\
\mathit{fwdIn}([1])(m)(x) &= \begin{cases} \mathit{uses}([1])(m)(x) & \text{if } x \in \{\mathtt{v}\} \\ \mathit{uses}([1])(m)(x) \oplus \mathit{fwdOut}([1])(m)(x) & \text{otherwise} \end{cases} \\
&= 0
\end{aligned}
$$

Note that for all I, x and m_1, m_2, $\mathit{fwdOut}(I)(m_1)(x) = \mathit{fwdOut}(I)(m_2)(x)$ holds whenever $x \in \mathit{defs}(I)$. The non-linear usages of y2 and w can be detected by observing that $\mathit{fwdOut}([6])(m)(\mathtt{y2}) = \top$ and $\mathit{fwdOut}([10])(m)(\mathtt{w}) = 0$ hold, where $[6] = I_{\mathtt{y2}}$ and $[10] = I_{\mathtt{w}}$. For all other pairs (I, x) with $x \in \mathit{defs}(I)$, $\mathit{fwdOut}(I)(m)(x) = 1$ holds and these variables are indeed used exactly once. $\diamond$

6.4 Soundness of usage analysis

For proving the soundness of the dataflow analysis we define a more fine-grained operational model $\Rightarrow$. This non-standard dynamic semantics reflects our understanding of limited reading access. Each time a variable is assigned to, the semantics associates a tag with the value saying how many times the value may be accessed for reading. We distinguish between unlimited reading capability, single reading capability and no reading capability. Variables of the latter category may not be accessed for reading but may be assigned to. Variables whose tag is of the 'read-once' form may be accessed for reading but not be assigned to, and the first reading operation changes the tag to 'no-reads'. Variables carrying an 'unlimited' tag may be read arbitrarily often and may also be reassigned to.

We show that the dynamic semantics including tags is sound with respect to the semantics $\rightarrow$ of IL: any $\Rightarrow$-execution abstracts to a $\rightarrow$-execution. On the other hand, $\Rightarrow$ is relative complete for $\rightarrow$ in the sense that for any $\rightarrow$-execution there is an annotation for the variables for which $\Rightarrow$ mirrors $\rightarrow$. Finally, we show that the dataflow analysis is sound for $\Rightarrow$, i.e. that annotations resulting from the dataflow analysis guarantee that a program does not get stuck by performing too many reading accesses to a variable or by reassigning too early.

6.4.1 Non-standard dynamic semantics

The capabilities consist of elements of the lattice $\mathcal{L}$: the unlimited capability is $\top$ and the 'read-once' and 'no-read' capabilities are given by 1 and 0 respectively. Capabilities are incorporated into the operational model by extending the codomain of states to $\mathcal{L} \times Val$, and extended states are ranged over by mappings $\Sigma : \mathbf{Var}_{\mathrm{P}} \rightharpoonup (\mathcal{L} \times Val)$. The update operations $\Sigma[x \mapsto (l,a)]$ and $\Sigma[x_1 \mapsto (l_1,a_1),\ldots,x_n \mapsto (l_n,a_n)] = \Sigma[x_1 \mapsto (l_1,a_1)]\ldots[x_n \mapsto (l_n,a_n)]$ (for distinct x_i) are given by the general notions defined in Chapter 2.

The evaluation relation $\Downarrow$ modifies the capabilities as described above

$$\frac{}{\Sigma, a \Downarrow \Sigma, a} \qquad\qquad \frac{\Sigma, x \Downarrow \Sigma_1, a}{\Sigma, x\ bop\ b \Downarrow \Sigma_1, BOP(a,b)}$$

$$\frac{x \in dom\,\Sigma \quad \Sigma(x) = (l,a) \quad l \sqsupseteq 1 \quad \Sigma_1 = \Sigma[x \mapsto (l \ominus 1, a)]}{\Sigma, x \Downarrow \Sigma_1, a}$$

$$\frac{\Sigma, x \Downarrow \Sigma_1, a \quad \Sigma_1, y \Downarrow \Sigma_2, b}{\Sigma, x\ bop\ y \Downarrow \Sigma_2, BOP(a,b)}$$

where we define

$$l_1 \ominus l_2 = \sqcup\{l \mid l \oplus l_2 = l_1\}.$$

In particular, the condition $l \sqsupseteq 1$ in the rule for $\Sigma, x \Downarrow \Sigma_1, a$ forbids reading from a variable tagged with 0. The complementary write permission is denoted by $\downarrow$.

Definition 34. *For $\Sigma : \mathbf{Var}_P \rightharpoonup (\mathcal{L} \times Val)$ and $x \in \mathbf{Var}_P$, we write $\Sigma \downarrow x$ if $\Sigma(x) = (l,a)$ implies $l \sqsupseteq 0$.*

The definition of $\ominus$ may be expanded using $\sqcup\emptyset = \bot$, which yields the table

$\ominus$	$\bot$	0	1	$\top$
$\bot$	0	$\bot$	$\bot$	$\bot$
0	$\bot$	0	$\bot$	$\bot$
1	1	1	0	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$

and the following properties.

Lemma 11. *Let $a,b,c \in \mathcal{L}$. Then*

1. $a \ominus a \sqsupseteq 0$
2. $(a \oplus b) \ominus b \sqsupseteq a$
3. $(1 \oplus a \oplus b) \ominus a \sqsupseteq 1$
4. $(a \ominus b) \ominus c = a \ominus (b \oplus c)$ *for* $b, c \neq \bot$.
5. $a \sqsupseteq b \neq \bot$ *implies* $a \ominus c \sqsupseteq b \ominus c$.

Proof. Inspection of the tables defining $\oplus$ and $\ominus$. □

The dynamic semantics for extended configurations is given by the relation $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$ which is parametric in an association function α. This function assigns to each pair (I,x) such that $I \in \mathbf{Instrs}_{\mathsf{P}}$ and $x \in uses(I)$ an element $\alpha(I,x) \in \mathcal{L}$, representing the tag to be associated with x during an assignment.

$$
\begin{gathered}
\text{JMP}\,\frac{}{(\Sigma, n, [k]\texttt{jmp}\; m) \Rightarrow_\alpha (\Sigma, k, \mathtt{N}(m))} \\[2ex]
\text{IF-T}\,\frac{\Sigma, x \Downarrow \Sigma_1, 0}{(\Sigma, n, [k]\texttt{if}\; x\; m_1\; m_2) \Rightarrow_\alpha (\Sigma_1, k, \mathtt{N}(m_1))} \\[2ex]
\text{IF-F}\,\frac{\Sigma, x \Downarrow \Sigma_1, a}{(\Sigma, n, [k]\texttt{if}\; x\; m_1\; m_2) \Rightarrow_\alpha (\Sigma_1, k, \mathtt{N}(m_2))}\; a \neq \mathsf{0} \\[2ex]
\text{ASS}\,\frac{\Sigma, e \Downarrow \Sigma_1, a}{(\Sigma, n, ass\vec{I}) \Rightarrow_\alpha (\Sigma_1[x \mapsto (\alpha(ass, x), a)], n, \vec{I})} \left\{ \begin{array}{c} ass = [k]x = e \\ \Sigma_1 \downarrow x \end{array} \right. \\[2ex]
\text{PHI}\,\frac{\forall 1 \leq i \leq k. \Sigma_i, x_{ij} \Downarrow \Sigma_{i+1}, a_i}{(\Sigma_1, n, \phi\vec{I}) \Rightarrow_\alpha (\Sigma, n, \vec{I})} \left\{ \begin{array}{c} \phi = [l]\mathbf{x}(k) = \mathbf{M}(m,k)\mathbf{n}(m) \\ n = n_j \\ \forall 1 \leq i \leq k. \Sigma_{k+1} \downarrow x_i \\ \Sigma = \Sigma_{k+1}[x_i \mapsto (\alpha(\phi, x_i), a_i)] \end{array} \right.
\end{gathered}
\tag{6.21}
$$

In order to justify the occurrence of lattice elements in the dynamic semantics, we observe that an entry $\Sigma(x) = (l, a)$ admits

- no reading access if $l = 0$ since $\Sigma, x \not\Downarrow \Sigma_1, b$ for all Σ_1 and b because the side condition $l \sqsupseteq 1$ is violated

- exactly one reading access if $l = 1$, since $\Sigma, x \Downarrow \Sigma_1, b$ implies $b = a$ and $\Sigma_1(x) = (1 \ominus 1, a) = (0, a)$.

- arbitrarily many reading accesses (including zero) if $l = \top$ as $\Sigma, x \Downarrow \Sigma_1, b$ implies $b = a$ and $\Sigma_1 = \Sigma$ due to $\top \ominus 1 = \top$

- a writing access exactly if $l \sqsupseteq 0$ holds, due to the side condition $\Sigma \downarrow x$ in rules ASS and PHI

Initialising annotations l for variable x during each assignment to x thus determines how often the assigned value *can* be read, but also how often it *must* be read prior to a subsequent assignment. In particular, a value with $l = 1$ *must* be read exactly once before it can be assigned to again, but also *can* be read from only once. Values with $l = 1$ are thus candidates for forwarding.

By incorporating the lattice elements into the operational model, the dataflow analysis may be proven sound in a different approach than [NNH99]. Configurations are naturally related to the functions *fwdIn*/*fwdOut* using the tags, and the duality between $\ominus$ and $\oplus$ embodies the directional relationship between (forward) program execution and (backward) dataflow analysis.

Before proving soundness, we relate $\Rightarrow$ to $\rightarrow$.

Definition 35. *The tag-free state* $\widehat{\Sigma} : \mathbf{Var}_{\mathtt{P}} \rightarrow Val$ *corresponding to* $\Sigma : \mathbf{Var}_{\mathtt{P}} \rightarrow (\mathcal{L} \times Val)$ *is given pointwise by*

$$\widehat{\Sigma}(x) = a \textit{ iff } \exists l.\Sigma(x) = (l,a).$$

For $\mathsf{S} = (\Sigma,n,B)$ *we also write* $\widehat{\mathsf{S}}$ *for the configuration* $(\widehat{\Sigma},n,B)$*.*

Proposition 31. *(Soundness of* $\Rightarrow$ *w.r.t.* $\rightarrow$*)*

$$\mathsf{S} \Rightarrow_{\alpha} \mathsf{T} \textit{ implies } \widehat{\mathsf{S}} \rightarrow \widehat{\mathsf{T}}$$

Proof. Induction on the rule used for $(\Sigma,n,B) \Rightarrow_{\alpha} (\Sigma',n',B')$.

Case JMP. For $B = [k]\mathtt{jmp}\ m$, the definition of rule JMP implies $\Sigma = \Sigma'$, $n' = n$ and $B' = \mathtt{N}(m)$. Thus, $\widehat{\Sigma} = \widehat{\Sigma'}$ holds and the claim holds trivially by rule Jmp.

Case IF-T. For $B = [k]\mathtt{if}\ x\ m\ l$, the definition of rule IF-T implies $\Sigma = \Sigma_1$, $n' = k$ and $B' = \mathtt{N}(m)$ where $\Sigma, x \Downarrow \Sigma_1, 0$. For $\Sigma, x \Downarrow \Sigma_1, 0$ to hold, we must have $\Sigma(x) = (l,0)$ with $1 \sqsubseteq l$ and $\Sigma_1 = \Sigma[x \mapsto (l \ominus 1, 0)]$. Consequently, $\widehat{\Sigma}(x) = 0$ and $\widehat{\Sigma} = \widehat{\Sigma_1}$, so the claim follows by rule If-t.

Case IF-F. Similar to the previous case.

Case ASS. For $B = ass\vec{I}$ and $ass = [k]x = e$, the definition of rule ASS implies $\Sigma' = \Sigma_1[x \mapsto (\alpha(ass,x),a]$, $n' = n$ and $B' = \vec{I}$ where $\Sigma, e \Downarrow \Sigma_1, a$. Consequently, $\widehat{\Sigma}, e \Downarrow a$, $\widehat{\Sigma_1} = \widehat{\Sigma}$ and

$$\widehat{\Sigma'} = \Sigma_1[x \mapsto \widehat{(\alpha(ass,x),a)}] = \widehat{\Sigma_1}[x \mapsto a] = \widehat{\Sigma}[x \mapsto a]$$

hold so the claim follows from the definition of rule Ass.

Case PHI. For $B = \phi\vec{I}$ and $\phi = [l]\mathbf{x}(k) = \mathbf{M}(k,m)\mathbf{n}(m)$, the definition of rule PHI implies

- $n = n_j$ for a $1 \leq j \leq m$,
- $\Sigma_i, x_{ij} \Downarrow \Sigma_{i+1}, a_i$ for some $\Sigma = \Sigma_1, \ldots, \Sigma_{k+1}$ and all $1 \leq i \leq k$,
- $\Sigma_{k+1} \downarrow x_i$ for all $1 \leq i \leq k$
- $n' = n$, $B' = \vec{I}$ and $\Sigma' = \Sigma_{k+1}[x_i \mapsto (\alpha(\phi,x),a_i)]$

By the definition of $\Downarrow$, $\widehat{\Sigma_i} = \widehat{\Sigma_1}$ and $\widehat{\Sigma}_i, x_i \Downarrow a_i$ hold for all $1 \leq i \leq k$. Also, $\widehat{\Sigma_{k+1}} = \widehat{\Sigma_1}$ and $\widehat{\Sigma'} = \widehat{\Sigma_1}[x_i \mapsto a_i]$, and the claim follows from the definition of rule Phi.

□

By combining Propositions 31 and 29 we obtain that $\widehat{\mathsf{T}} = \widehat{\mathsf{R}}$ holds whenever $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$ and $\mathsf{S} \Rightarrow_\beta \mathsf{R}$, for any α and β. As α, $\Downarrow$ and $\downarrow$ are deterministic, one may also show that $\mathsf{T} = \mathsf{R}$ holds if $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$ and $\mathsf{S} \Rightarrow_\alpha \mathsf{R}$ by adapting the proof of Proposition 29 to $\Rightarrow_\alpha$.

Definition 36. *For program* $\mathtt{P}$ *and configuration* $\mathsf{S} = (\sigma, n, B)$*, the annotation* $(\sigma)_\alpha$ *of* σ *according to annotation function* $\alpha : (\mathbf{Instrs}_\mathtt{P} \times \mathbf{Var}_\mathtt{P}) \to \mathcal{L}$ *is given by*

$$(\sigma)_\alpha(x) = (\sqcup_{x \in defs(I)} \alpha(I,x), a) \textit{ iff } \sigma(x) = a.$$

and the α*-annotated configuration* $(\mathsf{S})_\alpha$ *by* $((\sigma)_\alpha, n, B)$*.*

Consequently,

$$\widehat{(\sigma)_\alpha} = \sigma$$

and

$$(\widehat{\Sigma})_\alpha(x) = (\sqcup_{x \in defs(I)} \alpha(I,x), a) \text{ iff } \Sigma(x) = (l,a)$$

hold. For SSA programs, Definition 36 simplifies to

$$(\sigma)_\alpha(x) = (\alpha(I_x,x),a) \text{ iff } \sigma(x) = a$$

and hence

$$(\widehat{\Sigma})_\alpha(x) = (\alpha(I_x,x),a) \text{ iff } \Sigma(x) = (l,a).$$

Proposition 32. *(Completeness) There is an annotation function* α *such that*

$$\mathsf{S} \rightarrow \mathsf{T} \textit{ implies } (\mathsf{S})_\alpha \Rightarrow_\alpha (\mathsf{T})_\alpha.$$

Proof. Take $\alpha(I,x) = \top$ whenever $x \in defs(I)$ and observe that for this α, $\Sigma, e \Downarrow \Sigma_1, a$ and $\Sigma = (\sigma)_\alpha$ implies $\Sigma = \Sigma_1$ since $\top \ominus 1 = \top$ holds. □

Consequently, any $\rightarrow$-execution can be matched by a $\Rightarrow$-execution for the trivial $\alpha = \lambda x.\top$. This gives unlimited read/write capabilities to all variables and corresponds to communicating all variables through registers. The result of Proposition 32 is thus not satisfactory for reasoning about forwardability - it does not relate executions to solutions for the dataflow equations. In the next subsection, we will prove the dataflow equations sound by showing that association functions α originating from a solution for the equations respect $\rightarrow$.

In the following, we let S_0 also denote the initial state with respect to $\Rightarrow$. Confusion with the initial state with respect to $\rightarrow$ may always be resolved based on the context in which the notation is used.

Lemma 12. *If* $\mathsf{S}_0 \Rightarrow_\alpha^* \mathsf{S}$, $\mathsf{S} = (\Sigma, n, I\vec{I})$ *and* $I \neq \phi$, *then* $x \in uses(I)$ *implies* $x \in dom\,\Sigma$.

Proof. For $\mathsf{S}_0 \Rightarrow_\alpha^* \mathsf{S}$, Proposition 31 implies $\widehat{\mathsf{S}_0} \rightarrow^* \widehat{S}$, so for $x \in uses(I)$ Lemma 9 (first part) yields $x \in dom\,\widehat{\Sigma}$. Now apply $dom\,\widehat{\Sigma} = dom\,\Sigma$. □

Definition 37. *For* $\mathsf{S} = (\Sigma, n, B)$ *and* $f : \mathbf{Var}_\mathsf{P} \rightarrow \mathcal{L}$ *we write* $f \sqsubseteq \mathsf{S}$ *if for all* $x \in dom\,\Sigma$, $\Sigma(x) = (l,a)$ *implies* $f(x) \sqsubseteq l$.

6.4.2 Soundness for regular programs

We prove the equations (6.13) sound by showing that for any association α originating from a solution, $\Rightarrow_\alpha$ is implied by $\rightarrow$.

Definition 38. *Let* $\mathtt{P}$ *be a regular program. A pair of functions* $In, Out : \mathbf{Instrs}_\mathtt{P} \rightarrow \mathbf{Var}_\mathtt{P} \rightarrow \mathcal{L}$ *is called* a solution *for* $\mathtt{P}$ *if equations (6.13) are valid for the substitution* $In/fwdIn, Out/fwdOut$*.*

In the remainder of this subsection, let $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ be a regular program and In/Out a solution for $\mathtt{P}$.

Proposition 33. *If* $x \in \mathbf{Var}_\mathtt{P}$*,* $I \in \mathbf{Instrs}_\mathtt{P}$ *and* $J \in \mathtt{succ}(I)$ *then* $Out(I)(x) \sqsupseteq In(J)(x)$*.*

Proof. From $J \in \mathtt{succ}(I)$ we obtain $\mathtt{succ}(I) \neq \emptyset$, hence

$$Out(I)(x) = \sqcup_{K \in \mathtt{succ}(I)} In(K)(x) \sqsupseteq In(J)(x)$$

follows from equations (6.13) for any solution In/Out. □

Definition 39. *The association function* α_{Out} *is given by*

$$\alpha_{Out}(I, x) = Out(I)(x)$$

for all pairs (I, x) *such that* $I \in \mathbf{Instrs}_\mathtt{P}$ *and* $x \in defs(I)$*. For association functions* α *and* β *we write* $\alpha \sqsupseteq \beta$ *if* $\alpha(I, x) \sqsupseteq \beta(I, x)$ *for all* (I, x) *such that* $I \in \mathbf{Instrs}_\mathtt{P}$ *and* $x \in defs(I)$*.*

The following proposition represents the heart of the soundness proof. It guarantees that an execution sequence starting in S_0 whose capabilities are (at least as generous as) given by a solution to the dataflow equations can always proceed. Furthermore, the resulting configuration is related to the dataflow equations.

Proposition 34. *(Progress) Let* $\mathsf{S} = (\Sigma, n, I\vec{I})$ *and* $In(I) \sqsubseteq \mathsf{S}$*. Let* $\alpha \sqsupseteq \alpha_{Out}$ *and* $\mathsf{S}_0 \Rightarrow_\alpha^* \mathsf{S}$*. Then there is a (unique)* T *such that* $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$*. Furthermore,* T *fulfils* $Out(I) \sqsubseteq \mathsf{T}$*.*

Proof. Case distinction on the structure of I.

Case $x = a$. $\Sigma, a \Downarrow \Sigma, a$ holds by the definition of $\Downarrow$.

In order to show $\Sigma \downarrow x$, let $\Sigma(x) = (l, b)$. Since $In(I) \sqsubseteq \mathsf{S}$ implies $In(I)(x) \sqsubseteq l$, In fulfils equations (6.13), and $x \in defs(I)$ holds, we obtain

$$l \sqsupseteq In(I)(x) = uses(I)(x) = uses(a)(x) = 0.$$

For $\mathsf{T} = (\Sigma_1, n, \vec{I})$ where $\Sigma_1 = \Sigma[x \mapsto (\alpha(I,x), a)])$ we thus have $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$.

For showing $Out(I) \sqsubseteq \mathsf{T}$, let $y \in dom\, \Sigma_1$ with $\Sigma_1(y) = (\widetilde{l}, \widetilde{a})$. There are two cases.

1. If $x = y$, then $\widetilde{l} = \alpha(I,x) \sqsupseteq \alpha_{Out}(I,x) = Out(I)(x)$ follows.
2. If $x \neq y$ then $\Sigma(y) = (\widetilde{l}, \widetilde{a})$ follows from the definition of Σ_1, so $In(I) \sqsubseteq \mathsf{S}$ yields $In(I)(y) \sqsubseteq \widetilde{l}$. From $y \notin defs(I)$ we thus obtain

$$\widetilde{l} \sqsupseteq In(I)(y) = uses(I)(y) \oplus Out(I)(y) = Out(I)(y).$$

In both cases, $Out(I)(y) \sqsubseteq \widetilde{l}$ is fulfilled.

Case $x = x$. Since $\mathsf{S}_0 \Rightarrow_\alpha^* \mathsf{S}$ and $x \in uses(I)$ hold, Lemma 12 implies $x \in dom\, \Sigma$, so $\Sigma(x) = (l, a)$ for some l and a, and by assumption $In(I)(x) \sqsubseteq l$.

For $\Sigma_1 = \Sigma[x \mapsto (l \ominus 1, a)]$, $\Sigma, x \Downarrow \Sigma_1, a$ holds: In fulfils equations (6.13), so $l \sqsupseteq In(I)(x) = uses(I)(x) = uses(x)(x) = 1$.

Also, $\Sigma_1 \downarrow x$ holds, because $\Sigma_1(x) = (l', a')$ yields $a' = a$ and $l' = l \ominus 1 \sqsupseteq 0$.

For the unique $\mathsf{T} = (\Sigma_2, n, \vec{I})$ with $\Sigma_2 = \Sigma_1[x \mapsto (\alpha(I,x), a)]$ we thus have $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$.

For showing $Out(I) \sqsubseteq \mathsf{T}$, let $y \in dom\, \Sigma_2$ and $\Sigma_2(y) = (\widetilde{l}, \widetilde{a})$. There are two cases.

1. If $x = y$ then $\widetilde{l} = \alpha(I,x) \sqsupseteq \alpha_{Out}(I,x) = Out(I)(x)$ follows.
2. If $x \neq y$ then $\Sigma(y) = \Sigma_1(y) = \Sigma_2(y) = (\widetilde{l}, \widetilde{a})$ follows from the definition of Σ_2 and Σ_1. In particular, $x \in dom\, \Sigma$ holds, so the assumption $In(I) \sqsubseteq \mathsf{S}$ yields $In(I)(y) \sqsubseteq \widetilde{l}$. Since In fulfils equations (6.13), we obtain

$$\widetilde{l} \sqsupseteq In(I)(y) = uses(I)(y) \oplus Out(I)(y) = 0 \oplus Out(I)(y).$$

In both cases, $Out(I)(y) \sqsubseteq \widetilde{l}$ is fulfilled.

Case $x = z$ **where** $z \neq x$. Since $\mathsf{S}_0 \Rightarrow_\alpha^* \mathsf{S}$ and $z \in uses(I)$ hold, Lemma 12 implies $z \in dom\,\Sigma$, so $\Sigma(z) = (l, a)$ for some l and a, and by assumption $In(I)(z) \sqsubseteq l$.

For $\Sigma_1 = \Sigma[z \mapsto (l \ominus 1, a)]$, $\Sigma, z \Downarrow \Sigma_1, a$ holds: In fulfils equations (6.13), so

$$l \sqsupseteq In(I)(z) = uses(I)(z) \oplus Out(I)(z) = 1 \oplus Out(I)(z) \sqsupseteq 1. \tag{6.22}$$

Also, $\Sigma_1 \downarrow x$ holds, because $\Sigma_1(x) = (l', a')$ implies $\Sigma(x) = (l', a')$ by the definition of Σ_1. By the assumption $In(I) \sqsubseteq \mathsf{S}$, $0 = uses(I)(x) = In(I)(x) \sqsubseteq l'$ follows.

For the unique $\mathsf{T} = (\Sigma_2, n, \vec{I})$ with $\Sigma_2 = \Sigma_1[x \mapsto (\alpha(I,x), a)]$ we thus have $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$.

For showing $Out(I) \sqsubseteq \mathsf{T}$, let $y \in dom\,\Sigma_2$ and $\Sigma_2(y) = (\widetilde{l}, \widetilde{a})$. There are three cases.

1. If $x = y$ then $\widetilde{l} = \alpha(I,x) \sqsupseteq \alpha_{Out}(I,x) = Out(I)(x)$ follows.
2. If $y = z$ then $\widetilde{l} = l \ominus 1$ follows from $z \neq x$ and

   $$\Sigma_2 = \Sigma_1[x \mapsto (\alpha(I,x), a)] = \Sigma[z \mapsto (l \ominus 1, a)][x \mapsto (\alpha(I,x), a)].$$

   By property (6.22), $1 \sqsubseteq l$ holds, hence $l \in \{1, \top\}$.
   If $l = 1$ then $\widetilde{l} = l \ominus 1 = 0$ holds, and property (6.22) implies

   $$1 = l \sqsupseteq 1 \oplus Out(I)(z) = 1 \oplus Out(I)(y).$$

   Consequently, $Out(I)(y) \in \{0, \bot\}$ and $Out(I)(y) \sqsubseteq \widetilde{l}$.
   If $l = \top$ then $\widetilde{l} = l \ominus 1 = \top$, so $Out(I)(y) \sqsubseteq \widetilde{l}$ is trivially fulfilled.
3. If $x \neq y \neq z$ then $\Sigma(y) = (\widetilde{l}, \widetilde{a})$ holds due to

   $$\Sigma_2 = \Sigma_1[x \mapsto (\alpha(I,x), a)] = \Sigma[z \mapsto (l \ominus 1, a)][x \mapsto (\alpha(I,x), a)].$$

   The assumption $In(I) \sqsubseteq \mathsf{S}$ implies $In(I)(y) \sqsubseteq \widetilde{l}$, and as In fulfils the equations (6.13) and $y \notin defs(I)$ holds, we obtain

   $$\widetilde{l} \sqsupseteq In(I)(y) = uses(I)(y) \oplus Out(I)(y) = 0 \oplus Out(I)(y).$$

In all three cases, $Out(I)(y) \sqsubseteq \widetilde{l}$ follows.

Case $x = x\ bop\ a$ For $\mathsf{S}_0 \Rightarrow_\alpha^* \mathsf{S}$ and $x \in uses(I)$, Lemma 12 implies $x \in dom\ \Sigma$, so $\Sigma(x) = (l, a')$ holds for some l and a', and the assumption $In(I) \sqsubseteq \mathsf{S}$ yields $In(I)(x) \sqsubseteq l$.

For $\Sigma_1 = \Sigma[x \mapsto (l \ominus 1, a')]$ we obtain $\Sigma, x \Downarrow \Sigma_1, a'$: In fulfils equations (6.13), so

$$l \sqsupseteq In(I)(x) = uses(I)(x) = uses(x\,bop\,a)(x) = 1.$$

Also $\Sigma_1, a \Downarrow \Sigma_1, a$ holds, and thus $\Sigma, x\ bop\ a \Downarrow \Sigma_1, b$ for $b = BOP(a', a)$.

Furthermore, $\Sigma_1(x) = (\widetilde{l}, \widetilde{a})$ implies $\widetilde{l} = l \ominus 1$ and $\widetilde{a} = a'$, and $l \sqsupseteq 1$ yields $\widetilde{l} \sqsupseteq 0$ and thus $\Sigma_1 \downarrow x$.

For the unique $\mathsf{T} = (\Sigma_2, n, \vec{I})$ where

$$\begin{aligned} \Sigma_2 &= \Sigma_1[x \mapsto (\alpha(I,x), b)] \\ &= \Sigma[x \mapsto (l \ominus 1, a')][x \mapsto (\alpha(I,x), b)] \\ &= \Sigma[x \mapsto (\alpha(I,x), b)] \end{aligned}$$

we thus obtain $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$.

For showing $Out(I) \sqsubseteq \mathsf{T}$, let $y \in dom\ \Sigma_2$ and $\Sigma_2(y) = (\widetilde{l}, \widetilde{a})$. There are two cases.

1. If $x = y$ then $\widetilde{l} = \alpha(I,x) \sqsupseteq \alpha_{Out}(I,x) = Out(I)(x)$ follows.
2. If $x \neq y$ then $\Sigma(y) = (\widetilde{l}, \widetilde{a})$ follows from the definition of Σ_2, hence $y \in dom\ \Sigma$. The assumption $In(I) \sqsubseteq \mathsf{S}$ yields $In(I)(y) \sqsubseteq \widetilde{l}$, and from equations (6.13) and $y \notin defs(I)$ we obtain

$$\widetilde{l} \sqsupseteq In(I)(y) = uses(I)(y) \oplus Out(I)(y) = 0 \oplus Out(I)(y).$$

In both cases, $Out(I)(y) \sqsubseteq \widetilde{l}$ holds.

Remaining cases $x = z\ bop\ a$ **and** $x = z\ bop\ y$ Similar to the above cases. Consider $z = x$ and $z \neq x$ for the various forms of y.

Case $\texttt{jmp}\ m$ For the unique $\mathsf{T} = (\Sigma, k, \texttt{N}(m))$ where k is the label of I, $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$ holds.

In order to show $Out(I) \sqsubseteq \mathsf{T}$, let $y \in dom\,\Sigma$ and $\Sigma(y) = (l,a)$. The assumption $In(I) \sqsubseteq \mathsf{S}$ guarantees $In(I)(y) \sqsubseteq l$, and as In fulfils the equations (6.13),

$$l \sqsupseteq In(I)(y) = uses(I)(y) \oplus Out(I)(y) = Out(I)(y)$$

follows as required.

Case `if` $x\,m\,l$**.** Since $\mathsf{S}_0 \Rightarrow^*_\alpha \mathsf{S}$ and $x \in uses(I)$ hold, Lemma 12 implies $x \in dom\,\Sigma$, so $\Sigma(x) = (l,a)$ holds for some l and a, with (by assumption) $In(I)(x) \sqsubseteq l$.

For $\Sigma_1 = \Sigma[x \mapsto (l \ominus 1, a)]$, $\Sigma, x \Downarrow \Sigma_1, a$ holds: In fulfils equations (6.13), so

$$l \sqsupseteq In(I)(x) = uses(I)(x) \oplus Out(I)(x) = 1 \oplus Out(I)(x) \sqsupseteq 1.$$

If $a = \mathtt{0}$, $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$ holds for the unique $\mathsf{T} = (\Sigma_1, k, \mathtt{N}(m))$ where k is the label of I, and for $a \neq \mathtt{0}$ the same holds for $\mathsf{T} = (\Sigma_1, k, \mathtt{N}(l))$.

For showing $Out(I) \sqsubseteq \mathsf{T}$, let $y \in dom\,\Sigma_1$ and $\Sigma_1(y) = (\widetilde{l}, \widetilde{a})$. There are two cases.

1. If $x = y$ then Lemma 11 (items (2) and (5)) and $1 \oplus Out(I)(x) \neq \bot$ yield

$$\widetilde{l} = l \ominus 1 \sqsupseteq (1 \oplus Out(I)(x)) \ominus 1 \sqsupseteq Out(I)(x).$$

2. If $y \neq x$ then $\Sigma(y) = \Sigma_1(y) = (\widetilde{l}, \widetilde{a})$, so $\widetilde{l} \sqsupseteq In(I)(y)$ by assumption $\mathsf{S} \sqsupseteq In(I)$, hence

$$\widetilde{l} \sqsupseteq In(I)(y) = uses(I)(y) \oplus Out(I)(y) = 0 \oplus Out(I)(y) = Out(I)(y).$$

In both cases, we obtain $Out(I)(y) \sqsubseteq \widetilde{l}$.

□

As progress at each point in a computation is ensured, single steps $\mathsf{S}_i \Rightarrow_\alpha \mathsf{S}_{i+1}$ may be composed. The property $Out(I) \sqsupseteq In(J)$ for $J \in \mathtt{succ}(I)$ guarantees composability, as it ensures that the precondition $In(J) \sqsubseteq \mathsf{S}_{i+1}$ for step $i+1$ follows from the precondition $In(I) \sqsubseteq \mathsf{S}_i$ of step i. As all association functions respect the (deterministic) standard semantics $\rightarrow$ by Proposition 31, we thus obtain soundness for any association function $\alpha \sqsupseteq \alpha_{Out}$.

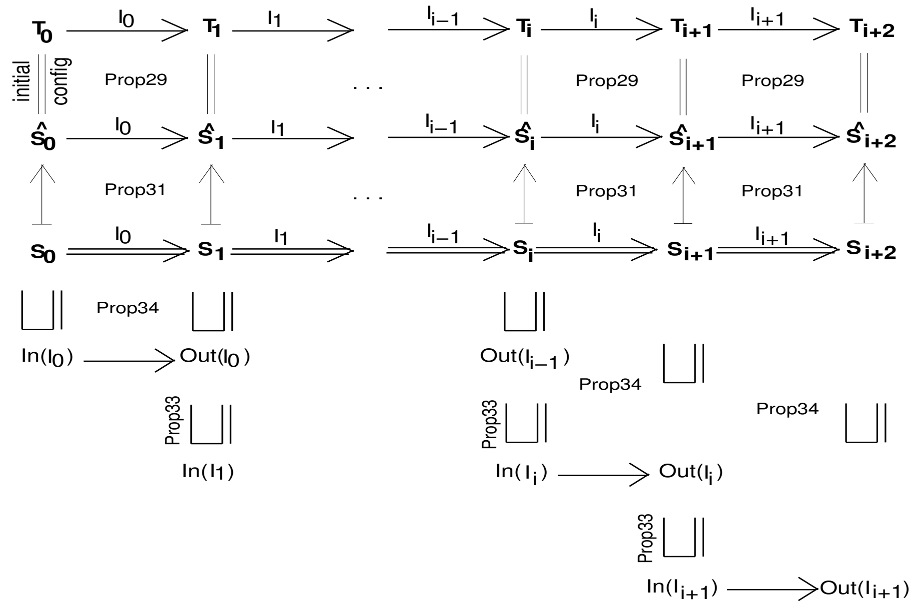


Figure 6.2: Relationship between dataflow solutions, $\Rightarrow_\alpha$ and $\rightarrow$

Theorem 6. *(Soundness of dataflow equations for regular programs) Let* $\alpha \sqsupseteq \alpha_{Out}$*. If* $\widehat{\mathsf{S}_0}$ *$\rightarrow$-terminates in a configuration* T *then* S_0 *$\Rightarrow_\alpha$-terminates in a configuration* S *with* $\widehat{\mathsf{S}} = \mathsf{T}$ *and* $\lambda x.0 \sqsubseteq \mathsf{S}$.

Proof. Let $\widehat{\mathsf{S}_0} = \mathsf{T}_0 \rightarrow \mathsf{T}_1 \rightarrow \ldots \rightarrow \mathsf{T}_n = \mathsf{T}$ be a $\rightarrow$-reduction sequence such that for $\mathsf{T}_i = (\sigma_i, n_i, B_i)$, $I_i = fst(B_i)$ holds for $1 \leq i < n$ and $|B_n| = 0$. As $|\mathtt{N}(\mathtt{entry})| > 0$ holds, we have $n > 0$. We show that there are $\mathsf{S}_1, \ldots, \mathsf{S}_n$ such that (see Figure 6.2) for all $0 \leq i < n$

1. $In(I_i) \sqsubseteq \mathsf{S}_i$
2. $\mathsf{S}_i \Rightarrow_\alpha \mathsf{S}_{i+1}$ and $Out(I_i) \sqsubseteq S_{i+1}$
3. $\widehat{\mathsf{S}_{i+1}} = \mathsf{T}_{i+1}$
4. $\mathsf{S}_0 \Rightarrow_\alpha^* S_{i+1}$

For $i = 0$, $\mathsf{S}_0 = (\Sigma_0, n_0, B_0)$ implies $dom\,\Sigma_0 = dom\,\sigma_0 = \emptyset$, so $In(I_0) \sqsubseteq \mathsf{S}_0$ holds trivially. $B_0 = \mathtt{N}(\mathtt{entry})$ implies $|B_0| > 0$, so Proposition 34 guarantees the existence of a unique

S_1 with $\mathsf{S}_0 \Rightarrow_\alpha \mathsf{S}_1$, and $Out(I_0) \sqsubseteq \mathsf{S}_1$. Finally, Proposition 31 implies $\widehat{\mathsf{S}_0} \rightarrow \widehat{\mathsf{S}_1}$, hence $\widehat{\mathsf{S}_1} = \mathsf{T}_1$ by Proposition 29.

For $i+1 < n$, the induction hypothesis implies $Out(I_i) \sqsubseteq \mathsf{S}_{i+1}$. Also, $|B_{i+1}| > 0$ holds, so $I_{i+1} \in \mathtt{succ}(I_i)$ follows by Lemma 8, and Proposition 33 yields

$$In(I_{i+1}) \sqsubseteq Out(I_i) \sqsubseteq \mathsf{S}_{i+1}.$$

As $\mathsf{S}_0 \Rightarrow_\alpha^* \mathsf{S}_{i+1}$ holds by induction hypothesis, Proposition 34 applies, hence there is a unique S_{i+2} with

$$\mathsf{S}_{i+1} \Rightarrow_\alpha \mathsf{S}_{i+2},$$

and

$$Out(I_{i+1}) \sqsubseteq \mathsf{S}_{i+2}$$

holds. The induction hypothesis also guarantees $\widehat{\mathsf{S}_{i+1}} = \mathsf{T}_{i+1}$, so Proposition 31 yields $\mathsf{T}_{i+1} \rightarrow \widehat{\mathsf{S}_{i+2}}$, resulting in

$$\widehat{\mathsf{S}_{i+2}} = \mathsf{T}_{i+2}$$

by Proposition 29. Also, from $\mathsf{S}_0 \Rightarrow_\alpha^* \mathsf{S}_{i+1}$ and $\mathsf{S}_{i+1} \Rightarrow_\alpha \mathsf{S}_{i+2}$, we obtain

$$\mathsf{S}_0 \Rightarrow_\alpha^* \mathsf{S}_{i+2},$$

completing the inductive proof.

Specialising to $i = n-1$ yields an $\mathsf{S} = \mathsf{S}_n$ with $\mathsf{S}_0 \Rightarrow_\alpha^* \mathsf{S}_n = \mathsf{S}$,

$$\widehat{\mathsf{S}} = \widehat{\mathsf{S}_n} = \mathsf{T}_n = \mathsf{T}$$

and $Out(I_{n-1}) \sqsubseteq \mathsf{S}_n = \mathsf{S}$. As $|B_n| = 0$ holds, Lemma 8 implies $\mathtt{succ}(I_{n-1}) = \emptyset$, so

$$\lambda x.0 \sqsubseteq Out(I_{n-1}) \sqsubseteq \mathsf{S}_n = \mathsf{S}$$

follows from equation (6.13). □

Notice that in Theorem 6, as well as in Proposition 34, the association function α is not required to correspond to a solution to the dataflow equations. This freedom may be used during the allocation phase in Chapter 7 for assigning even forwardable variables to registers.

6.4.3 Soundness for SSA programs

Soundness of the dataflow solutions for SSA programs is proven similarly to soundness for regular programs.

Definition 40. *For an SSA program* $\mathtt{P}$*, a pair of functions* $In, Out : \mathbf{Instrs}_\mathtt{P} \to \mathbf{N}_\mathtt{P} \to \mathbf{Var}_\mathtt{P} \to \mathcal{L}$ *is called a* solution *for* $\mathtt{P}$ *if equations (6.18) and (6.19) are fulfilled for the substitution In/fwdIn, Out/fwdOut.*

In the remainder of this subsection, let $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ be an SSA program and $In, Out : \mathbf{Instrs}_\mathtt{P} \to \mathbf{N}_\mathtt{P} \to \mathbf{Var}_\mathtt{P} \to \mathcal{L}$ a solution for $\mathtt{P}$.

Proposition 35. *Let* $x \in \mathbf{Var}_\mathtt{P}$*,* $I \in \mathbf{Instrs}_\mathtt{P}$*,* $m, m_1, m_2 \in \mathbf{N}_\mathtt{P}$ *arbitrary and* $n \in \mathbf{N}_\mathtt{P}$ *the label of* I*. Then*

1. $Out(I)(m_1)(x) = Out(I)(m_2)(x)$
2. $J \in \mathtt{succ}(I)$ *implies* $Out(I)(m)(x) \sqsupseteq In(J)(n)(x)$.

Proof. Both parts follow by applying equation (6.19).

1. If $\mathtt{succ}(I) = \emptyset$ then $Out(I)(m_1)(x) = 0 = Out(I)(m_2)(x)$ follows immediately, while $\mathtt{succ}(I) \neq \emptyset$ implies

$$Out(I)(m_1)(x) = \sqcup_{K \in \mathtt{succ}(I)} In(K)(n)(x) = Out(I)(m_2)(x).$$

2. Equation (6.19) implies $Out(I)(m)(x) = \sqcup_{K \in \mathtt{succ}(I)} In(K)(n)(x) \sqsupseteq In(J)(n)(x)$.

□

Definition 41. *The association function* α_{Out} *is given by*

$$\alpha_{Out}(I, x) = \sqcup_{m \in \mathbf{N}_\mathtt{P}} Out(I)(m)(x)$$

for all pairs (I, x) *such that* $I \in \mathbf{Instrs}_\mathtt{P}$ *and* $x \in defs(I)$.

Proposition 36. *(Progress) Let* $\mathsf{S} = (\Sigma, n, I\vec{I})$ *and* $In(I)(g) \sqsubseteq \mathsf{S}$ *for all* $g \in \mathbf{N}_\mathtt{P}$*. Let* $\alpha \sqsupseteq \alpha_{Out}$ *and* $\mathsf{S}_0 \Rightarrow_\alpha^* \mathsf{S}$*. Then there is a (unique)* T *such that* $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$*. Furthermore,* T *fulfils* $Out(I)(h) \sqsubseteq \mathsf{T}$ *for all* $h \in \mathbf{N}_\mathtt{P}$.

Proof. The proof proceeds by case distinction on I.

Case $I \notin$ Φ-Block Follows from Proposition 34.

Case $I = \phi$. For ϕ of the form $\mathbf{x}(k) = \mathbf{M}(m,k)\mathbf{n}(m)$, the soundness of $\Rightarrow$ (Proposition 31) and $\mathsf{S}_0 \Rightarrow^*_\alpha \mathsf{S}$ imply $\widehat{\mathsf{S}_0} \rightarrow^* \widehat{S}$, so by Lemma 9(part (2)) there is a unique j such that $n = n_j$, and $x_{ij} \in dom\,\widehat{\Sigma} = dom\,\Sigma$ holds for all $1 \leq i \leq k$. Consequently, for each $1 \leq i \leq k$, there are l_i and a_i such that $\Sigma(x_{ij}) = (l_i, a_i)$. For all $1 \leq i \leq k$ we have $In(I)(n) \sqsubseteq \mathsf{S}$ by assumption, hence $In(I)(n_j)(x_{ij}) \sqsubseteq l_i$. Furthermore, $In(I)(n_j)(x_{ij}) \sqsupseteq 1$ holds for all i, since $x_{ij} \in defs(I)$ implies

$$In(I)(n_j)(x_{ij}) = \oplus_{\mu=1}^{k} uses(x_{\mu j})(x_{ij}) = 1 \oplus \oplus_{\mu=1}^{k,\mu\neq i} uses(x_{\mu j})(x_{ij}) \sqsupseteq 1$$

and $x_{ij} \notin defs(I)$ implies

$$\begin{aligned} In(I)(n_j)(x_{ij}) &= (\oplus_{\mu=1}^{k} uses(x_{\mu j})(x_{ij})) \oplus Out(I)(n_j)(x_{ij}) \\ &= 1 \oplus (\oplus_{\mu=1}^{k,\mu\neq i} uses(x_{\mu j})(x_{ij})) \oplus Out(I)(n_j)(x_{ij}) \\ &\sqsupseteq 1. \end{aligned}$$

The proof now proceeds in four steps.

1. We first show that there are $\Sigma = \Sigma_1, \ldots, \Sigma_{k+1}$ such that for all $1 \leq i \leq k$ the following properties hold.

   - $\Sigma_i, x_{ij} \Downarrow \Sigma_{i+1}, a_i$
   - for all $\nu \leq k$, $\Sigma_{i+1}(x_{\nu j}) = (l_\nu \ominus (\oplus_{\mu=1}^{i} uses(x_{\nu j})(x_{\mu j})), a_\nu)$
   - $dom\,\Sigma_{i+1} = dom\,\Sigma$
   - if $y \in dom\,\Sigma_{i+1}$ and $y \neq x_{\nu j}$ holds for all $1 \leq \nu \leq k$ then $\Sigma_{i+1}(y) = \Sigma(y)$.

   For $i = 1$ we define $\Sigma_2 = \Sigma_1[x_{1j} \mapsto (l_1 \ominus 1, a_1)] = \Sigma[x_{1j} \mapsto (l_1 \ominus 1, a_1)]$. Then

   - $\Sigma_1, x_{1j} \Downarrow \Sigma_2, a_1$ is fulfilled, because $x_{1j} \in dom\,\Sigma_1 = dom\,\Sigma$ holds with $\Sigma_1(x_{1j}) = \Sigma(x_{1j}) = (l_1, a_1)$, where $l_1 \sqsupseteq In(I)(n_j)(x_{1j}) \sqsupseteq 1$ (see above).
   - for $1 \leq \nu \leq k$, we have

     if $x_{\nu j} \neq x_{1j}$: $\oplus_{\mu=1}^{i} uses(x_{\nu j})(x_{\mu j}) = uses(x_{\nu j})(x_{1j}) = 0$, hence

     $$\Sigma_2(x_{\nu j}) = \Sigma(x_{\nu j}) = (l_\nu, a_\nu) = (l_\nu \ominus (\oplus_{\mu=1}^{i} uses(x_{\nu j})(x_{\mu j})), a_\nu)$$

if $x_{\nu j} = x_{1j}$**:** $\oplus_{\mu=1}^{i} uses(x_{\nu j})(x_{\mu j}) = uses(x_{\nu j})(x_{1j}) = 1$ and $(l_1, a_1) = \Sigma(x_{1j}) = \Sigma(x_{\nu j}) = (l_\nu, a_\nu)$, hence

$$\Sigma_2(x_{\nu j}) = \Sigma_2(x_{1j}) = (l_1 \ominus 1, a_1) = (l_\nu \ominus (\oplus_{\mu=1}^{i} uses(x_{\nu j})(x_{\mu j})), a_\nu)$$

- $dom\,\Sigma_2 = \{x_{1j}\} \cup dom\,\Sigma_1 = \{x_{1j}\} \cup dom\,\Sigma = dom\,\Sigma$ by the definition of Σ_2 and because of $x_{1j} \in dom\,\Sigma$ (see above).
- for $y \in dom\,\Sigma_2$, $y \neq x_{\nu j}$ for all $1 \leq \nu \leq k$ implies $y \neq x_{1j}$, hence

$$\Sigma_2(y) = \Sigma_1[x_{1j} \mapsto \ldots](y) = \Sigma_1(y) = \Sigma(y).$$

For $1 < i \leq k$, we first show that there is a (necessarily unique) Σ_{i+1} such that $\Sigma_i, x_{ij} \Downarrow \Sigma_{i+1}, a_i$. The induction hypothesis (item (2)) yields

$$\Sigma_i(x_{\nu j}) = (l_\nu \ominus (\oplus_{\mu=1}^{i-1} uses(x_{\nu j})(x_{\mu j})), a_\nu)$$

for all $1 \leq \nu \leq k$, so in particular for $\nu = i$ we obtain

$$\Sigma_i(x_{ij}) = (l_i \ominus (\oplus_{\mu=1}^{i-1} uses(x_{ij})(x_{\mu j})), a_i).$$

Abbreviating $l_i \ominus (\oplus_{\mu=1}^{i-1} uses(x_{ij})(x_{\mu j}))$ by $\widetilde{l}$, we thus have $\Sigma_i(x_{ij}) = (\widetilde{l}, a_i)$, and we need to show $\widetilde{l} \sqsupseteq 1$. There are two cases.

If $x_{ij} \in defs(I)$**,** then

$$\begin{array}{rcl} l_i & \sqsupseteq & In(I)(n_j)(x_{ij}) = \oplus_{\mu=1}^{k} uses(x_{\mu j})(x_{ij}) \\ & = & 1 \oplus (\oplus_{\mu=1}^{k,\mu\neq i} uses(x_{\mu j})(x_{ij})) \\ & = & \left\{ \begin{array}{ll} 1 & \text{if} \quad \forall \mu \in \{1,\ldots,k\} \setminus \{i\}.uses(x_{\mu j})(x_{ij}) = 0 \\ \top & \text{if} \quad \exists \mu \in \{1,\ldots,k\} \setminus \{i\}.uses(x_{\mu j})(x_{ij}) \sqsupseteq 1 \end{array} \right. \end{array}$$

holds, where the first case, i.e. $l_i = 1$, implies

$$\oplus_{\mu=1}^{i-1} uses(x_{ij})(x_{\mu j}) = \oplus_{\mu=1}^{i-1} uses(x_{\mu j})(x_{ij}) = \oplus_{\mu=1}^{i-1} 0 = 0,$$

hence $\widetilde{l} = l_i \ominus 0 = l_i \sqsupseteq 1$, while the second case, i.e. $l_i = \top$, implies $\widetilde{l} = l_i \ominus \ldots = \top \sqsupseteq 1$ directly.

If $x_{ij} \notin \mathit{defs}(I)$**,** then

$$
\begin{aligned}
l_i &\sqsupseteq & In(I)(n_j)(x_{ij}) = (\oplus_{\mu=1}^{k} \mathit{uses}(x_{\mu j})(x_{ij})) \oplus Out(I)(n_j)(x_{ij}) \\
&= & 1 \oplus (\oplus_{\mu=1}^{k,\mu\neq i} \mathit{uses}(x_{\mu j})(x_{ij})) \oplus Out(I)(n_j)(x_{ij}) \\
&= & \begin{cases} 1 & \text{if } Out(I)(n_j)(x_{ij}) \sqsubseteq 0 \text{ and} \\ & \quad \forall \mu \in \{1,\ldots,k\} \setminus \{i\}. \mathit{uses}(x_{\mu j})(x_{ij}) = 0 \\ \top & \text{otherwise} \end{cases}
\end{aligned}
$$

holds, so $\widetilde{l} \sqsupseteq 1$ follows in a similar way as in the first case.

As both cases yield $\widetilde{l} \sqsupseteq 1$, the precondition for $\Downarrow$ is satisfied, and for

$$\Sigma_{i+1} = \Sigma_i[x_{ij} \mapsto (\widetilde{l} \ominus 1, a_i)]$$

all four items required of Σ_{i+1} are fulfilled:

- $\Sigma_i, x_{ij} \Downarrow \Sigma_{i+1}, a_i$ is fulfilled by the definition of $\Downarrow$
- for $1 \leq \nu \leq k$, $\Sigma_{i+1}(x_{\nu j}) = (l_\nu \ominus (\oplus_{\mu=1}^{i} \mathit{uses}(x_{\nu j})(x_{\mu j})), a_\nu)$ holds:

  if $x_{\nu j} \neq x_{ij}$**,** then $\mathit{uses}(x_{\nu j})(x_{ij}) = 0$ holds, so the second item of the induction hypothesis yields

$$
\begin{aligned}
\Sigma_{i+1}(x_{\nu j}) = \Sigma_i(x_{\nu j}) &= (l_\nu \ominus (\oplus_{\mu=1}^{i-1} \mathit{uses}(x_{\nu j})(x_{\mu j})), a_\nu) \\
&= (l_\nu \ominus (\oplus_{\mu=1}^{i} \mathit{uses}(x_{\nu j})(x_{\mu j})), a_\nu).
\end{aligned}
$$

  if $x_{\nu j} = x_{ij}$**,** then $(l_\nu, a_\nu) = \Sigma(x_{\nu j}) = \Sigma(x_{ij}) = (l_i, a_i)$ and

$$
\begin{aligned}
\widetilde{l} \ominus 1 &= (l_i \ominus (\oplus_{\mu=1}^{i-1} \mathit{uses}(x_{ij})(x_{\mu j}))) \ominus 1 \\
&= l_i \ominus ((\oplus_{\mu=1}^{i-1} \mathit{uses}(x_{ij})(x_{\mu j})) \oplus 1) \\
&= l_i \ominus (\oplus_{\mu=1}^{i} \mathit{uses}(x_{ij})(x_{\mu j})) \\
&= l_\nu \ominus (\oplus_{\mu=1}^{i} \mathit{uses}(x_{\nu j})(x_{\mu j}))
\end{aligned}
$$

  follow, hence

$$
\begin{aligned}
\Sigma_{i+1}(x_{\nu j}) = \Sigma_{i+1}(x_{ij}) &= (\widetilde{l} \ominus 1, a_i) \\
&= (l_\nu \ominus (\oplus_{\mu=1}^{i} \mathit{uses}(x_{\nu j})(x_{\mu j})), a_\nu).
\end{aligned}
$$

- the definition of Σ_{i+1} implies $dom\ \Sigma_{i+1} = \{x_{ij}\} \cup dom\ \Sigma_i$, so the claim follows using the induction hypothesis $dom\ \Sigma_i = dom\ \Sigma$ and the property $x_{ij} \in dom\ \Sigma$ which was shown at the very beginning of the proof.
- for $y \in dom\ \Sigma_{i+1}$ and $y \neq x_{\nu j}$ for all $1 \leq \nu \leq k$, the induction hypothesis yields $\Sigma_i(y) = \Sigma(y)$, hence

$$\Sigma_{i+1}(y) = \Sigma_i[x_{ij} \mapsto (\widetilde{l} \ominus 1, a_i)](y) = \Sigma_i(y) = \Sigma(y).$$

2. For $1 \leq i \leq k$, we now show that $\Sigma_{k+1} \downarrow x_i$ holds, i.e. that for $x_i \in dom\ \Sigma_{k+1}$ and $\Sigma_{k+1}(x_i) = (l, a)$, $l \sqsupseteq 0$ holds. There are two cases.

   Case $\exists \nu \leq k$ such that $x_i = x_{\nu j}$. Then $x_i = x_{\nu j} \in dom\ \Sigma_{k+1} = dom\ \Sigma$ follows from part (1), and $x_{\nu j} = x_i \in defs(I)$ holds, hence

$$\begin{aligned} l_\nu &\sqsupseteq In(I)(n_j)(x_{\nu j}) \\ &= \oplus_{\mu=1}^{k} uses(x_{\mu j})(x_{\nu j}) \\ &= 1 \oplus (\oplus_{\mu=1}^{k,\mu\neq\nu} uses(x_{\mu j})(x_{\nu j})) \\ &= \begin{cases} 1 & \text{if for all } \mu \in \{1,\ldots,k\} \setminus \{\nu\}, x_{\mu j} \neq x_{\nu j} \text{ holds} \\ \top & \text{otherwise} \end{cases} \end{aligned}$$

   and, by Lemma 11 (item (5))

$$\begin{aligned} l &= l_\nu \ominus (\oplus_{\mu=1}^{k} uses(x_{\nu j})(x_{\mu j})) \\ &\sqsupseteq \begin{cases} 1 \ominus (1 \oplus (\oplus_{\mu=1}^{k,\mu\neq\nu} uses(x_{\nu j})(x_{\mu j}))) & \text{if } x_{\mu j} \neq x_{\nu j} \text{ holds for all} \\ & \mu \in \{1,\ldots,k\} \setminus \{\nu\} \\ \top \ominus (\oplus_{\mu=1}^{k} uses(x_{\nu j})(x_{\mu j})) & \text{otherwise} \end{cases} \\ &= \begin{cases} 1 \ominus (1 \oplus 0) & \text{if } x_{\mu j} \neq x_{\nu j} \text{ holds for all } \mu \in \{1,\ldots,k\} \setminus \{\nu\} \\ \top & \text{otherwise} \end{cases} \\ &\sqsupseteq 0 \end{aligned}$$

   Case $x_i \neq x_{\nu j}$ holds for all $\nu \leq k$. For $x_i \in dom\ \Sigma_{k+1}$, part (1) (forth item) implies $\Sigma_{k+1}(x_i) = \Sigma(x_i)$, so the assumption $In(I)(n_j) \sqsubseteq \mathsf{S}$ together with $x_i \in defs(I)$ yields

$$l \sqsupseteq In(I)(n_j)(x_i) = uses(I)(n_j)(x_i) = \oplus_{\mu=1}^{k} uses(x_{\mu j})(x_i) = 0$$

where the last equality is valid as $uses(x_{\mu j})(x_i) = 0$ follows for each μ from $x_{\mu j} \neq x_i$.

3. Consequently, for the (unique!) $\mathsf{T} = (\Sigma', n, \vec{I})$ where

$$\Sigma' = \Sigma_{k+1}[x_1 \mapsto (l'_1, a_1), \ldots, x_k \mapsto (l'_k, a_k)]$$

and

$$l'_i = \alpha(I, x_i)$$

we obtain $\mathsf{S} \Rightarrow_\alpha \mathsf{T}$ by the rule PHI.

4. Finally, for proving $Out(I)(h) \sqsubseteq \mathsf{T}$ for all $h \in \mathbf{N}_\mathrm{P}$, we show that $y \in dom\,\Sigma'$ and $\Sigma'(y) = (l, a)$ implies $Out(I)(h)(y) \sqsubseteq l$. By Proposition 35 (part (1)), it suffices to show $Out(I)(n)(y) \sqsubseteq l$, so let $y \in dom\,\Sigma'$ and $\Sigma'(y) = (l, a)$. There are two cases.

Case $y = x_i$ for some $1 \leq i \leq k$. By the definition of SSA programs (Definition 27) there is at most one i with $y = x_i$. The definition of Σ' implies $l = l'_i$, so

$$\begin{aligned} l &= l'_i = \alpha(I, x_i) \sqsupseteq \alpha_{Out}(I, x_i) \\ &= \sqcup_{g \in \mathbf{N}_\mathrm{P}} Out(I)(g)(x_i) \sqsupseteq Out(I)(n)(x_i) = Out(I)(n)(y) \end{aligned}$$

Case $y \neq x_i$ holds for all $1 \leq i \leq k$. In this case, the definition of Σ' implies $y \in dom\,\Sigma_{k+1}$ and $\Sigma_{k+1}(y) = \Sigma'(y) = (l, a)$. There are two subcases.

If $y = x_{\nu j}$ for some $1 \leq \nu \leq k$, then part (1) (second item), $y \notin defs(I)$ and the assumption $In(I)(n_j) \sqsubseteq \mathsf{S}$ guarantee

$$\begin{aligned} l_\nu &\sqsupseteq In(I)(n_j)(y) \\ &= (\oplus_{\mu=1}^{k} uses(x_{\nu j})(x_{\mu j})) \oplus Out(I)(n_j)(y) \\ &= 1 \oplus (\oplus_{\mu=1}^{k, \mu \neq \nu} uses(x_{\nu j})(x_{\mu j})) \oplus Out(I)(n_j)(y) \\ &= \begin{cases} 1 & \text{if } Out(I)(n_j)(y) \sqsubseteq 0 \text{ and } x_{\mu j} \neq x_{\nu j} \\ & \quad \text{holds for all } \mu \in \{1, \ldots, k\} \setminus \{\nu\} \\ \top & \text{otherwise.} \end{cases} \end{aligned}$$

By Lemma 11 (item (5)) we obtain

$$\begin{array}{rcl}
l & = & l_\nu \ominus (\oplus_{\mu=1}^{k} uses(y)(x_{\mu j})) \\
 & \sqsupseteq & \begin{cases} 1 \ominus (1 \oplus (\oplus_{\mu=1}^{k,\mu\neq\nu} uses(y)(x_{\mu j}))) & \text{if } Out(I)(n_j)(y) \sqsubseteq 0 \\ \quad \text{and } x_{\mu j} \neq x_{\nu j} \text{ holds for all} & \mu \in \{1,\ldots,k\} \setminus \{\nu\} \\ \top \ominus \ldots & \text{otherwise} \end{cases} \\
 & = & \begin{cases} 0 & \text{if } Out(I)(n_j)(y) \sqsubseteq 0 \text{ and } x_{\mu j} \neq x_{\nu j} \text{holds for all} \\ & \mu \in \{1,\ldots,k\} \setminus \{\nu\} \\ \top & \text{otherwise} \end{cases} \\
 & \sqsupseteq & Out(I)(n_j)(y).
\end{array}$$

If $y \neq x_{\nu j}$ holds for all $1 \leq \nu \leq k$, then part (1) (forth item) implies $(l,a) = \Sigma_{k+1}(y) = \Sigma(y)$. From $y \neq x_i$ for all $1 \leq i \leq k$, we obtain $y \notin defs(I)$, hence

$$\begin{array}{rcl}
l \sqsupseteq In(I)(n_j)(y) & = & (\oplus_{\mu=1}^{k} uses(x_{\mu j})(y)) \oplus Out(I)(n_j)(y) \\
 & = & (\oplus_{\mu=1}^{k} 0) \oplus Out(I)(n_j)(y) \\
 & = & Out(I)(n_j)(y).
\end{array}$$

Both sub-cases yield $l \sqsupseteq Out(I)(n)(y)$ as required.

□

Theorem 7. *(Soundness of dataflow equations for SSA programs) Let $\alpha \sqsupseteq \alpha_{Our}$. If $\widehat{\mathsf{S}_0}$ $\rightarrow$-terminates in a configuration T then $\mathsf{S}_0 \Rightarrow_\alpha$-terminates in a configuration S with $\widehat{\mathsf{S}} = \mathsf{T}$ and $\lambda x.0 \sqsubseteq \mathsf{S}$.*

Proof. Similar to the proof of Theorem 6, using Propositions 35 and 36 instead of Propositions 33 and 34. □

6.4.4 Discussion

Our soundness proof does not directly follow the approach taken in [NNH99]. Instead of incorporating the dataflow solutions in the operational semantics by factoring the configuration space with respect to an equivalence relation, we incorporate them directly using annotated configurations. We expect that our approach also applies to liveness analysis, if the codomain of states is extended to pairs $(\mathcal{B} \times \mathit{Val})$. The non-standard semantics was canonically constructed from the lattice used in the dataflow analysis using $\ominus$. This allowed us to relate the fixed points of the dataflow equations to the operational behaviour based on the progress lemma. Future work is needed for comparing the two approaches in more depth and for evaluating how they generalise to other dataflow problems.

Chapter 7

Translating intermediate programs into ALEF

This chapter presents a translation from IL into ALEF based on solutions to the dataflow equations. Variables which have been identified as being linear with respect to all points of definition are candidates for forwarding. They can be passed through operand queues provided that all consuming instructions execute on the same functional unit. Remaining variables are communicated through registers. The translation is introduced in several stages which are shown to be functionally correct and faithful to the dataflow solution. The resulting ALEF program is well-typed in the sense of Chapter 5.

The first stage (Section 7.1) decides whether a variable can be forwarded based on the dataflow solution and the above restrictions. For programs in SSA form, the second stage (Section 7.2) eliminates Φ-blocks and inserts compensating instruction sequences. Although the resulting program does not fulfil the SSA requirements any longer, it is functionally equivalent to the original program, and forwardability of variables is preserved. The translation into ALEF is defined in Section 7.3, and is parametric in an allocation function ρ which assigns operand queue names to forwardable variables and register names to non-forwardable variables. In Section 7.4, we show that the resulting program is type-correct with respect to the calculus in Section 5.1 for any ρ that satisfies some mild compatibility conditions regarding the dataflow solutions. Section 7.5 describes how an allocation of queues can be obtained based on

colouring the conflict graphs for variables. Functional correctness of any allocation ρ which maps adjacent nodes to different queues or registers is proven in Section 7.6.
In order to achieve the outlined results, the translation itself is kept rather basic, and Section 7.7 consequently discusses a number of improvements and extensions.

7.1 Deciding forwardability

Based on a solution In/Out to the the dataflow equations and the intended translation scheme, the first step classifies each variable as being forwardable or not. Before giving the formal definition of forwardability, we briefly motivate its core conditions.
In Section 6.3.1, we observed that the information whether a variable is useless can be read off a solution to the dataflow equations for liveness by inspecting $liveOut(I)(x)$ where I is an assignment to x. Similarly, we now observe that forwardability information for x is captured in a solution In/Out to equations (6.13), in component $Out(I)(x)$ where $x \in defs(I)$. If $Out(I)(x) = \top$ holds then x is not forwardable since the dataflow analysis failed to prove linear usage. Variables with $Out(I)(x) = 0$ cannot be forwarded either, because this appearance of x was proven to be useless (other occurrences of x might still be used, so we cannot remove x completely). A linearly used appearance of x consist of an instruction I with $Out(I)(x) = 1$, and is forwardable if all its uses are in instructions executing on the same functional unit. In order to map variable x to an operand queue, two conditions must be fulfilled:

- *all* instructions I with $x \in defs(I)$ must satisfy $Out(I)(x) = 1$
- all uses of x are in instructions executing on the same functional unit

The first condition ensures that in a regular program, variables with multiple sites of definition are translated correctly. For example, variable `x` in program (7.1)

$$
\begin{array}{lcllcl}
\mathtt{N}(1) & = & [1]\mathtt{v} = \ldots & \mathtt{N}(6) & = & [6]\mathtt{x} = \mathtt{9} \\
 & & [2]\mathtt{if\ v\ 3\ 6} & & & [7]\mathtt{y} = \mathtt{x} + \mathtt{1} \\
\mathtt{N}(3) & = & [3]\mathtt{x} = \mathtt{3} & & & [8]\mathtt{jmp\ 9} \\
 & & [4]\mathtt{y} = \mathtt{5} & \mathtt{N}(9) & = & [9]\mathtt{z} = \mathtt{x} * \mathtt{y} \\
 & & [5]\mathtt{jmp\ 9} & & & [10]\ldots
\end{array}
\tag{7.1}
$$

must be communicated through a register. Its definition in instruction $[6]$ is used non-linearly (indeed, any solution to its dataflow equations fulfils $Out([6])(\mathtt{x}) = \top$), and the consuming instruction $[9]$ must hence be translated to $\mathtt{mul}\ r\ op_{\mathtt{y}}\ op_{\mathtt{z}}$ for some register r. Consequently, the assignment to $\mathtt{x}$ in instruction $[3]$ must also be translated to a register usage despite the linear use of the assigned value.

The second condition arises from the binding of operand queue names to functional units in ALEF. For example, the condition requires variables $\mathtt{x}$ and $\mathtt{y}$ in program (7.2)

$$\begin{array}{lcl} \mathtt{N}(1) & = & [1]\mathtt{v} = \ldots \\ & & [2]\mathtt{x} = \mathtt{5} \\ & & [3]\mathtt{y} = \mathtt{7} \\ & & [4]\mathtt{if\ v\ 5\ 6} \end{array} \quad \mathtt{N}(5) = [5]\mathtt{z} = \mathtt{x} + \mathtt{y} \quad \mathtt{N}(6) = [6]\mathtt{z} = \mathtt{x} * \mathtt{y} \tag{7.2}$$

to be communicated through registers, despite their linear usage. For suppose $\mathtt{x}$ or $\mathtt{y}$ are mapped to operand queues $\mathtt{q}_1$ and $\mathtt{q}_2$, respectively, then the typing rules for ALEF yield $\gamma(\mathtt{q}_1) = ALU \neq MUL = \gamma(\mathtt{q}_1)$ and $\gamma(\mathtt{q}_2) = ALU \neq MUL = \gamma(\mathtt{q}_2)$ and the resulting program is ill-typed and fails to execute.

In contrast, uses of x in IL-instructions of the form $y = x$ or in Φ-blocks do not constrain forwardability as these instructions will be translated to instructions $\mathtt{id}^{fu}\ op_x\ op_y$ which exist for each functional unit fu.

For regular programs, we require that *all* instructions J with $x \in uses(J)$ be mapped to the same functional unit. This is slightly conservative as disjoint sites of definition occasionally refer to different sets of using instructions. As explicitness of these structural dependencies are the main reason for using SSA, our requirement represents a trade-off for not using SSA. Indeed, SSA programs do not contain conflicts of the kind shown in program (7.1) as only one defining instruction exists for each variable.

In the following definition, we use the notation $Out(I)(x)$ for both regular and SSA programs, justified by part (1) of Proposition 35.

Definition 42. *Let* $\mathtt{P}$ *be a regular or SSA program and* In/Out *a solution for* $\mathtt{P}$. *Variable* $x \in \mathbf{Var}_{\mathtt{P}}$ *is called* forwardable to *fu if*

- $Out(I)(x) \notin \{\bot, \top\}$ *holds for all* $I \in \mathbf{Instrs}_{\mathtt{P}}$
- $Out(I)(x) = 1$ *holds for all* $I \in \mathbf{Instrs}_{\mathtt{P}}$ *with* $x \in defs(I)$

- *for all* $I \in \mathbf{Instrs}_{\mathtt{P}}$ *with* $x \in uses(I)$, *one of the following conditions holds*
  - $I \in \Phi$-*Block*
  - I *has the form* $[n]y = x$ *for some* y
  - I *has the form* $[n]y = z\ +\ e$ *or* $[n]y = z\ -\ e$ *and* $fu = ALU$
  - I *has the form* $[n]y = z\ *\ e$ *and* $fu = MUL$
  - I *has the form* $[n]\mathtt{if}\ x\ m\ l$ *and* $fu = BU$.

While the last two conditions are motivated by the above discussion, the first one is a technical condition whose significance will become apparent in the later development.

Example. In the SSA program (6.20), the solution discussed in Section 6.3.3 characterises variables `z` and `y` as being forwardable to `ALU` while `v` is forwardable to `BU`. Variables `z1`, `z2` and `y1` are forwardable to any *fu*, while `y2` and `w` are not forwardable.

$\diamond$

7.2 Eliminating Φ-instructions

For SSA programs, the second step consists of eliminating Φ-instructions. For this step, we assume that input programs are in standard edge-split form as described in Section 6.2.2.2. In particular, Φ-blocks must not occur as successors of conditional jumps and the order of Φ-instructions in each Φ-block must be such that a variable assigned to in the *i*-th row does not occur on the right-hand side in rows $l > i$.

We eliminate each Φ-block by inserting compensating assignments into its control flow predecessors, preceding the jump instructions. For example, eliminating instruction $[9]$ from program (6.20) requires the insertion of $\mathtt{y} = \mathtt{y1};\ \mathtt{z} = \mathtt{z1}$ between instructions $[4]$ and $[5]$ and of $\mathtt{y} = \mathtt{y2};\ \mathtt{z} = \mathtt{z2}$ between instructions $[7]$ and $[8]$.

Formally, for $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ and $fst(\mathtt{N}(l)) = \phi$, we define a program $\mathtt{P}_\phi$ which results from `P` by eliminating the instruction ϕ as follows.

- Any instruction $[n]\texttt{jmp}\ l$ is prefixed with the sequence

$$\vec{I}_n = \begin{array}{c} x_1 = x_{1j} \\ \vdots \\ x_k = x_{kj} \end{array}$$

where $\phi = [l]\mathbf{x}(k) = \mathbf{M}(k,m)\mathbf{n}(m)$ and $n = n_j$.

- ϕ is deleted from $\texttt{N}(l)$.
- $[n]\texttt{jmp}\ l$ is replaced by $[n]\texttt{jmp}\ l'$ where l' is the label of the (unique) $J \in \texttt{succ}(\phi)$.
- N, A and succ are updated to reflect the changes.

Example. For program (6.20) we indeed obtain

$$I_5 = \begin{array}{l} \texttt{y} = \texttt{y1} \\ \texttt{z} = \texttt{z1} \end{array} \quad \text{and} \quad I_8 = \begin{array}{l} \texttt{y} = \texttt{y2} \\ \texttt{z} = \texttt{z2} \end{array}$$

so the program $\texttt{P}_{[9]}$ resulting from eliminating instruction [9] is

$$\begin{array}{rcl} \texttt{N(1)} & = & [1]\texttt{v} = \texttt{7} \\ & & [2]\texttt{if v 3 6} \\ \texttt{N(3)} & = & [3]\texttt{y1} = \texttt{3} \\ & & [4]\texttt{z1} = \texttt{2} \\ & & [101]\texttt{y} = \texttt{y1} \\ & & [102]\texttt{z} = \texttt{z1} \\ & & [5]\texttt{jmp 10} \end{array} \qquad \begin{array}{rcl} \texttt{N(6)} & = & [6]\texttt{y2} = \texttt{7} \\ & & [7]\texttt{z2} = \texttt{y2} * \texttt{2} \\ & & [103]\texttt{y} = \texttt{y2} \\ & & [104]\texttt{z} = \texttt{z2} \\ & & [8]\texttt{jmp 10} \\ & & \\ \texttt{N(10)} & = & [10]\texttt{w} = \texttt{z} + \texttt{y} \end{array} \tag{7.3}$$

$\diamond$

Repeated elimination of ϕ-blocks transforms an edge-split SSA program into a regular program. The edge-split condition guarantees that at most one compensation sequence is inserted into each block, hence the ϕ-blocks may be eliminated in an arbitrary order. Forwardability of variables is preserved as solutions to the dataflow-equations after an elimination step coincide (modulo the relabelling of instructions) with those of the original program. Each use of a variable x_{ij} corresponds to a use in a new instruction $x_i = x_{ij}$ in a $\vec{I}_n$ with $n = n_j$. Furthermore, all compensating instruction sequences contain assignments to all x_i occuring in the LHS of a ϕ-block. Consequently, each variable y with $y \in \mathit{defs}(\phi)$ fulfils $y = x_i$ for some $1 \leq i \leq k$, and by Definition 27 this i is unique. Also for all $l > i$, $x_{lj} \neq y$ holds for all j. Consequently, for $n = n_j$ and $\vec{I}_n = I_1 \dots I_k$, instruction I_i has the form $y = x_{ij}$, and $\mathit{fwdOut}(I_{x_{ij}})(y)$ agrees with $\mathit{fwdOut}(\phi)(y)$. Likewise $\mathit{fwdIn}(I_1)(y) = \mathit{fwdIn}(\phi)(n)(y)$, since $I_1 \dots I_{i-1}$ contains exactly the assignments corresponding to position $n = n_j$,

$$\oplus_{l=1}^{i} \mathit{uses}(I_l)(n)(z) = \oplus_{l=1}^{i} \mathit{uses}(x_{lj})(z) = \mathit{uses}(\Phi')(n)(z)$$

where ϕ' contains the first i lines of ϕ, and $y \notin \mathit{defs}(I_l)$ for $l < i$.
Similarly, for variables y not occurring in $\mathit{defs}(\phi)$, $\mathit{fwdOut}(\phi)(n)(y) = \mathit{fwdOut}(I_k)(y)$ and $\mathit{fwdIn}(\phi)(n)(y) = \mathit{fwdIn}(I_1)(y)$ hold. Since $\vec{I}_n$ and ϕ have the same successors (the immediate successor of ϕ) and predecessors (the body of n's block), the dataflow equations agree. Consequently, each solution to the equations for the program containing ϕ corresponds to a solution for the program containing the $\vec{I}_n$ but not ϕ.

Example. The dataflow equations for the above (regular) program $\mathtt{P}_{[9]}$ are

$$\begin{array}{rcrcl}
 & & \mathit{fwdOut}([1])(\mathtt{v}) & = & 1 \\
\mathit{fwdOut}([3])(\mathtt{y1}) & = & \mathit{fwdOut}([4])(\mathtt{z1}) & = & 1 \\
\mathit{fwdOut}([101])(\mathtt{y}) & = & \mathit{fwdOut}([102])(\mathtt{z}) & = & 1 \\
 & & \mathit{fwdOut}([6])(\mathtt{y2}) & = & \top \\
 & & \mathit{fwdOut}([7])(\mathtt{z2}) & = & 1 \\
\mathit{fwdOut}([103])(\mathtt{y}) & = & \mathit{fwdOut}([104])(\mathtt{z}) & = & 1 \\
 & & \mathit{fwdOut}([10])(\mathtt{w}) & = & 0
\end{array} \tag{7.4}$$

The least fixed point corresponding to these equations yields the same forwardability as the one for the original SSA program: `y` and `z` are forwardable to `ALU` because *fwdOut* assigns the value 1 to their defining instructions [101]/[103] and [102]/[104]. Variable `v` is forwardable to `BU`, and variables `z1`, `z2` and `y1` are forwardable to any *fu*. Finally, `y2` and `w` are not forwardable. $\diamond$

Furthermore, eliminating a Φ-block respects the functional semantics. The condition on the shape of the matrix **M** ensures that the sequential execution of the compensating instruction sequence leads to the same result as the concurrent assignment during the execution of the original ϕ-block. In particular, the side condition $x_i \neq x_{lj}$ for all i and all $l > i$ in Definition 31 ensures that the correct values are copied into the x_i variables, as the definition of $\vec{I}_n$ respects the order of single ϕ-instructions in a Φ-block. This is formally proven in Proposition 37 using the following technical lemma.

Lemma 13. *If* $(\Sigma, m_1,\ \vec{ass}\vec{I}) \Rightarrow^*_\alpha (\Sigma', m_1, \vec{I})$ *then* $(\Sigma, m_2,\ \vec{ass}\vec{J}) \Rightarrow^*_\alpha (\Sigma', m_2, \vec{J})$ *for any* $m_2 \in \mathbf{N}_\mathrm{P}$ *and* $\vec{J}$.

Proof. Induction on the length n of $\vec{ass} = ass_1 \ldots ass_n$.
For $n = 1$, the rule ASS guarantees that ass_1 is of the form $[k]x = e$, and that $\Sigma, e \Downarrow \Sigma_1, a$ holds for some Σ_1 and a such that $\Sigma_1 \downarrow x$ and $\Sigma' = \Sigma_1[x \mapsto \alpha(ass_1, x), a]$. Consequently, the derivation

$$\frac{\Sigma, e \Downarrow \Sigma_1, a}{(\Sigma, m_2, ass_1\vec{J}) \Rightarrow_\alpha (\Sigma_1[x \mapsto (\alpha(ass_1, x), a)], m_2, \vec{J})} \left\{ \begin{array}{c} ass_1 = [k]x = e \\ \Sigma_1 \downarrow x \end{array} \right.$$

is possible.
For $n > 1$, $\vec{ass} = ass_1 \ldots ass_n$, $(\Sigma, m_1,\ \vec{ass}\vec{I}) \Rightarrow^*_\alpha (\Sigma', m_1, \vec{I})$ yields

$$(\Sigma, m_1,\ ass_1\, ass_2 \ldots ass_n\vec{I}) \Rightarrow^*_\alpha (\Sigma_1, m_1, ass_2 \ldots ass_n\vec{I})$$

for some Σ_1 with $ass_1 = [k]x = e$, $\Sigma, e \Downarrow \widetilde{\Sigma}, a$, $\widetilde{\Sigma} \downarrow x$, $\Sigma_1 = \widetilde{\Sigma}[x \mapsto (\alpha(ass_1, x), a)]$ and

$$(\Sigma_1, m_1, ass_2 \ldots ass_n\vec{I}) \Rightarrow^*_\alpha (\Sigma', m_1, \vec{I}).$$

These properties of $\widetilde{\Sigma}$, x and a allow us to derive

$$\frac{\Sigma, e \Downarrow \widetilde{\Sigma}, a}{(\Sigma, m_2,\ ass_1\, ass_2 \ldots ass_n\vec{J}) \Rightarrow^*_\alpha (\Sigma_1, m_2, ass_2 \ldots ass_n\vec{J})} \left\{ \begin{array}{c} ass_1 = [k]x = e \\ \Sigma_1 \downarrow x \end{array} \right.$$

and applying the induction hypothesis yields $(\Sigma_1, m_2, ass_2 \ldots ass_n\vec{J}) \Rightarrow^*_\alpha (\Sigma', m_2, \vec{J})$ so $(\Sigma, m_2,\ \vec{ass}\vec{J}) \Rightarrow^*_\alpha (\Sigma', m_2, \vec{J})$ can be derived. □

The proposition shows that prefixing the compensation code to the jump has the same effect as executing the jump followed by the ϕ-block.

Proposition 37. *Let $S = (\Sigma, l, [n]\texttt{jmp}\,h)$ and $\texttt{N}(h) = \phi\vec{I}$ where ϕ has the form $[h]\mathbf{x}(k) = \mathbf{M}(k,m)\mathbf{n}(m)$. For $n = n_j$ let $\vec{K}$ denote the code sequence $\vec{I}_n$, and let*

$$\mathsf{S} \Rightarrow_\alpha (\Sigma, n,\ \phi\vec{I}) \Rightarrow_\alpha (\Sigma', n, \vec{I})$$

be an execution for $\texttt{P}$. *Then*

$$(\Sigma, l,\ \vec{K}\,[n]\texttt{jmp}\,h) \Rightarrow_\alpha^* (\Sigma', n, \vec{I})$$

is an execution for $\texttt{P}_\phi$.

Proof. We will show that

$$(\Sigma, l, \vec{K}\vec{I}) \Rightarrow_\alpha^* (\Sigma', l, \vec{I})$$

holds. By Lemma 13, this will imply

$$(\Sigma, l, \vec{K}\vec{J}) \Rightarrow_\alpha^* (\Sigma', l, \vec{J})$$

for any $\vec{J}$. By choosing $\vec{J} = [n]\texttt{jmp}\,h$ we will thus obtain

$$(\Sigma, l,\ \vec{K}\,[n]\texttt{jmp}\,h) \Rightarrow_\alpha^* (\Sigma', l, [n]\texttt{jmp}\,h),$$

from which the claim follows due to $\texttt{N}'(h) = \vec{I}$ and

$$(\Sigma', l, [n]\texttt{jmp}\,h) \Rightarrow_\alpha (\Sigma', n, \texttt{N}'(h)).$$

In order to show $(\Sigma, l, \vec{K}\vec{I}) \Rightarrow_\alpha^* (\Sigma', l, \vec{I})$ observe that by the definition of rule PHI there are $\Sigma = \Sigma_1, \ldots, \Sigma_{k+1}$ such that for all $1 \leq i \leq k$,

$$\Sigma_i, x_{ij} \Downarrow \Sigma_{i+1}, a_i \text{ and } \Sigma_{k+1} \downarrow x_i$$

and

$$\Sigma' = \Sigma_{k'+1}[x_1 \mapsto (l_1, a_1), \ldots, x_k \mapsto (l_k, a_k)]$$

holds, where $l_i = \alpha(\phi, x_i)$. Each $\Sigma_i, x_{ij} \Downarrow \Sigma_{i+1}, a_i$ implies $\Sigma_i(x_{ij}) = (\widetilde{l_i}, a_i)$ for some $\widetilde{l_i} \sqsupseteq 1$ with $\Sigma_{i+1} = \Sigma_i[x_{ij} \mapsto (\widetilde{l_i} \ominus 1, a_i)]$.
This sequence of read operations for $\Sigma_i, x_{ij} \Downarrow \Sigma_{i+1\,j}, a_i$ is depicted horizontally at the top in Figure 7.1. The subsequent write operations during the assignment $\Sigma' = \Sigma_{k+1}[x_1 \mapsto (l_1, a_1), \ldots, x_k \mapsto (l_k, a_k)]$ are shown vertically on the right hand side. The remainder of

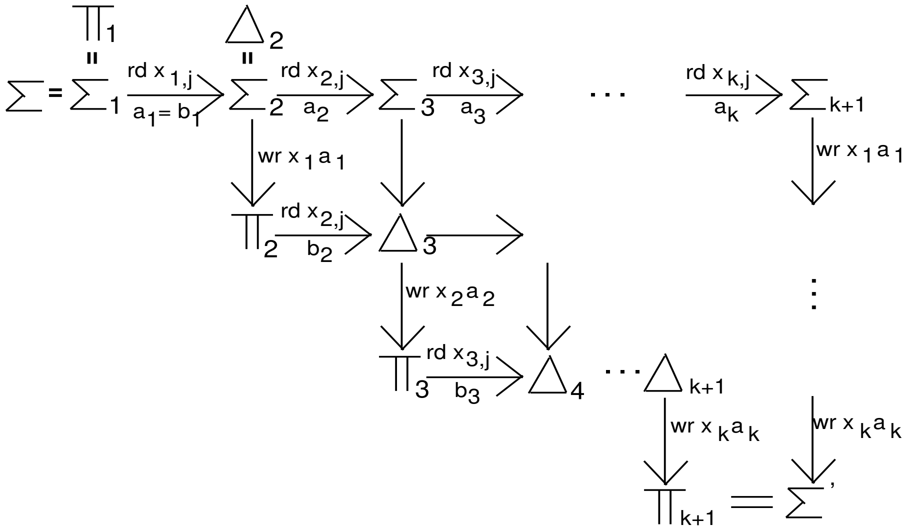


Figure 7.1: States Π_i and Δ_i.

this proof shows that executing each line of the Φ-block separately is possible due to the conditions in rules PHI and ASS, and that executing the compensating assignments in the same order as the entries in ϕ leads to the same final configuration Σ'. This is achieved by defining intermediate configurations Δ_i and Π_i where Π_i represents the configuration after executing assignments $1,\ldots,i-1$ and Δ_{i+1} denotes the configuration after executing assignments $1,\ldots,i-1$ and the reading operation of assignment i. In particular, we show that the lattice annotations obtained for the sequence of assignments coincide with the lattice annotations for ϕ in rule PHI, and that the same holds for the values a_i and b_i read during the two executions. This property rests on the restriction on the appearance of variables in ϕ for SSA programs in standard edge-split form (Definition 31): as a variable y occurring as an x_i on the left may not occur in a subsequent line on the right, the sequential (non-atomic) execution of the assignments will never overwrite a variable which is needed later.

Formally, for $1 \leq \nu \leq k+1$ we define states Π_ν by (see Figure 7.1)

$$\begin{aligned} \Pi_1 &= \Sigma_1 \\ \Pi_\nu &= \Delta_\nu[x_{\nu-1} \mapsto (l_{\nu-1}, b_{\nu-1})] \text{ for } \nu > 1 \end{aligned}$$

where Δ_ν and $b_{\nu-1}$ follow from $\Pi_{\nu-1}(x_{\nu-1\,j}) = (l'_{\nu-1}, b_{\nu-1})$ with $l'_{\nu-1} \sqsupseteq 1$ and

$$\Delta_\nu = \Pi_{\nu-1}[x_{\nu-1\,j} \mapsto (l'_{\nu-1} \ominus 1, b_{\nu-1})].$$

By definition, $(l'_1, b_1) = \Pi_1(x_{1j}) = \Sigma_1(x_{1j}) = (\widetilde{l}_1, a_1)$ holds, hence $l'_1 = \widetilde{l}_1 \sqsupseteq 1$, $b_1 = a_1$ and $\Delta_2 = \Pi_1[x_{1j} \mapsto (l'_1 \ominus 1, b_1)] = \Sigma_1[x_{1j} \mapsto (\widetilde{l}_1 \ominus 1, a_1)] = \Sigma_2$. By induction on $\nu \geq 2$ we now show $l'_\nu = \widetilde{l}_\nu \sqsupseteq 1$, $b_\nu = a_\nu$ and

$$\Delta_{\nu+1} = \Sigma_{\nu+1}[x_1 \mapsto (l_1, a_1)] \ldots [x_{\nu-1} \mapsto (l_{\nu-1}, a_{\nu-1})]. \tag{7.5}$$

For $\nu = 2$, Definition 31 yields $x_{2j} \neq x_1$ and thus

$$(l'_2, b_2) = \Pi_2(x_{2j}) = \Delta_2[x_1 \mapsto (l_1, b_1)](x_{2j}) = \Delta_2(x_{2j}) = \Sigma_2(x_{2j}) = (\widetilde{l}_2, a_2).$$

Consequently, $l'_2 = \widetilde{l}_2 \sqsupseteq 1$, $b_2 = a_2$ and

$$\begin{aligned}
\Delta_3 &= \Pi_2[x_{2j} \mapsto (l'_2 \ominus 1, b_2]) \\
&= \Delta_2[x_1 \mapsto (l_1, b_1)][x_{2j} \mapsto (l'_2 \ominus 1, b_2)] \\
&= \Delta_2[x_{2j} \mapsto (l'_2 \ominus 1, b_2)][x_1 \mapsto (l_1, b_1)] \\
&= \Sigma_2[x_{2j} \mapsto (l'_2 \ominus 1, b_2)][x_1 \mapsto (l_1, b_1)] \\
&= \Sigma_2[x_{2j} \mapsto (\widetilde{l}_2 \ominus 1, a_2)][x_1 \mapsto (l_1, a_1)] \\
&= \Sigma_3[x_1 \mapsto (l_1, a_1)]
\end{aligned}$$

For $\nu > 2$ we obtain $l'_{\nu-1} = \widetilde{l}_{\nu-1} \sqsupseteq 1$ and $w_{\nu-1} = a_{\nu-1}$ by induction hypothesis. Applying $x_i \neq x_{lj}$ for $l > i$ from Definition 31 (and the induction hypothesis) yields

$$\begin{aligned}
(l'_\nu, b_\nu) &= \Pi_\nu(x_{\nu j}) \\
&= \Delta_\nu[x_{\nu-1} \mapsto (l_{\nu-1}, b_{\nu-1})](x_{\nu j}) \\
&= \Delta_\nu(x_{\nu j}) = \Sigma_\nu[x_1 \mapsto (l_1, a_1)] \ldots [x_{\nu-2} \mapsto (l_{\nu-2}, a_{\nu-2})](x_{\nu j}) \\
&= \Sigma_\nu(x_{\nu j}) = (\widetilde{l}_\nu, a_\nu)
\end{aligned}$$

and thus $l'_\nu = \widetilde{l}_\nu \sqsupseteq 1$ and $b_\nu = a_\nu$. Consequently,

$$\begin{aligned}
\Delta_{\nu+1} &= \Pi_\nu[x_{\nu j} \mapsto (l'_\nu \ominus 1, b_\nu)] \\
&= \Delta_\nu[x_{\nu-1} \mapsto (l_{\nu-1}, b_{\nu-1})][x_{\nu j} \mapsto (l'_\nu \ominus 1, b_\nu)] \\
&= \Delta_\nu[x_{\nu-1} \mapsto (l_{\nu-1}, a_{\nu-1})][x_{\nu j} \mapsto (\widetilde{l}_\nu \ominus 1, a_\nu)] \\
&= \Sigma_\nu[x_1 \mapsto (l_1, a_1)] \ldots [x_{\nu-2} \mapsto (l_{\nu-2}, a_{\nu-2})] \\
&\qquad [x_{\nu-1} \mapsto (l_{\nu-1}, a_{\nu-1})][x_{\nu j} \mapsto (\widetilde{l}_\nu \ominus 1, a_\nu)] \\
&= \Sigma_\nu[x_{\nu j} \mapsto (\widetilde{l}_\nu \ominus 1, a_\nu)][x_1 \mapsto (l_1, a_1)] \ldots [x_{\nu-1} \mapsto (l_{\nu-1}, a_{\nu-1})] \\
&= \Sigma_{\nu+1}[x_1 \mapsto (l_1, a_1)] \ldots [x_{\nu-1} \mapsto (l_{\nu-1}, a_{\nu-1})]
\end{aligned}$$

completing the induction.

By definition, $\vec{K} = I_1 \ldots I_k$ where each assignment I_i has the form $x_i = x_{ij}$. By rule ASS we obtain $(\Pi_i, l, I_i \ldots I_k \vec{I}) \Rightarrow_\alpha (\Pi_{i+1}, l, I_{i+1} \ldots I_k \vec{I})$ for each i, due to

$\Pi_i, x_{ij} \Downarrow \Delta_{i+1}, a_i$: $\Pi_i(x_{ij}) = (l'_i, b_i) = (\widetilde{l}_i, a_i)$ with $\widetilde{l}_i \sqsupseteq 1$ and $\Delta_{i+1} = \Pi_i[x_{ij} \mapsto (l'_i \ominus 1, b_i)] = \Pi_i[x_{ij} \mapsto (\widetilde{l}_i \ominus 1, a_i)]$

$\Delta_{i+1} \downarrow x_i$: Equation (7.5) implies $\Delta_{i+1} = \Sigma_{i+1}[x_1 \mapsto (l_1, a_1)] \ldots [x_{i-1} \mapsto (l_{i-1}, a_{i-1})]$, and Definition 27 implies $x_i \neq x_l$ for $l < i$, hence for $x_i \in dom\, \Delta_{i+1}$ we obtain

$$\Delta_{i+1}(x_i) = \Sigma_{i+1}[x_1 \mapsto (l_1, a_1)] \ldots [x_{i-1} \mapsto (l_{i-1}, a_{i-1})](x_i) = \Sigma_{i+1}(x_i).$$

On the other hand, the definition of the Σ states yields

$$\Sigma_{k+1} = \Sigma_{i+1}[x_{i+1\,j} \mapsto \ldots] \ldots [x_{kj} \mapsto \ldots]$$

so using the property $x_{lj} \neq x_i$ for all $l > i$ from Definition 31 we obtain

$$\Sigma_{i+1}(x_i) = \ldots = \Sigma_{k+1}(x_i)$$

and thus

$$\Delta_{i+1}(x_i) = \Sigma_{k+1}(x_i).$$

Consequently, $\Delta_{i+1} \downarrow x_i$ holds since $\Sigma_{k+1} \downarrow x_i$ holds in rule PHI.

$\Pi_{i+1} = \Delta_{i+1}[x_i \mapsto (\alpha(I_i, x_i), a_i]$: by definition, $\Pi_{i+1} = \Delta_{i+1}[x_i \mapsto (l_i, b_i)]$ holds and we showed $b_i = a_i$. As the dataflow solutions for `P` and `P'` agree, $l_i = \alpha(\phi, x_i) = \alpha(I_i, x_i)$ holds and the claim follows.

Consequently, $(\Pi_1, l, I_1 \ldots I_k \vec{I}) \Rightarrow_\alpha^* (\Pi_{k+1}, l, \vec{I})$ holds, and the claim follows from $\Pi_1 = \Sigma_1 = \Sigma$, $\vec{K} = I_1 \ldots I_k$ and

$$\begin{aligned}\Pi_{k+1} &= \Delta_{k+1}[x_k \mapsto (l_k, a_k)] \\ &= \Sigma_{k+1}[x_1 \mapsto (l_1, a_1)] \ldots [x_{k-1} \mapsto (l_{k-1}, a_{k-1})][x_k \mapsto (l_k, a_k)] \\ &= \Sigma'.\end{aligned}$$

□

From now on we assume that all Φ-blocks of a program have been eliminated and that $\mathtt{P}$ is a regular program.

7.3 Translation

The translation $[\![\_]\!]$ of a regular IL program $\mathtt{P}$ into ALEF is defined by induction on the structure of $\mathtt{P}$. For each instruction form, a corresponding ALEF instruction is defined, and the translation of $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ results from translating each basic block $\mathtt{N}(m)$. The translation is parametric in the allocation of operand queues and registers to variables, which is represented by an allocation function $\rho : \mathbf{Var}_{\mathtt{P}} \to OP$. Type-correctness of the translation is independent of the exact allocation chosen as long as the following property is satisfied.

Definition 43. *An allocation function* $\rho : \mathbf{Var}_{\mathtt{P}} \to OP$ *is called* compatible with the dataflow solution In/Out *if for all operand queues q,*

$$\rho(x) = q \textit{ implies } x \textit{ forwardable to } \gamma(q).$$

The translation $[\![\_]\!]_\rho$ is defined by Table 7.1 where an extended ALEF instruction set is used which includes instructions with immediate operands. The choice to map an instruction $[n]x = y$ with $\rho(y) = r$ to $\mathtt{id}^{\mathtt{ALU}}$ instead of any other $\mathtt{id}^{fu}$ is arbitrary, and one could in fact choose different functional units each time such an instruction is translated.

The typing information for the new ALEF instructions is given in Table 7.2.

$[n]x = a$	$[n]\mathtt{ldc}\ a\ \rho(x)$
$[n]x = y$	$\begin{cases} [n]\mathtt{id}^{\gamma(q)}\ q\ \rho(x) & \text{if } \rho(y) = q \\ [n]\mathtt{id}^{\mathtt{ALU}}\ r\ \rho(x) & \text{if } \rho(y) = r \end{cases}$
$[n]x = y - a$	$[n]\mathtt{subi}\ a\ \rho(y)\ \rho(x)$
$[n]x = y + a$	$[n]\mathtt{addi}\ a\ \rho(y)\ \rho(x)$
$[n]x = y * a$	$[n]\mathtt{muli}\ a\ \rho(y)\ \rho(x)$
$[n]x = y - z$	$[n]\mathtt{sub}\ \rho(y)\ \rho(z)\ \rho(x)$
$[n]x = y + z$	$[n]\mathtt{add}\ \rho(y)\ \rho(z)\ \rho(x)$
$[n]x = y * z$	$[n]\mathtt{mul}\ \rho(y)\ \rho(z)\ \rho(x)$
$[n]\mathtt{if}\ x\ m\ l$	$[n]\mathtt{if}\ \rho(x)\ m\ l$
$[n]\mathtt{jmp}\ m$	$[n]\mathtt{jmp}\ m$
B	$[\![I_1]\!]_\rho \ldots [\![I_m]\!]_\rho$ if $B = I_1 \ldots I_m$
$\mathtt{P}$	$(\lambda m.[\![\mathtt{N}(m)]\!]_\rho, \mathtt{A})$

Table 7.1: Translation $[\![\mathtt{P}]\!]_\rho$ for IL program $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$

$code$	$FU(code)$	A_{code}	B_{code}
$\mathtt{addi}\ a\ op_1\ op_2$	ALU	op_1	op_2
$\mathtt{sub}\ op_1\ op_2\ op_3$	ALU	$op_1 \otimes op_2$	op_3
$\mathtt{subi}\ a\ op_1\ op_2$	ALU	op_1	op_2
$\mathtt{muli}\ a\ op_1\ op_2$	MUL	op_1	op_2

Table 7.2: Type system for extended instruction set

Example. We continue the example (7.3) from the previous section (program $\mathtt{P}_{[9]}$ and equations (7.4)). Take $\gamma(\mathtt{q}_1) = \gamma(\mathtt{q}_3) = \mathtt{ALU}$, $\gamma(\mathtt{q}_2) = \mathtt{MEM}$ and $\gamma(\mathtt{q}_4) = \mathtt{BU}$, and define ρ by

$$\begin{array}{rcl@{\qquad}rcl} \rho(\mathtt{v}) & = & \mathtt{q}_4 & \rho(\mathtt{z1}) & = & \mathtt{q}_1 \\ \rho(\mathtt{y1}) & = & \mathtt{q}_1 & \rho(\mathtt{z2}) & = & \mathtt{q}_2 \\ \rho(\mathtt{y2}) & = & \mathtt{r}_2 & \rho(\mathtt{z}) & = & \mathtt{q}_3 \\ \rho(\mathtt{y}) & = & \mathtt{q}_1 & \rho(\mathtt{w}) & = & \mathtt{r}_1 \end{array}$$

Then ρ is compatible with the forwardability information resulting from the least solution to the dataflow equations. For example, $\rho(\mathtt{y}) = \mathtt{q}_1$ holds, and $\mathtt{y}$ is indeed forwardable to $\gamma(\mathtt{q}_1) = \mathtt{ALU}$. On the other hand, $\mathtt{w}$ is not forwardable to any *fu* and is mapped to a register.

The resulting ALEF program $[\![\mathtt{P}_{[9]}]\!]_\rho = (\mathtt{N}, \mathtt{A})$ is

$$\begin{array}{rl@{\qquad}rl} \mathtt{N}(1) = [\![\mathtt{N}(1)]\!]_\rho = & [1]\mathtt{ldc}\ 7\ \mathtt{q}_4 & \mathtt{N}(6) = [\![\mathtt{N}(6)]\!]_\rho = & [6]\mathtt{ldc}\ 7\ \mathtt{r}_2 \\ & [2]\mathtt{if}\ \mathtt{q}_4\ 3\ 6 & & [7]\mathtt{muli}\ 2\ \mathtt{r}_2\ \mathtt{q}_2 \\ \mathtt{N}(3) = [\![\mathtt{N}(3)]\!]_\rho = & [3]\mathtt{ldc}\ 3\ \mathtt{q}_1 & & [103]\mathtt{id}^{\mathtt{ALU}}\ \mathtt{r}_2\ \mathtt{q}_1 \\ & [4]\mathtt{ldc}\ 2\ \mathtt{q}_1 & & [104]\mathtt{id}^{\mathtt{MEM}}\ \mathtt{q}_2\ \mathtt{q}_3 \\ & [101]\mathtt{id}^{\mathtt{ALU}}\ \mathtt{q}_1\ \mathtt{q}_1 & & [8]\mathtt{jmp}\ 10 \\ & [102]\mathtt{id}^{\mathtt{ALU}}\ \mathtt{q}_1\ \mathtt{q}_3 & \mathtt{N}(10) = [\![\mathtt{N}(10)]\!]_\rho = & [10]\mathtt{add}\ \mathtt{q}_3\ \mathtt{q}_1\ \mathtt{r}_1 \\ & [5]\mathtt{jmp}\ 10 & & \end{array}$$

and $\mathtt{A} = \{(1,3),(1,6),(3,10),(6,10)\}$. ◇

Notice that Definition 43 does not force the compiler to actually map forwardable variables to operand queues. The compiler may thus optimise the scheduling of operand communication by exploring different allocations.

The following two properties will be used in the following section.

Lemma 14. *If ρ is an allocation function for* $\mathtt{P}$*, $I \in$* $\mathbf{Instrs}_{\mathtt{P}}$ *and $x \in defs(I)$ then $\rho(x)$ is a factor of B_{code} where $[\![I]\!]_\rho = [n]code$ for some n.*

Proof. Inspection of Tables 7.1, 3.1 and 7.2. □

Lemma 15. *Let* ρ *be an allocation function for regular* $\mathtt{P}$ *which is compatible with a solution* In/Out *of the dataflow equations for* $\mathtt{P}$ *and let* $\rho(x) = q$*. For* $I \in \mathbf{Instrs}_{\mathtt{P}}$*,* $Out(I)(x) = 1$ *holds whenever* $x \in defs(I)$ *or* $Out(I)(x) \sqsupseteq 1$.

Proof. For $\rho(x) = q$, the definition of compatibility implies that x is forwardable to $\gamma(q)$, so by Definition 42 $Out(I)(x) \neq \top$ holds for all $I \in \mathbf{Instrs}_{\mathtt{P}}$, and $Out(I)(x) = 1$ holds for all I with $x \in defs(I)$. Thus, both $x \in defs(I)$ and $Out(I)(x) \sqsupseteq 1$ imply $Out(I)(x) = 1$. □

7.4 Type-correctness

In order to show that the translation preserves the forwarding structure, we relate dataflow solutions for IL to $\otimes$-types in ALEF. In the following discussion, let $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ be a fixed regular IL program and In/Out be a fixed (not necessarily least) solution to its dataflow equations.

Any allocation function ρ which is compatible with a dataflow solution for $\mathtt{P}$ leads to $[\![\mathtt{P}]\!]_\rho$ being well-typed in the typing calculus for sequential execution of ALEF programs in Section 5.1, where types arise from In/Out as follows.

Definition 44. *For allocation function* ρ *and* $I \in \mathbf{Instrs}_{\mathtt{P}}$*, let*

$$
\begin{aligned}
uses^{\rho}(I)(x) &= \begin{cases} \rho(x) & \textit{if } uses(I)(x) \sqsupseteq 1 \\ \mathbf{1} & \textit{otherwise} \end{cases} \\
Out^{\rho}(I)(x) &= \begin{cases} \mathbf{1} & \textit{if } x \notin defs(I), Out(I)(x) \sqsubseteq 0 \textit{ and } \rho(x) = q \\ \rho(x) & \textit{otherwise} \end{cases} \\
In^{\rho}(I)(x) &= \begin{cases} uses^{\rho}(I)(x) & \textit{if } x \in defs(I) \\ uses^{\rho}(I)(x) \otimes Out^{\rho}(I)(x) & \textit{otherwise} \end{cases}
\end{aligned}
$$

Then the types $uses^{\rho}(I)$*,* $In^{\rho}(I)$ *and* $Out^{\rho}(I)$ *are given by*

$$
\begin{aligned}
uses^{\rho}(I) &= \otimes_{x \in \mathbf{Var}_{\mathtt{P}}} uses^{\rho}(I)(x) \\
In^{\rho}(I) &= \otimes_{x \in \mathbf{Var}_{\mathtt{P}}} In^{\rho}(I)(x) \\
Out^{\rho}(I) &= \otimes_{x \in \mathbf{Var}_{\mathtt{P}}} Out^{\rho}(I)(x)
\end{aligned}
$$

Example. The IL program

$$[1]\mathtt{v} = 2\ [2]\mathtt{w} = \mathtt{v} + 5\ [3]\mathtt{if\ w\ 1\ 1} \tag{7.6}$$

has a solution fulfilling

$$Out([1])(\mathtt{v}) = Out([2])(\mathtt{w}) = 1 \text{ and } In([2])(\mathtt{v}) = In([3])(\mathtt{w}) = 1.$$

This solution is compatible with an allocation $\rho(\mathtt{v}) = \mathtt{q}_1$, $\rho(\mathtt{w}) = \mathtt{q}_2$ where $\gamma(\mathtt{q}_1) = \mathtt{ALU}$ and $\gamma(\mathtt{q}_2) = \mathtt{BU}$, and leads to

$$[\![[1][2][3]]\!]_\rho = [1]\mathtt{ldc}\ 2\ \mathtt{q}_1\ [2]\mathtt{addi}\ 5\ \mathtt{q}_1\ \mathtt{q}_2\ [3]\mathtt{if}\ \mathtt{q}_2\ 1\ 1 : \mathbf{1} \multimap \mathbf{1}$$

and

$$\begin{array}{rl}
In^\rho([1])(\mathtt{v}) = \mathbf{1} & In^\rho([1])(\mathtt{w}) = \mathbf{1} \\
Out^\rho([1])(\mathtt{v}) = \mathtt{q}_1 & Out^\rho([1])(\mathtt{w}) = \mathbf{1} \\
In^\rho([2])(\mathtt{v}) = \mathtt{q}_1 & In^\rho([2])(\mathtt{w}) = \mathbf{1} \\
Out^\rho([2])(\mathtt{v}) = \mathbf{1} & Out^\rho([2])(\mathtt{w}) = \mathtt{q}_2 \\
In^\rho([3])(\mathtt{v}) = \mathbf{1} & In^\rho([3])(\mathtt{w}) = \mathtt{q}_2 \\
Out^\rho([3])(\mathtt{v}) = \mathbf{1} & Out^\rho([3])(\mathtt{w}) = \mathbf{1}
\end{array}$$

In particular, $[\![[1][2][3]]\!]_\rho : In^\rho([1]) \multimap Out^\rho([3])$ holds. ◇

The translation of a single instruction may be typed according to its component in In/Out.

Proposition 38. *Let ρ be compatible with In/Out and $I \in \mathbf{Instrs}_\mathtt{P}$. Then*

1. $[\![I]\!]_\rho : In^\rho(I) \multimap Out^\rho(I)$ *and*
2. *for $J \in \mathtt{succ}(I)$ there is a factor $rg(Y)$ of $rg(Out^\rho(J))$ with*

$$Out^\rho(I) = In^\rho(J) \otimes rg(Y).$$

Proof. 1. Case distinction on the form of I.

Case $'[n]x = a'$**.** The definition of $[\![\_]\!]_\rho$ implies $[\![I]\!]_\rho = [\mathtt{n}]\mathtt{ldc}\ a\ \rho(x)$, hence

$$\begin{aligned} In^\rho(I) &= \otimes_{y \in \mathbf{Var}_\mathrm{P}} In^\rho(I)(y) = \mathbf{1} \otimes\ \otimes_{y \neq x}(\mathbf{1} \otimes Out^\rho(I)(y)) \\ Out^\rho(I) &= \otimes_{y \in \mathbf{Var}_\mathrm{P}} Out^\rho(I)(y) = \rho(x) \otimes \otimes_{y \neq x} Out^\rho(I)(y) \end{aligned}$$

and the typing rule for $\mathtt{ldc}$ succeeds for $X = \otimes_{y \neq x} Out^\rho(I)(y)$.

Case $'[n]x = z'$ **where** $x \neq z$ **and** $\rho(z) = q$. The definition of $[\![\_]\!]_\rho$ implies

$$[\![I]\!]_\rho = [n]\mathtt{id}^{\gamma(q)}\ q\ \rho(x),$$

and equation (6.12) implies

$$uses(I)(y) = \begin{cases} 1 & \text{if } y = z \\ 0 & \text{otherwise.} \end{cases}$$

Hence,

$$\begin{aligned} In^\rho(I) &= \mathbf{1} \otimes \rho(z) \otimes Out^\rho(I)(z) \otimes\ \otimes_{y \notin \{x,z\}}(\mathbf{1} \otimes Out^\rho(I)(y)) \\ &= q \otimes\ \otimes_{y \neq x} Out^\rho(I)(y) \\ Out^\rho(I) &= \rho(x) \otimes \otimes_{y \neq x} Out^\rho(I)(y) \end{aligned}$$

and the typing rule for $\mathtt{id}$ succeeds with $X = \otimes_{y \neq x} Out^\rho(I)(y)$, because the side condition SC reads $\gamma(q) = \gamma(q)$ and is trivially fulfilled.

Case $'[n]x = z'$ **where** $x \neq z$ **and** $\rho(z) = r$. The definition of $[\![\_]\!]_\rho$ yields

$$[\![I]\!]_\rho = [n]\mathtt{id}^{\mathtt{ALU}}\ r\ \rho(x),$$

and

$$uses(I)(y) = \begin{cases} 1 & \text{if } y = z \\ 0 & \text{otherwise} \end{cases}$$

holds by equation (6.12). Hence,

$$\begin{aligned} In^\rho(I) &= \mathbf{1} \otimes \rho(z) \otimes Out^\rho(I)(z) \otimes\ \otimes_{y \notin \{x,z\}}(\mathbf{1} \otimes Out^\rho(I)(y)) \\ &= r \otimes\ \otimes_{y \neq x} Out^\rho(I)(y) \\ Out^\rho(I) &= \rho(x) \otimes r \otimes\ \otimes_{y \notin \{x,z\}} Out^\rho(I)(y) \\ &= \rho(x) \otimes r \otimes\ \otimes_{y \neq x} Out^\rho(I)(y) \end{aligned}$$

due to $Out^\rho(I)(z) = r$ and $r \otimes r = r$. Again, the typing judgement for $[\![I]\!]_\rho$ follows for $X = \otimes_{y \neq x} Out^\rho(I)(y)$ because the side condition SC is void.

Case $'[n]x = x'$ **where** $\rho(x) = q$. From the definition of $[\![\_]\!]_\rho$ we obtain $[\![I]\!]_\rho = [n]\mathtt{id}^{\gamma(q)}\ q\ q$, hence

$$\begin{aligned} In^\rho(I) &= q \otimes \otimes_{y \neq x}(\mathbf{1} \otimes Out^\rho(I)(y)) \\ Out^\rho(I) &= q \otimes \otimes_{y \neq x} Out^\rho(I)(y) \end{aligned}$$

The claim follows for $X = \otimes_{y \neq x} Out^\rho(I)(y)$ and the side condition is fulfilled.

Case $'[n]x = x'$ **where** $\rho(x) = r$. The definition of $[\![\_]\!]_\rho$ yields $[\![I]\!]_\rho = [n]\mathtt{id}^{\mathtt{ALU}}\ r\ r$, hence

$$\begin{aligned} In^\rho(I) &= r \otimes \otimes_{y \neq x}(\mathbf{1} \otimes Out^\rho(I)(y)) \\ Out^\rho(I) &= r \otimes \otimes_{y \neq x} Out^\rho(I)(y) \end{aligned}$$

The claim follows for $X = \otimes_{y \neq x} Out^\rho(I)(y)$ and the side condition is fulfilled.

Cases $'[n]x = z\ bop\ a'$ **and** $'[n]x = x\ bop\ a'$**.** Similar to the cases $'[n]x = z'$ and $'[n]x = x'$, with $fu = \mathtt{MUL}$ for $bop \in \{*\}$ and $fu = \mathtt{ALU}$ for $bop \in \{-, +\}$.

Case $'[n]x = z\ bop\ v'$ **where** $z \neq x \neq v$**.** We treat the case $bop \in \{*\}$ and $\rho(z) = q_1 \neq q_2 = \rho(v)$ explicitly, all other cases being similar.

The definition of $[\![\_]\!]_\rho$ yields $[\![I]\!]_\rho = [n]\mathtt{mul}\ q_1\ q_2\ \rho(x)$, and equation (6.12) implies

$$\begin{aligned} In^\rho(I) &= In^\rho(I)(x) \otimes In^\rho(I)(z) \otimes In^\rho(I)(v) \otimes \otimes_{y \notin \{x,z,v\}} In^\rho(I)(y) \\ &= \mathbf{1} \otimes q_1 \otimes Out^\rho(I)(z) \otimes q_2 \otimes Out^\rho(I)(v) \otimes \\ &\quad \otimes_{y \notin \{x,z,v\}}(\mathbf{1} \otimes Out^\rho(I)(y)) \\ &= q_1 \otimes q_2 \otimes \otimes_{y \neq x} Out^\rho(I)(y) \\ Out^\rho(I) &= Out^\rho(I)(x) \otimes \otimes_{y \neq x} Out^\rho(I)(y) = \rho(x) \otimes \otimes_{y \neq x} Out^\rho(I)(y). \end{aligned}$$

With $X = \otimes_{y \neq x} Out^\rho(I)(y)$ the typing judgement $[\![I]\!]_\rho : X \otimes q_1 \otimes q_2 \multimap X \otimes \rho(x)$ is derivable. The side condition reads $\gamma(q_1) = \gamma(q_2) = \mathtt{MUL}$ and is fulfilled because the compatibility of ρ with In/Out implies that z and v

are forwardable to $\gamma(q_1)$ and $\gamma(q_2)$, respectively. By the syntactic form of I, these variables are only forwardable to $fu = \mathtt{MUL}$ by Definition 42, so $\gamma(q_1) = \gamma(q_2) = \mathtt{MUL}$ holds.

Other cases $'[n]x = z\ bop\ v'$**.** Similar.

Case $'[n]\mathtt{if}\ x\ m\ l'$**.** The definition of $[\![\_]\!]_\rho$ yields $[\![I]\!]_\rho = [n]\mathtt{if}\ \rho(x)\ m\ l$, and the earlier definition yield $defs(I) = \{x\}$, $uses(I)(x) = 1$ and $uses(I)(y) = 0$ for all $y \neq x$, hence

$$\begin{aligned} In^\rho(I) &= \rho(x) \otimes \otimes_{y \in \mathbf{Var}_\mathrm{P}} (\mathbf{1} \otimes Out^\rho(I)(y)) \\ Out^\rho(I) &= \otimes_{y \in \mathbf{Var}_\mathrm{P}} Out^\rho(I)(y) \end{aligned}$$

The claim follows with $X = \otimes_{y \in \mathbf{Var}_\mathrm{P}} Out^\rho(I)(y)$. The side condition reads $\gamma(\rho(x)) = \mathtt{BU}$ and is trivially fulfilled if $\rho(x) = r$, and is fulfilled by the definition of compatibility and forwardability if $\rho(x) = q$.

Case $'[n]\mathtt{jmp}\ m'$**.** The definition of $[\![\_]\!]_\rho$ yields $[\![I]\!]_\rho = [n]\mathtt{jmp}\ m$, and we obtain

$$In^\rho(I) = \otimes_{y \in \mathbf{Var}_\mathrm{P}} uses^\rho(I)(y) \otimes Out^\rho(I)(y) = \otimes_{y \in \mathbf{Var}_\mathrm{P}} Out^\rho(I)(y) = Out^\rho(I)$$

so with $X = In^\rho(I)$ the typing rule for `jmp` succeeds.

2. We show that for each $x \in \mathbf{Var}_\mathrm{P}$ there is a factor $rg(Y_x)$ of $rg(Out^\rho(J))$ such that $Out^\rho(I)(x) = In^\rho(J)(x) \otimes rg(Y_x)$ holds. Then $rg(Y) = \otimes_{x \in \mathbf{Var}_\mathrm{P}} rg(Y_x)$ is a factor of $rg(Out^\rho(J))$ and the claim follows due to

$$Out^\rho(I) = \otimes_{x \in \mathbf{Var}_\mathrm{P}} Out^\rho(I)(x) = \otimes_{x \in \mathbf{Var}_\mathrm{P}} (In^\rho(J)(x) \otimes rg(Y_x)) = In^\rho(J) \otimes rg(Y).$$

We argue by case distinction on the value of $Out^\rho(I)(x)$.

Case $Out^\rho(I)(x) = \mathbf{1}$**.** The definition of $Out^\rho(I)(x)$ implies $x \notin defs(I)$, $\rho(x) = q$ and $Out(I)(x) \sqsubseteq 0$. From Proposition 33 we obtain $In(J)(x) \sqsubseteq Out(I)(x) \sqsubseteq 0$, hence $uses(J)(x) \sqsubseteq 0$ and

$$uses^\rho(J)(x) = \mathbf{1}.$$

There are now two cases, depending on whether $x \in defs(J)$ holds or not.

If $x \in defs(J)$ then $In^{\rho}(J)(x) = uses^{\rho}(J)(x) = \mathbf{1}$ follows, so the claim holds with $Y_x = \mathbf{1}$.

If $x \notin defs(J)$ then $0 \sqsupseteq In(J)(x) = uses(J)(x) \oplus Out(J)(x)$ holds, hence $0 \sqsupseteq Out(J)(x)$ and $Out^{\rho}(J)(x) = \mathbf{1}$. Therefore,

$$In^{\rho}(J)(x) = uses^{\rho}(J)(x) \otimes Out^{\rho}(J)(x) = \mathbf{1}$$

and the claim holds for $Y_x = \mathbf{1}$.

In both cases, $rg(Y_x) = \mathbf{1}$ is trivially a factor of $rg(Out^{\rho}(J))$.

Case $Out^{\rho}(I)(x) = q$. The definition of $Out^{\rho}(I)(x)$ forces $\rho(x) = q$, and $x \in defs(I)$ or $Out(I)(x) \sqsupseteq 1$ hold. By Lemma 15, $Out(I)(x) = 1$ follows in either case, and from Proposition 33 we obtain $In(J)(x) \sqsubseteq Out(I)(x) = 1$, hence $In(J)(x) \in \{\perp, 1\}$. Again there are two cases.

If $x \in defs(J)$ then $\{\perp, 1\} \ni In(J)(x) = uses(J)(x) \in \{0, 1, \top\}$ holds, hence $1 = In(J)(x) = uses(J)(x)$ and $In^{\rho}(J)(x) = uses^{\rho}(J)(x) = q$. Take $Y_x = \mathbf{1}$.

If $x \notin defs(J)$, then $\{\perp, 1\} \ni In(J)(x) = uses(J)(x) \oplus Out(J)(x)$ holds. In the case of $uses(J)(x) = 1$, $Out(J)(x) \sqsubseteq 0$ follows, so $Out^{\rho}(J)(x) = \mathbf{1}$ and

$$In^{\rho}(J)(x) = uses^{\rho}(J)(x) \otimes Out^{\rho}(J)(x) = \rho(x) \otimes \mathbf{1} = q.$$

In the case of $uses(J)(x) = 0$ and $Out(J)(x) = 1$,

$$uses^{\rho}(J)(x) = \mathbf{1} \text{ and } Out^{\rho}(J)(x) = \rho(x) = q$$

follow, hence

$$In^{\rho}(J)(x) = uses^{\rho}(J)(x) \otimes Out^{\rho}(J)(x) = q.$$

In both cases, take $Y_x = \mathbf{1}$ which is a factor of $rg(Out^{\rho}(J))$. The case of $uses(J)(x) = 0$ and $Out(J)(x) = \perp$ cannot occur: from $\rho(x) = q$ and the definition of compatibility we know that x is forwardable to $\gamma(q)$, i.e. that $Out(J)(x) \notin \{\perp, \top\}$ holds.

Case $Out^{\rho}(I)(x) = r$. Then $\rho(x) = r$ follows, and there are three cases.

Case $uses(J)(x) = 0$ **and** $x \in defs(J)$**.** Then

$$In^{\rho}(J)(x) = uses^{\rho}(J)(x) = \mathbf{1}$$

holds, and we take $Y_x = r$. By Lemma 14, $\rho(x) = r = Y_x$ is a factor of $B_{[\![J]\!]_\rho}$, hence $Y_x \otimes Z = B_{[\![J]\!]_\rho}$ holds for some Z. The first part of this proposition implies $Out^{\rho}(J) = B_{[\![J]\!]_\rho} \otimes W$ for some W, so $Out^{\rho}(J) = Y_x \otimes Z \otimes W$ and $rg(Y_x) = r$ is a factor of $rg(Out^{\rho}(J))$.

Case $uses(J)(x) \sqsupseteq 1$ **and** $x \in defs(J)$**.** Then $Out^{\rho}(J)(x) = r$ holds due to $\rho(x) = r$, and $In^{\rho}(J)(x) = uses^{\rho}(J)(x) = r$. We take $Y_x = \mathbf{1}$.

Case $x \notin defs(J)$**.** Again, $Out^{\rho}(J)(x) = r$ holds due to $\rho(x) = r$, and we have $uses^{\rho}(J)(x) \in \{\mathbf{1}, r\}$, so $In^{\rho}(J)(x) = r$ follows and we take $Y_x = \mathbf{1}$.

□

The situation in the case "$Out^{\rho}(I)(x) = q$" in the proof of part (2) motivates the condition $Out(J)(x) \neq \top$ for all I in the definition of forwardability (Definition 42). Consider the program P,

$$\begin{array}{rclrclrcl}
\mathtt{N}(1) & = & [1]\mathtt{v} = \ldots & \mathtt{N}(3) & = & [3]\mathtt{x} = 2 & \mathtt{N}(5) & = & [5]\mathtt{if\ v\ 3\ 5} \\
 & & [2]\mathtt{if\ v\ 3\ 5} & & & [4]\mathtt{y} = \mathtt{x} + 1 & & &
\end{array}$$

and its dataflow equations for x

$$\begin{array}{rcl}
\mathit{fwdIn}([1])(\mathtt{x}) & = & 0 \oplus \mathit{fwdOut}([1])(\mathtt{x}) = \mathit{fwdIn}([2])(\mathtt{x}) \\
\mathit{fwdIn}([2])(\mathtt{x}) & = & 0 \oplus \mathit{fwdOut}([2])(\mathtt{x}) = \mathit{fwdIn}([3])(\mathtt{x}) \sqcup \mathit{fwdIn}([5])(\mathtt{x}) \\
\mathit{fwdIn}([3])(\mathtt{x}) & = & 0 \\
\mathit{fwdOut}([3])(\mathtt{x}) & = & \mathit{fwdIn}([4])(\mathtt{x}) = 1 \oplus \mathit{fwdOut}([4])(\mathtt{x}) \\
\mathit{fwdOut}([4])(\mathtt{x}) & = & 0 \\
\mathit{fwdIn}([5])(\mathtt{x}) & = & 0 \oplus \mathit{fwdOut}([5])(\mathtt{x}) = \mathit{fwdIn}([3])(\mathtt{x}) \sqcup \mathit{fwdIn}([5])(\mathtt{x})
\end{array}$$

In particular, the last equation simplifies to

$$\mathit{fwdIn}([5])(\mathtt{x}) \quad = \quad 0 \sqcup \mathit{fwdIn}([5])(\mathtt{x}) \tag{7.7}$$

for which a (non-minimal) fixed point is

$$\mathit{fwdIn}([5])(\mathtt{x}) = \top.$$

A solution In/Out based on this fixed point fulfils

$$\mathit{In}([1])(\mathtt{x}) = \mathit{Out}([1])(\mathtt{x}) = \mathit{In}([2])(\mathtt{x}) = \mathit{Out}([2])(\mathtt{x}) = \top.$$

The condition $\mathit{Out}(I)(x) \neq \top$ in Definition 42 ensures that for this solution variable $\mathtt{x}$ is not forwardable, despite being used linearly.

Suppose the condition was dropped. Then, an allocation $\rho(\mathtt{x}) = q$ would be possible due to $\mathit{fwdOut}([3])(\mathtt{x}) = 1$, leading to

$$\begin{aligned} \mathit{In}^{\rho}([1])(\mathtt{x}) &= \mathbf{1} \otimes \mathit{Out}^{\rho}([1])(\mathtt{x}) = q \\ \mathit{In}^{\rho}([2])(\mathtt{x}) &= \mathbf{1} \otimes \mathit{Out}^{\rho}([2])(\mathtt{x}) = q \\ \mathit{In}^{\rho}([3])(\mathtt{x}) &= \mathbf{1} \end{aligned}$$

and Lemma 15 and Proposition 38 (part (2)) would be violated as $[3] \in \mathtt{succ}([2])$ holds. Furthermore, $\mathit{In}^{\rho}([1])(\mathtt{x}) = q$ means that an initial ALEF configuration would have to provide a value in q which results in functionally incorrect behaviour for $\mathtt{v} = 0$ as instruction $[\![[4]]\!]_{\rho}$ increments the head value of q.

On the other hand, Definition 42 does allow $\mathtt{x}$ to be forwarded to $\mathtt{ALU}$. For this, however, a different fixed point must be chosen. In particular, the least fixed point $\mathit{fwdIn}([5])(\mathtt{x}) = 0$ to equation (7.7) leads to

$$\mathit{In}([1])(\mathtt{x}) = \mathit{Out}([1])(\mathtt{x}) = \mathit{In}([2])(\mathtt{x}) = \mathit{Out}([2])(\mathtt{x}) = 0.$$

This solution fulfils the condition in Definition 42 and also Lemma 15, and for $\rho(\mathtt{x}) = q$ it yields

$$\begin{aligned} \mathit{In}^{\rho}([1])(\mathtt{x}) &= \ldots = \mathit{Out}^{\rho}([2])(\mathtt{x}) = \mathbf{1} \\ \mathit{In}^{\rho}([3])(\mathtt{x}) &= \mathbf{1} \\ \mathit{In}^{\rho}([5])(\mathtt{x}) &= \mathit{Out}^{\rho}([5])(\mathtt{x}) = \mathbf{1}. \end{aligned}$$

Hence, Proposition 38 (part (2)) is satisfied and functionally correct behaviour is observed as an initial configuration does not need to provide a value in q.

For registers, a similar mismatch between $Out^{\rho}(I)$ and $In^{\rho}(J)$ motivates the factor Y in Proposition 38, and indeed requires most registers to be initialised at the start of an ALEF execution.

Example. Consider the IL program

$$[1]\mathtt{v} = 2\ [2]\mathtt{w} = 5\ [3]\mathtt{z} = 6 \tag{7.8}$$

and the allocation ρ with $\rho(x) = r_x$ for $x \in \{\mathtt{v}, \mathtt{w}, \mathtt{z}\}$ and $r_x \neq r_y$ for $x \neq y$. We obtain

$$\begin{array}{rcl} In^{\rho}([1]) & = & r_{\mathtt{w}} \otimes r_{\mathtt{z}} \\ In^{\rho}([2]) & = & r_{\mathtt{v}} \otimes r_{\mathtt{z}} \\ In^{\rho}([3]) & = & r_{\mathtt{v}} \otimes r_{\mathtt{w}} \end{array} \quad \text{and} \quad \begin{array}{rcl} [\![[1]]\!]_{\rho} & = & \mathtt{ldc}\ 2\ r_{\mathtt{v}} \\ [\![[2]]\!]_{\rho} & = & \mathtt{ldc}\ 5\ r_{\mathtt{w}} \\ [\![[3]]\!]_{\rho} & = & \mathtt{ldc}\ 6\ r_{\mathtt{z}} \end{array}$$

and for $I \in \{[1], [2], [3]\}$, $Out^{\rho}(I) = r_{\mathtt{v}} \otimes r_{\mathtt{w}} \otimes r_{\mathtt{z}}$ and $[\![I]\!] : In^{\rho}(I) \multimap Out^{\rho}(I)$ follow, hence

$$\begin{array}{rcl} Out^{\rho}([1]) & = & In^{\rho}([2]) \otimes r_{\mathtt{w}} \\ Out^{\rho}([2]) & = & In^{\rho}([3]) \otimes r_{\mathtt{z}} \end{array}$$

and $[\![[1]\,[2]\,[3]]\!]_{\rho} : In^{\rho}([1]) \multimap Out^{\rho}([3])$. $\diamond$

The definition of In^{ρ} and Out^{ρ} yields the following result.

Corollary 3. *Let ρ be compatible with In/Out and $B = I_1 \ldots I_n$ with $|B| > 0$. If $rg(Y)$ is a factor of $rg(Out^{\rho}(I_1))$ then*

$$[\![B]\!]_{\rho} : In^{\rho}(I_1) \otimes rg(Y) \multimap Out^{\rho}(I_n)$$

and $Out^{\rho}(I_n) = rg(Out^{\rho}(I_1)) \otimes X$ for some X.

Proof. Induction on n.

Case $n = 1$. Applying Proposition 38 (part (1)) yields $[\![I_1]\!]_{\rho} : In^{\rho}(I_1) \multimap Out^{\rho}(I_1)$, so by the results of Chapter 5, $[\![I_1]\!]_{\rho} : In^{\rho}(I_1) \otimes rg(Y) \multimap Out^{\rho}(I_1)$ holds for any factor $rg(Y)$ of $Out^{\rho}(I_1)$. Take $X = Out^{\rho}(I_1)$.

Case $n > 1$. For $B = I_1 I_2 \ldots I_n$ the base case yields

$$[\![I_1]\!]_\rho : In^\rho(I_1) \otimes rg(Y) \multimap Out^\rho(I_1)$$

and Proposition 38 (part (2)) yields

$$Out^\rho(I_1) = In^\rho(I_2) \otimes rg(Y_2)$$

for some factor $rg(Y_2)$ of $rg(Out^\rho(I_2))$, i.e.

$$rg(Y_2) \otimes W = rg(Out^\rho(I_2)) \text{ for some } W.$$

Consequently, the induction hypothesis can be applied to $I_2 \ldots I_n$, resulting in

$$[\![I_2 \ldots I_n]\!]_\rho : In^\rho(I_2) \otimes rg(Y_2) \multimap Out^\rho(I_n)$$

and by applying the cut rule we obtain

$$[\![I_1 \ldots I_n]\!]_\rho : In^\rho(I_1) \otimes rg(Y) \multimap Out^\rho(I_n)$$

as claimed. The induction hypothesis also results in $Out^\rho(I_n) = rg(Out^\rho(I_2)) \otimes V$ for some V. The identity $Out^\rho(I_2) \otimes rg(Y_2) = Out^\rho(I_2)$ yields

$$\frac{[\![I_2]\!]_\rho : In^\rho(I_2) \multimap Out^\rho(I_2)}{[\![I_2]\!]_\rho : In^\rho(I_2) \otimes rg(Y_2) \multimap Out^\rho(I_2)}$$

and by applying Lemma 5 (in Chapter 5) we obtain

$$\begin{aligned} Out^\rho(I_2) &= rg(In^\rho(I_2) \otimes rg(Y_2)) \otimes Out^\rho(I_2) \\ &= rg(Out^\rho(I_1)) \otimes Out^\rho(I_2). \end{aligned}$$

Consequently, $rg(Out^\rho(I_2)) = rg(Out^\rho(I_1)) \otimes rg(Out^\rho(I_2))$ holds and thus

$$Out^\rho(I_n) = rg(Out^\rho(I_2)) \otimes V = rg(Out^\rho(I_1)) \otimes rg(Out^\rho(I_2)) \otimes V,$$

hence $Out^\rho(I_n) = rg(Out^\rho(I_1)) \otimes X$ for $X = rg(Out^\rho(I_2)) \otimes V$.

□

We are thus able to show well-typedness of the result of the translation.

Theorem 8. *If* ρ *is an allocation compatible with a solution In/Out of a regular program* $\mathtt{P}$ *then* $\Sigma \vdash [\![P]\!]_\rho : \circ$ *for some* Σ*.*

Proof. For $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ and any $m \in dom\ N$, Corollary 3 with $rg(Y) = \mathbf{1}$ yields

$$[\![\mathtt{N}(m)]\!]_\rho : In^\rho\,(fst(\mathtt{N}(m))) \multimap Out^\rho\,(lst(\mathtt{N}(m)))$$

and

$$Out^\rho\,(lst(\mathtt{N}(m))) = rg(Out^\rho\,(fst(\mathtt{N}(m)))) \otimes X_m$$

for some X_m. For the abbreviations

$$A_m = In^\rho\,(fst(\mathtt{N}(m))) \quad \text{and} \quad B_m = Out^\rho\,(lst(\mathtt{N}(m)))$$

there are thus ALEF basic blocks $\mathtt{N}(m) = [\![\mathtt{N}(m)]\!]_\rho$ such that

$$\mathtt{N}(m) : A_m \multimap B_m$$

and $B_m = rg(Out^\rho\,(fst(\mathtt{N}(m)))) \otimes X_m$. Take

$$\Sigma = \cup_{m \in dom\ \mathtt{N}} \mathtt{N}_m : A_m \multimap B_m.$$

The graph of basic blocks in ALEF contains exactly the same arrows as the IL program, by Table 7.1. Consequently, for $(m,k) \in \mathtt{A}_{ALEF}$ we obtain $fst(\mathtt{N}(k)) \in \mathtt{succ}(lst(\mathtt{N}(m)))$, and by Proposition 38 (part (2)) there is a factor $rg(Y_{mk})$ of $rg(Out^\rho(fst(\mathtt{N}(k))))$ such that

$$B_m = A_k \otimes rg(Y_{mk})$$

holds. Thus,

$$B_m = A_k \otimes rg(Y_{mk}) \text{ and } rg(Y_{mk}) \otimes rg(Z) = rg(Out^\rho(fst(\mathtt{N}(k)))) = rg(B_k)$$

hold for some Z whenever $(m,k) \in \mathtt{A}_{ALEF}$, so the condition on arrows in rule PROG-R is fulfilled. □

The result of Theorem 8 may be used to justify the condition $Out(J)(x) \neq \bot$ for all I in the definition of forwardability (Definition 42).

Example. The dataflow equations for variable w in program (7.9)

$$\begin{array}{lclclcl} \mathtt{N}(1) & = & [1]\mathtt{v}=17 & \quad & \mathtt{N}(3) & = & [3]\mathtt{w}=2 \\ & & [2]\mathtt{if\ v\ 3\ 6} & & & & [4]\mathtt{if\ v\ 5\ 6} \\ \mathtt{N}(5) & = & [5]\mathtt{x}=\mathtt{w} & & \mathtt{N}(6) & = & [6]\mathtt{jmp\ 6} \end{array} \tag{7.9}$$

are

I	$In(I)(\mathtt{w})$	$Out(I)(\mathtt{w})$
[1]	$In([3])(\mathtt{w}) \sqcup In([6])(\mathtt{w})$	$In([3])(\mathtt{w}) \sqcup In([6])(\mathtt{w})$
[2]	$In([3])(\mathtt{w}) \sqcup In([6])(\mathtt{w})$	$In([3])(\mathtt{w}) \sqcup In([6])(\mathtt{w})$
[3]	0	$In([5])(\mathtt{w}) \sqcup In([6])(\mathtt{w})$
[4]	$In([5])(\mathtt{w}) \sqcup In([6])(\mathtt{w})$	$In([5])(\mathtt{w}) \sqcup In([6])(\mathtt{w})$
[5]	1	0
[6]	$In([6])(\mathtt{w})$	$In([6])(\mathtt{w})$

The least solution In/Out arises from choosing $In([6])(\mathtt{w}) = \perp$:

$$\begin{array}{rcl} Out([3])(\mathtt{w}) = In([4])(\mathtt{w}) = Out([4])(\mathtt{w}) & = & 1 \\ In([6])(\mathtt{w}) = Out([6])(\mathtt{w}) & = & \perp \\ In([1])(\mathtt{w}) = Out([1])(\mathtt{w}) = In([2])(\mathtt{w}) = Out([2])(\mathtt{w}) & = & 0 \end{array}$$

If the condition $\forall I.Out(J)(x) \neq \perp$ in Definition 42 was dropped, w would be forwardable, and we could choose an allocation ρ which maps w to operand queue q, and v and x to (say) $\mathtt{r}_1$ and $\mathtt{r}_2$. However, this allocation falsifies Proposition 38(part (2)) as it yields $Out^{\rho}([4])(\mathtt{w}) = \mathtt{q}$ and $In^{\rho}([6])(\mathtt{w}) = 1$. More importantly, Theorem 8 is also violated as the translation $[\![\mathtt{P}_{(7.9)}]\!]_{\rho}$ is ill-typed. The principal typings

$$\begin{array}{lclclcl} [\![\mathtt{N}(1)]\!]_{\rho} & :: & \mathbf{1} \multimap \mathtt{r}_1 & \quad & [\![\mathtt{N}(5)]\!]_{\rho} & :: & \mathtt{q} \multimap \mathtt{r}_2 \\ [\![\mathtt{N}(3)]\!]_{\rho} & :: & \mathtt{r}_1 \multimap \mathtt{q} \otimes \mathtt{r}_1 & & [\![\mathtt{N}(6)]\!]_{\rho} & :: & \mathbf{1} \multimap \mathbf{1} \end{array}$$

of the translations

$$\begin{array}{lclclcl} [\![\mathtt{N}(1)]\!]_{\rho} & = & [1]\mathtt{ldc\ 17\ r}_1 & \quad & [\![\mathtt{N}(3)]\!]_{\rho} & = & [3]\mathtt{ldc\ 2\ q} \\ & & [2]\mathtt{if\ r}_1\ \mathtt{3\ 6} & & & & [4]\mathtt{if\ r}_1\ \mathtt{5\ 6} \\ [\![\mathtt{N}(5)]\!]_{\rho} & = & [5]\mathtt{id}^{\gamma(\mathtt{q})}\ \mathtt{q\ r}_2 & & [\![\mathtt{N}(6)]\!]_{\rho} & = & [6]\mathtt{jmp\ 6} \end{array}$$

cannot be unified. Indeed, all alternative fixed points to $In([6])(\mathtt{w})$ yield solutions In/Out for which w is not forwardable. $\diamond$

Similar to the situation for the condition $Out(I)(x) \neq \top$, the restriction $Out(I)(x) \neq \bot$ may often be satisfied without loosing forwardability, if a non-minimal fixed point is chosen.

Example. The dataflow equations for variable w for program

$$\begin{array}{rclrclrcl} \mathtt{N(1)} & = & [1]\mathtt{v}=3 & \mathtt{N(4)} & = & [4]\mathtt{jmp}\ 4 & \mathtt{N(5)} & = & [5]\mathtt{x}=\mathtt{w} \\ & & [2]\mathtt{w}=6 & & & & & & \\ & & [3]\mathtt{if}\ \mathtt{v}\ 4\ 5 & & & & & & \end{array} \tag{7.10}$$

are

I	$In(I)(\mathtt{w})$	$Out(I)(\mathtt{w})$
$[1]$	0	0
$[2]$	0	$In([4])(\mathtt{w}) \sqcup In([5])(\mathtt{w})$
$[3]$	$In([4])(\mathtt{w}) \sqcup In([5])(\mathtt{w})$	$In([4])(\mathtt{w}) \sqcup In([5])(\mathtt{w})$
$[4]$	$In([4])(\mathtt{w})$	$In([4])(\mathtt{w})$
$[5]$	1	0

The least solution In/Out arises from choosing $In([4])(\mathtt{w}) = \bot$:

$$\begin{array}{rcl} In([4])(\mathtt{w}) = Out([4])(\mathtt{w}) & = & \bot \\ Out([2])(\mathtt{w}) = In([3])(\mathtt{w}) = Out([3])(\mathtt{w}) & = & 1 \end{array}$$

If the condition $\forall I.Out(I)(x) \neq \bot$ was dropped, w would be forwardable with respect to this solution. Again, choosing $\rho(\mathtt{w}) = \mathtt{q}$ would violate Proposition 38(part (2)) as it would imply $Out^{\rho}([3])(\mathtt{w}) = \mathtt{q}$ and $In^{\rho}([4])(\mathtt{w}) = \mathbf{1}$. In contrast to the previous example, Theorem 8 would still hold, but not for the types constructed in its proof: the resulting program $[\![\mathtt{P}_{(7.10)}]\!]_{\rho}$ may only be unified after weakening the typing of $[\![\mathtt{N(4)}]\!]_{\rho}$ by q. However, the solution arising from the alternative fixed point $In([4])(\mathtt{w}) = 1$ fulfils

$$Out([2])(\mathtt{w}) = In([3])(\mathtt{w}) = Out([3])(\mathtt{w}) = In([4])(\mathtt{w}) = Out([4])(\mathtt{w}) = 1$$

and allows w to be forwarded. For the same allocation as above, we obtain

$$In^{\rho}(I)(\mathtt{w}) = \begin{cases} \mathbf{1} & \text{if } I \in \{[1],[2]\} \\ \mathtt{q} & \text{otherwise} \end{cases} \quad \text{and} \quad Out^{\rho}(I)(\mathtt{w}) = \begin{cases} \mathbf{1} & \text{if } I \in \{[1],[5]\} \\ \mathtt{q} & \text{otherwise.} \end{cases}$$

Hence, Proposition 38 is fulfilled, $[\![\mathtt{P}_{(7.10)}]\!]_{\rho}$ is well-typed and its typing arises from In/Out as described in the proof of Theorem 8.

For the solutions arising from the remaining choices $In([4])(\mathtt{w}) = 0$ and $In([4])(\mathtt{w}) = \top$, variable w is not forwardable. $\diamond$

As the earlier example program (7.8) shows, initial ALEF configurations have to provide values in nearly all registers, despite the requirement that any IL-variable be assigned to prior to its first use. This property results from using the least upper bound operation in the definition of *fwdOut* in equations (6.13) and the slack-less equality $fwdIn(I)(x) = uses(I)(x)$ if $x \in defs(I)$. While the condition on variable assignments ensures that this mismatch does not lead to functionally incorrect behaviour, a tighter correspondence between In and In^{ρ} would be desirable. This may be possible by a second dataflow analysis in forward-direction after the decision on forwardability has been made. Such a dual analysis might resemble the effect of the operation $\ominus$ (see Section 6.4), but a detailed study is left for future research. An alternative may be to employ translations τ^{ρ} which relate $\oplus$ and $\otimes$ homomorphically, like

$$\begin{array}{rclcrcl} \tau^{\rho}_x(\top) & = & \rho(x) & \qquad & \tau^{\rho}_x(0) & = & \mathbf{1} \\ \tau^{\rho}_x(1) & = & \rho(x) & & \tau^{\rho}_x(\bot) & = & \mathbf{1} \end{array}$$

with subsequent definitions

$$defs(I)(x) = \begin{cases} 1 & \text{if } x \in defs(I) \\ 0 & \text{otherwise} \end{cases}$$

and

$$\begin{array}{rclcrcl} \tau^{\rho}_{defs}(I) & = & \otimes_{x \in defs(I)} \tau^{\rho}_x(defs(I)(x)) & \qquad & \tau^{\rho}_{In}(I) & = & \otimes_{x \in \mathbf{Var}_{\mathtt{P}}} \tau^{\rho}_x(In(I)(x)) \\ \tau^{\rho}_{uses}(I) & = & \otimes_{x \in uses(I)} \tau^{\rho}_x(uses(I)(x)) & & \tau^{\rho}_{Out}(I) & = & \otimes_{x \in \mathbf{Var}_{\mathtt{P}}} \tau^{\rho}_x(Out(I)(x)). \end{array}$$

7.5 Allocating operand queues and registers

Operand queues and registers are assigned to variables by defining the function $\rho : \mathbf{Var}_{\mathtt{P}} \to OP$. For register allocation, many algorithms exist in the literature which ensure that variables which share a register cannot be in use at the same time. These algorithms are often presented as (extensions of) graph colouring problems where nodes represent variables and undirected arcs (x, y) represent conflicts. Conflicts arise between variables x and y if there is an instruction I at which both x and y are live. Any node colouring of the conflict graph which maps adjacent nodes to different colours yields a register allocation in which conflicting variables are assigned to different registers. The task to colour the nodes of a graph with a *minimal* number of colours can be proven to be NP-complete. In practice, algorithms therefore compute non-optimal solutions which use (in general) a higher number of registers than necessary. Numerous varieties of register allocation algorithms exist, coping with finite supply of registers, restrictions on the usability of registers for values of a particular type, or supporting register windowing for procedure calls.

For allocating operand queues and registers to IL programs, we assume an infinite supply of operand queues and registers, leaving the combination with register or operand queue *spilling* [App98a] for future research. Restricting ourselves to the most basic case, we allocate registers to non-forwardable variables using the above colouring approach and assign operand queues to forwardable variables by colouring a a family of graphs, one for each type of functional unit.

Due to the order of values inside operand queues, the conflict condition for operand queue mapped variables differs from that for register mapped variables. Instead of employing joint liveness, variables x and y which are forwardable to the same functional unit conflict if there is a path of instructions in $\mathtt{P}$ which assigns to x and y in the order *x before y* reads in the order *y before x*.

Definition 45. *For IL program $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ and $x \in \mathbf{Var}_{\mathtt{P}}$, a path π in $\mathtt{P}$ is called an x-*path *if $x \in defs(\pi(1))$ and*

- *if π is finite and n its length, then $x \in uses(\pi(n))$ holds and $x \notin defs(\pi(i))$ for all $1 < i < n$*

- *if* π *is infinite then* $x \notin defs(\pi(i))$ *holds for all* $1 < i$

Conflicts between variables which are forwardable to the same *fu* may be expressed as containment between paths.

Definition 46. *For* $x \neq y \in \mathbf{Var}_{\mathtt{P}}$*, a (finite or infinite) x-path* π contains *a finite y-path* τ *if*

- *there is a* $k \geq 0$ *such that for all* $i \in dom\, \tau$*,* $\tau(i) = \pi(i+k)$ *holds, and*
- *if* π *is finite and the last instructions of* π *and* τ *coincide, then this last instruction is of the form* $z = y\ bop\ x$ *for some* z*.*

In particular, no x-path contains an infinite y-path. In contrast, a finite x-path $(I_1, \ldots, I_n)$ contains $(I_2, \ldots, I_{n-1})$ always, and contains $(I_2, \ldots, I_n)$ if I_n is of the form $z = y\ bop\ x$, but not if I_n is of the form $z = x\ bop\ y$.

Definition 47. *For regular* $\mathtt{P}$*, solution* In/Out *and functional unit fu, the (undirected) conflict graph* $\mathcal{G}_{fu} = (\mathcal{V}_{fu}, \mathcal{E}_{fu})$ *consists of nodes* $x \in \mathbf{Var}_{\mathtt{P}}$ *which are forwardable to fu with respect to* In/Out *and edges* (x,y) *whenever there is an x-path* π *and a y-path* τ *and* π *contains* τ*.*
The register conflict graph $\mathcal{G}_{Reg} = (\mathcal{V}_{Reg}, \mathcal{E}_{Reg})$ *is given by the variables* $x \in \mathbf{Var}_{\mathtt{P}}$ *as nodes and conflict edges for all pairs* (x,y) *such that there is an instruction* $I \in \mathbf{Instrs}_{\mathtt{P}}$ *with* $x, y \in liveOut(I)$*.*

Example. Consider the three programs

$$
\begin{array}{rl}
\mathtt{N(1)} = & [1]\mathtt{x} = \mathtt{4} \\
 & [2]\mathtt{y} = \mathtt{7} \\
 & [3]\mathtt{z} = \mathtt{x} - \mathtt{y}
\end{array}
\qquad
\begin{array}{rl}
\mathtt{N(1)} = & [1]\mathtt{x} = \mathtt{4} \\
 & [2]\mathtt{y} = \mathtt{7} \\
 & [3]\mathtt{z} = \mathtt{y} - \mathtt{x}
\end{array}
\qquad
\begin{array}{rl}
\mathtt{N(1)} = & [1]\mathtt{x} = \mathtt{2} \\
 & [2]\mathtt{y} = \mathtt{5} \\
 & [3]\mathtt{z} = \mathtt{7} \\
 & [4]\mathtt{jmp}\ \mathtt{5} \\
\mathtt{N(5)} = & [5]\mathtt{x} = \mathtt{x} + \mathtt{z} \\
 & [6]\mathtt{y} = \mathtt{y} + \mathtt{1} \\
 & [7]\mathtt{z} = \mathtt{z} - \mathtt{1} \\
 & [8]\mathtt{if}\ \mathtt{z}\ \mathtt{9}\ \mathtt{5} \\
\mathtt{N(9)} = & [9]\mathtt{z} = \mathtt{x} - \mathtt{y}
\end{array}
\tag{7.11}
$$

In the program on the left, there is one $\mathtt{x}$-path $\pi = ([1],[2],[3])$ and one $\mathtt{y}$-path $\tau = ([2],[3])$ and both variables are forwardable to ALU. Although all instructions of τ occur in π, τ is not contained in π as the instruction $[3]$ does not have the form $z = \mathtt{y}\ bop\ \mathtt{x}$. Indeed, no conflict exists between x and y and they may share the same operand queue. The second program has the same x-paths and y-paths, but this time instruction $[3]$ is of the form $z = \mathtt{y}\ bop\ \mathtt{x}$. Consequently, a conflict exists between x and y. The conflict graph $\mathcal{G}_{\mathtt{ALU}}$ of variables forwardable to ALU contains an edge between x and y and a colouring will hence map them to different operand queues. This is necessary for functionally correct execution: suppose an allocation ρ was admitted with $\rho(\mathtt{x}) = \rho(\mathtt{y}) = q$. Then the translated program $[\![\mathtt{P}]\!]_\rho$ would read

$$[1]\mathtt{ldc}\ 4\ q\ [2]\mathtt{ldc}\ 7\ q\ [3]\mathtt{sub}\ q\ q\ \rho(\mathtt{z})$$

and the last instruction would assign -3 to $\rho(\mathtt{z})$ instead[1] of 3.

In the third program, x and y are forwardable to ALU, and z is not forwardable. There are three x-paths

$$\begin{aligned} \pi_1 &= ([1],[2],[3],[4],[5]) \\ \pi_2 &= ([5],[6],[7],[8],[9]) \\ \pi_1 &= ([5],[6],[7],[8],[5]) \end{aligned}$$

and three y-paths

$$\begin{aligned} \tau_1 &= ([2],[3],[4],[5],[6]) \\ \tau_2 &= ([6],[7],[8],[9]) \\ \tau_3 &= ([6],[7],[8],[5],[6]) \end{aligned}$$

No π_i contains any τ_j, and no τ_j contains any π_i, hence no conflict exists between x and y. In particular π_2 does not contain τ_2 as instruction $[9]$ is not of the form $z = \mathtt{y}\ bop\ \mathtt{x}$. For $\gamma(q) = \mathtt{ALU}$ the allocation $\rho(\mathtt{x}) = \rho(\mathtt{y}) = \mathtt{q}_1$, $\rho(\mathtt{z}) = \mathtt{r}_1$ yields the translation

$\mathtt{N}(1) =$	$[1]\mathtt{ldc}\ 2\ \mathtt{q}_1$	$\mathtt{N}(5) =$	$[5]\mathtt{add}\ \mathtt{q}_1\ \mathtt{r}_1\ \mathtt{q}_1$	$\mathtt{N}(9) =$	$[9]\mathtt{sub}\ \mathtt{q}_1\ \mathtt{q}_1\ \mathtt{r}_1$
	$[2]\mathtt{ldc}\ 5\ \mathtt{q}_1$		$[6]\mathtt{inc}\ \mathtt{q}_1\ \mathtt{q}_1$		
	$[3]\mathtt{ldc}\ 7\ \mathtt{r}_1$		$[7]\mathtt{dec}\ \mathtt{r}_1\ \mathtt{r}_1$		
	$[4]\mathtt{jmp}\ 5$		$[8]\mathtt{if}\ \mathtt{r}_1\ 9\ 5$		

[1]In fact, for commutative *bop* the second condition in Definition 46 may be dropped.

with principal typings $\mathtt{N(1)} : \mathbf{1} \multimap \mathtt{q}_1 \otimes \mathtt{q}_1 \otimes \mathtt{r}_1$, $\mathtt{N(5)} : \mathtt{q}_1 \otimes \mathtt{r}_1 \multimap \mathtt{q}_1 \otimes \mathtt{r}_1$ and $\mathtt{N(9)} : \mathtt{q}_1 \otimes \mathtt{q}_1 \multimap \mathtt{r}_1$. Weakening the typing for $\mathtt{N(5)}$ by $\mathtt{q}_1$ and the typing for $\mathtt{N(9)}$ by $\mathtt{r}_1$ yields

$$
\begin{aligned}
\mathtt{N(1)} &: \mathbf{1} \multimap \mathtt{q}_1 \otimes \mathtt{q}_1 \otimes \mathtt{r}_1 \\
\mathtt{N(5)} &: \mathtt{q}_1 \otimes \mathtt{q}_1 \otimes \mathtt{r}_1 \multimap \mathtt{q}_1 \otimes \mathtt{q}_1 \otimes \mathtt{r}_1 \\
\mathtt{N(9)} &: \mathtt{q}_1 \otimes \mathtt{q}_1 \otimes \mathtt{r}_1 \multimap \mathtt{r}_1
\end{aligned}
$$

and makes the program type-correct. It is also functionally correct due to $\mathtt{N(5)}$'s invariant: the head value of $\mathtt{q}_1$ is the updated value of $\mathtt{x}$, the second element of $\mathtt{q}_1$ contains $\mathtt{y}$ and $\mathtt{r}_1$ contains the value of $\mathtt{z}$. Thus, instruction [9] correctly subtracts $\mathtt{y}$ from $\mathtt{x}$. $\diamond$

Allocating queue and registers comprises node-colouring the conflict graphs.

Definition 48. *For regular* $\mathtt{P}$ *and solution In/Out, allocation function* $\rho : \mathbf{Var}_{\mathtt{P}} \to OP$ *is called* an allocation for $\mathtt{P}$ *if*

- $\rho(x) = q$ *implies* $x \in \mathcal{V}_{\gamma(q)}$
- $\rho(x) = \rho(y) = r$ *implies* $(x,y) \notin \mathcal{E}_{Reg}$
- $\rho(x) = \rho(y) = q$ *implies* $(x,y) \notin \mathcal{E}_{\gamma(q)}$

Lemma 16. *If* ρ *is an allocation for* $\mathtt{P}$ *with respect to solution In/Out then* ρ *is compatible with In/Out.*

Proof. Let $\rho(x) = q$ and suppose x is not forwardable to $\gamma(q)$. Then $x \notin \mathcal{V}_{\gamma(q)}$ so $\rho(x) \neq q$. Contradiction. $\square$

By Theorem 8, translating $\mathtt{P}$ with respect to an allocation for $\mathtt{P}$ consequently leads to an ALEF program which is well-typed. Variables which are forwardable to several functional units can be inserted into any graph, each choice yielding a different instantiation of the polymorphic instruction $\mathtt{id}^{fu}$.

7.6 Functional correctness

We finally prove functional correctness of the translation. This is achieved by relating the ALEF execution of $[\![\mathtt{P}]\!]_\rho$ to the execution of P in the non-standard dynamic semantics $\Rightarrow_\alpha$. We first define the association function α_ρ which arises from allocation ρ according to the read/write capabilities motivated earlier: register-mapped variables are associated to the lattice element $\top$ and queue-mapped ones to 1. For an allocation arising from a dataflow solution, the resulting α_ρ fulfils the condition of Theorem 6 which guarantees that the functional correctness also holds with respect to the standard semantics $\rightarrow$ of IL.

We then establish a satisfaction relation between IL states S and ALEF states C which expresses the fact that values in IL-component σ occur as entries in Q or R in C, depending on their entry in the association function α_ρ.

The satisfaction relation is shown to be preserved by program execution, including the order of values in operand queues. In particular, this result implies that loops fulfil an invariant similar to the one observed in the rightmost program in (7.11).

Again, we consider P to be a fixed regular program and In/Out a fixed solution for P. Furthermore, we let ρ be an allocation for P with respect to In/Out according to Definition 48.

Definition 49. *The association function* α_ρ *is given by*

$$\alpha_\rho(I,x) = \begin{cases} 1 & \textit{if } x \in \textit{defs}(I) \textit{ and } \rho(x) = q \\ \top & \textit{if } x \in \textit{defs}(I) \textit{ and } \rho(x) = r \end{cases}$$

The following Lemma ensures that $\Rightarrow_{\alpha_\rho}$ is faithful to the standard execution $\rightarrow$ of P, based on Theorem 6.

Lemma 17. $\alpha_\rho \sqsupseteq \alpha_{Out}$.

Proof. Let $x \in \textit{defs}(I)$.

If $\rho(x) = q$ then x is forwardable to $\gamma(q)$ as ρ is compatible with In/Out, so $x \in \textit{defs}(I)$ implies $Out(I)(x) = 1$, so $\alpha_\rho(I,x) = 1 = \alpha_{Out}(I,x)$.

If $\rho(x) = r$ then $\alpha_\rho(I,x) = \top$, so $\alpha_\rho(I,x) \sqsupseteq \alpha_{Out}(I,x)$ is trivially fulfilled. $\square$

In particular, Lemma 17 explains why Theorem 6 was stated for arbitrary $\alpha \sqsupseteq \alpha_{Out}$, although the proof used only properties of α_{Out}. Consider an allocation which maps some variables to registers, regardless of their forwardability. Such an allocation is admitted by all definitions we gave so far, and should certainly be admitted for translation into ALEF, too. However, its association function does not necessarily correspond to any solution to the dataflow equations. As an extreme example consider the trivial allocation ρ which maps *all* variables to registers. Its association function is $\alpha_\rho(I,x) = \top$ whenever $x \in \mathit{defs}(I)$ holds, but $\lambda x.\top$ is in general not a fixed point for equations (6.13). Thus, stating Theorem 6 only for associations arising from solutions would have ruled out allocation ρ.

The stated version of Theorem 6 leaves the decision whether a forwardable variable is indeed forwarded to the allocation phase rather than to the dataflow analysis. This is the right choice as many compilers aim to perform program analysis independently from architectural limitations such as the availability of a certain number of operand queues or registers.

The satisfaction relation between IL-configurations S and ALEF-configurations C is motivated as follows. Firstly, $S = (\Sigma, n, B)$ may contain entries $\Sigma(x) = (\top, a_1)$ and $\Sigma(y) = (\top, a_2)$ for variables $x, y \in \mathbf{Var}_{\mathsf{P}}$ with $\rho(x) = r = \rho(y)$. On the other hand, the register component R of an ALEF configuration $\mathsf{C} = (\mathsf{Q}, \mathsf{R}, \mathsf{M}, \widetilde{\kappa})$ maps r to a single value. Our definition relating S to C therefore requires that such variables x and y cannot both be live-in at $\mathit{fst}(B)$, and stipulates that the entry in Σ for the live variable is the one found in C.

Second, the order of values in operand queues is not specified in Σ. Our definition requires for each $q \in Q$ that each entry a in $\mathsf{Q}(q)$ corresponds precisely to one variable x with $\Sigma(x) = (1, a)$ and $\rho(x) = q$. Furthermore, the order of values $a_1 \ldots a_n$ in $\mathsf{Q}(q)$ must correspond to the order of the last n instructions $I_1, \ldots, I_n$ in the IL-execution history leading to S, where each I_i assigns value a_i to a variable x with $\rho(x) = q$.

Definition 50. *Let ρ be an allocation for regular P, $\mathsf{S} = (\Sigma, \_, \_)$ and $\mathsf{C} = (\mathsf{Q}, \mathsf{R}, \mathsf{M}, \widetilde{\kappa})$. We write $\mathsf{C} \models_{\alpha_\rho} S$ if*

1. *there are $\mathsf{S}_1, \ldots, \mathsf{S}_n$ such that $\mathsf{S}_n = \mathsf{S}$ and $\mathsf{S}_i \Rightarrow_{\alpha_\rho} \mathsf{S}_{i+1}$ for $0 \leq i < n$, where S_0 is the initial configuration.. We write $\mathsf{S}_i = (\Sigma_i, m_i, B_i)$.*

2. *if* $x \in dom\,\Sigma$, $\Sigma(x) = (\top, a)$ *and* $\rho(x) = r$ *then* $r \in dom\,\mathsf{R}$.

3. *if* $r \in dom\,\mathsf{R}$ *and* $|B_n| > 0$ *then there is at most one* $x \in liveIn(fst(B_n))$ *with* $\rho(x) = r$, *and any such* x *fulfils* $\Sigma(x) = (\top, \mathsf{R}(r))$.

4. *for each* $q \in Q$, $|\mathsf{Q}(q)| = |\{x \in dom\,\Sigma \,|\, \rho(x) = q, \exists b.\ \Sigma(x) = (1, b)\}|$ *holds.*

5. *for each* $q \in Q$ *and* $\mathsf{Q}(q) = a_1 \dots a_m$ *there are* $x_1, \dots, x_m \in \mathbf{Var}_\mathsf{P}$ *and* $I_1, \dots, I_m \in \mathbf{Instrs}_\mathsf{P}$ *such that for all* $1 \leq i \leq m$

   - $I_i = fst(B_{j_i})$ *for some* $0 \leq j_1 \leq \dots \leq j_m < n$
   - $x_i \in defs(I_i)$
   - $\rho(x_i) = q$
   - $\Sigma(x_i) = (1, a_i)$
   - $x_i \notin defs(J) \cup uses(J)$ *for* $J = fst(B_k)$ *and all* $j_i < k < n$

   and $\cup_{i=1}^m x_i = \{x \in dom\,\Sigma \,|\, \rho(x) = q, \exists b.\ \Sigma(x) = (1, b)\}$.

In particular, the length of $\mathsf{Q}(q)$ equals the number of variables mapped to q which have been assigned to most recently but not yet been read, with assigning instructions $fst(B_{j_i})$.

Before proving that the satisfaction relation is preserved by execution, we collect two facts.

Lemma 18. *If* $\mathsf{S}_0 \Rightarrow^*_{\alpha_\rho} \mathsf{S}$, $\mathsf{S} = (\Sigma, \_, B)$, $|B| > 0$ *and* $x \in dom\,\Sigma$, *then*

- $\rho(x) = r$ *implies* $\Sigma(x) = (\top, a)$ *for some* a.
- $\rho(x) = q$ *implies* $\Sigma(x) = (l, a)$ *for some* $l \in \{0, 1\}$.

Proof. Since $\mathsf{S}_0 = (\Sigma_0, \_, B_0)$ is initial, $x \notin dom\,\Sigma_0$ holds, so for $x \in dom\,\Sigma$ and $\Sigma(x) = (l, a)$ we must have $l = \alpha_\rho \underbrace{\ominus 1 \dots \ominus 1}_{n}$ for some n. By the definitions of α_ρ and $\ominus$, $\rho(x) = r$ implies $\alpha_\rho = \top$, hence $l = \top$.

For $\rho(x) = q$ we obtain $\alpha_\rho = 1$, hence $l \in \{\perp, 0, 1\}$. On the other hand, the definitions of compatibility and forwardability imply $Out(fst(B))(x) \in \{0, 1\}$, hence $\perp \neq In(fst(B))(x)$. From $In(fst(B)) \sqsubseteq \mathsf{S}$ we thus obtain $l \in \{0, 1\}$. $\square$

Lemma 19. *If ρ is an allocation for* $\mathtt{P}$, $\rho(x) = q$ *and* π *is a path in* $\mathtt{P}$ *with* $x \in \mathit{defs}(\pi(1))$ *and* $x \notin \mathit{defs}(\pi(i)) \cup \mathit{uses}(\pi(i))$ *for all* $i > 1$*, then* π *is the prefix of an* x*-path in* $\mathtt{P}$.

Proof. If π is infinite, then the claim holds trivially as π is an x-path.
For finite π, and n its length, suppose $\mathtt{succ}(\pi(n)) = \emptyset$ holds. Then equations (6.13) imply $\mathit{Out}(\pi(n))(x) = 0$, and for all $1 < i \leq n$, $\mathit{uses}(\pi(i))(x) = 0$ holds due to $x \notin \mathit{uses}(\pi(i))$, hence

$$\begin{aligned} 0 &= \mathit{Out}(\pi(n))(x) = \mathit{In}(\pi(n))(x) \sqsubseteq \mathit{Out}(\pi(n-1))(x) = \mathit{In}(\pi(n-1))(x) \sqsubseteq \\ \dots &\sqsubseteq \mathit{Out}(\pi(1))(x) = 1 \end{aligned}$$

where $\mathit{Out}(\pi(1))(x) = 1$ follows from Lemmas 16 and 15. This contradicts the structure of the lattice $\mathcal{L}$, as $0 \not\sqsubseteq 1$ holds. Hence $\mathtt{succ}(\pi(n)) \neq \emptyset$ follows.
Similarly, $J \in \mathtt{succ}(\pi(n))$ with $x \in \mathit{defs}(J)$ but $x \notin \mathit{uses}(J)$ leads to the same contradiction $0 \sqsubseteq \mathit{Out}(\pi(n))(x) \sqsubseteq \dots \sqsubseteq \mathit{Out}(\pi(1))(x) = 1$ since $0 = \mathit{In}(J)(x) \sqsubseteq \mathit{Out}(\pi(n))(x)$ holds. Consequently, $x \in \mathit{defs}(J)$ implies $x \in \mathit{uses}(J)$, and the path resulting from appending any such J to π is an x-path whose existence we had to show.
If there is no $J \in \mathtt{succ}(\pi(n))$ with $x \in \mathit{defs}(J)$, then we choose an arbitrary J from $\mathtt{succ}(\pi(n)) \neq \emptyset$ and extend π by J. The resulting path is of the form required in the claim, and is of length $n+1$.
By repeating this process, we either obtain a path of which π is a prefix, or we extend π to an infinite path for which the claim holds trivially. $\square$

The proof of functional correctness shows that initial configurations S_0 and C_0 are always related, and that each $\Rightarrow_{\alpha_\rho}$-step corresponds to an ALEF step, preserving the satisfaction relationship between configurations. We first state the result for assignments.

Proposition 39. *Let* $\mathsf{C}_0 = ([\,],[\,],\mathsf{M},\widetilde{\kappa})$, $\mathsf{C} = (\mathsf{Q},\mathsf{R},\mathsf{M},\widetilde{\kappa})$, $\mathsf{S} = (\Sigma, m, B)$ *and* $|B| > 0$. *Then*

- $\mathsf{C}_0 \models_{\alpha_\rho} \mathsf{S}_0$
- *If* $\mathit{fst}(B)$ *is an assignment ass,* $\mathsf{C} \models_{\alpha_\rho} \mathsf{S}$ *and* $\mathsf{S} \Rightarrow_{\alpha_\rho} \mathsf{T}$ *then there is a unique* D *such that* $\mathsf{C} \xrightarrow{[\![ass]\!]_\rho} \mathsf{D}$. *Furthermore,* $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$ *and* $\mathsf{D} = (\_,\_,\mathsf{M},\widetilde{\kappa})$ *hold.*

Proof. For the first claim, observe that the definition of S_0 implies $\Sigma_0 = []$, hence for $n = 0$ all points in Definition 50 are trivially fulfilled.

For the second item, we first note that for $\mathsf{C} \models_{\alpha_\rho} \mathsf{S}$ and $\mathsf{S} \Rightarrow_{\alpha_\rho} \mathsf{T}$ with $\mathsf{S} = (\Sigma, m,\ ass\vec{I})$, the definition of $\Rightarrow_{\alpha_\rho}$ implies $\mathsf{T} = (\Sigma_2, m, \vec{I})$ for the unique

$$\Sigma_2 = \Sigma_1[x \mapsto (\alpha_\rho(ass, x), a)]$$

where *ass* is of the form $x = e$ and $\Sigma, e \Downarrow \Sigma_1, a$ and $\Sigma_1 \downarrow x$ hold. The proof proceeds by case distinction on the structure of *ass*, and we treat one representative case.

Case $'[\_]x = x'$**.** The definition of $\Sigma, x \Downarrow \Sigma_1, a$ implies $\Sigma(x) = (l, a)$ for some $l \sqsupseteq 1$ with $\Sigma_1 = \Sigma[x \mapsto (l \ominus 1, a)]$, and $\Sigma_1 \downarrow x$ implies $l \ominus 1 \sqsupseteq 0$. There are two cases, depending on the value of $\rho(x)$.

Case $\rho(x) = r$**.** The definition of $[\![\_]\!]_\rho$ yields $[\![ass]\!]_\rho = \mathtt{id}^{\mathtt{ALU}}\, r\, r$, and the assumption $\mathsf{C} \models_{\alpha_\rho} S$ implies that there is an n with

(1) $\mathsf{S}_0 \Rightarrow_{\alpha_\rho}^{n} \mathsf{S}$, and in particular $ass = fst(B_n) = fst(B)$.

(2) if $y \in dom\,\Sigma$, $\Sigma(y) = (\top, b)$ and $\rho(y) = r'$ then $r' \in dom\,\mathsf{R}$.

(3) for $r' \in dom\,\mathsf{R}$ there is at most one $y \in liveIn(ass)$ with $\rho(y) = r'$, and any such y fulfils $\Sigma(y) = (\top, \mathsf{R}(r'))$.

(4) for each $q \in Q$, $|\mathsf{Q}(q)| = |\{y \in dom\,\Sigma \,|\, \rho(y) = q, \exists b.\ \Sigma(y) = (1, b)\}|$ holds.

(5) for each $q \in Q$ and $\mathsf{Q}(q) = a_1 \ldots a_m$ there are $x_1, \ldots, x_m \in \mathbf{Var}_{\mathsf{P}}$ and $I_1, \ldots, I_m \in \mathbf{Instrs}_{\mathsf{P}}$ such that for all $1 \leq i \leq m$

- $I_i = fst(B_{j_i})$ for some $0 \leq j_1 \leq \ldots \leq j_m < n$
- $x_i \in defs(I_i)$
- $\rho(x_i) = q$
- $\Sigma(x_i) = (1, a_i)$
- $x_i \notin defs(J) \cup uses(J)$ for $J = fst(B_k)$ and all $j_i < k < n$

and $\cup_{i=1}^{m} x_i = \{y \in dom\,\Sigma \,|\, \rho(y) = q, \exists b.\ \Sigma(y) = (1, b)\}$.

Step 1. We first show that there is a D with $\mathsf{C} \xrightarrow{[\![ass]\!]_\rho} \mathsf{D}$.

From $\mathsf{S}_0 \Rightarrow^n_{\alpha_\rho} \mathsf{S}$ (fact (1)) and $\mathsf{S} \Rightarrow_{\alpha_\rho} \mathsf{T}$ we obtain $\mathsf{S}_0 \Rightarrow^{n+1}_{\alpha_\rho} \mathsf{T}$.

We noticed $\Sigma(x) = (l,a)$ above, so Lemma 18 implies $l = \top$, hence $\Sigma_1 = \Sigma$ and fact (2) imply $r \in dom\, \mathsf{R}$. Furthermore, $x \in uses(ass)$ holds, hence $x \in liveIn(ass)$, and fact (3) yields $\Sigma(x) = (\top, \mathsf{R}(r))$. From $\Sigma(x) = (l,a)$ and uniqueness of the map R we thus obtain $\mathsf{R}(r) = a$.

Consequently, the definition of ALEF execution yields $\mathsf{C} \xrightarrow{[\![ass]\!]_\rho} \mathsf{D}$ for the unique $\mathsf{D} = \mathsf{C}$.

Step 2. Second, we show that for $\mathsf{D} = \mathsf{C}$, the relation $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$ holds.

1. $\mathsf{S}_0 \Rightarrow^{n+1}_{\alpha_\rho} \mathsf{T}$ was shown above
2. For $y \in dom\, \Sigma_2$, $\Sigma_2(y) = (\top, b)$ and $\rho(y) = r'$ there are two cases.
   - If $x = y$ then $r' = \rho(y) = \rho(x) = r$, hence $r' \in dom\, \mathsf{R}$
   - If $x \neq y$ then $y \in dom\, \Sigma_2$ implies $y \in dom\, \Sigma_1 = dom\, \Sigma$ with $\Sigma(y) = \Sigma_2(y) = (\top, b)$, so fact (2) implies $r' \in dom\, \mathsf{R}$.

   In both cases, $r' \in dom\, \mathsf{R}$ holds.
3. For $r' \in dom\, \mathsf{R}$ and $|\vec{I}| > 0$, suppose that $\{y,z\} \subseteq liveIn(fst(\vec{I}))$ holds for some $y \neq z$ with $\rho(y) = r' = \rho(z)$. From $\mathtt{succ}(ass) = \{fst(\vec{I})\}$ and equations (6.9) we obtain $\{y,z\} \subseteq liveOut(ass)$. Consequently, y and z are adjacent in $\mathcal{G}_{Reg}$ so $\rho(y) \neq \rho(z)$ as ρ is an allocation for $\mathtt{P}$. This contradicts $\rho(y) = r' = \rho(z)$, hence there is at most one $y \in liveIn(fst(\vec{I}))$ with $\rho(y) = r'$.

   It remains to show that any such y fulfils $\Sigma_2(y) = (\top, \mathsf{R}(r'))$. There are two cases.
   - if $y = x$, then $r' = r$ follows, and the definition of α_ρ yields
   $$\Sigma_2(y) = \Sigma_2(x) = (\alpha_\rho(ass, x), a) = (\top, \mathsf{R}(r)).$$
   - if $y \neq x$ then equations (6.9) imply
   $$y \in liveIn(fst(\vec{I})) \subseteq liveOut(ass),$$
   hence $y \in liveIn(ass)$ holds due to $y \notin \{x\} = defs(ass)$. By fact (3), $\Sigma_2(y) = \Sigma_1(y) = \Sigma(y) = (\top, \mathsf{R}(r'))$ follows.

4. For $q \in Q$, we have to show

$$|Q(q)| = |\{y \in dom\,\Sigma_2 \,|\, \rho(y) = q, \exists b.\ \Sigma_2(y) = (1,b)\}|.$$

Abbreviating $\{y \in dom\,\Sigma_2 \,|\, \rho(y) = q, \exists b.\ \Sigma_2(y) = (1,b)\}$ by M_q, any $y \in M_q$ fulfils $y \neq x$ as $\rho(y) = q \neq r = \rho(x)$ holds. The definition of Σ_2 thus implies

$$\begin{aligned}
M_q &= \{y \in dom\,\Sigma_2 \,|\, \rho(y) = q, \exists b.\ \Sigma_2(y) = (1,b)\} \\
&= \{y \in dom\,\Sigma_2 \,|\, y \neq x, \rho(y) = q, \exists b.\ \Sigma_2(y) = (1,b)\} \\
&= \{y \in dom\,\Sigma_1 \,|\, y \neq x, \rho(y) = q, \exists b.\ \Sigma_1(y) = (1,b)\} \\
&= \{y \in dom\,\Sigma \,|\, y \neq x, \rho(y) = q, \exists b.\ \Sigma(y) = (1,b)\} \\
&= \{y \in dom\,\Sigma \,|\, \rho(y) = q, \exists b.\ \Sigma(y) = (1,b)\}
\end{aligned}$$

so fact (4) yields $|Q(q)| = |M_q|$.

5. For each $q \in Q$ and $\mathsf{Q}(q) = a_1 \ldots a_m$ we have to show that there are $z_1, \ldots, z_m \in \mathbf{Var}_{\mathsf{P}}$ and $J_1, \ldots, J_m \in \mathbf{Instrs}_{\mathsf{P}}$ such that for all $1 \leq i \leq m$
   - $J_i = fst(B_{l_i})$ for some $0 \leq l_1 \leq \ldots \leq l_m < n+1$
   - $z_i \in defs(J_i)$
   - $\rho(z_i) = q$
   - $\Sigma(z_i) = (1, a_i)$
   - $z_i \notin defs(K) \cup uses(K)$ for $K = fst(B_h)$ and all $l_i < h < n+1$

   and $\cup_{i=1}^{m} z_i = M_q$ for the M_q as defined above.
   Defining $z_i = x_i$ and $J_i = I_i$ from fact (5) for $1 \leq i \leq m$ yields
   - $J_i = I_i = fst(B_{l_i})$ for $j_i = l_i$, hence $1 \leq l_1 \leq \ldots \leq l_m < n < n+1$
   - $z_i = x_i \in defs(I_i) = defs(J_i)$
   - $\rho(z_i) = \rho(x_i) = q$
   - each z_i fulfils $z_i \neq x$ due to $\rho(z_i) = q \neq r = \rho(x)$, hence $\Sigma_2(z_i) = \Sigma_1(z_i) = \Sigma(z_i) = \Sigma(x_i) = (1, a_i)$.
   - for $l_i < h < n$ we obtain $z_i = x_i \notin defs(K) \cup uses(K)$ for $K = fst(B_h)$

by fact (5), and for $n \leq h < n+1$ we have $h = n$, so

$$\begin{aligned} z_i = x_i \notin \{x\} &= \mathit{defs}(\mathit{ass}) \cup \mathit{uses}(\mathit{ass}) \\ &= \mathit{defs}(\mathit{fst}(B_h)) \cup \mathit{uses}(\mathit{fst}(B_h)) \end{aligned}$$

and $\cup_{i=1}^m z_i = \cup_{i=1}^m x_i = M_q$ using the above equality

$$\{y \in \mathit{dom}\, \Sigma \,|\, \rho(y) = q, \exists b.\ \Sigma(y) = (1,b)\} = M_q.$$

Case $\rho(x) = q$**.** The definition of $[\![\_]\!]_\rho$ yields $[\![\mathit{ass}]\!]_\rho = \mathtt{id}^{\gamma(q)}\, q\, q$. Lemma 16 implies that ρ is compatible with the solution In/Out of P, so x is forwardable to $\gamma(q)$. The assumption $\mathsf{C} \models_{\alpha_\rho} \mathsf{S}$ implies that there is an n with

(1) $\mathsf{S}_0 \Rightarrow^n_{\alpha_\rho} \mathsf{S}$, and in particular $\mathit{ass} = \mathit{fst}(B_n)$.

(2) if $y \in \mathit{dom}\, \Sigma$, $\Sigma(y) = (\top, b)$ and $\rho(y) = r$ then $r \in \mathit{dom}\, \mathsf{R}$.

(3) for $r \in \mathit{dom}\, \mathsf{R}$ there is at most one $y \in \mathit{liveIn}(\mathit{ass})$ with $\rho(y) = r$, and any such y fulfils $\Sigma(y) = (\top, \mathsf{R}(r))$.

(4) for each $q' \in Q$, $|\mathsf{Q}(q')| = |\{y \in \mathit{dom}\, \Sigma \,|\, \rho(y) = q', \exists b.\ \Sigma(y) = (1,b)\}|$ holds.

(5) for each $q' \in Q$ and $\mathsf{Q}(q') = a_1 \ldots a_m$ there are $x_1, \ldots, x_m \in \mathbf{Var}_\mathsf{P}$ and $I_1, \ldots, I_m \in \mathbf{Instrs}_\mathsf{P}$ such that for all $1 \leq i \leq m$

- $I_i = \mathit{fst}(B_{j_i})$ for some $0 \leq j_1 \leq \ldots \leq j_m < n$
- $x_i \in \mathit{defs}(I_i)$
- $\rho(x_i) = q'$
- $\Sigma(x_i) = (1, a_i)$
- $x_i \notin \mathit{defs}(J) \cup \mathit{uses}(J)$ for $J = \mathit{fst}(B_k)$ and all $j_i < k < n$

and $\cup_{i=1}^m x_i = \{y \in \mathit{dom}\, \Sigma \,|\, \rho(y) = q', \exists b.\ \Sigma(y) = (1,b)\}$.

Step 1. We first show that there is a D with $\mathsf{C} \xrightarrow{[\![\mathit{ass}]\!]_\rho} \mathsf{D}$.

From fact (1) and the assumption $\mathsf{S} \Rightarrow_{\alpha_\rho} \mathsf{T}$ we obtain $\mathsf{S}_0 \Rightarrow^{n+1}_{\alpha_\rho} \mathsf{T}$. We noticed $\Sigma(x) = (l, a)$ for some $l \sqsupseteq 1$ above, and Lemma 18 now implies $l = 1$ due to $\rho(x) = q$, hence $\Sigma(x) = (1, a)$ and $\Sigma_1 = \Sigma[x \mapsto (0, a)]$. Therefore, $x \in M_q$ where

$$M_q = \{y \in \mathit{dom}\, \Sigma \,|\, \rho(y) = q, \exists b. \Sigma(y) = (1,b)\}.$$

By fact (4), $\mathsf{Q}(q)$ has thus at least one element, i.e. there is an $m \geq 1$ with $\mathsf{Q}(q) = a_1 \dots a_m$. By fact (5), there are $x_1, \dots, x_m$ and $I_1, \dots, I_m$ such that $M_q = \cup_{i=1}^{m} x_i$ and

- $I_i = fst(B_{j_i})$ for some $0 \leq j_1 \leq \dots \leq j_m < n$
- $x_i \in defs(I_i)$
- $\rho(x_i) = q$
- $\Sigma(x_i) = (1, a_i)$
- $x_i \notin defs(J) \cup uses(J)$ for $J = fst(B_k)$ and all $j_i < k < n$.

$x \in M_q$ implies that $x = x_i$ holds for some $1 \leq i \leq m$, and we now show $x = x_1$, i.e. $a = a_1$.

Suppose $x \neq x_1$. The sequence of instructions

$$fst(B_{j_1}), fst(B_{j_1+1}), \dots, fst(B_{j_2}), \dots, fst(B_{j_m}), \dots, ass = fst(B_n)$$

is a path π_0 in $\mathtt{P}$ since $0 \leq j_1 \leq \dots \leq j_m < n$ and IL execution $\Rightarrow_{\alpha_\rho}$ respects the relation `succ` of $\mathtt{P}$ by Proposition 31 and Lemma 8. Furthermore, fact (5) yields $x_1 \in defs(I_1) = defs(B_{j_1})$ and $x_1 \notin defs(K) \cup uses(K)$ for all $K \neq I_1$ on π_0. By Lemma 19, π_0 is hence the prefix of an x_1-path π.

Since $x = x_i$ holds for some $1 \leq i < m$, $x_i \in defs(I_i)$ and $x \in uses(ass)$,

$$\tau = I_i, \dots, I_m, \dots, ass$$

is an x-path – fact (5) guarantees that ass is the first use of x_i following I_i). Thus π contains τ. Contradiction: $\rho(x) = q = \rho(x_1)$ holds by definition of x_1, but the nodes x and x_1 are adjacent in $\mathcal{G}_{\gamma(q)}$ as π contains τ, so $\rho(x) \neq \rho(x_1)$ as ρ is an allocation for program $\mathtt{P}$. Therefore, $x = x_1$ holds.

For the unique $\mathsf{D} = (\mathsf{P}, \mathsf{R}, \mathsf{M}, \widetilde{\kappa})$ with $\mathsf{P} = \mathsf{Q}[q \mapsto a_2 \dots a_m\, a]$ we thus have $\mathsf{C} \xrightarrow{[\![ass]\!]_\rho} \mathsf{D}$.

Step 2. Second, we show that $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$ holds.

1. We have $\mathsf{S}_0 \Rightarrow_{\alpha_\rho}^{n+1} \mathsf{T}$ as noticed above.
2. For $y \in dom\, \Sigma_2$, $\Sigma_2(y) = (\top, b)$ and $\rho(y) = r$, $y \neq x$ follows due to $\rho(x) = q$. Hence $y \in dom\, \Sigma$ with $(\top, b) = \Sigma_2(y) = \Sigma_1(y) = \Sigma(x)$, so fact (2) implies $r \in dom\, \mathsf{R}$.

3. For $r \in dom\, \mathsf{R}$, $|\vec{I}| > 0$ and variables $y \neq z$ with $\rho(y) = r = \rho(z)$ and $\{y,z\} \subseteq liveIn(fst(\vec{I}))$, we have $\{y,z\} \subseteq liveOut(ass)$ by equations (6.9) and $\mathtt{succ}(ass) = \{fst(\vec{I})\}$. Consequently, y and z are adjacent in $\mathcal{G}_{Reg}$ so $\rho(y) \neq \rho(z)$.
   For $y \in liveIn(fst(\vec{I}))$ with $\rho(y) = r$, $y \neq x$ holds due to $\rho(x) = q$, and $y \in liveIn(fst(\vec{I}))$ implies $y \in liveOut(ass) \subseteq liveIn(ass)$ by equations (6.9) as $y \notin \{x\} = defs(ass)$. Hence, $\Sigma_2(y) = \Sigma(y) = (\top, \mathsf{R}(r))$ by fact (3).
4. For $q' \in Q$ there are two cases.
   - If $q \neq q'$ then any $y \in dom\,\Sigma$ with $\rho(y) = q'$ and $\Sigma(y) = (1,b)$ for some b fulfils $x \neq y$ due to $\rho(x) = q$, hence

$$
\begin{array}{rcl}
|\mathsf{P}(q')| & = & |\mathsf{Q}(q')| \\
& = & |\{y \in dom\,\Sigma \,|\, \rho(y) = q', \exists b.\Sigma(y) = (1,b)\}| \\
& = & |\{y \in dom\,\Sigma \,|\, y \neq x, \rho(y) = q', \exists b.\Sigma(y) = (1,b)\}| \\
& = & |\{y \in dom\,\Sigma_1 \,|\, y \neq x, \rho(y) = q', \exists b.\Sigma_1(y) = (1,b)\}| \\
& = & |\{y \in dom\,\Sigma_2 \,|\, y \neq x, \rho(y) = q', \exists b.\Sigma_2(y) = (1,b)\}| \\
& = & |\{y \in dom\,\Sigma_2 \,|\, \rho(y) = q', \exists b.\Sigma_2(y) = (1,b)\}|
\end{array}
$$

   holds by fact (4) and the definitions of P and Σ_2.
   - If $q = q'$ then fact (4), $\rho(x) = q$ and $\Sigma(x) = (1,a)$ imply

$$
\begin{array}{rcl}
|\mathsf{P}(q')| & = & |\mathsf{Q}(q)| \\
& = & |\{y \in dom\,\Sigma \,|\, \rho(y) = q, \exists b.\Sigma(y) = (1,b)\}| \\
& = & |\{x\} \cup \\
& & \{y \in dom\,\Sigma \,|\, x \neq y, \rho(y) = q, \exists b.\Sigma(y) = (1,b)\}| \\
& = & |\{x\} \cup \\
& & \{y \in dom\,\Sigma_1 \,|\, x \neq y, \rho(y) = q, \exists b.\Sigma_1(y) = (1,b)\}| \\
& = & |\{x\} \cup \\
& & \{y \in dom\,\Sigma_2 \,|\, x \neq y, \rho(y) = q, \exists b.\Sigma_2(y) = (1,b)\}| \\
& = & |\{y \in dom\,\Sigma_2 \,|\, \rho(y) = q, \exists b.\Sigma_2(y) = (1,b)\}|
\end{array}
$$

where the last step follows from $\Sigma_2(x) = (\alpha_\rho(ass,x),a)$ where $\alpha_\rho(ass,x) = 1$ holds by Definition 49 due to $x \in defs(ass)$ and $\rho(x) = q$.

5. for $q' \in Q$ and $\mathsf{P}(q') = b_1 \dots b_\mu$ we have to show that there are $z_1, \dots, z_\mu$ and $J_1, \dots, J_\mu$ such that $\cup_{i=1}^{\mu} z_i = \{y \in dom\ \Sigma_2 \mid \rho(y) = q', \exists b. \Sigma_2(y) = (1,b)\}$ and for all $1 \leq i \leq \mu$ the following properties hold.
   - $J_i = fst(B_{l_i})$ for some $0 \leq l_1 \leq \dots \leq l_\mu < n+1$
   - $z_i \in defs(J_i)$
   - $\rho(z_i) = q'$
   - $\Sigma_2(z_i) = (1,b_i)$
   - $z_i \notin defs(K) \cup uses(K)$ for $K = fst(B_h)$ and all $l_i < h < n+1$

   Again, there are two cases.
   - If $q \neq q'$ then we take (for all i) $z_i = x_i$ and $J_i = I_i$ from fact (5), and obtain
     - $J_i = I_i = fst(B_{j_i}) = fst(B_{l_i})$ for $j_i = l_i$, hence $0 \leq l_1 \leq \dots l_\mu < n < n+1$
     - $z_i = x_i \in defs(I_i) = defs(J_i)$
     - $\rho(z_i) = \rho(x_i) = q'$
     - each z_i fulfils $z_i \neq x$ due to $\rho(z_i) = q' \neq q = \rho(x)$, hence $\Sigma_2(z_i) = \Sigma_1(z_i) = \Sigma(z_i) = \Sigma(x_i) = (1,b_i)$
     - $z_i = x_i \notin defs(K) \cup uses(K)$ for $K = fst(B_h)$ and all $l_i < h < n$ by fact (5), and $z_i = x_i \notin defs(fst(B_n)) \cup uses(fst(B_n)) = defs(ass) \cup uses(ass) = \{x\}$ holds due to $z_i \neq x$.
   - If $q = q'$ then $\mu = m$ and

$$b_i = \begin{cases} a_{i+1} & \text{for } i < m \\ a_1 & \text{for } i = m \end{cases}$$

     follow, and we define

$$(z_i, J_i) = \begin{cases} (x_{i+1}, I_{i+1}) & \text{for } i < m \\ (x, ass) & \text{for } i = m \end{cases}$$

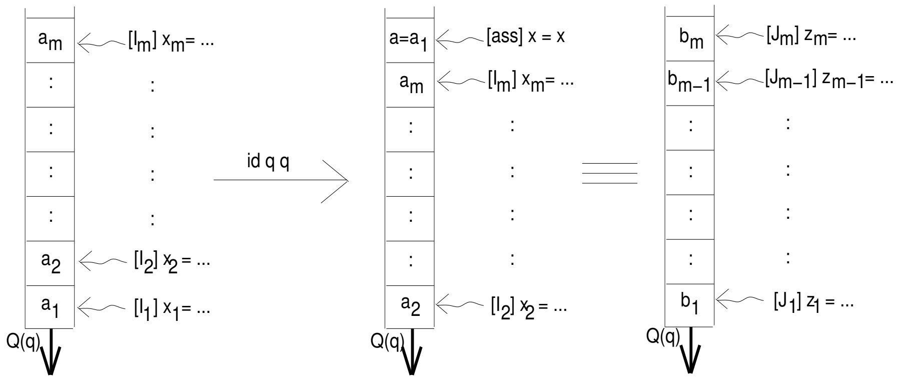


Figure 7.2: Change of $\mathsf{Q}(q)$ when executing $[\![ass]\!]_\rho = \mathtt{id}\ q\ q$.

where the x_i and I_i are those from fact (5). Then (see Figure 7.2)

$$\begin{aligned} \cup_{i=1}^{m} z_i &= \{z_m\} \cup_{i=1}^{m-1} z_i \\ &= \{x\} \cup_{i=1}^{m-1} x_{i+1} \\ &= \{x_1\} \cup_{i=1}^{m-1} x_{i+1} \\ &= \cup_{i=1}^{m} x_i \end{aligned}$$

using $x = x_1$ from above, and with the definition

$$l_i = \begin{cases} j_{i+1} & \text{for } i < m \\ n & \text{for } i = m \end{cases}$$

we obtain

- $0 < j_1 < j_2 = l_1 < j_3 = l_2 < \ldots < j_m = l_{m-1} < n = l_m < n+1$, $J_i = I_{i+1} = fst(B_{j_{i+1}}) = fst(B_{l_i})$ for $1 \leq i < m$ and $J_m = ass = fst(B_n) = fst(B_{l_m})$
- $z_i = x_{i+1} \in defs(I_{i+1}) = defs(J_i)$ for $1 \leq i < m$ and $z_m = x \in defs(ass) = defs(J_m)$
- $\rho(z_i) = \rho(x_{i+1}) = q' = q$ for $1 \leq i < m$ and $\rho(z_m) = \rho(x) = q$
- for $1 \leq i \neq i' \leq m$ we have $z_i \neq z_{i'}$, because

$$\begin{aligned} \cup_{i=1}^{m} z_i &= \cup_{i=1}^{m} x_i \\ &= \{y \in dom\,\Sigma \,|\, \rho(y) = q, \exists b.\ \Sigma(y) = (1,b)\} \\ &= \{y \in dom\,\Sigma_2 \,|\, \rho(y) = q, \exists b.\ \Sigma_2(y) = (1,b)\} \end{aligned}$$

where the last equality was shown in step (4). For $i \neq m$ we thus obtain $z_i \neq x$ and

$$\Sigma_2(z_i) = \Sigma_1(z_i) = \Sigma(z_i) = \Sigma(x_{i+1}) = (1, a_{i+1}) = (1, b_i),$$

while $i = m$ yields $\Sigma_2(z_m) = \Sigma_2(x) = (1, a) = (1, a_1) = (1, b_m)$.

– for $1 \leq i < m$, $z_i = x_{i+1} \notin \mathit{defs}(K) \cup \mathit{uses}(K)$ holds by fact (5) for $K = \mathit{fst}(B_k)$ and all $j_{i+1} < k < n$, i.e. $l_i < k < n$, and for $k = n$, $z_i \notin \{x\} = \mathit{defs}(\mathit{fst}(B_k)) \cup \mathit{uses}(\mathit{fst}(B_k))$ holds due to $z_i \neq x$ shown above. For $i = m$, the requirement reads $z_m = x \notin \mathit{defs}(\mathit{fst}(B_k)) \cup \mathit{uses}(\mathit{fst}(B_k))$ for all $n = l_m = l_i < k < n+1$ and is thus trivially fulfilled as no such k exists.

$\square$

A similar result relates IL states and ALEF configurations for jump instructions.

Proposition 40. *Let* C, S *and* B *as before. If* $\mathit{fst}(B)$ *is a jump instruction* jump, $\widetilde{\kappa} = \mathit{nil}$, $\mathsf{C} \models_{\alpha_\rho} \mathsf{S}$ *and* $\mathsf{S} \Rightarrow_{\alpha_\rho} \mathsf{T}$ *then there is a unique* D *such that* $\mathsf{C} \xrightarrow{[\![\mathit{jump}]\!]_\rho} \mathsf{D}$. *Furthermore,* $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$ *and* $\mathsf{D} = (\_, \_, \mathsf{M}, \iota)$ *hold where* $\mathsf{T} = (\_, \_, \mathtt{N}(\iota))$.

Proof. There are two cases, depending on the form of *jump*. In both cases, the proof follows the same style as the proof of the previous proposition.

Case $\mathit{jump} = [h]\,\mathtt{jmp}\; m_1$**.** The definition of $[\![\_]\!]_\rho$ yields

$$[\![\mathit{jump}]\!]_\rho = [h]\,\mathtt{jmp}\; m_1,$$

and for $\mathsf{S} \Rightarrow_{\alpha_\rho} \mathsf{T}$, rule JMP implies $\mathsf{T} = (\Sigma, h, \mathtt{N}(m_1))$.

The assumption $\mathsf{C} \models_{\alpha_\rho} S$ implies that there is an n with

(1) $\mathsf{S}_0 \Rightarrow^n_{\alpha_\rho} \mathsf{S}$, and in particular $\mathit{jump} = \mathit{fst}(B_n) = \mathit{fst}(B)$.

(2) if $y \in \mathit{dom}\,\Sigma$, $\Sigma(y) = (\top, b)$ and $\rho(y) = r'$ then $r' \in \mathit{dom}\,\mathsf{R}$.

(3) for $r' \in \mathit{dom}\,\mathsf{R}$ there is at most one $y \in \mathit{liveIn}(\mathit{jump})$ with $\rho(y) = r'$, and any such y fulfils $\Sigma(y) = (\top, \mathsf{R}(r'))$.

(4) for each $q \in Q$, $|\mathsf{Q}(q)| = |\{y \in dom\,\Sigma \,|\, \rho(y) = q, \exists b.\ \Sigma(y) = (1,b)\}|$ holds.

(5) for each $q \in Q$ and $\mathsf{Q}(q) = a_1 \ldots a_m$ there are variables $x_1, \ldots, x_m \in \mathbf{Var}_{\mathsf{P}}$ and $I_1, \ldots, I_m \in \mathbf{Instrs}_{\mathsf{P}}$ such that for all $1 \leq i \leq m$

- $I_i = fst(B_{j_i})$ for some $0 \leq j_1 \leq \ldots \leq j_m < n$
- $x_i \in defs(I_i)$
- $\rho(x_i) = q$
- $\Sigma(x_i) = (1, a_i)$
- $x_i \notin defs(J) \cup uses(J)$ for $J = fst(B_k)$ and all $j_i < k < n$

and $\cup_{i=1}^{m} x_i = \{y \in dom\,\Sigma \,|\, \rho(y) = q, \exists b.\ \Sigma(y) = (1,b)\}$.

For the unique $\mathsf{D} = (\mathsf{Q}, \mathsf{R}, \mathsf{M}, m_1)$, $\mathsf{C} \xrightarrow{[\![jump]\!]_\rho} \mathsf{D}$ holds, and the claim regarding ι is fulfilled. For showing $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$, we observe that the first two components Q and R of D agree with those of C, and that the first component Σ of T agrees with the first component of S, so all five claims follow from facts (1) to (5).

Case $jump = [h]\texttt{if}\ x\ m_1\ m_2$**.** The definition of $[\![\_]\!]_\rho$ yields

$$[\![jump]\!]_\rho = [h]\texttt{if}\ \rho(x)\ m_1\ m_2,$$

and for $\mathsf{S} \Rightarrow_{\alpha_\rho} \mathsf{T}$, rules IF-T and IF-F imply that $\Sigma, x \Downarrow \Sigma_1, a$ holds for some a with $\mathsf{T} = (\Sigma_1, h, \mathtt{N}(m_1))$ if $a = 0$ and $\mathsf{T} = (\Sigma_1, h, \mathtt{N}(m_2))$ if $a \neq 0$. The definition of $\Downarrow$ implies $\Sigma(x) = (l, a)$ for some $l \sqsupseteq 1$ and $\Sigma_1 = \Sigma[x \mapsto (l \ominus 1, a)]$, and the assumption $\mathsf{C} \models_{\alpha_\rho} S$ implies that there is an n with

(1) $\mathsf{S}_0 \Rightarrow^n_{\alpha_\rho} \mathsf{S}$, and in particular $jump = fst(B_n) = fst(B)$.

(2) if $y \in dom\,\Sigma$, $\Sigma(y) = (\top, b)$ and $\rho(y) = r'$ then $r' \in dom\,\mathsf{R}$.

(3) for $r' \in dom\,\mathsf{R}$ there is at most one $y \in liveIn(jump)$ with $\rho(y) = r'$, and any such y fulfils $\Sigma(y) = (\top, \mathsf{R}(r'))$.

(4) for each $q \in Q$, $|\mathsf{Q}(q)| = |\{y \in dom\,\Sigma \,|\, \rho(y) = q, \exists b.\ \Sigma(y) = (1,b)\}|$ holds.

(5) for each $q \in Q$ and $\mathsf{Q}(q) = a_1 \ldots a_m$ there are variables $x_1, \ldots, x_m \in \mathbf{Var}_{\mathsf{P}}$ and $I_1, \ldots, I_m \in \mathbf{Instrs}_{\mathsf{P}}$ such that for all $1 \leq i \leq m$

- $I_i = fst(B_{j_i})$ for some $0 \leq j_1 \leq \ldots \leq j_m < n$

- $x_i \in \mathit{defs}(I_i)$
- $\rho(x_i) = q$
- $\Sigma(x_i) = (1, a_i)$
- $x_i \notin \mathit{defs}(J) \cup \mathit{uses}(J)$ for $J = \mathit{fst}(B_k)$ and all $j_i < k < n$

and $\cup_{i=1}^{m} x_i = \{y \in \mathit{dom}\,\Sigma \mid \rho(y) = q, \exists b.\ \Sigma(y) = (1, b)\}$.

Again, there are two cases, depending on the value of $\rho(x)$.

Case $\rho(x) = r$**.** The proof proceeds in two steps.

Step 1. We first show that there is a D with $\mathsf{C} \xrightarrow{[\![\mathit{jump}]\!]_\rho} \mathsf{D}$.

From $\mathsf{S}_0 \Rightarrow_{\alpha_\rho}^{n} \mathsf{S}$ (fact (1)) and $\mathsf{S} \Rightarrow_{\alpha_\rho} \mathsf{T}$ we obtain $\mathsf{S}_0 \Rightarrow_{\alpha_\rho}^{n+1} \mathsf{T}$.

We noticed $\Sigma(x) = (l, a)$ above, so Lemma 18 implies $l = \top$, hence $\Sigma_1 = \Sigma$, and fact (2) implies $r \in \mathit{dom}\,\mathsf{R}$. Consequently, fact (3) yields $\Sigma(x) = (\top, \mathsf{R}(r))$ as $x \in \mathit{uses}(\mathit{jump}) \subseteq \mathit{liveIn}(\mathit{jump})$ holds. Uniqueness of the map R and $\Sigma(x) = (l, a)$ thus result in $\mathsf{R}(r) = a$.

Consequently, for $\mathsf{D} = (\mathsf{Q}, \mathsf{R}, \mathsf{M}, \iota)$ where

$$\iota = \begin{cases} m_1 & \text{if } a = 0 \\ m_2 & \text{if } a \neq 0 \end{cases}$$

we obtain $\mathsf{C} \xrightarrow{[\![\mathit{jump}]\!]_\rho} \mathsf{D}$, and the value in the last position agrees with the basic block $\mathtt{N}(\iota)$ in T.

Step 2. We now show $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$.

1. $\mathsf{S}_0 \Rightarrow_{\alpha_\rho}^{n+1} \mathsf{T}$ was shown above
2. For $y \in \mathit{dom}\,\Sigma_1$, $\Sigma_1(y) = (\top, b)$ and $\rho(y) = r'$ there are two cases.
   - If $x = y$ then $r' = \rho(y) = \rho(x) = r$, hence $r' \in \mathit{dom}\,\mathsf{R}$
   - If $x \neq y$ then $y \in \mathit{dom}\,\Sigma_1$ implies $y \in \mathit{dom}\,\Sigma$ with $\Sigma(y) = \Sigma_1(y) = (\top, b)$, so fact (2) implies $r' \in \mathit{dom}\,\mathsf{R}$.

   In both cases, $r' \in \mathit{dom}\,\mathsf{R}$ holds.
3. For $r' \in \mathit{dom}\,\mathsf{R}$ suppose there are variables $y \neq z \in \mathit{liveIn}(\mathit{fst}(\mathtt{N}(\iota)))$ with $\rho(y) = r' = \rho(z)$. Then equations (6.9) and $\mathit{fst}(\mathtt{N}(\iota)) \in \mathtt{succ}(\mathit{jump})$

yield $\{y,z\} \subseteq \mathit{liveOut}(\mathit{jump})$. Consequently, y and z are adjacent in $\mathcal{G}_{Reg}$ so $\mathsf{\rho}(y) \neq \mathsf{\rho}(z)$ as $\mathsf{\rho}$ is an allocation for $\mathtt{P}$. This contradicts $\mathsf{\rho}(y) = r' = \mathsf{\rho}(z)$, hence at most one $y \in \mathit{liveIn}(\mathit{fst}(\mathtt{N}(\iota)))$ with $\mathsf{\rho}(y) = r'$ exists.

For showing that any such y fulfils $\Sigma_1(y) = (\top, \mathsf{R}(r'))$, there are two cases.

- if $y = x$ then $r' = r$ and $\Sigma_1(y) = \Sigma_1(x) = (\top, \mathsf{R}(r))$ follows
- if $y \neq x$ then equations (6.9) imply

  $$y \in \mathit{liveIn}(\mathit{fst}(\mathtt{N}(\iota))) \subseteq \mathit{liveOut}(\mathit{jump}),$$

  hence $y \in \mathit{liveIn}(\mathit{jump})$ holds due to $y \notin \emptyset = \mathit{defs}(\mathit{jump})$. By fact (3), $\Sigma_1(y) = \Sigma(y) = (\top, \mathsf{R}(r'))$ follows.

4. For $q \in Q$, we have to show

   $$|\mathsf{Q}(q)| = |\{y \in \mathit{dom}\,\Sigma_1 \,|\, \mathsf{\rho}(y) = q, \exists b.\ \Sigma_1(y) = (1,b)\}|.$$

   Abbreviating $\{y \in \mathit{dom}\,\Sigma_1 \,|\, \mathsf{\rho}(y) = q, \exists b.\ \Sigma_1(y) = (1,b)\}$ by M_q, any $y \in M_q$ fulfils $y \neq x$ as $\mathsf{\rho}(y) = q \neq r = \mathsf{\rho}(x)$ holds. The definition of Σ_1 thus implies

   $$\begin{aligned} M_q &= \{y \in \mathit{dom}\,\Sigma_1 \,|\, \mathsf{\rho}(y) = q, \exists b.\ \Sigma_1(y) = (1,b)\} \\ &= \{y \in \mathit{dom}\,\Sigma_1 \,|\, y \neq x, \mathsf{\rho}(y) = q, \exists b.\ \Sigma_1(y) = (1,b)\} \\ &= \{y \in \mathit{dom}\,\Sigma \,|\, y \neq x, \mathsf{\rho}(y) = q, \exists b.\ \Sigma(y) = (1,b)\} \\ &= \{y \in \mathit{dom}\,\Sigma \,|\, \mathsf{\rho}(y) = q, \exists b.\ \Sigma(y) = (1,b)\} \end{aligned}$$

   so fact (4) yields $|\mathsf{Q}(q)| = |M_q|$.

5. For each $q \in Q$ and $\mathsf{Q}(q) = a_1 \ldots a_m$ we have to show that there are $z_1, \ldots, z_m \in \mathbf{Var}_{\mathtt{P}}$ and $J_1, \ldots, J_m \in \mathbf{Instrs}_{\mathtt{P}}$ such that for all $1 \leq i \leq m$
   - $J_i = \mathit{fst}(B_{l_i})$ for some $0 \leq l_1 \leq \ldots \leq l_m < n+1$
   - $z_i \in \mathit{defs}(J_i)$
   - $\mathsf{\rho}(z_i) = q$
   - $\Sigma(z_i) = (1, a_i)$

- $z_i \notin defs(K) \cup uses(K)$ for $K = fst(B_h)$ and all $l_i < h < n+1$

and $\cup_{i=1}^{m} z_i = M_q$ for the M_q as defined above.

Defining $z_i = x_i$ and $J_i = I_i$ from fact (5) for $1 \leq i \leq m$ yields

- $J_i = I_i = fst(B_{l_i})$ for $j_i = l_i$, and $1 \leq l_i \leq \ldots \leq l_i < n < n+1$ holds
- $z_i = x_i \in defs(I_i) = defs(J_i)$
- $\rho(z_i) = \rho(x_i) = q$
- each z_i fulfils $z_i \neq x$ due to $\rho(z_i) = q \neq r = \rho(x)$. Therefore, $\Sigma_2(z_i) = \Sigma_1(z_i) = \Sigma(z_i) = \Sigma(x_i) = (1, a_i)$.
- for $l_i < h < n$ we obtain $z_i = x_i \notin defs(K) \cup uses(K)$ for $K = fst(B_h)$ by fact (5), and for $n \leq h < n+1$ we have $h = n$, so

$$\begin{aligned} z_i = x_i \notin \{x\} &= defs(jump) \cup uses(jump) \\ &= defs(fst(B_h)) \cup uses(fst(B_h)) \end{aligned}$$

and $\cup_{i=1}^{m} z_i = \cup_{i=1}^{m} x_i = M_q$ using the above equality

$$\{y \in dom\, \Sigma \,|\, \rho(y) = q, \exists b.\ \Sigma(y) = (1,b)\} = M_q.$$

Case $\rho(x) = q$. Again, the proof proceeds in two steps.

Step 1. We first show that there is a D with $\mathsf{C} \xrightarrow{[\![jump]\!]_\rho} \mathsf{D}$.

From $\mathsf{S}_0 \Rightarrow_{\alpha_\rho}^{n} \mathsf{S}$ (fact (1)) and $\mathsf{S} \Rightarrow_{\alpha_\rho} \mathsf{T}$ we obtain $\mathsf{S}_0 \Rightarrow_{\alpha_\rho}^{n+1} \mathsf{T}$.

We noticed $\Sigma(x) = (l, a)$ above, so Lemma 18 implies $l = 1$, hence $\Sigma_1 = \Sigma[x \mapsto (0, a)]$, and

$$x \in M_q = \{y \in dom\, \Sigma \,|\, \rho(y) = q, \exists b. \Sigma(y) = (1,b)\}.$$

By fact (4), $|\mathsf{Q}(q)| > 1$ follows, and for $\mathsf{Q}(q) = a_1 \ldots a_m$ there are (by fact (5)) $x_1 \ldots x_m$ and $I_1 \ldots I_m$ such that $I_i = fst(B_{j_i})$ holds for some $0 \leq j_1 \leq \ldots \leq j_m < n$ and all properties in fact (5) are fulfilled. Therefore, $a = a_i$ holds for some $1 \leq i \leq m$.

Similarly to the proof of the case $'x = x'$, we show $x = x_1$, i.e. $a = a_1$. Suppose $x \neq x_1$. Then the sequence

$$fst(B_{j_1}) \ldots fst(B_{j_2}) \ldots fst(B_{j_m}) \ldots jump = fst(B_n)$$

is a path π_0 In $\mathtt{P}$. Also, for $1 \leq k$, fact (5) implies $x_1 \in \mathit{defs}(\pi_0(1))$ and $x_1 \notin \mathit{defs}(\pi_0(k)) \cup \mathit{uses}(\pi_0(k))$. By Lemma 19, π_0 is hence the prefix of an x_1-path π. Since $x = x_i$ holds for some $1 \leq i < n$, $x_i \in \mathit{defs}(I_i)$ and $x \in \mathit{uses}(\mathit{jump})$, the path

$$\tau = I_i, \ldots, I_m, \ldots, \mathit{jump}$$

is an x-path – fact (5) guarantees that jump is the first use of x_i following I_i). Thus π contains τ, in contradiction to $\rho(x_1) = q = \rho(x)$, since ρ is an allocation for $\mathtt{P}$. Therefore, $x = x_1$ holds.

For $\mathsf{D} = (\mathsf{P}, \mathsf{R}, \mathsf{M}, \iota)$ where

$$\iota = \begin{cases} m_1 & \text{if } a = 0 \\ m_2 & \text{if } a \neq 0 \end{cases}$$

and $\mathsf{P} = \mathsf{Q}[q \mapsto a_2 \ldots a_m]$ we obtain $\mathsf{C} \xrightarrow{[\![\mathit{jump}]\!]_\rho} \mathsf{D}$, and the value ι in the last component of D coincides with the component $\mathtt{N}(\iota)$ in T.

Step 2. We now show $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$.

1. $\mathsf{S}_0 \Rightarrow^{n+1}_{\alpha_\rho} \mathsf{T}$ was shown above
2. For $y \in \mathit{dom}\, \Sigma_1$, $\Sigma_1(y) = (\top, b)$ and $\rho(y) = r'$, $y \neq x$ follows due to $\rho(x) = q$, so $y \in \mathit{dom}\, \Sigma_1$ implies $y \in \mathit{dom}\, \Sigma$ with $\Sigma(y) = \Sigma_1(y) = (\top, b)$, and fact (2) implies $r' \in \mathit{dom}\, \mathsf{R}$.
3. For $r' \in \mathit{dom}\, \mathsf{R}$ suppose there are variables $y \neq z \in \mathit{liveIn}(\mathit{fst}(\mathtt{N}(\iota)))$ with $\rho(y) = r' = \rho(z)$. Then equations (6.9) and $\mathit{fst}(\mathtt{N}(\iota)) \in \mathtt{succ}(\mathit{jump})$ yield $\{y, z\} \subseteq \mathit{liveOut}(\mathit{jump})$. Consequently, y and z are adjacent in $\mathcal{G}_{\mathit{Reg}}$ so $\rho(y) \neq \rho(z)$ as ρ is an allocation for $\mathtt{P}$. This contradicts $\rho(y) = r' = \rho(z)$, hence at most one $y \in \mathit{liveIn}(\mathit{fst}(\mathtt{N}(\iota)))$ exists with $\rho(y) = r'$.

   Again, any such y fulfils $y \neq x$ due to $\rho(y) = r' \neq q = \rho(x)$ and equations (6.9) and $\emptyset = \mathit{defs}(\mathit{jump})$ imply

   $$y \in \mathit{liveIn}(\mathit{fst}(\mathtt{N}(\iota))) \subseteq \mathit{liveOut}(\mathit{jump}) \subseteq \mathit{liveIn}(\mathit{jump})$$

   By fact (3), $\Sigma_1(y) = \Sigma(y) = (\top, \mathsf{R}(r'))$ follows.

4. For $q' \in Q$, we show

$$|\mathsf{P}(q')| = |\{y \in dom\,\Sigma_1 \,|\, \rho(y) = q', \exists b.\ \Sigma_1(y) = (1,b)\}|.$$

- If $q \neq q'$ then any $y \in dom\,\Sigma$ with $\rho(y) = q'$ and $\Sigma(y) = (1,b)$ for some b fulfils $x \neq y$ due to $\rho(x) = q$, hence

$$\begin{aligned}
|\mathsf{P}(q')| &= |\mathsf{Q}(q')| \\
&= |\{y \in dom\,\Sigma \,|\, \rho(y) = q', \exists b.\Sigma(y) = (1,b)\}| \\
&= |\{y \in dom\,\Sigma \,|\, y \neq x, \rho(y) = q', \exists b.\Sigma(y) = (1,b)\}| \\
&= |\{y \in dom\,\Sigma_1 \,|\, y \neq x, \rho(y) = q', \exists b.\Sigma_1(y) = (1,b)\}| \\
&= |\{y \in dom\,\Sigma_1 \,|\, \rho(y) = q', \exists b.\Sigma_1(y) = (1,b)\}|
\end{aligned}$$

  holds by fact (4) and the definitions of P and Σ_2.

- If $q = q'$ then fact (4), $\rho(x) = q$ and $\Sigma(x) = (1,a)$ imply

$$\begin{aligned}
|\mathsf{P}(q)| &= |\mathsf{Q}(q)| - 1 \\
&= |\{y \in dom\,\Sigma \,|\, \rho(y) = q, \exists b.\Sigma(y) = (1,b)\}| - 1 \\
&= |\{x\} \cup \\
&\quad \{y \in dom\,\Sigma \,|\, x \neq y, \rho(y) = q, \exists b.\Sigma(y) = (1,b)\}| - 1 \\
&= |\{y \in dom\,\Sigma_1 \,|\, x \neq y, \rho(y) = q, \exists b.\Sigma_1(y) = (1,b)\}| \\
&= |\{y \in dom\,\Sigma_1 \,|\, \rho(y) = q, \exists b.\Sigma_1(y) = (1,b)\}|
\end{aligned}$$

5. for $q' \in Q$ and $\mathsf{P}(q') = b_1 \ldots b_\mu$ we have to show that there are $z_1, \ldots, z_\mu$ and $J_1, \ldots, J_\mu$ such that $\cup_{i=1}^{\mu} z_i = \{y \in dom\,\Sigma_1 \,|\, \rho(y) = q', \exists b.\Sigma_1(y) = (1,b)\}$ and for all $1 \leq i \leq \mu$ the following properties hold.

   - $J_i = fst(B_{l_i})$ for some $0 \leq l_1 \leq \ldots \leq l_\mu < n+1$
   - $z_i \in defs(J_i)$
   - $\rho(z_i) = q'$
   - $\Sigma_1(z_i) = (1, b_i)$
   - $z_i \notin defs(K) \cup uses(K)$ for $K = fst(B_g)$ and all $l_i < g < n+1$

   Again, there are two cases.

- If $q \neq q'$ then we take (for all i) $z_i = x_i$ and $J_i = I_i$ from fact (5), and obtain
  - $J_i = I_i = \mathit{fst}(B_{j_i}) = \mathit{fst}(B_{l_i})$ for $j_i = l_i$, hence $0 \leq l_1 \leq \ldots l_\mu < n < n+1$
  - $z_i = x_i \in \mathit{defs}(I_i) = \mathit{defs}(J_i)$
  - $\rho(z_i) = \rho(x_i) = q'$
  - each z_i fulfils $z_i \neq x$ due to $\rho(z_i) = q' \neq q = \rho(x)$, hence $\Sigma_1(z_i) = \Sigma(z_i) = \Sigma(x_i) = (1, b_i)$
  - $z_i = x_i \notin \mathit{defs}(K) \cup \mathit{uses}(K)$ for $K = \mathit{fst}(B_g)$ and all $l_i < g < n$ by fact (5) and $z_i = x_i \notin \mathit{defs}(\mathit{fst}(B_n)) \cup \mathit{uses}(\mathit{fst}(B_n)) = \{x\}$ holds due to $z_i \neq x$.
- If $q = q'$ then $\mu = m - 1$, and $b_i = a_{i+1}$ holds for $1 \leq i < m$ (where m is the length of $\mathsf{Q}(q)$), and we define

  $$(z_i, J_i) = (x_{i+1}, I_{i+1})$$

  (again for $1 \leq i < m$), where the x_i and I_i are those from fact (5). Then

  $$\cup_{i=1}^{\mu} z_i = \cup_{i=2}^{m} x_i = \{y \in \mathit{dom}\,\Sigma_1 \,|\, \rho(y) = q, \exists b. \Sigma(y) = (1, b)\},$$

  and with the definition

  $$l_i = j_{i+1} \text{ for } 1 \leq i < m$$

  we obtain
  - $0 < j_1 < j_2 = l_1 < j_3 = l_2 < \ldots < j_m = l_{m-1} < n < n+1$, $J_i = I_{i+1} = \mathit{fst}(B_{j_{i+1}}) = \mathit{fst}(B_{l_i})$ for $1 \leq i < m$
  - $z_i = x_{i+1} \in \mathit{defs}(I_{i+1}) = \mathit{defs}(J_i)$ for $1 \leq i < m$
  - $\rho(z_i) = \rho(x_{i+1}) = q' = q$ for $1 \leq i < m$
  - from

    $$\begin{aligned} \cup_{i=1}^{\mu} z_i &= \cup_{i=2}^{m} x_i \\ &= \{y \in \mathit{dom}\,\Sigma \,|\, \rho(y) = q, \exists b.\ \Sigma(y) = (1, b)\} \setminus \{x\} \\ &= \{y \in \mathit{dom}\,\Sigma_1 \,|\, \rho(y) = q, \exists b.\ \Sigma_2(y) = (1, b)\} \end{aligned}$$

we obtain for $z_i \neq z_{i'}$ for $1 \leq i \neq i' \leq m$, hence $z_i \neq x$ follows for $i \neq \mu$, and we obtain

$$\Sigma_1(z_i) = \Sigma(x_{i+1}) = (1, a_{i+1}) = (1, b_i)$$

for $1 \leq i \leq \mu = m-1$

– for $1 \leq i \leq \mu = m-1$, $z_i = x_{i+1} \notin \mathit{defs}(K) \cup \mathit{uses}(K)$ holds by fact (5) for $K = \mathit{fst}(B_k)$ and all $j_{i+1} < k < n$, i.e. $l_i < k < n$, and for $k = n$, $z_i \notin \{x\} = \mathit{defs}(\mathit{fst}(B_k)) \cup \mathit{uses}(\mathit{fst}(B_k))$ follows from $z_i \neq x$.

□

Consequently, the execution of basic blocks in IL and ALEF agree.

Proposition 41. *For $B \in \mathbf{Blocks}_{\mathsf{P}}$ let $\mathsf{S} = (\Sigma, n, B)$, $\mathsf{C} = (\mathsf{Q}, \mathsf{R}, \mathsf{M}, nil)$ and $\mathsf{C} \models_{\alpha_\rho} \mathsf{S}$. If $\mathsf{S} \Rightarrow^{|B|}_{\alpha_\rho} \mathsf{T}$ then there is a unique D such that $\mathsf{C} \xrightarrow{[\![B]\!]_\rho} \mathsf{D}$. Furthermore, for $\mathsf{D} = (\mathsf{Q}', \mathsf{R}', \mathsf{M}', \widetilde{k})$ and $\mathsf{T} = (\Sigma', n', B')$ the following hold*

- $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$
- $|B'| = 0$ *implies* $\widetilde{k} = nil$
- $|B'| > 0$ *implies* $\widetilde{k} \neq nil$ *and* $B' = \mathtt{N}(k)$.

Proof. Induction on $|B|$. For $|B| = 1$, $\mathit{fst}(B)$ is either a jump instruction *jump* or an assignment *ass*.
In the former case, there is a unique D with $\mathsf{C} \xrightarrow{[\![\mathit{fst}(B)]\!]_\rho} \mathsf{D}$ by Proposition 40, and $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$ holds. Hence $\mathsf{C} \xrightarrow{[\![B]\!]_\rho} \mathsf{D}$, and by the rules JMP, IF-T and IF-F, $B' = \mathtt{N}(l)$ follows for some l. By Definition 24, $|B'| > 1$ holds, and $nil \neq k = l$ indeed holds by Proposition 40.
In the latter case, the second part of Proposition 39 implies $\mathsf{C} \xrightarrow{[\![\mathit{fst}(B)]\!]_\rho} \mathsf{D}$ for some unique D, and $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$ holds. From $|B| = 1$, $|B'| = 0$ follows by rule ASS, and the second part of Proposition 39 indeed guarantees $\widetilde{k} = nil$ as the last components of C and D agree.

For $|B| > 1$, $fst(B) = ass$ holds by Definition 24, and for $\mathsf{S} \Rightarrow_{\alpha_\rho} \mathsf{S}_1 \Rightarrow_{\alpha_\rho}^{|B|-1} \mathsf{T}$ and $\mathsf{S}_1 = (\Sigma_1, m_1, B_1)$, rule ASS implies $B = ass\,B_1$ with $|B_1| \geq 0$. Applying the second part of Proposition 39 yields a (unique) E such that $\mathsf{C} \xrightarrow{[\![fst(B)]\!]_\rho} \mathsf{E}$, and $\mathsf{E} \models_{\alpha_\rho} \mathsf{S}_1$ holds, and the last component of E equals that of C and is hence *nil*. Applying the induction hypothesis to $\mathsf{S}_1 \Rightarrow_{\alpha_\rho}^{|B|-1} \mathsf{T}$ and $\mathsf{E} \models_{\alpha_\rho} \mathsf{S}_1$ yields a (unique) D with $\mathsf{E} \xrightarrow{[\![B_1]\!]_\rho} \mathsf{D}$, and $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$ and the two claims regarding the last component of D hold. Composing $\mathsf{C} \xrightarrow{[\![fst(B)]\!]_\rho} \mathsf{E}$ and $\mathsf{E} \xrightarrow{[\![B_1]\!]_\rho} \mathsf{D}$ thus yields $\mathsf{C} \xrightarrow{[\![B]\!]_\rho} \mathsf{D}$ for a D of the required shape. ☐

Consequently, $\Rightarrow_{\alpha_\rho}$-execution is faithfully matched by ALEF execution:

Theorem 9. *Let* $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ *be a regular IL-program and* ρ *an allocation for* $\mathtt{P}$*. If* S_0 $\Rightarrow_{\alpha_\rho}$*-terminates in a configuration* T *then there are unique* D *and* w *such that for* $\mathsf{C} = ([\,],[\,],\mathsf{M},\mathtt{entry})$*,* $\mathsf{C} \xrightarrow{w} \mathsf{D}$ *holds with* $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$*.*

Proof. By the first part of Proposition 39, $\mathsf{C}_0 = ([\,],[\,],\mathsf{M}, nil)$ fulfils $\mathsf{C}_0 \models_{\alpha_\rho} \mathsf{S}_0$, where $\mathsf{S}_0 = ([\,], \mathtt{entry}, \mathtt{N}(\mathtt{entry}))$ by Definition 32. As $|\mathtt{N}(\mathtt{entry})| > 0$ holds by Definition 24, there is a unique T_1 such that $\mathsf{S}_0 \Rightarrow_{\alpha_\rho}^{|\mathtt{N}(\mathtt{entry})|} \mathsf{T}_1$, and applying Proposition 41 yields a unique D_1 with $\mathsf{C}_0 \xrightarrow{[\![\mathtt{N}(\mathtt{entry})]\!]_\rho} \mathsf{D}_1$, hence $\mathsf{C} \xrightarrow{\mathtt{entry}} \mathsf{D}$ by rule EXECUTE. Furthermore, $\mathsf{D}_1 \models_{\alpha_\rho} \mathsf{T}_1$ holds, for $\mathsf{T}_1 = (\Sigma, m, B)$, $|B| = 0$ implies $\mathsf{D}_1 = (\_,\_,\_, nil)$, while $|B| = 1$ implies $B = \mathtt{N}(l)$ for some l, and $\mathsf{D}_1 = (\_,\_,\_, l)$. With an induction on the number of basic blocks executed during the reduction $\mathsf{S} \Rightarrow_{\alpha_\rho} \mathsf{T}$ the claim follows. ☐

By combining Theorems 6 and 9 we thus obtain functional correctness of the translation based on dataflow solutions for regular programs.

Corollary 4. *Let* $\mathtt{P} = (\mathtt{N}, \mathtt{A}, \mathtt{succ}, \mathtt{entry})$ *be a regular IL-program and* ρ *an allocation for* $\mathtt{P}$*. If* S_0 $\rightarrow$*-terminates in a configuration* T *then there is are unique* D *and* w *such that for* $\mathsf{C} = ([\,],[\,],\mathsf{M},\mathtt{entry})$*,* $\mathsf{C} \xrightarrow{w} \mathsf{D}$ *holds, and* $\mathsf{D} \models_{\alpha_\rho} \mathsf{T}$*.*

For SSA programs, the similar result follows by Theorem 7 and Proposition 37.

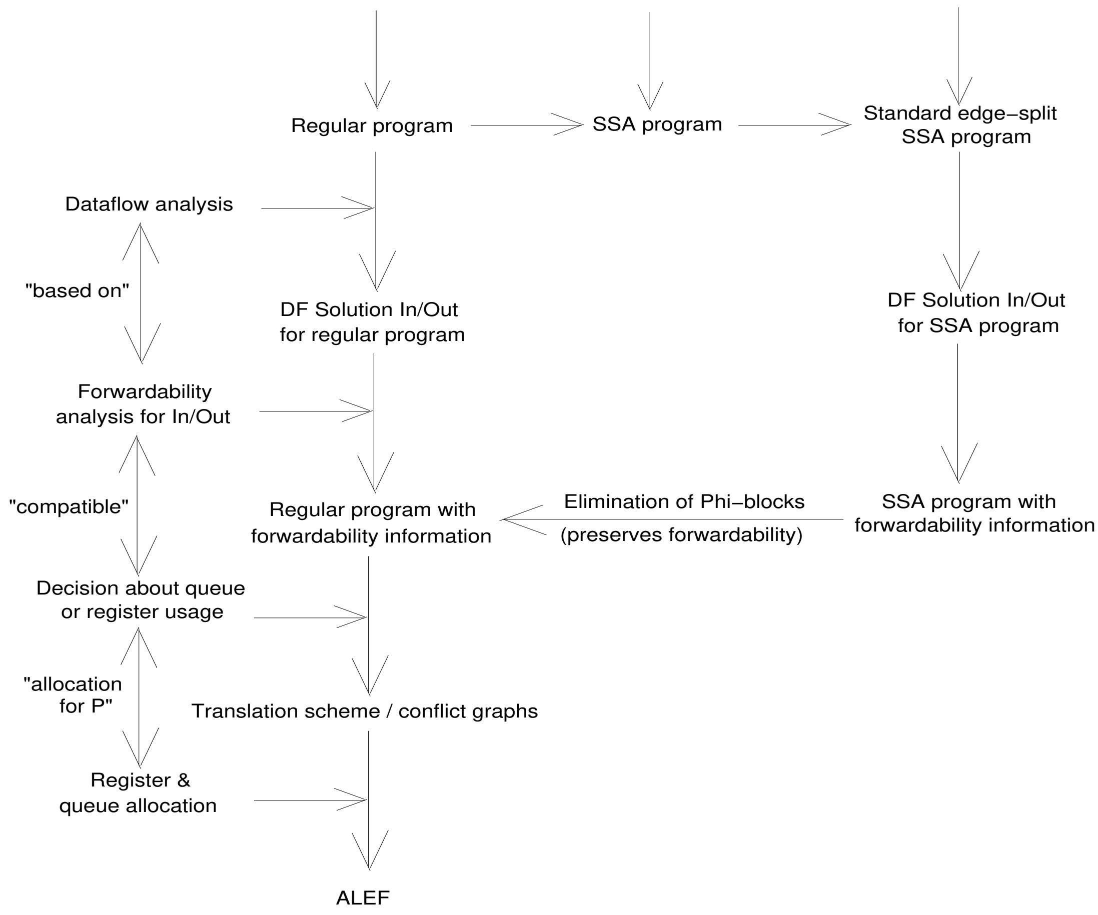


Figure 7.3: Compilation from IL to ALEF

7.7 Discussion

7.7.1 Summary of the compilation

Chapters 6 and 7 outlined the back-end of a compiler with a program analysis phase for an intermediate language and a translation into ALEF. The various routes a compilation may follow are shown in Figure 7.3.

Starting from a regular IL program, forwardability information may either be obtained directly from a solution to the dataflow equations for variable usage. Alternatively, the program may be transformed into (standard edge-split) SSA form as described in Section 6.2. We may also accept SSA programs which did not arise from such a transformation by checking that the syntactic requirements are met. In both cases, dataflow equations and forwardability analysis are performed on the SSA code, before Φ-blocks are eliminated in a forwardability-preserving (and functionally correct) way. Subsequently, for each forwardable variable it is decided whether forwarding should actually be used, possibly governed by restrictions on the number of queues and their length. Allocating queues and registers by colouring the conflict graphs yields the final ALEF program. Provided that each step in the translation fulfilled the requirements of the corresponding propositions (compatibility, etc.) the resulting ALEF program agrees functionally with the input program, and is well-typed where the typing corresponds to the chosen dataflow solution.

7.7.2 Extensions and future work

In order to obtain the formal results, we presented a rather simple dataflow analysis and a very basic translation. The following sections outline some improvements which would be desirable for a full compiler, as well as some topics for more theoretical future work.

7.7.2.1 Alternative patterns of usage

Multiple usage Instead of requiring a value to be used exactly once on each path, one can require a value to be used exactly k times, for arbitrary fixed k. The corresponding dataflow analysis extends $\mathcal{L}$ to the flat lattice $\mathcal{L}_k$ of values $0, \ldots k$, with addition defined by

$\oplus$	$\bot$	0	1	$\cdots$	k-1	k	$\top$
$\bot$	$\bot$	$\bot$	1	$\cdots$	k-1	k	$\top$
0	$\bot$	0	1	$\cdots$	k-1	k	$\top$
1	1	1	2	$\cdots$	k	$\top$	$\top$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\top$	$\top$	$\top$
k-1	k-1	k-1	k	$\top$	$\top$	$\top$	$\top$
k	k	k	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

and $\ominus$ defined by $l_1 \ominus l_2 = \sqcup\{l \mid l \oplus l_2 = l_1\}$.

In the context of ALEF, the case $k = 2$ is of particular interest, as variables with exactly two uses can be translated into a duplication instruction, provided that all pairs of consuming instructions execute on the same sets of functional units. For example the IL-program $[1]\mathtt{x} = 4;\ [2]\mathtt{y} = \mathtt{x} * 2;\ [3]\mathtt{z} = \mathtt{y} + \mathtt{x}$ may be translated to

$$[1]\mathtt{ldc}\ 4\ \mathtt{q}_1\ [1\mathtt{a}]\mathtt{dupl}^{\mathtt{MUL}}\ \mathtt{q}_1\ \mathtt{q}_3\ \mathtt{q}_1\ [2]\mathtt{muli}\ 2\ \mathtt{q}_1\ \mathtt{q}_2\ [3]\mathtt{add}\ \mathtt{q}_2\ \mathtt{q}_3\ \mathtt{r}_1$$

where we assume $\gamma(\mathtt{q}_1) = \mathtt{MUL}$ and $\gamma(\mathtt{q}_2) = \gamma(\mathtt{q}_3) = \mathtt{ALU}$. In an extended ALEF instruction set, an instruction might exist which combines the first two instructions to $[1]\mathtt{ldc\_d}\ 4\ \mathtt{q}_3\ \mathtt{q}_1$.

Variables which are used exactly $k > 2$ times may in principle be translated into a cascade of duplication instructions. For example,

$$[1]\mathtt{x} = 4;\ [2]\mathtt{y} = \mathtt{x} * \mathtt{x};\ [3]\mathtt{z} = \mathtt{y} + \mathtt{x}$$

may be translated to

$$[1]\mathtt{ldc}\ 4\ \mathtt{q}_1\ [1\mathtt{a}]\mathtt{dupl}^{\mathtt{MUL}}\ \mathtt{q}_1\ \mathtt{q}_1\ \mathtt{q}_1\ [1\mathtt{b}]\mathtt{dupl}^{\mathtt{MUL}}\ \mathtt{q}_1\ \mathtt{q}_3\ \mathtt{q}_1\ [2]\mathtt{mul}\ \mathtt{q}_1\ \mathtt{q}_1\ \mathtt{q}_2\ [3]\mathtt{add}\ \mathtt{q}_2\ \mathtt{q}_3\ \mathtt{r}_1.$$

As k grows however, we expected that register usage will outperform `dupl`-cascades.

Up to k uses More generously, it may be possible to include paths with *at most* k uses, rather than requiring *exactly* k uses. By modifying the definition of *fwdOut* to

$$\mathit{fwdOut}(I)(x) = \begin{cases} \bot & \text{if } \mathtt{succ}(I) = \emptyset \\ \sqcup_{J \in \mathtt{succ}(I)} \mathit{fwdIn}(J)(x) & \text{otherwise} \end{cases}$$

values which are not consumed may be left over at the end of a program run. Inside loops, no spare values are admitted, and the above definition in fact simplifies to

$$\mathit{fwdOut}(I)(x) = \sqcup_{J \in \mathtt{succ}(I)} \mathit{fwdIn}(J)(x)$$

using $\sqcup_\emptyset \ldots = \bot$. On the other hand, this modification may require one to revisit the translation between dataflow solutions and ALEF types.

Unbalanced branches In situations where the two branches of a conditional jump differ in the number of uses of a variable, the compiler may insert additional duplication instructions. Likewise, by inserting `skip` instructions we can remove redundant copies of variables, and inserting `id`-instructions allows us to move values between operand queues connected to different functional units. We expect that these techniques in combination will make *all* variables forwardable. Whenever a variable is accessed, a duplication instruction is inserted, and one copy is used immediately while the other one is kept for later. In principle, forwarding should be possible for all variables for logical reasons: in Chapter 5 we observed that registers play the role of exponentials in the linear setting, so the duplication instruction should eliminate the need for registers.

7.7.2.2 Efficient translations

Translating Φ-instructions into several `id` instructions often results in an intolerably high static and dynamic instruction count. Consider for example the Java excerpt

```
int f(){
  int x=3;
  int y=2;
  for (int z=10, z>0, z--){x=x+y;}
  x=x+1
}
```
(7.12)

and its naive translation into SSA-IL

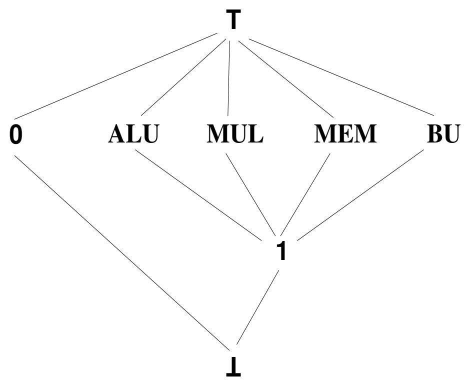
$$
\begin{aligned}
\mathtt{N(1)} &= [1]\mathtt{x1} = 3\\
&\quad [2]\mathtt{y1} = 2\\
&\quad [3]\mathtt{z1} = 10\\
&\quad [4]\mathtt{jmp}\ 5\\
\mathtt{N(5)} &= [5]\begin{pmatrix}\mathtt{x2}\\ \mathtt{z2}\end{pmatrix} = \begin{pmatrix}\mathtt{x1} & \mathtt{x3}\\ \mathtt{z1} & \mathtt{z3}\end{pmatrix}\begin{pmatrix}4\\ 9\end{pmatrix}\\
&\quad [6]\mathtt{if}\ \mathtt{z2}\ 10\ 7
\end{aligned}
\qquad
\begin{aligned}
\mathtt{N(7)} &= [7]\mathtt{x3} = \mathtt{x2} + \mathtt{y1}\\
&\quad [8]\mathtt{z3} = \mathtt{z2} - 1\\
&\quad [9]\mathtt{jmp}\ 5\\
\mathtt{N(10)} &= [10]\mathtt{x4} = \mathtt{x2} + 1
\end{aligned}
$$

In our translation the Φ-block results in four `id`-instructions, two at the end of block 1 and one at the end of block 7. All these instructions represent the only uses of their respective operand `x1`, `x3`, `z1` and `z3`. Their results are used exactly once, and are indeed both forwardable to `ALU`. All this information is captured in the dataflow analysis and a less wasteful translation would reduce the number of `id` instructions, in effect reversing the splitting of variables performed during the transformation into SSA.

In general, optimisation can be achieved either by transformations during compilation or by post-compilation optimisation, similar to peephole optimisation. The presence of types in the compilation output is beneficial in both cases. For example, `dec` $r\ q_1$ `id`$^{\gamma(q_1)}$ $q_1\ q_2$ may only be replaced by `dec` $r\ q_2$ if the value consumed by the second instruction actually is the one resulting from the first one. These context-dependencies are captured in types, as was shown when considering data-dependence between instruction with output-conflicts in Chapter 4.

7.7.2.3 Integrating dataflow analysis and forwardability

The decision whether a value is forwardable occurs as a separate step in our approach, between the dataflow analysis and the allocation. It can however be integrated with the dataflow analysis by using the more sophisticated lattice



The functional units sit flatly between 1 and ⊤ but behave like 1 with respect to ⊕. The element 1 is used for polymorphism in `id` instructions arising from assignments $y = x$. Alternatively, it may be possible to integrate the forwardability decision with processor-dependent compiler phases such as queue and register allocation.

7.7.2.4 Allocation for distributed and bounded execution

The conflict graph constructed in Section 7.5 is suited for allocating operand queues with respect to sequential execution of ALEF programs. Race conditions arising from out-of-order execution do not surface using x-paths and are therefore not eliminated. Neither did we require the colouring to satisfy length limitation of operand queues – in fact an optimal colouring will most likely yield long operand queues.

One approach for obtaining allocations for distributed and finite-queue execution consists of post-processing sequential allocations using the techniques from Chapter 4. More elegantly, it may be possible to capture additional requirements in the conflict graph or the dataflow equations. For example, edges may be introduced between intermediate variables whose defining instruction(s) execute on different functional units. This generalises the forwardability predicate by complementing the destination information with the source of a value, similar to pipeline-dependencies on the ALEF level. Leaving the details for future research, we expect that data-dependencies and finiteness of queues may be tackled by restricting the graph colourings or by a more fine-grained dataflow analysis which keeps track of values which are jointly in-flight under distributed execution.

7.7.2.5 Relating program analysis frameworks

The construction of non-standard dynamic semantics in Chapter 6 followed a very natural regime: incorporate the lattice of the dataflow equations into the dynamic semantics, taking into account that the dataflow analysis proceeds in the opposite direction as the program execution by dualising $\oplus$ to $\ominus$. If this approach can be generalised to arbitrary dataflow-equations, proving these correct would amount to relating the natural dynamic semantics arising from (a solution to) the dataflow equations to the standard dynamic semantics. Further effort is needed to see whether this is indeed possible, to formalise such correspondences in frameworks such Nielsen et al's [NNH99], and to explore whether a connection to general approaches for relating syntax and operational semantics exists.

We noted in Section 7.4 that the correspondence between dataflow solutions and low-level types is slightly unsatisfactory and future work should consider homomorphic relationships between $\oplus$ and $\otimes$. As was mentioned before, one possibility may be to perform a second dataflow analysis which promotes the decision whether a forwardable value is actually sent through an operand queue downwards along the program execution flow.

Recent work in the programming language community established structural relations between flow analysis, abstract interpretation and type systems [NNH99] [PP98] [Hei95], often by expressing particular (classes of) program analysis problems in several frameworks and formally relating the corresponding calculi. In contrast, our correspondence between dataflow equations and low-level types involved an (albeit simple) compilation from an intermediate language to assembly language. A future challenge will consequently consist of combining compilation with translations between several formalisms and to extend the types-in-compilation approach to other program analysis techniques.

7.7.3 Conclusion

The dataflow analysis and compilation substantiate the argument that a programming language based approach is beneficial for linking processor structure and program analysis, allowing compilers to optimise operand communication. We draw two conclusions.

1. Ensuring safe operand usage in ALEF programs can be lifted from the assembly level to earlier stages of the compiler. This demonstrates that our approach of typing the operand usage fits well with other approaches pursued in the typed-assembly-language community, and with the types-in-compilation paradigm.

2. The restrictions we imposed on ALEF programs for being type-correct are generous enough to accommodate many forwardable variables. In particular, the unification condition during the composition of basic blocks appeared rather restrictive on the assembly level, but is naturally met by programs originating from IL via the correspondence between dataflow solutions and ALEF typings.

The next chapter will describe an initial implementation of the compilation and report on results obtained from translating Java code into ALEF.

Chapter 8

Implementation

In this chapter, we describe a prototypical implementation of our compilation approach and give experimental results for a standard benchmark suite. The purpose of this implementation is to validate some of the calculi described in the earlier chapters rather than to provide a full compiler of high efficiency. We chose ML as implementation language, using the Moscow-ML compiler kit [RRS00].

Concentrating on the regular fragment of IL, our implementation follows the approach of Chapters 6 and 7. Dataflow analysis and register and queue allocation phases are implemented using a simple iterative algorithm and greedy colouring algorithms respectively. While for production-strength compilers more efficient implementations are necessary, the chosen algorithms provide sufficient insight for our purpose.

In order to exercise the translation on larger programs, we implemented two symbolic conversions from a subset of the Java Virtual Machine Language (JVML, [LY97]) into IL. Although some manual intervention is currently required to convert JVM code into the format accepted by our conversion tool, an overall translation process is achieved accepting JAVA source code and emitting ALEF code. We exercised this compiler flow on the Linpack benchmark suite (Java version [DWM]), a numerically intensive linear equation solver commonly used for evaluating parallelising compilers and computer architectures [Don89].

The following section summarises the conversion schemes for translating JVML code into IL. Subsequently, Sections 8.2 and 8.3 present and discuss our experimental results.

8.1 Analysis of JVM bytecode

We briefly summarise the Java virtual machine language (JVML) before outlining how JVML code may be converted into IL.

8.1.1 Java virtual machine code

The semantics of JVML is defined (see [LY97]) by an operational model for bytecode instructions which manipulate an operand stack, local variables, an object heap, a call stack of method invocations and a set of available classes. Both translations concentrate on the bodies of methods, interpret the invocation of a method as a single symbolic instruction and aim at converting the operand stack carried values into forwardable entities. Ignoring call stack, heap and class-file environment, the semantics of an instruction can be approximated by triples of the form (s, LV, pc) where s is a stack of 32-bit values, LV a map from local variables to values and pc the program counter. Table 8.1 shows a subset of JVML's integer instructions and their semantic actions. Arguments i of $iinc$, $iload$ and $istore$ represent the index of the local variable to be manipulated, and the static semantics of JVM code ensures that variable i is a legal entry in LV. The instruction $ireturn$ pushes the top value of the current operand stack to the operand stack of the calling method and then returns, and the primed terms in Table 8.1 represent the respective components of the calling frame.

For converting JVML code into IL instructions, we considered two translation schemes which differ in the way IL variables are assigned to entities on the operand stack. Due to the limited instruction sets of both IL and ALEF, we decided to implement symbolic conversions which model the consumption of operands correctly but interpret some instructions in a functionally imprecise way. For example, the execution of Linpack programs requires the implementation of floating point instructions (of double precision), which our conversions replace by integer operations on symbolic data. Likewise, method invocations and function calls translate into symbolic consumption and creation of operands and return values rather than in assembly-level procedure calls. While this symbolic approach suffices for analysing the usage of operands, it precludes the possibility of executing the resulting code in a meaningful way.

Bytecode	*Semantics*
`iadd`	$(a\,b\,s, LV, pc) \rightarrow (b+a)\,s, LV, pc+1)$
`isub`	$(a\,b\,s, LV, pc) \rightarrow (b-a)\,s, LV, pc+1)$
`imul`	$(a\,b\,s, LV, pc) \rightarrow (b*a)\,s, LV, pc+1)$
`iinc` $i\ a$	$(s, LV, pc) \rightarrow (s, LV[i \mapsto (LV\ i)+a], pc+1)$
`iload` i	$(s, LV, pc) \rightarrow ((LV\ i)\,s, LV, pc+1)$
`istore` i	$(a\,s, LV, pc) \rightarrow (s, LV[i \mapsto a], pc+1)$
`iconst` a	$(s, LV, pc) \rightarrow (a\,s, LV, pc+1))$ where $a \in \{-1, \ldots, 5\}$
`ineg`	$(a\,s, LV, pc) \rightarrow ((-a)\,s, LV, pc+1)$
`dup`	$(a\,s, LV, pc) \rightarrow (a\,a\,s, LV, pc+1)$
`pop`	$(a\,s, LV, pc) \rightarrow (s, LV, pc+1)$
`goto` pc'	$(s, LV, pc) \rightarrow (s, LV, pc')$
`if_icmplt` pc'	$(a\,b\,s, LV, pc) \rightarrow \begin{cases} (s, LV, pc') & \text{if } b < a \\ (s, LV, pc+1) & \text{otherwise} \end{cases}$
`iflt` pc'	$(a\,s, LV, pc) \rightarrow \begin{cases} (s, LV, pc') & \text{if } a < 0 \\ (s, LV, pc+1) & \text{otherwise} \end{cases}$
`ireturn`	$(a\,s, LV, pc) \rightarrow (a\,s', LV', pc')$

Table 8.1: Operational semantics of some JVML instructions

Both translations introduce IL variables for operand stack communicated values and translate each JVML instruction into a sequence of IL instructions such that creation and consumption of stack values corresponds exactly to assignment and usage of these *stack variables*. Furthermore, the translations employ *local variables* for encoding JVM variables in *LV* and *auxiliary variables* for modelling the effect of method return and symbolic instruction execution. The three categories of IL variables are separated by syntactic means in IL, and we write $\mathtt{x_i}$ for stack variables, $\mathtt{y_i}$ for local variables and $\mathtt{w_i}$ for auxiliary variables. The forwardability analysis concentrates on variables of the first category.

8.1.2 Translating each stack position into a variable

The first translation (henceforth referred to as scheme A) statically assigns one IL variable to each position of the operand stack. Consequently, the number of stack variables employed coincides with the maximal height of the operand stack. This height is statically known at compile time of a JAVA program and explicitly available in a JVML class-file. For example, the JVM program

$$\begin{array}{ll} 0\ \mathtt{iload}\ 1 & 4\ \mathtt{iadd} \\ 1\ \mathtt{iconst}\ 2 & 5\ \mathtt{isub} \\ 2\ \mathtt{iload}\ 3 & 6\ \mathtt{iadd} \\ 3\ \mathtt{iconst}\ 3 & 7\ \mathtt{ireturn} \end{array} \tag{8.1}$$

uses an operand stack of height 4 and is translated into the IL program

$$\begin{array}{ll} [0]\mathtt{x}_0 = \mathtt{y}_1 & [4]\mathtt{x}_2 = \mathtt{x}_2 + \mathtt{x}_3 \\ [1]\mathtt{x}_1 = 2 & [5]\mathtt{x}_1 = \mathtt{x}_1 - \mathtt{x}_2 \\ [2]\mathtt{x}_2 = \mathtt{y}_3 & [6]\mathtt{x}_0 = \mathtt{x}_0 + \mathtt{x}_1 \\ [3]\mathtt{x}_3 = 3 & [7]\mathtt{w}_0 = \mathtt{x}_0 \end{array} \tag{8.2}$$

where each variable $\mathtt{x_i}$ corresponds to the stack position i. Local variables are translated bijectively into IL variables of the second category (see instructions [0] and [2]), while auxiliary variables are created as needed for the correct modelling of operand consumption (instruction [7]).

Bytecode	*IL code*	*sp update*
`iadd`	$\mathtt{x}_{sp-2} = \mathtt{x}_{sp-2} + \mathtt{x}_{sp-1}$	$sp := sp - 1$
`isub`	$\mathtt{x}_{sp-2} = \mathtt{x}_{sp-2} - \mathtt{x}_{sp-1}$	$sp := sp - 1$
`imul`	$\mathtt{x}_{sp-2} = \mathtt{x}_{sp-2} * \mathtt{x}_{sp-1}$	$sp := sp - 1$
`iinc` $i\ a$	$\mathtt{y}_{\mathtt{i}} = \mathtt{y}_{\mathtt{i}} + a$	–
`iload` i	$\mathtt{x}_{sp} = \mathtt{y}_{\mathtt{i}}$	$sp := sp + 1$
`istore` i	$\mathtt{y}_{\mathtt{i}} = \mathtt{x}_{sp-1}$	$sp := sp - 1$
`iconst` a	$\mathtt{x}_{sp} = a$	$sp := sp + 1$
`ineg`	$\mathtt{x}_{sp-1} = \sim \mathtt{x}_{sp-1}$	–
`dup`	$\left\{ \begin{array}{c} \mathtt{w}_0 = \mathtt{x}_{sp-1} \\ \mathtt{x}_{sp} = \mathtt{w}_0 \\ \mathtt{x}_{sp-1} = \mathtt{w}_0 \end{array} \right\}$	$sp := sp + 1$
`pop`	$\mathtt{w}_0 = \mathtt{x}_{sp-1}$	$sp := sp - 1$

Table 8.2: Translation scheme A for non-jumps

The translation inside each basic block follows the sequence of instructions using the translation scheme given in Table 8.2, where the additional unary IL operator $\sim$ denotes negation. The effect of instruction execution is modelled by internally maintaining the stack pointer *sp* of variables $\mathtt{x}_{\mathtt{i}}$. For each instruction, the context conditions given in [LY97] ensure that Table 8.2 is well-defined as the stack pointer never decreases beyond zero and is always greater than or equal to the highest index used in the corresponding line in Table 8.2. Most JVM instructions pop their operands off the stack during execution, which yields a single usage for most stack-communicated values. This pattern may be extended to stack-manipulating instructions `pop` and `dup` at the cost of additional assignments to auxiliary variables. Alternatively, one could implement `pop` without emitting any IL code by purely decrementing the stack pointer. Likewise, `dup` could be implemented by a single instruction $\mathtt{x}_{sp} = \mathtt{x}_{sp-1}$. The translation shown in Table 8.2 thus sacrifices runtime efficiency for the uniform consumption of stack variables and for the coherence with the behaviour of the corresponding ALEF instructions `skip` and `dupl`.

Bytecode	*IL code*	*sp update*
goto pc'	jmp pc'	–
if_icmplt pc'	$\left\{\begin{array}{c} \mathtt{w}_0 = \mathtt{x}_{sp-2} \\ \mathtt{w}_0 = \mathtt{x}_{sp-1} \\ \mathtt{if}\ \mathtt{w}_0\ pc'\ pc+1 \end{array}\right\}$	$sp := sp-2$
iflt pc'	$\left\{\begin{array}{c} \mathtt{w}_0 = \mathtt{x}_{sp-1} \\ \mathtt{if}\ \mathtt{w}_0\ pc'\ pc+1 \end{array}\right\}$	$sp := sp-1$
ireturn	$\left\{\begin{array}{c} \mathtt{w}_0 = \mathtt{x}_{sp-1} \\ \vdots \\ \mathtt{w}_0 = \mathtt{x}_0 \end{array}\right\}$	$sp := 0$

Table 8.3: Translation scheme A for jumps

At the end of basic blocks, programs compose correctly by the stipulation in the JVM specification that all control flow edges deliver operand stacks of the same height - indeed, even the stacks' shapes must coincide, i.e. the types of values at each stack position must agree. As our translation does not distinguish between values of different types, monitoring the height of the operand stack suffices. All conditional jump instructions are symbolically modelled by the IL conditional `if`, and the consumption of operands is translated into assignments to an auxiliary variable (Table 8.3).

Example. Consider the Java function `f1`

```
int f1(){
  int sum = 5;
  for (int i =0; i<5; i++){sum += 3;}
  return sum;
}
```
(8.3)

and its translation into JVML

0 iconst 5	4 goto 13	14 iconst 5
1 istore 1	7 iinc 1 3	15 if_icmplt 7
2 iconst 0	10 iinc 2 1	18 iload 1
3 istore 2	13 iload 2	19 ireturn

Inserting explicit unconditional jumps for the fall-through case of conditional jumps, the translation scheme A results in the IL program

$$
\begin{array}{rclcrcl}
\mathtt{N(0)} & = & [\mathtt{0}]\mathtt{x}_0 = \mathtt{5} & \quad & \mathtt{N(13)} & = & [\mathtt{13}]\mathtt{x}_0 = \mathtt{y}_2 \\
 & & [\mathtt{1}]\mathtt{y}_1 = \mathtt{x}_0 & & & & [\mathtt{14}]\mathtt{x}_1 = \mathtt{5} \\
 & & [\mathtt{2}]\mathtt{x}_0 = \mathtt{0} & & & & [\mathtt{15a}]\mathtt{w}_0 = \mathtt{x}_0 \\
 & & [\mathtt{3}]\mathtt{y}_2 = \mathtt{x}_0 & & & & [\mathtt{15b}]\mathtt{w}_0 = \mathtt{x}_1 \\
 & & [\mathtt{4}]\mathtt{jmp}\ \mathtt{13} & & & & [\mathtt{15}]\mathtt{if}\ \mathtt{w}_0\ \mathtt{7}\ \mathtt{18} \\
\mathtt{N(7)} & = & [\mathtt{7}]\mathtt{y}_1 = \mathtt{y}_1 + \mathtt{3} & & & & \\
 & & [\mathtt{10}]\mathtt{y}_2 = \mathtt{y}_2 + \mathtt{1} & & \mathtt{N(18)} & = & [\mathtt{18}]\mathtt{x}_0 = \mathtt{y}_1 \\
 & & [\mathtt{100}]\mathtt{jmp}\ \mathtt{13} & & & & [\mathtt{19}]\mathtt{w}_0 = \mathtt{x}_0
\end{array}
$$

The stack regime results in each assignment to a stack variable $\mathtt{x_i}$ being used linearly, as the $\mathtt{x}_0$-paths ([0], [1]), ([2], [3]), ([13], [14], [15a]) and ([18], [19]) and the $\mathtt{x}_1$-path ([14], [15a], [15b]) show. Indeed, both variables are forwardable to any functional unit as all consuming instructions are of the form $x = y$ and will be translated to `id` instructions in ALEF. ◇

All stack variables are used linearly as most JVM instructions pop their operands from the stack. Instructions whose sole purpose consists of modifying the stack (`pop` and `dup`) conform to this discipline by virtue of the assignments to auxiliary variables. However, not all stack positions translate into *forwardable* stack variables as variable names are extensively reused, and the respective values are consumed by instructions executing on different functional units.

Example. The program

```
int f2(){
  int sum = 5; int j = 2;
  for (int i =0; i<5; i++){j++; sum = sum+j;}
  return sum * j;
}
```
(8.4)

results in the JVML code

0 iconst 5	5 istore 3	14 iadd	21 if_icmplt 19
1 istore 1	6 goto 19	15 istore 1	24 iload 1
2 iconst 2	9 iinc 2 1	16 iinc 3 1	25 iload 2
3 istore 2	12 iload 1	19 iload 3	26 imul
4 iconst 0	13 iload 2	20 iconst 5	27 ireturn

which translates into

$$\begin{aligned}
\mathtt{N}(0) = \quad & [0]\ \mathtt{x}_0 = 5 & \mathtt{N}(9) = \quad & [9]\ \mathtt{y}_2 = \mathtt{y}_2 + 1 \\
& [1]\ \mathtt{y}_1 = \mathtt{x}_0 & & [12]\ \mathtt{x}_0 = \mathtt{y}_1 \\
& [2]\ \mathtt{x}_0 = 2 & & [13]\ \mathtt{x}_1 = \mathtt{y}_2 \\
& [3]\ \mathtt{y}_2 = \mathtt{x}_0 & & [14]\ \mathtt{x}_0 = \mathtt{x}_0 + \mathtt{x}_1 \\
& [4]\ \mathtt{x}_0 = 0 & & [15]\ \mathtt{y}_1 = \mathtt{x}_0 \\
& [5]\ \mathtt{y}_3 = \mathtt{x}_0 & & [16]\ \mathtt{y}_3 = \mathtt{y}_3 + 1 \\
& [6]\ \mathtt{jmp}\ 19 & & [100]\ \mathtt{jmp}\ 19
\end{aligned}$$

$$\begin{aligned}
\mathtt{N}(19) = \quad & [19]\ \mathtt{x}_0 = \mathtt{y}_3 & \mathtt{N}(24) = \quad & [24]\ \mathtt{x}_0 = \mathtt{y}_1 \\
& [20]\ \mathtt{x}_1 = 5 & & [25]\ \mathtt{x}_1 = \mathtt{y}_2 \\
& [21\mathrm{a}]\ \mathtt{w}_0 = \mathtt{x}_0 & & [26]\ \mathtt{x}_0 = \mathtt{x}_0 * \mathtt{x}_1 \\
& [21\mathrm{b}]\ \mathtt{w}_0 = \mathtt{x}_1 & & [27]\ \mathtt{w}_0 = \mathtt{x}_0 \\
& [21]\ \mathtt{if}\ \mathtt{w}_0\ 9\ 24 & &
\end{aligned}$$

The usages of $\mathtt{x}_0$ and $\mathtt{x}_1$ are linear - the dataflow equations have a solution with

$$\begin{aligned}
\mathit{Out}([0])(\mathtt{x}_0) &= \mathit{Out}([2])(\mathtt{x}_0) = \mathit{Out}([4])(\mathtt{x}_0) = \mathit{Out}([12])(\mathtt{x}_0) = \mathit{Out}([14])(\mathtt{x}_0) \\
&= \mathit{Out}([19])(\mathtt{x}_0) = \mathit{Out}([24])(\mathtt{x}_0) = \mathit{Out}([26])(\mathtt{x}_0) = 1
\end{aligned}$$

and

$$\mathit{Out}([13])(\mathtt{x}_1) = \mathit{Out}([20])(\mathtt{x}_1) = \mathit{Out}([25])(\mathtt{x}_1) = 1$$

but neither variable is forwardable due to the conflicting uses in $[14]\ \mathtt{x}_0 = \mathtt{x}_0 + \mathtt{x}_1$ and $[26]\ \mathtt{x}_0 = \mathtt{x}_0 * \mathtt{x}_1$. ◇

Translation according to scheme A consequently results in a low number of IL variables (and in a fast computation of dataflow solutions), but also limits the opportunities for forwarding.

Bytecode	internal stack (pre)	*IL code*	internal stack (post)
iadd	$x\,y\,S$	$\mathtt{x_v} = y + x$	$\mathtt{x_v}\,S$
isub	$x\,y\,S$	$\mathtt{x_v} = y - x$	$\mathtt{x_v}\,S$
imul	$x\,y\,S$	$\mathtt{x_v} = y * x$	$\mathtt{x_v}\,S$
iinc $i\ a$	S	$\mathtt{y_i} = \mathtt{y_i} + a$	S
iload i	S	$\mathtt{x_v} = \mathtt{y_i}$	$\mathtt{x_v}\,S$
istore	$x\,S$	$\mathtt{y_i} = x$	S
iconst a	S	$\mathtt{x_v} = a$	$\mathtt{x_v}\,S$
ineg	xS	$\mathtt{x_v} = \sim x$	$\mathtt{x_v}\,S$
dup	$x\,S$	$\left\{\begin{array}{l} \mathtt{w_0} = x \\ \mathtt{x_{v_1}} = \mathtt{w_0} \\ \mathtt{x_{v_2}} = \mathtt{w_0} \end{array}\right\}$	$\mathtt{x_{v_2}}\mathtt{x_{v_1}}\,S$
pop	$x\,S$	$\mathtt{w_0} = x$	S

Table 8.4: Translation scheme B for non-jumps

8.1.3 Translating each push operation into a variable

The second translation (referred to as scheme B) aims to make more variables forwardable. This is achieved by introducing a fresh IL variable for each assignment to a stack position. Instead of monitoring the height of the operand stack using a stack pointer, the translation internalises the stack behaviour by maintaining a stack of IL variables. For simple instructions, the translation scheme is given in Table 8.4 where in each line, $\mathtt{x_v}$ represents a globally fresh variable name of the stack variable category. Again, instructions pop and dup could be implemented more efficiently than by assignments to auxiliary variables. The translation results in an SSA-like usage of variables as no variable is (statically) assigned to repeatedly. For example, example program (8.1) from the previous section is translated into

$$\begin{array}{ll} [0]\mathtt{x_0} = \mathtt{y_1} & [4]\mathtt{x_4} = \mathtt{x_2} + \mathtt{x_3} \\ [1]\mathtt{x_1} = 2 & [5]\mathtt{x_5} = \mathtt{x_1} - \mathtt{x_4} \\ [2]\mathtt{x_2} = \mathtt{y_3} & [6]\mathtt{x_6} = \mathtt{x_0} + \mathtt{x_5} \\ [3]\mathtt{x_3} = 3 & [7]\mathtt{w_0} = \mathtt{x_6} \end{array} \tag{8.5}$$

Bytecode	internal stack (pre)	*IL code*	internal stack (post)
goto pc'	S	jmp pc'	S
if_icmplt pc'	xyS	$\left\{\begin{array}{c}\mathtt{w}_0 = y \\ \mathtt{w}_0 = x \\ \mathtt{if}\ \mathtt{w}_0\ pc'\ pc+1\end{array}\right\}$	S
iflt pc'	xS	$\left\{\begin{array}{c}\mathtt{w}_0 = x \\ \mathtt{if}\ \mathtt{w}_0\ pc'\ pc+1\end{array}\right\}$	S
ireturn	$xx_1 \ldots x_k$	$\left\{\begin{array}{c}\mathtt{w}_0 = x_k \\ \vdots \\ \mathtt{w}_0 = x_1 \\ \mathtt{w}_0 = x\end{array}\right\}$	ε

Table 8.5: Translation scheme B for jumps

if the creation of fresh $\mathtt{x}_i$ variables follows the natural numbers.

At basic block boundaries, compensation code is introduced similar to the one arising from the elimination of Φ-blocks, without prior explicit insertion of Φ-instructions. Again based on the JVM requirement that stacks delivered by separate control flow edges must match, a cascade of assignments $\mathtt{x_i} = \mathtt{x_j}$ is inserted where each variable on the left is (syntactically) the one expected by the following basic block and each x_j is the variable name at the same stack position as delivered by the current basic block. Jump instructions are translated as shown in Table 8.5. and a prior phase is employed to ensure (standard) edge-split form. For example, the Java function

```
final int abs (int i) {return (i >= 0) ? i : -i;}
```

results in JVM code

```
0 iload 1       8 iload 1
1 iflt 8        9 ineg
4 iload 1       10 ireturn
5 goto 10
```

which translates to

$$\begin{array}{rclrcl} \mathtt{N}(0) & = & [0]\mathtt{x}_0 = \mathtt{y}_1 & \mathtt{N}(8) & = & [8]\mathtt{x}_2 = \mathtt{y}_1 \\ & & [1]\mathtt{if}\ \mathtt{x}_0\ 8\ 4 & & & [9]\mathtt{x}_3 =\sim \mathtt{x}_2 \\ & & & & & [100]\mathtt{x}_1 = \mathtt{x}_3 \\ \mathtt{N}(4) & = & [4]\mathtt{x}_1 = \mathtt{y}_1 & & & [101]\mathtt{jmp}\ 10 \\ & & [5]\mathtt{jmp}\ 10 & \mathtt{N}(10) & = & [10]\mathtt{w}_0 = \mathtt{x}_1 \end{array}$$

The stack at the entry of block $\mathtt{N}(10)$ contains one element, and instruction $[100]$ is the glue instruction moving $\mathtt{x}_3$ to $\mathtt{x}_1$. Our implementation differs slightly from our theoretical treatment in Section 7.2 as basic block $\mathtt{N}(4)$ is not equipped with compensation code, and $\mathtt{N}(10)$ instead operates on the stack delivered by $\mathtt{N}(4)$ directly. Notice that in contrast to scheme A, most $\mathtt{x}_\mathtt{i}$ variables are now not only used linearly but also forwardable.

Example. Returning to the two example programs from the previous section, program (8.3) translates to

$$\begin{array}{rclrcl} \mathtt{N}(0) & = & [0]\mathtt{x}_0 = 5 & \mathtt{N}(13) & = & [13]\mathtt{x}_2 = \mathtt{y}_2 \\ & & [1]\mathtt{y}_1 = \mathtt{x}_0 & & & [14]\mathtt{x}_3 = 5 \\ & & [2]\mathtt{x}_1 = 0 & & & [15\mathtt{a}]\mathtt{w}_0 = \mathtt{x}_2 \\ & & [3]\mathtt{y}_2 = \mathtt{x}_1 & & & [15\mathtt{b}]\mathtt{w}_0 = \mathtt{x}_3 \\ & & [4]\mathtt{jmp}\ 13 & & & [15]\mathtt{if}\ \mathtt{w}_0\ 7\ 18 \\ \mathtt{N}(7) & = & [7]\mathtt{y}_1 = \mathtt{y}_1 + 3 & & & \\ & & [10]\mathtt{y}_2 = \mathtt{y}_2 + 1 & \mathtt{N}(18) & = & [18]\mathtt{x}_4 = \mathtt{y}_1 \\ & & [100]\mathtt{jmp}\ 13 & & & [19]\mathtt{w}_0 = \mathtt{x}_4 \end{array}$$

using scheme B, and all stack variables $\mathtt{x}_\mathtt{i}$ are used linearly and indeed forwardable to any functional unit. Likewise, program (8.4) translates to

$$\begin{aligned}
\mathtt{N}(0) &= [0]\ \mathtt{x}_0 = 5 & \mathtt{N}(9) &= [9]\ \mathtt{y}_2 = \mathtt{y}_2 + 1\\
&\ [1]\ \mathtt{y}_1 = \mathtt{x}_0 & &\ [12]\ \mathtt{x}_8 = \mathtt{y}_1\\
&\ [2]\ \mathtt{x}_1 = 2 & &\ [13]\ \mathtt{x}_9 = \mathtt{y}_2\\
&\ [3]\ \mathtt{y}_2 = \mathtt{x}_1 & &\ [14]\ \mathtt{x}_{10} = \mathtt{x}_8 + \mathtt{x}_9\\
&\ [4]\ \mathtt{x}_2 = 0 & &\ [15]\ \mathtt{y}_1 = \mathtt{x}_{10}\\
&\ [5]\ \mathtt{y}_3 = \mathtt{x}_2 & &\ [16]\ \mathtt{y}_3 = \mathtt{y}_3 + 1\\
&\ [6]\ \mathtt{jmp}\ 19 & &\ [100]\ \mathtt{jmp}\ 19\\
\mathtt{N}(19) &= [19]\ \mathtt{x}_3 = \mathtt{y}_3 & \mathtt{N}(24) &= [24]\ \mathtt{x}_5 = \mathtt{y}_1\\
&\ [20]\ \mathtt{x}_4 = 5 & &\ [25]\ \mathtt{x}_6 = \mathtt{y}_2\\
&\ [21a]\ \mathtt{w}_0 = \mathtt{x}_3 & &\ [26]\ \mathtt{x}_7 = \mathtt{x}_5 * \mathtt{x}_6\\
&\ [21b]\ \mathtt{w}_0 = \mathtt{x}_4 & &\ [27]\ \mathtt{w}_0 = \mathtt{x}_7\\
&\ [21]\ \mathtt{if}\ \mathtt{w}_0\ 9\ 24 & &
\end{aligned}$$

and all stack variables $\mathtt{x}_\mathtt{i}$ are used linearly and are indeed forwardable: $\mathtt{x}_5$ and $\mathtt{x}_6$ are forwardable to `MUL`, $\mathtt{x}_8$ and $\mathtt{x}_9$ to `ALU` and the remaining $\mathtt{x}_\mathtt{i}$ to any functional unit. ◇

In general, some stack variables remain non-forwardable in scheme B such as $\mathtt{x}_0$ in program

$$\begin{aligned}
\mathtt{N}(0) = {}& [0]\mathtt{x}_0 = \mathtt{y}_1 & \mathtt{N}(5) = {}& [5]\mathtt{x}_2 = 3 & \mathtt{N}(8) = {}& [8]\mathtt{x}_4 = 7\\
& [1]\mathtt{x}_1 = 2 & & [6]\mathtt{x}_3 = \mathtt{x}_0 + \mathtt{x}_2 & & [9]\mathtt{x}_5 = \mathtt{x}_0 * \mathtt{x}_4\\
& [2]\mathtt{if}\ \mathtt{x}_1\ 5\ 8 & & [7]\mathtt{w}_0 = \mathtt{x}_3 & & [10]\mathtt{w}_0 = \mathtt{x}_5
\end{aligned}$$

which results from the JVML program

```
0 iload 1      5 iconst 3     8 iconst 7
1 iconst 2     6 iadd         9 imul
2 iflt 8       7 ireturn      10 ireturn.
```

However, the experimental results in the following section indicate that this pattern of operand usage does not occur often in JVM code generated by `javac`.

8.1.4 Implemented instruction set

We implemented both translation schemes for an instruction set which extends the one given in Table 8.1 by (see Table 8.6)

instructions operating on 64-bit values: The definition of the JVM stipulates that values of datatypes `double` and `long` are stored using two operand stack positions of width 32 bit each. As both translation schemes relate single stack positions to IL variables, two variables are used for to represent a 64-bit value. Consequently, most instructions involving datatypes `double` and `long` result in a sequence of (at least) two IL assignments. Furthermore, the values of types `double` and `long` are represented symbolically by integer values (in both components) as functionally precise assembly instructions are not available in IL or ALEF.

instructions involving JVM references and arrays: References into the object heap are represented as 32-bit entities in JVML and are manipulated on the operand stack. As our conversions do not model the structure of the heap, instructions operating on references are again interpreted symbolically. Likewise, arrays of various datatypes are treated symbolically due to the absence of a memory component and references in IL.

additional stack-manipulating instructions: These are converted in a way similar to `pop`, `iconst` and `ireturn`. As mentioned above the conversion of duplication is such that linear usage is preserved, at the cost of an additional assignment to an auxiliary variable.

more variants of conditionals: Comparison operations and conditional branches are parameterised by the comparison operator ($\leq, <, =, \neq, \geq, >, \ldots$) used, but all variants are translated to IL's single `if` instruction.

instructions for object creation and field access: as the object heap is not modelled in IL, we convert `new` into the creation of a symbolic value representing the resulting object reference on the operand stack. Instructions `getfield`, `putfield` and `getstatic` operate on symbolic data.

Category	*Bytecodes*
64-bit operations	`dload`, `dstore`, `dconst`, `dneg`, `dmul`, `ddiv`, `dadd`, `dsub`, `dcmpl`, `dcmpg`
arrays and references	`anewarray`, `newarray`, `multianewarray`, `aconst_null`, `aaload`, `aastore`, `aload`, `astore`, `iaload`, `iastore`, `daload`, `dastore`
stack manipulation	`ldc`, `bipush`, `l2d`, `i2d`, `d2i`, `ldc2_w`, `i2l`, `dup2`, `sipush`,`return`, `dreturn`, `areturn`
conditionals	`if_acmp op`, `if op`, `if_icmp op`
object management	`new`, `getfield`, `putfield`, `getstatic`
method invocation	`invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual`,

Table 8.6: Implemented JVML instructions

method invocations: Depending on the number and types of arguments, and the type of the return value of the method being called, a varying number of operands is popped from / pushed onto the operand stack. As the signature of the called method is statically known at compile time, our translations insert as many assignments to auxiliary variables and stack variables as there are operands consumed and returned, respectively. Again, `double` and `long` operands are modelled by pairs of assignments.

This instruction set allows us to convert all but one of the fifteen methods of the Linpack benchmark suite (Java version). The remaining method `actionPerformed` uses exception handling and was therefore not considered.

8.2 Results

8.2.1 Conversion into IL

We first present results for the conversion from JVM to IL.

Method name	Basic blocks	Bytecode instrs	IL instrs	Stack variables total	linear	fwd	Local variables
abs	4	9	21	4	4	3	3
daxpy	16	88	134	9	9	3	13
ddot	15	85	114	7	7	1	13
dgefa	17	145	228	10	10	6	15
dgesl	21	191	321	12	12	7	13
dmxpy	7	30	52	8	8	4	8
dscal	10	46	77	6	6	2	9
epslon	4	31	69	5	5	1	11
idamax	29	132	207	4	4	3	12
matgen	19	89	155	6	6	5	10
Linpack	1	27	48	3	3	3	1
second	3	17	42	4	4	0	1
run_benchmark	16	244	472	7	7	2	22
init	1	383	602	5	5	5	7

Table 8.7: Linpack results for scheme A: Translation JVM → IL

Table 8.7 shows for each method the number of basic blocks and the number of instructions of the JVM code as obtained from `javac` in the leftmost columns. For translation scheme *A*, the fourth column gives the number of resulting IL instructions. The increase over the number of JVM instructions results from the conversion of (single) JVM instructions operating on values of type `double` into pairs of IL-instructions and from additional instructions involving auxiliary variables or symbolic data. The remaining columns show the total number of stack variables $\mathtt{x_i}$, the share of linearly used and forwardable stack variables, respectively, and the number of IL variables of the local-variable category. For example, method `abs` uses four stack variables, corresponding to the maximal operand stack height of the original JVM code. All four variables are used linearly each time they occur, and all but one are forwardable. Additionally, three local IL variables are used. The data given in the table confirms our characterisation of translation scheme *A*: all stack variables are used linearly as values

Method name	Basic blocks	Bytecode instrs	IL instrs	Stack variables total	linear	fwd	Local variables
abs	4	9	23	11	11	11	3
daxpy	16	88	134	82	82	82	13
ddot	15	85	114	72	72	72	13
dgefa	17	145	228	126	126	126	15
dgesl	21	191	321	176	176	176	13
dmxpy	7	30	52	28	28	28	8
dscal	10	46	77	43	43	43	9
epslon	4	31	69	43	43	43	11
idamax	29	132	215	119	119	119	12
matgen	19	89	157	75	75	75	10
Linpack	1	27	48	23	23	23	1
second	3	17	42	24	24	24	1
run_benchmark	16	244	476	280	280	280	22
init	1	383	602	300	300	300	7

Table 8.8: Linpack results for scheme B: Translation JVM → IL

are popped from the stack during their first usage. However, the repeated usage of variables for communicating different entities results in drastic variations in forwardability. For some methods, all stack variables are forwardable (`init`, `Linpack`), while other methods do not benefit from forwarding at all (`second`). On average, 57.3% of the stack variables are forwardable.

The same data for conversion scheme B is given in Table 8.8. While the first two columns agree with Table 8.7, the differences in the third column result from compensation code at the boundary of basic blocks in the under scheme B. The fourth column shows that scheme B uses far more stack variables. Again confirming our expectations, these are all used linearly and forwardable. Our results thus indicate that the operand stack may de facto be implemented using forwarding. This is of particular interest for compilation into native code as is performed by just-in-time compilers like HotSpot [Sun01] or high-performance compilers like Jalapeno [AAB+00].

Method name	ALEF instructions	Registers		Stack registers		Queues	
		no fwd	fwd	no fwd	fwd	no fwd	fwd
abs	21	9	6	4	1	0	2
daxpy	134	23	20	9	6	0	2
ddot	114	22	21	7	6	0	1
dgefa	228	26	20	10	4	0	6
dgesl	321	27	20	12	5	0	7
dmxpy	52	17	13	8	4	0	2
dscal	77	16	13	6	4	0	1
epslon	69	19	18	5	4	0	1
idamax	207	17	14	4	1	0	3
matgen	155	18	13	6	1	0	3
Linpack	48	6	3	3	0	0	1
second	42	8	8	4	4	0	0
run_benchmark	472	31	29	7	5	0	2
init	602	14	9	5	0	0	5

Table 8.9: Linpack results for scheme A: Translation JVM → ALEF

8.2.2 Translation into ALEF

IL code resulting from the two conversion schemes was translated into ALEF using the approach of Chapters 6 and 7. For scheme A, Table 8.9 shows the size of the resulting ALEF programs, and the number of registers and operand queues needed. Register usage is given as the total number of allocated registers and as registers employed for stack variables (*stack registers*). For comparison, we also give the data resulting from an allocation where all variables are mapped to registers, regardless of their forwardability. Columns marked *no fwd* thus indicate register requirements in traditional assembly code, while the column marked *fwd* contain the values where all forwardable variables are indeed mapped to operand queues. The table shows that for those cases where there are enough forwardable variables, few queues suffice to reduce the number of (stack) registers significantly.

The number of registers and queues allocated is a means to evaluate the *static* costs of forwarding in terms of hardware needed for a processor implementation. However, one of the motivations for introducing forwarding queues is the reduction of *dynamic* costs such as the energy consumption during access operations. Therefore, a more relevant performance measure is the number of read and write operations to registers and queues during program execution. As a realistic simulation of the emitted ALEF code is precluded by the symbolic nature of our translation, we decided to approximate the number of access operations using a symbolic interpretation which executes each instruction once. Although the resulting control flow does not represent a legal program execution, we expect that the number of access operations along this trace gives a rough indication of the benefits of forwarding. Table 8.10 presents the resulting data, again for each function of the Linpack suite separately. The first two columns contain the total number of register read and write operations (including registers used for local and auxiliary variables) for the translation where no forwarding is used. The following four columns show the effect of forwarding, and give the number of read and write operations to registers and operand queues, respectively. Notice that the total number of read and write operations for the forwarding and non-forwarding cases agree. Differences between the number of reads and writes for operand queues result from the hypothetical trace in combination with non-empty operand stacks at basic block boundaries in JVM programs.

Again, the performance benefit of translation scheme *A* varies across the different methods. Forwarding is (obviously) of no benefit if no forwardable variables exist (method `second`), and yields little performance benefit for programs with few forwardable stack variables (methods `dscal`, `ddot` and `epslon`). On the other hand, for JVM programs where a large percentage of operand stack positions can be turned into forwardable variables, more than half of the dynamic register access operations may be replaced by queue operations (method `Linpack`). On average, forwarding reduces the number of register read and write operations to 76.8% and 79.1%, respectively, of the corresponding values for the non-forwarding case.

For conversion scheme *B*, registers and queues allocated and the number of access operations are shown in Tables 8.11 and 8.12. Regarding the number of allocated

Method name	No forwarding Registers		Forwarding Registers		Forwarding Queues	
	R	W	R	W	R	W
abs	15	17	10	11	5	6
daxpy	133	118	115	100	18	18
ddot	114	99	112	97	2	2
dgefa	206	211	196	201	10	10
dgesl	291	300	234	243	57	57
dmxpy	48	45	32	29	16	16
dscal	76	67	72	63	4	4
epslon	61	65	59	63	2	2
idamax	173	178	114	115	59	63
matgen	126	136	81	90	45	46
Linpack	33	47	10	24	23	23
second	31	39	31	39	0	0
run_benchmark	417	456	401	440	16	16
init	448	601	148	301	300	300

Table 8.10: Register/queue access operations (scheme A)

Method	ALEF	Registers		Stack registers		Queues	
name	instructions	no fwd	fwd	no fwd	fwd	no fwd	fwd
abs	23	9	5	4	0	0	2
daxpy	134	23	14	9	0	0	4
ddot	114	22	15	7	0	0	4
dgefa	228	26	16	10	0	0	5
dgesl	321	27	15	12	0	0	5
dmxpy	52	17	9	8	0	0	4
dscal	77	16	10	6	0	0	5
epslon	69	19	14	5	0	0	5
idamax	215	17	13	4	0	0	2
matgen	157	18	12	6	0	0	3
Linpack	48	6	3	3	0	0	1
second	42	8	4	4	0	0	5
run_benchmark	476	31	24	7	0	0	6
init	602	14	9	5	0	0	2

Table 8.11: Linpack results for scheme B: Translation JVM $\longrightarrow$ ALEF

registers, Table 8.11 confirms that all stack registers may be replaced by forwarding queues and again, a low number of queues suffices. Also, if no forwarding is used, the number of registers holding stack variables (column five of Table 8.11) agrees with the height of the operand stack (column five of Tables 8.7 or 8.9). This is a consequence of the instruction order in the translated program and will be briefly discussed in the next section.

As all stack variables are forwarded, the effect on the dynamic number of access operations is even more favourable than that of scheme A. Table 8.12 indicates that stack variables are responsible for a large amount of the total dynamic register usage. Replacing these registers by forwarding eliminates at least half of the register read operations, with best case reductions of up to 75% (method `second`) and average reduction of 65%. At the same time, register write operations are reduced also by up to 75% (method `ddot`), with an average reduction of 62%.

8.3 Discussion

The results indicate that many opportunities for forwarding exist in application programs, which can be extracted via a compilation to JVML and subsequent conversion of the operand stack into forwarding. Our translation followed a simple pattern where the order of IL (and ALEF) instructions corresponds to the order of JVM instructions. While this is sufficient for analysing forwardability, it is wasteful regarding the number of registers and operand queues allocated, as these favour different instruction orders. A pathological example is the JVM instruction sequence

$$\mathtt{iconst}\ 1 \quad \ldots \quad \mathtt{iconst}\ 10 \quad \underbrace{\mathtt{iadd} \quad \ldots \quad \mathtt{iadd}}_{9}$$

which (for scheme B) results in IL code

$$\mathrm{x}_0 = 1 \quad \ldots \quad \mathrm{x}_9 = 10 \quad \mathrm{x}_{10} = \mathrm{x}_8 + \mathrm{x}_9 \quad \ldots \quad \mathrm{x}_{18} = \mathrm{x}_{17} + \mathrm{x}_0.$$

The operands are consumed in the reverse order of their creation. Consequently, the operand stack height cannot be converted into the length of a single operand queue, and ten queues are required. Reordering the IL instructions to

$$\mathrm{x}_8 = 9 \quad \mathrm{x}_9 = 10 \quad \mathrm{x}_{10} = \mathrm{x}_8 + \mathrm{x}_9 \quad \mathrm{x}_7 = 8 \quad \mathrm{x}_{10} = \mathrm{x}_8 + \mathrm{x}_{10} \ldots$$

Method name	No forwarding Registers		Forwarding Registers		Forwarding Queues	
	R	W	R	W	R	W
abs	17	19	6	6	11	13
daxpy	133	118	51	36	82	82
ddot	114	99	41	26	73	73
dgefa	206	211	80	85	126	126
dgesl	291	300	115	124	176	176
dmxpy	48	45	20	17	28	28
dscal	76	67	33	24	43	43
epslon	61	65	18	22	43	43
idamax	181	186	62	59	119	127
matgen	126	138	51	61	75	77
Linpack	33	47	10	24	23	23
second	31	39	7	15	24	24
run_benchmark	409	460	129	176	280	284
init	448	601	148	301	300	300

Table 8.12: Register/queue access operations (scheme B)

results in a program which needs two registers if no forwarding is used and one queue in the presence of forwarding. In addition to implementing more sophisticated allocation algorithms, a compiler may consequently reduce the number of registers and queues by reordering the IL instructions. The task of transforming a stack oriented instruction order into a queue oriented one resembles the task of optimising the evaluation order of expressions. We therefore expect that modifications of standard algorithms [ASU86] will lead to improvements in Tables 8.9 and 8.11. Furthermore, code optimisations as discussed in Section 7.7 and the usage of dedicated floating-point instructions should be beneficial for decreasing the instruction count and the number of operands communicated.

Finally, the exceptional case of scheme B where a variable is communicated to different branches and read by instructions executing on different functional units did not occur in our experiments. This may either be a characteristic of the chosen benchmark programs or represent an artefact of `javac`'s compilation strategy. Code inspection suggests that the latter is that case as `javac` uses the operand stack almost exclusively for references and intermediate values during the evaluation of expressions while Java program variables result in local JVM variables. Consequently, most basic blocks produced by `javac` operate on an initially empty stack and also deliver an empty stack. Further exploration of different compilation schemes of Java programs into JVM and development of direct translation from Java into IL are needed to evaluate the benefits of forwarding across basic block boundaries more thoroughly.

Chapter 9

Discussion

9.1 Summary

This thesis studied computational models resulting from the interaction between asynchronous processor operation and operand forwarding. We

- proposed to employ programming language concepts such as the separation into static and dynamic aspects using structural techniques
- unified existing architectures with forwarding by a novel computational model, asynchronous queue machines
- introduced an assembly language which makes forwarding explicit and supports the formal study of various dynamic models of operation, based on structured operational semantics
- demonstrated that typing calculi support the reasoning about forwarding schemes
- presented the program analysis for forwardability at the level of an intermediate language using dataflow equations
- exhibited a compilation approach which transforms intermediate language programs into assembly programs whose low-level structure is guaranteed to conform to the type systems

- supported the theoretical investigations with results obtained from a prototypical implementation.

To our knowledge, this work represents the first systematic investigation into the interactions between program analysis, asynchronous architectures and forwarding. Although we concentrated on very limited operational models and simplifying architectural abstractions, we argue that this thesis demonstrates the validity of the advocated approach. The implementational results show that despite these restrictions a compile-time analysis can uncover a relevant number of forwarding opportunities. This motivates the further development of mathematically rigorous reasoning techniques at the boundary between computer architecture, program analysis and optimisation, and the scheduling of communication.

9.2 Shortcomings and future work

We finally outline some topics for future research, ranging from more detailed implementations of the proposed calculi to more ambitious research into advanced program analysis and applications of typing frameworks.

9.2.1 Full implementation

Our implementation concentrated on the most basic analysis of regular programs and their translation into ALEF. A future implementation should realise some of the modifications mentioned at the end of Chapter 7 including multiple forwarding, balancing of branches with different usage patterns and compilation of SSA code. This implementation should interface to a compilation framework including other optimisation phases and better graph colouring algorithms. The target language should be an extended ALEF instruction set with integer and floating point operations, and the operational semantics should include cost measures regarding the latency and energy consumption of registers and operand queues. This would enable a detailed exploration of different compilation strategies and forwarding schemes for a high-level programming language. Benefits for just-in-time compilation of JVM code could subsequently be explored by turning the two symbolic conversion schemes into full translations.

9.2.2 Extensions of the computational model

The computational model of ALEF ignores several modern architectural issues, including speculation, branch prediction, and precise interrupts, and does not support procedure calls. In the future, the interaction of all these topics with forwarding should be explored, and we hope that formalisms such as the ones we presented will be useful for this task.

The forwarding capabilities of ALEF's computational model are less powerful than dynamic forwarding schemes of some hardware-based implementations. These restrictions appeared necessary for achieving an initial understanding of forwarding and for developing the presented reasoning formalisms. Relaxing these limitations could be supported by our techniques in three stages.

9.2.2.1 Dependent forwarding

As operand queue names and registers appear fixed in an ALEF instruction's opcode, the decision whether and to which queue a value should be forwarded must be made at compile-time. When motivating the introduction of registers, we saw that this scheme results in additional instructions if values are repeatedly accessed or used by different functional units. Operationally, it is not difficult to extend the instruction set to enable more flexible forwarding patterns. For example, assigning different destinations to forwarded values depending on the outcome of a branch may be modelled by splitting branch instructions into one instruction which computes the outcome of a branch condition and one instruction which actually performs the jump operation. The Fred architecture [RB96] employed a similar scheme, where code inside basic blocks is subsequently reordered to allow the former instruction to be computed earlier. As a result, instructions of the target basic blocks may be loaded from memory earlier, similar to our scheme of interleaving program execution and refilling of instruction pipelines in Chapter 4. Likewise, current architectures often store the branch status in a dedicated register inside the branch unit [HP96]. Making this value explicitly available to other instructions yields the above dependent forwarding. Unfortunately, incorporating this dependency in the static semantics appears not to be equally straight-forward. The interaction between control flow (outcome of branch conditions) and data flow (forwarding behaviour) may in principle be captured by dependent types such as the dependent function space with argument type `bool`. Future work is needed to see how this can be integrated into the rule for composing basic blocks and how the effect of successive branch instructions can be modelled correctly. As an alternative to dependent types in this context, union and intersection types should be explored.

9.2.2.2 Indirect forwarding

Additional forwarding flexibility may be obtained by introducing instructions which can issue forwarding request or modify the destination queue of other values or the queue in which an instruction expects its operands. At first, this may be achieved by interpreting some values as the index of operand queues. A more structured approach turns operand queue names into forwardable entities. Ultimately, this yields computational models with *dynamically programmable forwarding* where instructions may influence each other's forwarding behaviour by communicating appropriate queue names. Both schemes are expected to admit extremely powerful forwarding schemes which are difficult to reason about without formal support. On the other hand, type theory and concurrency theory offer calculi for combining types with computation, and for communicating names. These calculi might give guidance as to what restrictions need to be imposed on forwarding disciplines in order to obtain flexible and yet analysable behaviour.

9.2.2.3 Multi-clustered architectures

Advancing integration density offers the opportunity to implement multi-processor architectures on a single chip. For example, [ONH$^+$96]'s architecture contains several computational nodes, each equipped with some local memory. Nodes are connected by a communication network and may share data using global on-chip caches. Alternatively, multi-threaded architectures [TEL95] execute instructions from different processor contexts simultaneously. These may either compute a given task cooperatively by operating mainly independently and exchanging results only occasionally, or may stem from independent application programs. Understanding and verifying the issue logic and the interaction between different components will become more difficult as the number of components increases. In our asynchronous framework, nodes of such processors may be modelled as a collection of functional units and local forwarding capabilities, with additional operand queues and registers for global data exchange. The set of forwarding queues would consequently be partitioned into private queues for exchanging data within a node and public queues for communication between nodes. In order to support reasoning about concurrent execution in a compositional way, these abstraction barriers should be reflected in the type system, so that local forwarding may be reasoned about locally before the global communication is considered.

Again, similar topics have been studied in concurrency theory, and solutions such as restricting the visibility of (operand queue) names should enable one to incorporate the distinction public/private into the type system.

9.2.3 Connections to linear logic and other intermediate forms

We employed a rather intuitive interpretation of the linear logic-based notation, without attempting to establish deeper connections to formal proof theory. A more detailed exploration might not only help in modelling more complex forwarding schemes using other (linear) logical connectives, but might also allow us to relate executional models of different granularity. For example, different interleavings in the distributed model of execution resemble the structure of different derivations for the same final sequent. A particularly useful result would be a logics-based mechanism to relate the instruction-level interleaving to the fine-grained interleaving which arises from the small-step transition system of the micro-instructions *read* and *write*.

Since its inception [Gir86], linear logic has been seen as a logic for reasoning about resource consumption. Most widely known are its applications to concurrency formalisms such as the π-calculus [BS94] and Petri nets [MOM91] [EW90] [BG90], and to functional programming languages [Wad90] [TW99] [Bar96]. In particular, the latter work aims to detect linear usage of values in functional programs, similar to our usage analysis in Chapter 6. Due to the high level of functional programming languages, these calculi are not immediately applicable to the analysis of low-level architectural behaviour. However, if these calculi could be transferred from high-level programming language to functional intermediate or low-level forms [App98a], [FSDF93], forwardability analysis may be possible. In fact, functional intermediate forms and SSA are closely related [App98b], suggesting that it might be possible to obtain a structural correspondence between linearity types for a functional language and the dataflow solutions in our imperative intermediate language. A particular challenge will be to avoid the overhead of copying or moving operands at the boundaries of basic blocks which appears to arise in the translation of [App98b]. A possible compromise between functional abstraction and low-level behaviour might be to investigate the relation between λ-lifting/dropping [Dan99] and control flow properties such as the dominance property in the imperative intermediate language, similar to lightweight closures [SW97]. An alternative may be to follow the sequence of compilation steps of [MWCG98].

9.2.4 Explicitness of architectural features

The decision to model operand forwarding explicitly on the assembly level made the detailed calculus dependent on architectural configurations. We discussed the benefits of this design decision for reasoning and verification, as well as for compiler-based optimisation of operand communication. Future work should explore whether our calculi can be made parametric in configurations (see below) and how they apply to existing hardware-based forwarding schemes in more detail. Practically, algorithms for scheduling the communication of operands should be developed which are parametric in the performance and availability of individual forwarding paths. In addition to yielding specific compiler optimisation phases, the inclusion of performance characteristics in architectural descriptions would then allow one to study and compare architectural families abstractly by classifying them according to the performance of the various operand communication mechanisms.

More generally, future work should evaluate programming language based reasoning for other architectures, synchronous or asynchronous. As was alluded to in the discussion of Chapter 4, (virtually) reconfigurable architectures might be particularly well-suited due to their interaction between configurational changes and program execution. Other recent architectures which exposed the communication structure to the compiler and should thus profit from static semantics include the already mentioned TTA and RAW models of computation, or digital signal processors.

9.2.5 Formalisations

We expect that most of the material in Chapters 3 to 5 can be formalised in a state-of-the art theorem prover such as Isabelle/HOL or PVS. This might not only reveal more insight into the subtleties involved in relating the operational models to each other and to the static semantics, but would also link our work to formalisations of traditional hardware implementations of forwarding schemes such as Tomasulo's algorithm [McM98] [HSG98] and score-boards [MP96].

Our architectural models were restricted as mappings of instructions to functional units, number and types of functional units and bindings of operand and instruction queues to functional units were fixed. In contrast, a future compiler should be parametric in architectural descriptions of the target machine. At least two generalisations would be desirable. Firstly, verification should be modular in the *configuration* of an architectural scheme. For example, a system designer might need to explore the consequences of introducing a second functional unit of type `ALU` or of associating a different number of operand queues to a particular functional unit. Secondly, it may be desirable to vary the *architectural constraints*, for example by allowing functional units to be served by several instruction queues. Despite the conceptual simplicity, the entities of our models yield a high degree of design freedom: instruction queues might serve single or multiple functional units of identical or different type, operand queues might be associated to instruction queues or functional units (again either exclusively or non-exclusively), and the network might restrict the paths which a forwarded value may travel. Additional complexity arises if instructions may be allocated to instruction pipelines dynamically.

Evaluating the effect of any such architectural choice on the legality of forwarding schemes requires formal support. A type-theoretic solution may be to understand architectural configurations as contextual declarations which represent associations between entities of different categories:

$iq : fu$**:** an instruction queue for instructions executing on functional unit fu

fu : `ALU`**:** declaring functional unit fu to be of type `ALU`

$q : fu$**:** allowing operand queue q to hold values sent from any functional unit to fu

$q : fu' \to fu$**:** allowing operand queue q to hold values for the specific forwarding path $fu' \to fu$

$q : iq$**:** an operand queue whose entries can only be consumed by instructions in instruction queue iq

In a first step, conformance of a program to a specific configuration is expressed by including the context in the typing derivation. For example, a rule such as

$$\frac{\Gamma \vdash \mathtt{q}_1 : iq \quad \Gamma \vdash iq : fu \quad \Gamma \vdash fu : \mathtt{ALU}}{\Gamma \vdash [n]\mathtt{dec}\ \mathtt{q}_1\ \mathtt{q}_2 : X \otimes \mathtt{q}_1 \multimap X \otimes \mathtt{q}_2}$$

captures the fact that instruction $[n]\mathtt{dec}\,\mathtt{q}_1\,\mathtt{q}_2$ may only be mapped to an instruction queue iq which serves a functional unit of type ALU, where $\mathtt{q}_1$ is associated to iq. The three hypothesis are instantiations of a generic look-up rule

$$\frac{x : y \in \Gamma}{\Gamma \vdash x : y}$$

Consistency of configurations over the whole program is ensured in a cut rule

$$\frac{\Gamma \vdash s : A \multimap B \quad \Gamma \vdash t : B \multimap C}{\Gamma \vdash st : A \multimap C}$$

and a similar rule for combining basic blocks. Based on meta-theoretic results regarding substitution, weakening and strengthening, it may subsequently be possible to make the architectural configuration less opaque, and thus the typing calculi less dependent on exact architectural naming conventions.

In a second step, constraints on the *formation* of contexts may be used to classify architectural families. Examples are the restriction that instruction queues and functional units are in one-to-one correspondence or the stipulation that no forwarding path between two particular functional units exists.

This *logical frameworks*-perspective [HHP87] [Pfe01] should support the exploration of the relationship between architectural constraints and processor configurations and their consequences on forwarding schemes. In particular, we expect that serialisation constraints and their discharge may be incorporated in this framework by using judgements for pipeline-dependencies. Furthermore, such a formalisation would stress the connections between ALEF and the TIC/TAL/PCC work.

As mentioned in Chapter 2, we have had some initial success in formalising ALEF's types using Ohori's SCC [Oho99b]. The proof-theoretic foundations of transforming a language with a single, central stack and a single operational model (JVM) into one with several, distributed queues and multiple operational models (ALEF) are beyond the scope of this thesis, and left for future research. Notice however, that such an understanding might represent the formal basis of conversions between JVML and ALEF as presented in Chapter 8.

9.3 A future application

The need to verify structural correctness of assembly code has recently been recognised more widely as programs are dynamically (re)compiled and parts of applications are dynamically downloaded from untrustworthy sources. The concept of proof-carrying code (PCC) offers mathematically sound technology to enforce security properties and to ensure that performance requirements are met. In future, mobile applications will pose additional challenges regarding power consumption and resource capabilities of the underlying machinery. As statically fixed processor architectures appear unlikely to offer the flexibility necessary for applications of different domains, virtual architectures and reconfiguration will become more prevalent. In particular, it is anticipated that in addition to application programs, parts of configurational descriptions may only be available dynamically. This will require to specify available resources at abstract levels. Due to the inherent complexity of underlying hardware, reconfiguration and mobile code, formal verification techniques will be mandatory. A natural extension of the PCC approach under these conditions are *proof-carrying processor configurations* (PCPC). These bundle descriptions of configurations with formal proofs which are unforgeable, witness the sanity of the configurations with respect to the underlying hardware and certify the appropriateness for the intended application program.

Bibliography

[AAB+00] Bowen Alpern, Dick Attanasio, John J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, D. Lieber, V. Litvinov, Mark Mergen, Ton Ngo, J. R. Russell, Vivek Sarkar, Manuel J. Serrano, Janice Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, 39(1), February 2000.

[AC01] David Aspinall and Adriana Compagnoni. Heap bounded assembly language. Submitted, 2001.

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 243–253. ACM Press, January 2000.

[Aga97] Johan Agat. Types for register allocation. In Chris Clack, Kevin Hammond, and Tony Davie, editors, *Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97)*, volume 1467 of *Lecture Notes in Computer Science*, pages 92–111. Springer, September 1997.

[AM99] D. K. Arvind and Robert D. Mullins. A fully asynchronous superscalar architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 17–22. IEEE Computer Society Press, October 1999.

[AM00] D. K. Arvind and Robert D. Mullins. Instruction issue and data forwarding mechanisms for asynchronous superscalar processor. In *Proceedings of the Workshop on Complexity-Effective Design at the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, June 2000.

[Amj01] Hasan Amjad. BDD Representation Judgements in HOL: A Performance Evaluation. In Paul Jackson and Richard Boulton, editors, *Supplementary Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'01)*, Informatics Research Report EDI-INF-RR-0046. Division of Informatics, University of Edinburgh, UK, September 2001.

[Ana99] C. Scott Ananian. The Static Single Information Form. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1999.

[AP99] Tamarah Arons and Amir Pnueli. Verifying Tomasulo's algorithm by refinement. In *Proceedings of the 12th International Conference on VLSI Design (VLSI'99)*, pages 306–309. IEEE Computer Society, January 1999.

[App98a] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, Cambridge, UK, 1998.

[App98b] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, April 1998.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 247–258, June 2001.

[AS99] Arvind and Xiaowei Shen. Using Term Rewriting Systems to Design and Verify Processors. *IEEE Micro, Special Issue on Modeling and Validation of Microprocessors*, May/June 1999.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers*. Addison Wesley, 1986.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 1–11. ACM Press, New York, January 1988.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland, 1996.

[BBCZ98] Sergey Berezin, Armin Biere, Edmund Clarke, and Yunshan Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *Lecture Notes in Computer Science*. Springer, 1998.

[BCL+94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[BCS97] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software – Practice and Experience*, 27(6):701–724, June 1997.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Proceedings of the 6th International Conference on Computer-Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer, June 1994.

[Ber01] Lennart Beringer. Typing assembly programs with explicit forwarding. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Proceedings of* *the Fourth International Symposium on Theoretical Aspects of Computer Software (TACS'01)*, volume 2215 of *Lecture Notes in Computer Science*. Springer, October 2001.

[BFGW97] Howard Barringer, D. Fellows, Graham D. Gough, and A. Williams. Abstract modelling of asynchronous micropipeline systems using Rainbow. In C. D. Kloos and E. Cerny, editors, *Proceedings of the 13th IFIP WG 10.5 Conference on Computer Hardware Description Languages and Their Applications (CHDL'97)*, pages 285–304, April 1997.

[BG90] Carolyn Brown and Doug Gurr. A categorical linear framework for Petri nets. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 208–218. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[BG92] Gérard Berry and Georges Gonthier. The synchronous programming language Esterel: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BJQ00] Jonathan P. Bowen, He Jifeng, and Xu Qiwen. An animatable operational semantics of the Verilog hardware description language. In Shaoying Liu, John A. McDermid, and Michael G. Hinchey, editors, *Proceedings of the Third IEEE International Conference on Formal Engineering Methods (ICFEM'00)*, pages 199–207. IEEE Computer Society Press, September 2000.

[BKM96] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 275–293. Springer, November 1996.

[BM92] Rod Burstall and James McKinna. Deliverables: A categorical approach to program development in type theory. Technical Report ECS-LFCS-

92-242, Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland, October 1992.

[BM00] Graham Birtwistle and Matthew Morley. Case study: Specifying and checking tk, an asynchronous Amulet-like microprocessor. In *Workshop on Asynchronous INTerfaces: tools, techniques, and implementations (AINT'00)*, July 2000.

[BP96] Gianfranco Bilardi and Keshav Pingali. A framework for generalized control dependence. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*, pages 291–300, May 1996.

[BS94] G. Bellin and P. J. Scott. On the π-calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, December 1994.

[BS95] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits*. Springer, 1995.

[Bur69] Rodney M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[Bur87] Steven M. Burns. Automated Compilation of Concurrent Programs into Self-timed Circuits. Master's thesis, California Institute of Technology, Pasadena, California, December 1987.

[Car82] Luca Cardelli. *An Algebraic Approach to Hardware Description and Verification*. PhD thesis, Department of Computer Science, University of Edinburgh, April 1982.

[CCL$^+$97] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou. Automated verification with abstract state machines using multiway decision graphs. In Kropf [Kro97], pages 79–113.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[CM91] Henk Corporaal and Hans Mulder. MOVE: A framework for high-performance processor design. In Anne Copeland MacCallum, editor, *Proceedings of the 4th Annual Conference on Supercomputing*, pages 692–701. IEEE Computer Society Press, November 1991.

[CM99a] Antonio Cerone and George J. Milne. A methodology for the formal analysis of asynchronous micropipelines. Technical Report 99-33, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072, Australia, October 1999.

[CM99b] Karl Crary and Greg Morrisett. Type structure for low-level programming languages. In Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Proceedings of the 26th International Colloquium on Automata, Languages, and Programming (ICALP'99)*, volume 1644 of *Lecture Notes in Computer Science*, pages 40–54. Springer, July 1999.

[CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M.K. Srivas. Effective theorem proving for hardware verification. In T. Kropf and R. Kumar, editors, *Proceedings of the Second International Conference on Theorem Provers in Circuit Design (TPCD'94)*, volume 901 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, September 1994. Springer.

[Cur92] Paul Curzon. A verified compiler for a structured assembly language. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*. IEEE Computer Society Press, 1992.

[Cyr93] David A. Cyrluk. Microprocessor Verification in PVS: A Methodology and Simple Example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, Stanford Research Institute, December 1993.

[Dan99] Olivier Danvy. An Extensional Characterization of Lambda-Lifting and Lambda-Dropping. Technical Report BRICS-RS-99-21, Basic Research Institute in Computer Science, Aarhus, Denmark, 1999.

[DGY93] Ilana David, Ran Ginosar, and Michael Yoeli. Self-timed architecture of a reduced instruction set computer. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies,* volume A-28 of *IFIP Transactions*, pages 29–43. Elsevier Science Publishers, 1993.

[DN97] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, Department of Computer Science, University of Utah, September 1997.

[Don89] Jack J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report 89-85, Department of Computer Science, University of Tennessee, Knoxville, TN, October 1989.

[DP97] Werner Damm and Amir Pnueli. Verifying out-of-order executions. In Hon F. Li and David K. Probst, editors, *Advances in Hardware Design and Verification: Proceedings of the IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME'97)*, pages 23–47. Chapman & Hall, October 1997.

[DWM] Jack Dongarra, Reed Wade, and Paul McMahan. Linpack benchmark – Java version. http://www.netlib.org/benchmark/linpackjava.

[End96] Philip B. Endecott. *SCALP: A Superscalar Asynchronous Low-Power Processor*. PhD thesis, Department of Computer Science, University of Manchester, February 1996.

[EW90] U. Engberg and G. Winskel. Petri nets as models of linear logic. In André Arnold, editor, *Proceedings of the 15th Colloquium on Trees in Algebra and Programming (CAAP'90)*, volume 431 of *Lecture Notes in Computer Science*, pages 147–161. Springer, May 1990.

[FDG+96] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and S. Temple. AMULET2e. In C. Muller-Schloer, F. Geerinckx, B. Stanford-Smith, and R. van Riet, editors, *Proceedings the 6th Annual OMI Conference on Embedded Microprocessor Systems (EMSYS'96)*, September 1996.

[FLT97] Marcio Merino Fernandes, Josep Llosa, and Nigel Topham. Using queues for register file organization in VLIW architectures. Technical Report ECS-CSG-29-97, Computer Systems Group, Department of Computer Science, University of Edinburgh, Scotland, February 1997.

[FLT98] Marcio Merino Fernandes, Josep Llosa, and Nigel Topham. Extending a VLIW architecture model. Technical Report ECS-CSG-34-97, Computer Systems Group, Department of Computer Science, University of Edinburgh, Scotland, January 1998.

[Fly95] Michael J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones & Bartlett Publishing, 1995.

[FM98] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java$^{\text{tm}}$ bytecode language. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 310–328. ACM Press, October 1998.

[FM99] Stephen N. Freund and John C. Mitchell. A formal framework for the Java bytecode language and verifier. In *Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 147–166, October 1999.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93), Albuquerque, NM, USA*, volume 28(6) of *ACM SIGPLAN Notices*, pages 237–247. ACM Press, 1993.

[GFC99] J. D. Garside, S. B. Furber, and S.-H. Chung. AMULET3 revealed. In *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'99)*, pages 51–59. IEEE Computer Society Press, April 1999.

[GG97] D. A. Gilbert and J. D. Garside. A result forwarding mechanism for asynchronous pipelined systems. In *Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'97)*, pages 2–11. IEEE Computer Society Press, April 1997.

[Gir86] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 46:1–102, 1986.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Cetin K. Koc, David Naccache, and Christof Paar, editors, *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.

[Goo93] Kees G.W. Goosens. *Embedding Hardware Description Languages in Proof Systems*. PhD thesis, Department of Computer Science, University of Edinburgh, 1993.

[HA99] James C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. In *Proceedings of the Tenth International Conference on VLSI (VLSI'99)*, Lisbon, Portugal, December 1999.

[Hau95] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[HBB94] Henrik Hulgaard, Steven M. Burns, and Gaetano Borriello. Testing asynchronous circuits: A survey. Technical Report 94-03-06, Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195, USA, March 1994.

[Hei95] Nevin Heintze. Control-flow analysis and type systems. In *Proceedings of the International Static Analysis Symposium (SAS'95)*, volume 983 of *Lecture Notes in Computer Science*, pages 189–206. Springer, September 1995.

[HGS99] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. In *Proceedings of the International Conference on Correct Hardware and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 8–22. Springer, September 1999.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the 2nd Annual IEEE Symposium on Logic in Computer Science (LICS'87), Ithaca, NY*, pages 194–204. IEEE Computer Society Press, June 1987.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HSG98] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Decomposing the Proof of Correctness of Pipelined Microprocessors. In Alan J. Hu and Moshe Y. Vardi, editor, *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*. Springer, June 1998.

[HSG99] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 47–59. Springer, July 1999.

[HT98] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java virtual machine subroutines. In Giorgio Levi, editor, *Proceeding of the 5th International Symposium on Static Analysis (SAS'98)*, volume 1503 of *Lecture Notes in Computer Science*, pages 17–32. Springer, September 1998.

[JM01] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 396–410. Springer, March 2001.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In Robert Cartwright, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'93)*, pages 78–89. ACM Press, June 1993.

[JT01] Alexander H. Jackson and Andrew M. Tyrrell. Asynchronous embryonics with reconfiguration. In *Proceedings of the 4th International Conference on Evolvable Systems*, volume 2210 of *Lecture Notes in Computer Science*. Springer, October 2001.

[KCL+99] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, November 1999.

[KMP99] Daniel Kröning, Silvia M. Müller, and Wolfgang Paul. A rigorous correctness proof of the Tomasulo scheduling algorithm with precise interrupts. In *Proceedings of he 3rd World Multiconference on Systemics, Cybernetics and Informatics (SCI'99) and the 5th International Conference on Information Systems, Analysis and Synthesis (ISAS'99)*, July 1999.

[KO01] Shin-ya Katsumata and Atsushi Ohori. Proof-Directed De-compilation of Low-Level Code. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming, (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 352–366. Springer, April 2001.

[Kro97] Thomas Kropf, editor. *Formal Hardware Verification: Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*. Springer, 1997.

[Laf88] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[LBM96] Ying Liu, Graham Birtwistle, and Matthew Morley. Specifying and property checking the Amulet1 address interface. In *Designing Correct Circuits*, September 1996.

[LGS95] Trevor W. S. Lee, Mark R. Greenstreet, and Carl-Johan Seger. Automatic verification of asynchronous circuits. *IEEE Design & Test of Computers*, Spring 1995:24–30, 1995. Extended version appeared as Technical Report TR93-40, Department of Computer Science, University of British Columbia, Vancouver, Canada.

[LT79] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flow graph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[MA00] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher order logic. In David A. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE'00)*, volume 1831 of *Lecture Notes in Computer Science*, pages 7–24. Springer, June 2000.

[MAK00] Simon W Moore, Ross J Anderson, and Markus G Kuhn. Improving smartcard security using self-timed circuit technology. In *Fourth ACiD-WG Workshop*, February 2000.

[Man00] Rajit Manohar. A case for asynchronous computer architecture. In *Proceedings of the Workshop on Complexity-Effective Design at the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, June 2000.

[Mar86] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Federick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *Proceedings of the Second ACM SIGPLAN Workshop on Compiler Support for System Software (WCSSS'99)*, pages 25–35, May 1999.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, volume 1473 of *Lecture Notes in Computer Science*. Springer, March 1998.

[McM98] Kenneth L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In Alan J. Hu and Moshe Y. Vardi, editor, *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*. Springer, June 1998.

[McM00] Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1–3):279–309, May 2000.

[Mel88] Tom F. Melham. Using recursive types to reason about hardware and higher order logic. In G.J. Milne, editor, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 27–50. North-Holland, July 1988.

[MHC97] Jon Mountjoy, Pieter Hartel, and Henk Corporaal. Modular Operational Semantic Specification of Transport Triggered Architectures. In *Proceedings of the 13th IFIP WG 10.5 Conference on Computer Hardware Description Languages and Their Applications (CHDL'97)*, pages 260 – 279. Chapman and Hall, London, April 1997.

[MLM+97] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997.

[MLM99] Rajit Manohar, Tak-Kwan Lee, and Alain J. Martin. Projection: A synthesis technique for concurrent systems. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'99)*, pages 125–134, April 1999.

[MOM91] N. Marti-Oliet and J. Meseguer. From Petri nets to linear logic through categories: A survey. *International Journal of Foundations of Computer Science (IJFCS)*, 2(4):297–400, December 1991.

[Moo98] J. Strother Moore. Symbolic simulation: An ACL2 approach. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *Lecture Notes in Computer Science*. Springer, 1998.

[MP96] Silvia M. Müller and Wolfgang J. Paul. Making the original scoreboard mechanism deadlock free. In *Proceedings of the Fourth Israel Symposium on Theory of Computing and Systems (ISTCS'96)*, pages 92–99. IEEE Computer Society Press, 1996.

[MRW96] Simon Moore, Peter Robinson, and Steve Wilcox. Rotary pipeline processors. In *IEE Proceedings on Computers and Digital Techniques*, volume 143 (5), pages 259–265, September 1996.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[Mul01] Robert D. Mullins. *Dynamic instruction scheduling and data forwarding in asynchronous superscalar processors*. PhD thesis, Division of Informatics, University of Edinburgh, 2001.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 85–97. ACM Press, January 1998.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, January 1997.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[Oho99a] Atsushi Ohori. A Curry-Howard isomorphism for compilation and program execution. In Jean-Yves Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, volume 1581 of *Lecture Notes in Computer Science*. Springer, April 1999.

[Oho99b] Atsushi Ohori. The logical abstract machine: A Curry-Howard isomorphism for machine code. In *Proceedings of the 4th Fuji International*

Symposium on Functional and Logic Programming (FLOPS'99), volume 1722 of *Lecture Notes in Computer Science*, pages 300–318. Springer, November 1999.

[OIU+01] Motokazu Ozawa, Masashi Imai, Voichiro Ueno, Hiroschi Nakamura, and Takashi Nanya. Performance evaluation of Cascade ALU architecture for asynchronous super-scalar processors. In *Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'01)*, March 2001.

[ONH+96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. *ACM SIGPLAN Notices*, 31(9):2–11, September 1996. Co-published as SIGOPS Operating Systems Review **30**(5), December 1996, and as SIGARCH Computer Architecture News, **24**(special issue), October 1996.

[OZGS99] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally verifying IEEE compliance of floating-point hardware. In *Intel Technology Journal*, volume Q1. Intel Corporation, 1999.

[Pav94] Nigel Charles Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, 1994.

[PCHS00] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Department of Computer Science, Carnegie-Mellon University, December 2000.

[Pfe01] Frank Pfenning. Logical frameworks—a brief introduction. In *Proceedings of the Marktoberdorf NATO Summer School*, July 2001.

[PHA98] Lisa A. Poyneer, James C. Hoe, and Arvind. A TRS Model for a Modern Microprocessor. Technical Report Computation Structures Group Memo

408, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1998.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, University of Aarhus, September 1981.

[PP98] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages, San Diego (POPL'98), California*, pages 197–208, 1998.

[RB95] William F. Richardson and Erik Brunvand. Fred: An architecture for a self-timed decoupled computer. Technical Report UUCS-95-008, Department of Computer Science, University of Utah, May 1995.

[RB96] William F. Richardson and Erik Brunvand. Architectural considerations for a self-timed decoupled processor. In *IEE Proceedings on Computers and Digital Techniques*, volume 143 (5), pages 251–257, September 1996.

[Ric96] William F. Richardson. *Architectural considerations in a Self-Timed Processor Design*. PhD thesis, Department of Computer Science, University of Utah, February 1996. Also published as Technical Report CSTD-96-001.

[RRS00] Sergei Romanenko, Claudio Russo, and Peter Sestoft. *Moscow ML Owner's Manual, Version 2.00*, June 2000. Available from http://www.dina.kvl.dk/∼sestoft/mosml.html.

[SA95] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 116–129, June 1995.

[SA98a] Xiaowei Shen and Arvind. Design and Verification of Speculative Processors. In *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems (FTH'98) at the Fourth International Conference on Mathematics of Program Construction (MPC'98)*, June 1998.

[SA98b] Xiaowei Shen and Arvind. Modelling and Verification of ISA implementations. In *Proceedings of the Australasian Computer Architecture Conference*, February 1998.

[SA98c] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 149–160. ACM Press, 1998.

[SB93] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. Technical Report UBC-TR-93-08, Department of Computer Science, University of British Columbia, July 1993.

[SBN82] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples*. McGraw-Hill, 1982.

[Seg93] Carl-Johan H. Seger. An introduction to formal hardware verification. Technical Report UBC-TR-92-113, Department of Computer Science, University of British Columbia, 1993.

[SH98] Jun Sawada and Warren A. Hunt. Processor verification with precise exceptions and speculative execution. In Alan J. Hu and Moshe Y. Vardi, editor, *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 135–146. Springer, June 1998.

[Sha01] Zhong Shao. Type-based certifying compilation. In *Tutorial presented at the ACM SIGPLAN Conference on Programming Language*

Design and Implementation (PLDI'01), June 2001. Available from http://flint.cs.yale.edu/flint/publications.

[SJD98] J. U. Skakkebaek, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In Alan J. Hu and Moshe Y. Vardi, editor, *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 98–109. Springer, June 1998.

[SKLY97] A. Semenov, A. M. Koelmans, L. Lloyd, and A. Yakovlev. Designing an asynchronous processor using Petri nets. *IEEE Micro*, 17(2):54–64, 1997.

[SM95] M.K. Srivas and S.P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, 1995.

[SRC97] Mandayam Srivas, Harald Rueß, and David Cyrluk. Hardware verification using PVS. In Kropf [Kro97], pages 156–205.

[SRU01] Juric Silc, Borut Robic, and Theo Ungerer. Asynchrony in parallel computing: from dataflow to multithreading. In Frank Columbus, editor, *Progress in Computer Research*, volume 1, pages 1–33. Nova Science Publ., 2001. Also published as Technical Report CSD-TR-97-4, Jozef Stefan Institute, University of Ljubljana.

[SS99] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 443–454. Springer, July 1999.

[SSM94] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. Counterflow pipeline processor architecture. Technical Report SMLI-TR-94-25, Sun Microsystems Research, April 1994.

[SSTP02] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 217–232, January 2002.

[Sun01] Sun Microsystems. *The Java HotSpot Virtual Machine (Technical White Paper)*, 2001. http://java.sun.com/products/hotspot.

[Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. The 1988 Turing Award Lecture.

[SW97] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, January 1997.

[SWG94] Eric Stoltz, Michael Wolfe, and Michael P. Gerlek. Constant propagation: a fresh, demand-driven look. In Ed Deaton, Dave Oppenheim, Joseph Urban, and Hal Berghel, editors, *Proceedings of the 1994 ACM Symposium on Applied Computing (SAC'94)*, pages 400–404. ACM Press, 1994.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TE01] Krishnaprasad Thirunarayan and Robert L. Ewing. Structural operational semantics for a portable subset of behavioral VHDL-93. *Formal Methods in System Design*, 18(1):69–88, 2001.

[TEL95] Dean M. Tullesen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, volume 23(6) of *ACM SIGARCH Computer Architecture News*, pages 392–403, June 1995.

[Thi98] Peter Thiemann. Formalizing resource allocation. In *Proceedings of the Workshop on Types in Compilation (TIC'98)*, volume 1473 of *Lecture Notes in Computer Science*. Springer, March 1998.

[Tho64] James E. Thornton. Parallel operation in the control data 6600. In *Proceedings of the Fall Joint Computer Conference*, volume 26(2), pages 33–40. American Federation of Information Processing Society, 1964. Reprinted in [SBN82], pages 730–742.

[TKM$^+$02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, 2(2):25–35, March/April 2002. Also available at http://www.cag.lcs.mit.edu/raw/documents/.

[TMC$^+$96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*, volume 31(6) of *SIGPLAN Notices*. ACM Press, June 1996.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and development*, 11:25–33, January 1967.

[TW99] David N. Turner and Philip Wadler. Operational interpretations of Linear Logic. *Theoretical Computer Science*, 227:231–248, September 1999.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceeding of the 7th International Conference on Functional Programming and Computer Architecture*, pages 1–11. ACM Press, June 1995.

[Uri00] Tomás E. Uribe. Combinations of model checking and theorem proving. In *Proceedings of the Third International Workshop on Frontiers of Combining Systems (FROCOS'00)*, volume 1794 of *Lecture Notes in Computer Science*, pages 151–170. Springer, March 2000.

[VBF+97] W. Visser, H. Barringer, D. Fellows, G. Gough, and A. Williams. Efficient CTL* model checking for analysis of rainbow designs. In H. F. Li and D. K. Probst, editors, *Advances in Hardware Design and Verification: Proceedings of the IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME'97)*. Chapman and Hall, October 1997.

[vT93] John Peter van Tassel. *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant*. PhD thesis, Computer Laboratory, University of Cambridge, 1993.

[WA01] Tony Werner and Venkatesh Akella. An Asynchronous Superscalar Architecture for Exploiting Instruction-Level Parallelism. In *Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'01)*, March 2001.

[Wad90] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, pages 561–581. North Holland, 1990.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.

[XH01] Hongwei Xi and Robert Harper. A dependently typed assembly language. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, volume 36, 10 of *ACM SIGPLAN notices*, pages 169–180. ACM Press, September 2001.

[YGL00] Alex Yakovlev, Luis Gomes, and Luciano Lavagno, editors. *Hardware Design and Petri Nets*. Kluwer Academic Publishers, February 2000.