An Algebraic Semantics of Prolog Control

Brian James Ross

Doctor of Philosophy

University of Edinburgh

1992

Abstract

The conceptual distinction between logic and control is an important tenet of logic programming. In practice, however, logic program languages use control strategies which profoundly affect the computational behaviour of programs. For example, sequential Prolog's depth–first–left–first control is an unfair strategy under which nontermination can easily arise if programs are ill–structured. Formal analyses of logic programs therefore require an explicit formalisation of the control scheme. To this ends, this research introduces an algebraic process semantics of sequential logic programs written in Milner's Calculus of Communicating Systems (CCS). The main contribution of this semantics is that the control component of a logic programming language is concisely modelled. Goals and clauses of logic programs correspond semantically to sequential AND and OR agents respectively, and these agents are suitably defined to reflect the control strategy used to traverse the AND/OR computation tree for the program. The main difference between this and other process semantics which model concurrency is that the processes used here are sequential. The primary control strategy studied is standard Prolog's left–first–depth–first control. CCS is descriptively robust, however, and a variety of other sequential control schemes are modelled, including breadth–first, predicate freezing, and nondeterministic strategies. The CCS semantics for a particular control scheme is typically defined hierarchically. For example, standard Prolog control is initially defined in basic CCS using two control operators which model goal backtracking and clause sequencing. Using these basic definitions, higher–level bisimilarities are derived, which are more closely mappable to Prolog program constructs. By using various algebraic properties of the control operators, as well as the stream domain and theory of observational equivalence from CCS, a programming calculus approach to logic program analysis is permitted. Some example applications using the semantics include proving program termination, verifying transformations which use cut, and characterising some control issues of partial evaluation. Since process algebras have already been used to model concurrency, this thesis suggests that they are an ideal means for unifying the operational semantics of the sequential and concurrent paradigms of logic programming.

Acknowledgements

I owe thanks to:

- The University of Edinburgh for support through a University of Edinburgh Postgraduate Studentship and an Overseas Research Student (ORS) award. I thank Paul Wilk and Jim Howe for supporting my applications for these scholarships, without which I would have been unable to study in Edinburgh.

- My advisor, Alan Smaill. I cannot overstate how much I have appreciated his knowledgable direction and enthusiasm, and for believing in what I was doing. Without Alan, this thesis would have greatly suffered.

- My examiners, Pat Hill and Stuart Anderson, for their suggestions for improvements to the thesis; my other advisors, Paul Wilk and Colin Stirling, for their input; Chris Tofts, a genuine CCS guru (and a patient one) whose advice was indispensable; and Harvey Abramson, who gave some initial hints on how to proceed.

- Emilio Agustin, Alan Black, Andy Bowles, Rolando Carrera-Sanchez, Matthew Crocker, Carla Gomes, Ian Lewin, Nelson Ludlow, Suresh Manandhar, Dave Moffat, Peter Nowell, Robert Scott, Maria Vargas-Vera, Eva Bayerlein, Maria Cabral, Regina and Alvaro Fernandes, Bob Kemp, Franziska Maier, Jussi Stader, Rajiv Trehan, Joke van de Plassche, and many others, for the good times.

- Two special friends: Flávio Corrêa da Silva, a friend and companion who made the last year of my degree a lot of fun; and Mike Cramer, whose compassion, intelligence, and humour is always a source of inspiration.

- My parents, Jack and Marlene Ross, sisters Susanne and Leanne, and grandmother Vera Pearce, for their love and support.

- Three institutions which saved my sanity: the Pleasance gym with its karate club and excellent Nautilus facilities; the Usenet; and The Mambo Club, for improving my dancing and perception.

Declaration

I declare that this thesis has been composed by myself and that, unless otherwise stated, the work described in it is my own.

Some of the work contained in this thesis has appeared before in a different form (Ross 1991*c*) (Ross and Smaill 1991) (Ross 1991*a*) (Ross 1991*b*).

Brian J. Ross

Contents

List of Figures

Chapter 1

Introduction

Formal methods of software engineering attempt to introduce mathematical rigour into the development and analysis of computer software. The main motivation of the formal approach is to make software verifiably more reliable, which is a quality sorely lacking in informal *ad hoc* software development. A second motivation is to form the necessary foundations for sound computer–based automatic programming and verification systems, which require the entire programming environment to be formally modelled. To this end, there has been much attention given toward developing new formal methods of program specification, derivation, and verification, as well as developing high–level programming languages.

Logic programming is an active research area which is making contributions towards formal software engineering and high–level programming language design. A tenet of logic programming is that mathematical logic can be used as a practical programming language, and hence declarative logic programs may act as their own specifications (Kowalski 1974). A declarative logic program inherently reflects a high–level human–oriented statement of what the program is intended to compute. This contrasts to von Neumann languages, where the relationship between programs and their specifications is much less direct, and which require formal language semantics to help map programs to their intended declarative meanings. This semantic mapping stage is unnecessary with declarative logic programs, since they directly specify the relation they compute. In addition, logic programming conceptually distinguishes the declarative logic from the control (Kowalski 1979). This permits the specification inherent in a logic program to be analysed without regard to the particular refutation schemes used to infer results from it.

Most practical logic programming languages use simple refutation procedures which are efficient to implement. For example, Prolog's standard refutation procedure uses a depth–first search rule and a left–to–right computation rule, which is efficiently implemented on current computer hardware[1]. Unfortunately, this strategy is unfair: the depth–first search can proceed down infinite branches of the search tree, to the detriment of computational completeness. In addition, Prolog's standard strategy selects clauses and goals by their order within the program, and computations therefore depend upon the textual form of programs. As a consequence, logic programs must be tailored with the effects of this control strategy in mind, or else inefficient or non–terminating computations can arise. The problem is further compounded when extra–logical control features like the cut are considered. All of these operational phenomena of Prolog detract from the declarative aspirations of logic programming.

Some examples of the idiosyncracies of Prolog are as follows. Consider the following simple Prolog program:

$ancestor(X, Y) :- parent(X, Z),\ ancestor(Z, Y).$
$parent(bob, tim).$
$parent(bob, jim)$
$parent(mary, bob).$

The *ancestor* predicate is written with Prolog's standard control strategy specifically in mind. If its goals are permuted,

$$ancestor(X, Y) :- ancestor(Z, Y),\ parent(X, Z).$$

then a looping computation occurs due to left–recursion. The operational semantics of the program is made more complex if a cut is used,

$$ancestor(X, Y) :- parent(X, Z),\ ancestor(Z, Y),\ !.$$

This cut prunes the backtracking in the goals prior to it, and also prunes search in any ancestor clauses following this one. Any adequate operational account of Prolog needs to formally account for all the above operational phenomena. Subtle changes to Prolog programs, which might fully respect the integrity of the program's declarative semantics, may result in undesirable and erroneous computational behaviour. The fact

[1] Henceforth, "Prolog" refers to sequential Prolog as described in (Clocksin and Mellish 1981).

that cuts cause confusion amongst both novice and experienced programmers further presses the case for a more rigorous semantic modelling of Prolog.

Formal methods of program analysis require the use of a semantics of the programming language. Programming language semantics are mathematical systems which formally model aspects of the computational behaviour of programming languages, and thereby allow formal reasoning to be undertaken about the language and programs written in it. The behaviour modelled by the semantics varies according to the needs of the analysis. Some semantics are concerned with modelling the computed input–output relations of programs, while others can model phenomena such as resource usage, termination and non–termination, and other more abstract notions. Language semantics are strongly influenced by the particular mathematical tools in which they are defined, such as lambda calculus, first–order predicate logic, or algebraic rewrite rules.

Proving properties of Prolog programs is an active area of research. When unfair control strategies are being considered, the formal analyses of programs require a semantic representation of the control. As shown above, Prolog program computations are fundamentally dependent upon the syntactic structure of the program, such as goal and clause order. Therefore, a minimum requirement is that the computational effects of goal and clause order are formally accounted for. Extra–logical facilities such as the cut also must be modelled if they are to be used. Different operational semantics for Prolog have been suggested for this purpose, including denotational, meta–interpretive, and proof–theoretic semantics. The utility of these formalisms is largely dependent upon the mathematical model used and the intended application. For example, denotational semantics use lambda calculus, which lends itself well to the modelling of all Prolog features under one formalism. The abstract and complex domain spaces which arise, however, make it more suitable for language analysis, and decidedly unsuitable for proving properties of individual Prolog programs. Meta-interpreters allow the concise representation of many complex logic program control schemes, with the advantage that such representations are executable. Meta–interpreters are not intended to be applied towards proving program properties, as their operational semantics depends upon that of the meta–language.

Semantic formalisms used for imperative languages are not effective for modelling logic programs, as logic program computations are quite different in nature to their imperative counterparts. Imperative programs are characterised by transitions over a state, which is normally the value of variables in the memory. This is exploited by Hoare logics, predicate transformers, and other formalisms. Logic program computations, however, are nondeterministic rather than deterministic, and the state of a logic program computation cannot be modelled by functions over the memory (there is no notion of "memory" to be modelled in the first place). Instead, logic program computations are better modelled by computation trees, such as an SLD or AND/OR trees. Perhaps because the logic programming paradigm is considerably newer than other more conventional ones, there has not been as much attention given towards deriving useful mathematical models of nondeterministic computations.

This thesis presents a new algebraic process semantics for sequential logic programs, and in particular, Prolog. An AND/OR process model of Prolog computations is used, in which logic program components are modelled by sequential processes or agents. The semantics permits the formal representation of and reasoning about logic program control, and can be used as a calculus of logic program control tailored to a particular control strategy of interest. This is useful for applications such as the analysis of Prolog program termination and transformation properties, which require a formal operational semantics of the inference scheme.

The semantics is written in Milner's Calculus of Communicating Systems (CCS) (Milner 1989). Although CCS is normally applied towards concurrency, this thesis establishes its suitability as a tool for describing and analysing sequential logic programs. The use of a theory intended to model concurrency towards modelling the sequential logic programming paradigm is not as controversial as expected, since the characterisation of sequential logic program computations using sequential AND/OR process trees is strongly related to theoretical models of concurrent logic program control. Sequential logic program computations are effectively characterised by AND/OR trees. Ascribing processes to the nodes of these AND/OR trees results in an operational semantics for the language. Rather than using concurrent processes, however, sequential ones are defined. CCS effectively models networks of sequential processes, as it contains the necessary mathematical means with which sequentiality can be defined and reasoned with. This application is in line with the philosophical motivations behind

CCS discussed in the preface of (Milner 1989):

> People will use [a theory] only if it enlightens their design and analysis of systems; therefore the experiment is to determine the extent to which it is *useful*, the extent to which the design process and analytical methods are indeed improved by the theory. The experimental element is present because the degree of this usefulness is hard [...] to predict from the mathematical nature of the theory [...] In computer science there is something which one may call the *pertinence* of a theory which must be judged by experiment.

In this respect, this thesis shows that CCS's theory of concurrency is a very pertinent and pragmatic theory for describing and analysing sequential logic programs.

A flavour of the CCS semantics is now shown. The above example program is transliterated into a corresponding CCS semantic representation. This representation uses control operators which correspond to the particular control strategy under which the program will be executed. For example, if standard Prolog control is to be used, the above program is represented in the semantics as:

$$\begin{array}{rcl}
ancestor(X,Y) & \stackrel{\text{def}}{=} & ancestor_1(X,Y) \\
ancestor_1(X,Y) & \stackrel{\text{def}}{=} & parent(X,Z) \rhd ancestor(Z,Y) \\
\\
parent(X,Y) & \stackrel{\text{def}}{=} & parent_1(X,Y) \mathbin{\hat{;}} parent_2(X,Y) \mathbin{\hat{;}} parent_3(X,Y) \\
parent_1(X,Y) & \stackrel{\text{def}}{=} & (X,Y) = (bob, tim) \\
parent_2(X,Y) & \stackrel{\text{def}}{=} & (X,Y) = (bob, jim) \\
parent_3(X,Y) & \stackrel{\text{def}}{=} & (X,Y) = (mary, bob)
\end{array}$$

The control operators $\hat{;}$ and $\rhd$ represent clause sequencing and goal backtracking respectively, and taken together, model Prolog's depth–first–left–first control. The semantics of the cut is similarly modelled. Appending a cut to the ancestor clause results in the CCS expression,

$$ancestor_1(X,Y) \stackrel{\text{def}}{=} parent(X,Z) \rhd ancestor(Z,Y) \rhd\!|_{\ell}\ True$$

where $\rhd\!|_{\ell}$ is a cut operator. These control operators are defined in terms of basic CCS expressions. By applying CCS theory to these operator definitions, different computational phenomena of the source program can be studied. For example, one can apply CCS to this representation to simulate symbolic computation. Using CCS's stream notation, the execution of *parent* results in the stream

$$parent(X,Y) \;\approx\; \overline{succ(\theta_1)} \,.\, \overline{succ(\theta_2)} \,.\, \overline{succ(\theta_3)} \,.\, \overline{done} \,.\, \mathbf{0}$$

Here, $\overline{succ(\theta_i)}$ are actions representing computed answer substitutions θ_i which have the form $\theta_i = \{X \leftarrow bob, Y \leftarrow tim\}$. The $\overline{done}$ action indicates termination, and $\mathbf{0}$ denotes inactivity. The $\approx$ symbol is an operator denoting observational equivalence. Two CCS expressions are observationally equivalent if they generate the same observable output or behaviour. With some restrictions, observationally equivalent expressions are substitutive with one another in CCS. More analytical reasoning can be performed. CCS's domain of streams can be used to analyse termination characteristics of the program, for example, how a non–terminating *parent* clause would affect the execution of *ancestor*. The algebraic basis of these operators permits formal algebraic reasoning to be performed on programs with respect to Prolog's standard control scheme. For example, different associative, distributive, and non–commutative properties which are derived for these operators can be employed within proofs of program properties. Another powerful property given by the underlying CCS formalism is the ability to perform *bisimulations* on agents. Two agents are said to be bisimilar if they are equivalent with respect to some established notion of behavioural equality, for example, if they generate the same stream of computed results. Bisimulation proofs are similar in flavour to inductive proofs, and proceed by showing that an equivalence relation holds for all the possible states and observed behaviours of two agents.

Besides modelling Prolog's depth–first–left–first control with cut, the semantics of some other sequential control schemes is investigated. A variety of breadth–first control strategies, a basic predicate freezing scheme, and some nondeterministic schemes are modelled. The semantics is also applied towards proving properties of logic programs, such as termination, the validity of source–to–source transformations, and partial evaluation transformations. Given that CCS has already been applied towards concurrent logic program languages, this thesis suggests that process algebras are a means of unifying the operational semantics of sequential and concurrent logic programming paradigms.

1.1 Thesis outline

Chapter 2 reviews some logic programming terminology, and gives a brief outline of the CCS formalism used throughout the thesis. This chapter is not intended as a tutorial introduction to either logic programming or CCS; the reader is referred to other sources for a fuller introduction to these topics.

Chapter 3 is the main chapter of the thesis. The topic of programming language semantics is first reviewed. The basic AND/OR process tree model which underlies the semantics is described. A CCS semantics of standard Prolog control is then presented. The central component of the semantics are two control operators which model goal backtracking and clause sequencing. The semantics of these operators is first defined in terms of rudimentary CCS primitives. Then, using these low–level definitions, high–level bisimilarities or behavioural equivalences are defined for the operators. The semantics of Prolog's cut is similarly modelled. The issue of dataflow is addressed. Since logical variables are not supported in CCS, some extensions to CCS are necessary in order to handle the Herbrand domain of logic programs.

Properties of the CCS semantics of Prolog are derived in chapter 4. First, some termination properties are derived which describe the behaviour of the semantics with respect to streams of computed answer substitutions. Some compositional properties of the semantics are proven, such as associativity, distributivity, and non–commutativity. The concept of well–termination is addressed, which refers to the well–behavedness of the semantics with respect to a termination protocol. The correctness and semantic completeness of the semantics is also established with respect to three other semantic models of Prolog: SLD resolution and the immediate consequence operator T_P; a functional semantics for Prolog; and a Prolog meta–interpreter for Prolog.

The semantics is applied in program termination analysis in chapter 5. The program termination problem as it pertains to Prolog is reviewed. A simple and general technique for analysing the program termination characteristics which exploits well–founded orderings of predicative arguments is outlined. A number of examples are given.

Chapter 6 applies the semantics towards validating source–to–source Prolog transformations which use cut. Cuts can be inserted into programs to prune unwarranted search. The transformations verified here exploit the determinism of program components. The semantic treatment of the cut, as well as CCS's bisimulation proof technique, are well–suited to this application.

A final application of the semantics is given in chapter 7, in which the semantics is used to characterise partial evaluation transformations of Prolog programs. Partial evaluation is accomplished by performing bisimilar transformations on a program's CCS representation. This permits a rudimentary semantics of the control component of partial evaluation algorithms, and explains how this control is related to the control used to execute the source and residual programs.

The robustness of CCS towards modelling other sequential control schemes is explored in chapter 8. Three different breadth–first control strategies are modelled. As with the semantics of standard control, these semantics are defined hierarchically. A simple predicate freezing mechanism is modelled. Some nondeterministic control strategies are studied, which can also be considered to be semantic models of simple parallel computation strategies. The application of these semantics within program analysis is discussed. The possibility of intercomposing different control operators to model hybrid control schemes is an attractive feature.

A summary of major results concludes the thesis in chapter 9. Future directions of this research are discussed, including using the semantics in a semi–automated environment, and the issue of extending the semantics to handle concurrency.

Chapter 2

Preliminary Definitions

Some preliminary definitions are given for logic programming and CCS, which are used throughout the rest of the thesis.

2.1 Logic programming

The following definitions are in (Lloyd 1984), and a gentle introduction to them is in (Hogger 1990). A *program clause* is a sentence of first–order predicate logic having the form

$$\forall\, (H \vee \neg G_1 \vee \cdots \vee \neg G_n) \qquad (n \geq 0)$$

where H, G_i are atoms of arity ≥ 0, H is a positive atom, and all the G_i are negated. The logic program syntax adopted for program clauses is

$$H :-\ G_1, ..., \ G_n. \qquad (n > 0)$$

for *rules*, where H is the clause head and the goals G_i comprise the body, and

$$H. \qquad (n = 0)$$

for *assertions* or *facts*. A *query* is a clause without a positive atom H,

$$\forall\, (\neg G_1 \vee \cdots \vee \neg G_n) \qquad (n \geq 0)$$

which has the logic program syntax

$$?-\ G_1, ..., \ G_n. \qquad (n > 0)$$

The term *clause* will henceforth refer to program clauses. A *predicate* is an ordered list of clauses with the same head name and arity (for simplicity, it is assumed that a given predicate name will have only one arity value). A *logic program* is a finite set of predicates. Clause headers and goals will often be indexed to indicate their relative order within predicates and clause bodies respectively.

The arguments of the atoms in logic programs are *terms*, which are compositions of functors, constants (0 arity functors), and logical variables. A shorthand notation for a tuple $(t_1, \ldots, t_k)$ of terms is $\tilde{t}$. Given a logic program P, the *Herbrand universe* U_P for P is the set of all ground terms formed from the constants and functions appearing in P. The *Herbrand base* B_P is the set of all ground atoms of P with the set of terms from the Herbrand universe as arguments.

A *substitution* θ is a set of variable-term replacement pairs $\{X_1 \leftarrow t_1, \cdots, X_k \leftarrow t_k\}$, meaning each logical variable X_i is simultaneously bound to the corresponding term t_i. The empty substitution is denoted ϵ. The *application* of a substitution θ to a term t is denoted $t\theta$, and means that every occurrence of every variable X_i in t that exists in a substitution pair $X_i \leftarrow t_i$ is replaced by t_i. For example, if $t = a(X, b(Y, X, Z), Z)$, and $\theta = \{X \leftarrow w, Z \leftarrow f(X), W \leftarrow b\}$, then $t\theta = a(w, b(Y, w, f(X)), f(X))$. When applying a substitution to a tuple of terms, as in $\tilde{t}\theta$, the θ distributes to each term within $\tilde{t}$. The *composition* of substitutions of θ and γ is denoted $\theta \circ \gamma$, and has the property that $t(\theta \circ \gamma) = (t\theta)\gamma$. A substitution θ is called a *unifier* for a finite set of expressions S if $S\theta$ is a singleton. θ is a most general unifier (mgu) if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$.

Unification algorithms are used to determine the mgu's of terms to be unified. In order for unification to be sound, it is assumed that an *occur check* is performed, which checks that a variable is never bound to a term which contains that variable as a subterm. It is henceforth assumed that an occur check is performed during unification. Due to the computational inefficiency of performing an occur check, however, it is often dropped in practical implementations of logic program languages.

The logic programming paradigm refers to the following computational interpretation for the above definitions. A *computation rule* R is a function which selects one atom from a list of atoms constituting a program query. For example, Prolog uses the rule that the *first* goal in a sequence of goals is always selected:

$$R(\;A_1,\, A_2,\, \ldots,\, A_k.\;) \;=\; A_1$$

Let goal Q_i be "$?-\ A_1, \ldots, A_m, \ldots, A_k.$", and clause C_{i+1} be "$A :-\ B_1, \ldots, B_q.$", and R be a computation rule. Then Q_{i+1} is *derived* from Q_i and C_{i+1} using mgu θ_{i+1} via R if: (i) A_m is the atom selected by R; (ii) $A_m\theta_{i+1} = A\theta_{i+1}$; and (iii) G_{i+1} is the goal

$$?- \ (A_1, ..., A_{m-1}, B_1, ..., B_q, A_{m+1}, ..., A_k) \ \theta_{i+1}$$

Q_{i+1} is the *resolvent* of Q_i and C_{i+1}.

Let P be a program, Q a goal, and R a computation rule. An *SLD–derivation*[1] of $P \cup \{\neg Q\}$ via R consists of a finite or infinite sequence $Q_0, Q_1, ...$ of goals ($Q = Q_0$), a sequence $C_1, C_2, ...$ of clause variants from P, and a sequence $\theta_1, \theta_2, ...$ of mgu's, such that each Q_{i+1} is derived from Q_i and C_{i+1} using θ_{i+1} via R. The selected clause from $C_1, C_2, ...$ is a variant in the sense that it is *standardised apart*, which means that the logic variables within it are uniquely named with respect to any logic variable used in the derivation so far. This is done to avoid variable clashes during unification. An *SLD–refutation* of $P \cup \{\neg Q\}$ via R is a finite SLD–derivation which has the empty clause as the last goal in the derivation. A *failed SLD–derivation* is one that ends in a non–empty goal with the property that the selected atom in the goal does not resolve with any clause.

SLD resolution computes values during the accumulation of substitutions during the resolution steps for an SLD–refutation. The final binding obtained represents an instantiation of terms for which the refutation holds. This *computed binding substitution* is a computed result for the program. When multiple refutations are queried from the SLD refutation algorithm, each computed binding substitution represents another solution to the query. SLDNF resolution is SLD resolution with *negation as failure*, which permits negative literals to be used in queries and clause bodies. A safe negation as failure rule states that, if ground A has a finitely failed SLD–derivation, then infer $\neg A$.

SLD–resolution can be modelled by an *SLD–tree*. Given a program P, a goal G, and a computation rule R, an SLD–tree for $P \cup \{G\}$ via R is defined as follows Each node of the tree denotes a (possibly empty) goal. The root node is G. If $?- A_1, ..., A_m, ..., A_k \ (k \geq 1)$ is a node, and A_m is the atom selected by R, then this node has a descendent for each clause resolvable with A_m, and each of these descendents denotes the corresponding resolvent goal. Nodes which denote the empty clause have

[1] SLD is an acronym for Linear resolution with Selection function for Definite clauses.

no descendents, as do nodes which have no resolvents. In the above, the structure of the SLD–tree is dependent upon the goal selection rule R used, which is fixed for the entire tree. Changing R will change the SLD–tree derived.

A *Herbrand interpretation* I is an assignment of truth values to the atoms in B_P. A *Herbrand model* for a logic program P is a Herbrand interpretation which logically satisfies all the program clauses in P. However, there can be many different Herbrand models for a program, many of which contain more elements from B_P than are necessary to satisfy P. A *minimal model* MM_P is the (unique) minimal set of atoms from B_P which satisfy the completion of P. An important result is that computed results arising from SLD resolution on a program P are precisely those belonging to MM_P.

A formal technique for linking the declarative and operational semantics of logic programs is by the use of an immediate consequence operator T_P, defined as:

$$T_P(I) = \{ \quad A \in B_P : A \leftarrow B_1, ..., B_n. \ \textit{is a ground instance} \\ \textit{of a clause in } P, \textit{ and } B_1, ..., B_n \subseteq I \quad \}$$

MM_P can be incrementally constructed from T_P. Initially, the set

$$T_P^1(\emptyset) = \{A \in B_P : A \leftarrow \textit{true is a ground instance of an assertion in } P\}$$

is created, which represents the set of atoms from B_P derivable from program assertions. Then T_P is recursively applied to itself:

$$T_P^n(\emptyset) = T(T_P^{n-1}(\emptyset))$$

It turns out that, because T_P is monotonic and continuous,

$$\lim_{n \to \infty} T_P^n(\emptyset) = MM_P$$

2.2 Basic CCS syntax and semantics

This section briefly reviews the essential elements of CCS used in this thesis. This is not intended as a tutorial introduction to CCS; see (Milner 1989) for a comprehensive treatment. Applications of the semantics use relatively little CCS theory, and the understanding of basic termination and transformation proofs does not require intimate knowledge of CCS.

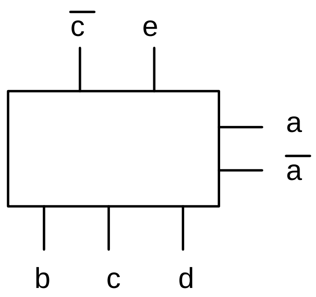

Figure 2.1: An agent

An *agent* or *process* is a mechanism whose behaviour is characterised by discrete actions. An agent can be conceptualised as a black box with input–output lines extending from it which are labelled with action names (figure 2.1). Actions are transmitted along input lines (non–overbarred) and output lines (overbarred). The behaviour of the agent is characterised by the nature of the transmission of actions. Research in process algebras is concerned with the establishing of useful theories for modelling and analysing this characterisation of agent behaviour. Of particular interest are networks of autonomously executing agents which interact via discrete communications with one another, as this is a popular model of concurrency. Here, the input–output lines of agents are shared between sets of agents. Agents can be hierarchical: an entire network can be considered to define an agent, and the internal definitions of the agents in this network can be likewise defined by networks of agents.

CCS is an algebra which allows the description and analysis of the behaviour of complex networks of agents. Agents are described using a set of *agent expressions*, $\mathcal{E}$, which use five basic operators. Letting E range over $\mathcal{E}$, then $\mathcal{E}$ are the formulae recursively constructed using the following expressions (the meanings of which are described in detail later):

$\alpha.E$	Prefix
$\sum_{i \in I} E_i$	Summation
$E_1 \mid E_2$	Composition
$E \backslash L$	Restriction
$E[f]$	Relabelling

Milner defines the semantics of these equational operators using the transitional rules of figure 2.2. These transitions are sequents in which the expression below the line can be inferred when the expressions above the line (if any) hold. Throughout the figure,

$$
\textbf{Act}\quad \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \textbf{Sum}_\mathbf{j}\quad \frac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i\in I} E_i \xrightarrow{\alpha} E'_j} \quad (j \in I)
$$

$$
\textbf{Com}_\mathbf{1}\quad \frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F} \qquad \textbf{Com}_\mathbf{2}\quad \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}
$$

$$
\textbf{Com}_\mathbf{3}\quad \frac{E \xrightarrow{\ell} E' \qquad F \xrightarrow{\overline{\ell}} F'}{E|F \xrightarrow{\tau} E'|F'}
$$

$$
\textbf{Res}\quad \frac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L} \quad (\alpha, \overline{\alpha} \notin L) \qquad \textbf{Rel}\quad \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}
$$

$$
\textbf{Con}\quad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad (A \stackrel{\mathrm{def}}{=} P)
$$

Figure 2.2: Transitional semantics of basic CCS calculus

the expression $E \xrightarrow{\alpha} E'$ represents the transition of agent E into agent E' through the action α. The E' expression is an *α–derivative* or *derivative* of E. When multiple transitions occur as in $E \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$ then $\alpha_1 \cdots \alpha_n$ is an *action sequence* of E. The E expression is conceptually equivalent to the notion of a *state* in the same sense as in automata theory. An agent's state is defined by the behaviour of the agent at a particular moment in time. Because an expression E unambiguously determines a behaviour, E itself can be regarded as denoting a state.

The meaning of the transitions in figure 2.2 are now described. The **Act** transition represents an agent transition in terms of its immediate actions α. The symbol "." temporally separates actions. A stream of actions is represented as

$$\alpha_1 \,.\, \alpha_2 \,.\, \cdots$$

where $\alpha_i \in A \cup \overline{A}$ are *action labels*. $\mathcal{A}$ is a set of action *names*, and $\overline{\mathcal{A}}$ is the set of *co-names*. By convention, names are used for input actions, and co-names for output actions. The set of *labels* $\mathcal{L}$ is $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$. The set of *actions* Act is $Act = \mathcal{L} \cup \{\tau\}$, where τ is a distinguished silent action[2]. An example transition using **Act** is $a.b.E \xrightarrow{a}$

[2] Because of the determinism modelled in the thesis, τ will not be of great concern.

$b.E \xrightarrow{b} E$. A notation for multiple transitions like this is $a.b.E \stackrel{ab}{\Rightarrow} E$.

The $\mathbf{Sum}_j$ notation represents a choice of possible behaviours. For example, the binary expression $E_1 + E_2$ means that behaviours E_1 and E_2 are alternative choices of behaviour. The summation generalises to any finite number of terms. The choice operator $+$ is normally treated as being nondeterministic in CCS. In practice, however, deterministic choices are often made. An example of **Sum** is the transition $(a.P + \overline{b}.Q) \xrightarrow{a} P$. This represents the action a being chosen, which causes the alternative choice of the other term to be pre–empted.

The **Com** transitions describe agent composition, which is the basic operator for concurrency. Agent composition represents how agents behave, both autonomously ($\mathbf{Com_1}, \mathbf{Com_2}$) and interactively ($\mathbf{Com_3}$). A fundamental activity within CCS is the *handshake*, which is a successful simultaneous communication between two agents. In order for a handshake to occur, two agents must simultaneously execute identical immediate actions, one of which is a co–action of the other. In $\mathbf{Com_1}$ and $\mathbf{Com_2}$, if either agent on the left or right of the composition operator $|$ can produce a single transition, then the whole expression makes the transition. However, if both expressions make complementary transitions, a handshake results. This is represented by the silent τ action. For example, by using $\mathbf{Com_1}$, the following transition can be done:

$$(a.P + \overline{b}.Q) \mid (\overline{a}.R + c.S) \xrightarrow{a} P \mid (\overline{a}.R + c.S)$$

while with $\mathbf{Com_3}$,

$$(a.P + \overline{b}.Q) \mid (\overline{a}.R + c.S) \xrightarrow{\tau} P \mid R$$

since communication can occur between the terms $a.P$ and $\overline{a}.R$, resulting in the occurrence of a silent "τ" action.

The **Res** transition represents *restriction.* Restriction removes the specified actions in set L from being observed externally. This is useful for hiding actions from external observation. For example, in $(a.P + \overline{b}.Q)\backslash\{b\}$, the restriction causes b's output to be suppressed; any transition of the term with $\overline{b}$ will be replaced by a τ action.

The **Rel** transition is a *relabelling* rule. A relabelling function $f : \mathcal{L} \rightarrow \mathcal{L}$ renames actions. A notation for finite relabelling functions is $[\ a_1/b_1,\ \cdots\ , a_k/b_k\ ]$ where each b_i is renamed by a_i. For example, in $(a.P + \overline{b}.Q)[a/c]$, any transition using the action a will be relabelled: $c.(P[a/c])$.

The **Con** defines agent *constants*, and is the basic means for creating recursive agent definitions. A constant is an agent whose meaning is defined by an agent expression. For every constant A, there exists an equation "$A \stackrel{\text{def}}{=} E$", where E is an expression using the aforementioned $\mathcal{E}$ operators. The definition of an agent constant is semantically equivalent to the constant reference itself. The *null* or inactive agent is denoted $\mathbf{0}$. An example of an agent constant is $P \stackrel{\text{def}}{=} a.P + \overline{b}.Q$.

A common form of CCS expressions is $(P_1 \mid \ ... \ \mid P_n) \setminus L$. The *expansion law* converts such an expression into one having a summation of terms with all immediate actions prefixed onto corresponding agent states. The (simplified) expansion law is as follows. Let $P = (P_1 \mid \ ... \ \mid P_n) \setminus L$ with $n \geq 1$. Then

$$\begin{array}{rcl} P & = & \sum\{\alpha.(P_1|...|P_i'|...|P_n)\backslash L \ : \ P_i \xrightarrow{\alpha} P_i', \alpha \notin L \cup \overline{L})\} \\ & + & \sum\{\tau.(P_1|...|P_i'|...|P_j'|...|P_n)\backslash L \ : \ P_i \xrightarrow{\beta} P_i', \ P_j \xrightarrow{\overline{\beta}} P_j', i < j\} \end{array}$$

The first summation represents the agents which autonomously change state. The second summation represents the agents which change state interactively with one another (via silent τ actions), which happens when a β and $\overline{\beta}$ synchronise, or handshake.

Basic CCS uses the agent constant schema $P \stackrel{\text{def}}{=} E$, where P is an agent name and E is a CCS expression. However, it is convenient for agents to have arguments which can be actions, terms, or agent expressions themselves. CCS can be extended to handle value–passing by treating an agent constant with an argument as representing a family of agent expressions enumerated over the possible values that the argument may have. For example, an agent $P(I) \stackrel{\text{def}}{=} E$ in which I is an integer value > 0 represents the set of agents,

$$\begin{array}{l} P_1 \stackrel{\text{def}}{=} E_1 \\ P_2 \stackrel{\text{def}}{=} E_2 \\ \cdots \end{array}$$

where each definition is specialised for a particular integer I. Agent arguments are similarly enumerated. This idea will be used in the thesis for extending CCS to handle logical variables over the Herbrand universe of a program.

Recursive agent expressions in CCS are treated as constant expressions. For example, an alternative treatment of the recursive constant expression $A \stackrel{\text{def}}{=} a.A$ is $\mathbf{fix}(X = a.X)$, in which a new inference rule for the **fix** operator is used, which is similar to **Con** except that the solution to the recursive equation is obtained. A result of this interpretation of recursive equations is that, as long as some guardedness criteria are followed, recursive equations have unique solutions (Milner 1989, page 65). If two recursive equations have the same external behaviours, it can then be concluded that they both define the same agent. This uniqueness result will be assumed whenever recursive agents are being considered.

A significant part of CCS theory is devoted to various concepts of behavioural equality. Three basic theories of behavioural equivalence in CCS are given in (Milner 1989): (i) $\sim$ strong equivalence; (ii) $=$ equality or observation congruence; and (iii) $\approx$ observation (or weak) equivalence. The type of equivalence chosen depends upon the granularity of behaviour needed to distinguish the type of concurrency being studied. Observation equivalence is the most practical for this thesis. It is the least strict equivalence which still preserves syntactic substitutivity (or congruence) with respect to the constrained use of CCS used here. In basic terms, two agents are observationally equivalent if they generate the same observable behaviour under the same conditions. Since the majority of control schemes studied here are deterministic ones, there is no need to unduly account for the nondeterminism encountered with concurrent computations. Observation equivalence is not strictly a congruence relation, as it is not fully substitutive within expressions. The problem arises because observation equivalence is not preserved by summation. For example, $b.\mathbf{0} \approx \tau.b.\mathbf{0}$, but $a.\mathbf{0} + b.\mathbf{0} \not\approx a.\mathbf{0} + \tau.b.\mathbf{0}$. This is not a practical problem here, since substitutions within expressions in summations are never done.[3] The summation expressions used in the thesis denote deterministic choices between success and failure. The equality theory used here therefore reduces to *trace equivalence*; proofs, however, will respect the integrity of the observation equivalence relation,

A *bisimilarity* is an observed behavioural equivalence amongst a pair of agents. Two agents are shown to be bisimilar with respect to some given theory of behavioural equality by performing a *bisimulation* on them. The bisimulation proofs in this thesis use observational equivalence as the theory of equality, and the general form of these proofs is as follows. Let $A \stackrel{\hat{\alpha}}{\Rightarrow} A'$ represent the transition of A into A' where the action sequence $\hat{\alpha}$ is one where all silent τ actions are removed. Then $P \approx Q$ iff, for all $\alpha \in Act$,

[3] The nondeterministic control schemes studied in section 8.5 are an exception.

(i) Whenever $P \xrightarrow{\alpha} P'$, then for some Q', $Q \overset{\hat{\alpha}}{\Rightarrow} Q'$, and $P' \approx Q'$.
(ii) Whenever $Q \xrightarrow{\alpha} Q'$, then for some P', $P \overset{\hat{\alpha}}{\Rightarrow} P'$, and $P' \approx Q'$.

where $Act = \mathcal{L} \cup \{\tau\}$, and $\mathcal{L}$ is the universe of actions and co–actions. To prove bisimilarity between two expressions, it must be shown that, for all possible α, their α–derivatives generate the same behaviours. Practically speaking, bisimulation proofs proceed on a case–by–case basis, each case being a specific state transition for an agent.

Chapter 3

An Algebraic Semantics of Prolog

The semantics of Prolog in this chapter is the main focus of the thesis. This algebraic process semantics is written in Milner's CCS formalism. It is ideally suited for modelling the control component Prolog, and as a result, will be applied later in the thesis in program analyses which require a formal account of the control strategy.

The topic of program language semantics is surveyed in section 3.1. Section 3.2 discusses some design principles of the semantics. Section 3.3 presents a CCS semantics of the basic control component of Prolog computation within CCS. This is first done at an operator level, and then at a higher level using CCS bisimilarities. The semantics of the cut is given in section 3.4. The topic of dataflow is discussed in section 3.5. A discussion concludes the chapter in section 3.6.

3.1 Review

3.1.1 Programming language semantics

Programming language semantics mathematically describe the computational behaviour of programming languages and programs written in them. Semantics are normally intended to describe languages and programs at a conceptually abstract level so that higher–level reasoning about them can be undertaken. Semantic formalisms differ widely according to their intended application. Operational semantics model the computational mechanisms underlying a language, often in terms of an abstract machine. Such semantics are intended to describe the functionality of programming languages in a way which lends insight into issues of language design and implementation. Program specification semantics describe the meaning of programs at a more human–oriented, problem–domain level, and are used for program verification and derivation.

Semantics can be distinguished by the mathematical formalism in which they are defined, such as lambda calculus, predicate logic, or algebras. The suitability of different mathematical formalisms varies according to their intended use and the programming paradigm they are modelling. For example, the lambda calculus used in denotational semantics (Stoy 1977) is descriptively rich enough to describe a wide variety of programming languages. Predicate logic has been successfully applied in formal program derivation and verification, as logic can act as a semantic link between logical specifications and the programming language, as well as justify the soundness of derivation steps between the two (Hoare 1969) (Hoare 1985*b*) (Hehner 1984*c*). Algebraic semantics have been applied towards both language description (Plotkin 1981) and formal program development (Jones 1986) (Sannella 1988). Algebras are also useful for describing the semantics of concurrency (Hoare 1985*a*) (Hennessy 1988) (Milner 1989).

Axiomatic semantics are commonly used for program development. They are termed "axiomatic" because the semantics takes the form of a formal logical theory in which axioms describe the semantics of program language constructs and valid derivations within the theory. Axiomatic semantics for partial and total correctness are readily derived for imperative–style programming languages, since the deterministic state transitions over the memory values which characterise them are easily modelled in relational logic. Semantics for partial correctness establish the soundness of results for terminating programs. Semantics for total correctness additionally specify the conditions under which programs terminate, and therefore determine the soundness and completeness of computations. The axiomatisation of terminating computations is generally undecidable according to the Halting problem. This is not usually a limitation, since most practical programs used in everyday applications have decidable termination properties. Different types of axiomatic semantics include Hoare logics (Hoare 1969) (Apt 1981), predicate transformers (Dijkstra 1975) (Gries 1981), and program relations (Hehner 1984*c*).

3.1.2 The semantics of logic programs and Prolog

A primary motivation behind logic programming is that mathematical logic can be a practical programming language, and hence declarative logic programs may act as their own specifications (Kowalski 1974) (Kowalski 1985). A clear conceptual separation can be made between the declarative semantics of the program and the control used to infer results from it (Kowalski 1979). A declarative logic program is a high–level human–oriented statement of what the program is intended to compute. The behaviour of the program is dependent upon the inference strategy applied to this declarative representation. This is in contrast with von Neumann languages, where control is explicitly encoded in programs, and the relation between a program and a declarative statement of what it is intended to compute is much less direct. For example, to verify imperative programs, a mapping of the program to its operational meaning is done via a program language semantics, and then this expression is refined towards a higher level declarative specification (Loeckx and Sieber 1984).

Declarative and operational semantics have been established for Horn clause logic programs (van Emden and Kowalski 1976) (Apt and van Emden 1982) (Lloyd 1984). Many semantic interpretations of logic programs have been derived, and a comprehensive survey is beyond the scope of this review. Model–theoretic semantics treat logic programs as logical theories, in which the program statements are interpreted as logical formulae over the Herbrand universe. Procedural interpretations of logic programs are concerned with the behaviour of computations when logical deductions are performed on a program and query using SLD resolution. An important result is that the success set computed by SLD resolution (the set of atoms that have SLD–refutations) is equivalent to the minimal Herbrand model for the completion of the program. Operational semantics of logic programs are concerned with behavioural effects of the implementation, such as the particular computation and search strategies, extra–logical facilities such as the cut, and the treatment of negation as failure.

In practise, the conceptual separation of the declarative program logic from the inference system control is not guaranteed. Logic program implementations such as Prolog use unfair and unsafe computation schemes, and computed success sets are not necessarily equivalent to what is specified by the program's declarative semantics. Because fair strategies are inefficient to implement, practical implementations of logic programming languages use weaker inference strategies which are practical to implement on current hardware. For example, Prolog with its standard control strategy has proven to be a popular logic programming language. This depth–first traversal of the SLD tree is efficiently implemented on conventional stack–based architecture. Unfortunately, depth–first search is unfair, and termination is not assured: divergence results if the program is ill–structured so that the search is forced to proceed down an infinite branch of the computation tree. Therefore, the correctness and completeness of Prolog programs fundamentally depends upon the textual order of goals and clauses, and practical computations require programs to be tailored correctly.

There is much recent effort in deriving new semantic models of logic program control, and in particular, Prolog's depth–first left–to–right control strategy. A semantics of control is necessary for analysing the behaviour of particular control strategies, and proving program properties such as termination and the correctness of source–to–source program transformations which are affected by control.

Meta–interpretive accounts of logic program computations have been suggested (Hill and Lloyd 1988) (Sterling and Shapiro 1986). Meta–interpretive semantics describe computations via the definition of a logical meta–interpreter. Meta–interpreters allow the writing of concise and executable operational semantics of different programming languages, which makes them ideal for language prototyping. Meta–interpreters can model the control component of a particular logic programming language, but in order to do so, the operational semantics of the meta–language itself must be accounted for. For example, given a meta–interpreter for Prolog that is to be executed in Prolog, the semantics of Prolog's depth–first–left–first control are relevant to any complete operational analyses of the meta–interpreter axioms.

Many denotational semantics of Prolog programs have been proposed (Fitting 1985) (Jones and Mycroft 1984) (North 1986) (Arbab and Berry 1987) (Debray and Mishra 1988) (Billaud 1990) (Baudinet 1988) (Nicholson and Foo 1989). These semantics are compositional – the semantics of a program is defined in terms of the semantics of its constituent components. Denotational semantics are also founded in the lambda calculus, whose descriptive robustness permits essentially all of Prolog's features to be modelled within one formalism.

Proof theoretic accounts of logic program have been suggested (Andrews 1991)

(Hallnas and Schroeder-Heister 1988). A tenet of these approaches is that logic, being an essential component of the logic programming paradigm, should likewise be used to describe the inference strategy. Control is modelled by axioms of logical inference, while at the same time, the declarative semantics of the logic program is retained in the formalism.

There are many other approaches to describing the operational semantics of Prolog. (Francez *et al.* 1985) derives Hoare–style axiomatisations of the computation tree. (Drabent 1987) use inductive assertions for axiomatising the mode relations within the left–to–right backtracking of goals. (Deransart 1988) present an attribute grammar account of logic program control.

3.2 Overview of semantic model

The CCS semantics of Prolog presented in this thesis is an *algebraic AND/OR process semantics of sequential Prolog.* It is a distributed operational semantics in which Prolog behaviour is modelled by networks of communicating processes. The rationale behind this model will now be outlined.

A tenet of logic programming is that logic programs are directly interpretable as high–level declarative statements of first–order predicate logic (Lloyd 1984). In the discussion of the logical basis of logic programming in section 2.1, it is shown how program clauses and queries are interpretable as statements of first–order predicate logic, and how logically valid deductions can be deduced from them by applying SLD resolution. In particular, the goals within clauses and program queries are logically delineated by logical AND ($\wedge$) connectives, and clauses by logical OR ($\vee$). This logical structure can be further extended to a logic program's execution via SLD trees. Each node of the tree represents the current state of the goal being processed, and branches represent different resolution strategies. The whole SLD tree represents the evolution of SLD derivations performed on the program and query.

Another way of representing logic program computations is through the use of an *AND/OR tree*. AND/OR trees are declarative representations of logical dependencies in logical inferences and computations (Harel 1980). The "AND" and "OR" labels refer to the logical contribution of the nodes: at least one OR node must be success (logical *OR*), while all brethren AND nodes must be resolved (logical *AND*). Logic pro-

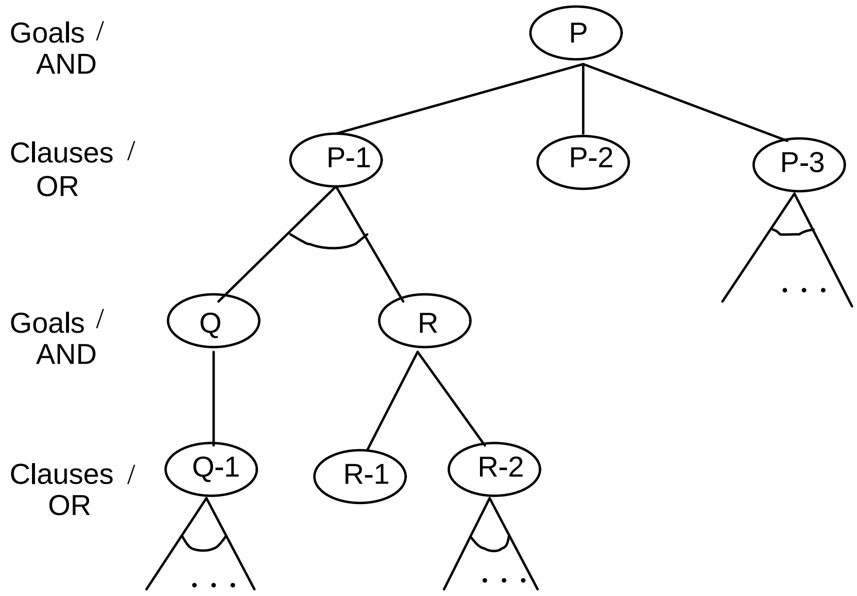

Figure 3.1: AND/OR tree for logic programs

grams have a natural AND/OR tree interpretation (Lindstrom and Panangaden 1984). An AND/OR tree defines the search space over the Herbrand universe of a logic program, and computations which occur over this search space correspond to what is denoted by the declarative semantics of the program. Program clauses map to OR nodes, and and goals in clauses and queries map to AND nodes. AND/OR trees therefore denote the declarative requirements that program components fulfill without forcing a rigid procedural interpretation on the details of the derivation process: the only procedural requirements specified are the logical satisfactions denoted by AND and OR nodes themselves. AND/OR trees differ from SLD trees in that they explicitly represent the logical contribution that program components have on the computation. Such information is not rendered within SLD trees, as the state of the computation is denoted only by a dynamically changing program goal set, along with computed binding substitutions. AND/OR trees are especially useful for representing concurrent logic program computations, as the loose procedural requirements assigned to AND and OR nodes are ideal for describing concurrency. The terms *AND parallelism* and *OR parallelism* refer to the fact that all the AND and OR nodes can be executed in parallel; the only procedural requirement is that all AND nodes succeed, and that one OR node succeeds.

While AND/OR trees are suitable for representing the declarative semantics of logic programs, they do not model the control during computations. Logic program implementations require the introduction of control mechanisms in order to make efficient execution possible, whereas AND/OR trees do not specify control at all. What is needed is a way to represent the interaction and communication amongst the nodes in an AND/OR tree. The *process* concept allows the injection of an operational semantics to AND/OR trees. A process or agent is a mechanism whose behaviour is characterised by discrete actions. The process paradigm is a powerful means with which to conceptualise computations which are characterised by discrete autonomous mechanisms that communicate with one another. This is naturally applicable to concurrent logic program computations, since concurrently executing program components can be modelled as concurrent communicating processes (Lindstrom and Panangaden 1984) (Conery and Kibler 1985) (Waern 1986). They are particularly successful within the concurrent logic program framework, as the modular nature of concurrent logic programs coincides with the compositional nature of process semantics.

AND/OR process models combine the declarative semantics of AND/OR trees with the operational semantics of processes. Given an AND/OR tree, each AND and OR node is modelled as a process which has encoded within it a particular control regimen and communication protocol. Doing so introduces an *operational semantics* to the AND/OR tree: the AND and OR processes determine the manner in which the tree is explored.[1]

Communicating networks of processes are conveniently described by *process algebras* (Hoare 1985*a*) (Hennessy 1988) (Milner 1989). Process algebras allow the detailed description and analysis of complex networks of concurrent processes. The particular process algebra used in this thesis is Milner's Calculus of Communicating Systems (CCS) (Milner 1989). CCS has been applied to the semantics of concurrent Prolog implementations in (Beckman *et al.* 1986) (Beckman 1986), A unique contribution of this thesis, however, is that *sequential* AND and OR agents are defined, rather than concurrent agents as in most other process models. This might seem controversial at the outset, as process models are typically intended for describing concurrent com-

[1] One peculiarity of nomenclature is that AND and OR nodes are modelled as OR and AND agents respectively. This arises because AND and OR agents define the operational behaviour of descendent agents, while AND and OR nodes define the logical contribution of those nodes themselves.

putations. However, (Saraswat 1989) states that sequential control can be considered to be a specialisation of concurrent control; as a result, the modelling of sequential Prolog using tools normally reserved for concurrency is worth consideration.

Unlike imperative computations, logic program computations are not easily characterised as functions over the memory state, since there is no memory involved. Rather than state transitions over a store, logic program states are more naturally modelled by computation trees: the state of a computation is directly reflected by the structure of the computation tree. It turns out that the process algebra paradigm is ideal for describing the semantics of sequential logic program control, since AND/OR trees can be modelled by process algebras. As with concurrent computations, sequential logic program computations are also characterised by AND/OR trees whose structure dynamically changes during the computation. This tree structure is effectively modelled by concurrent processes, under the provision that they interact sequentially. Sequential computation is indeed a specialised control strategy within concurrent computation, as can be seen in formal treatments of sequencing in concurrency theory (Hoare 1985*a*) (Milner 1989). Deriving sequential models of logic program computation entails specialising process interaction to comply with a sequential control regimen. A result of this thesis is that different inference strategies used by different logic program control schemes simply require the definition of appropriate processes.

The sequential AND/OR process semantics in this thesis is operationally similar to the concurrent ones in (Lindstrom and Panangaden 1984) and (Conery and Kibler 1985), the main difference being its sequentiality. As described above, the execution of a Prolog program will result in the creation and deletion of agents. Viewed as a whole, the dynamically changing structure of the network of agents parallels the computation tree for the program. The composition of agents themselves mirror the syntactic structure of Prolog programs. An AND agent is created for each clause in a predicate, and an OR agent is created for each predicate. AND agents spawn OR agents, and vice versa, according to the call structure of goals in the Prolog program. Communication between agents takes the form of either value arguments when an agent is invoked, or action messaged when an agent completes a computation. The above description is very general; the following sections will describe the semantics in greater detail.

The semantics is written as a process algebra using Milner's CCS. A Prolog program has corresponding CCS expressions defined for it, which is henceforth collectively called the program's CCS semantics. CCS is a concise and descriptively powerful calculus of concurrency, and its lean set of control operators quite readily models sequential Prolog control. CCS's compositionality of agents is well–suited to describing the modular nature of logic programs. CCS's ability for hierarchically modelling computations is also ideal for the hierarchical treatment of Prolog control done within the semantics. CCS theory itself allows many computational phenomena of logic program computation to be studied, such as computed streams of results, symbolic computations, behavioural equivalence, non–termination, and looping. CCS's stream domain is a powerful means of analysing many Prolog program properties, such as termination and transformations.

The semantics is conceptually divided into control and dataflow, and these two aspects of the semantics will be presented separately. A semantics of control involves the modelling of Prolog's left–to–right computation or goal selection rule and depth–first search rule. Dataflow is concerned with the representation of the data objects over the Herbrand universe, the application of answer substitutions through the program, and the structure of computed results. Because a main motivation in this thesis is to formally describe logic program control, dataflow is often abstracted away. It is assumed in these cases that the data domain being used is the same as with pure logic programs – a domain using logical variables over the Herbrand universe. However, the semantics is easily extended to model the details of dataflow.

3.3 Control

This section presents a CCS semantics of the basic control strategy of Prolog. The definitions of some semantic operators for control are discussed in 3.3.1. Section 3.3.2 extends this semantics by considering a higher–level description of control. An example translation and symbolic computation is in section 3.3.3. See section 4.4 for a detailed comparison of the semantics and SLD resolution.

3.3.1 Control operators

The two events affecting the standard control strategy of Prolog are success and termination, which are represented by the actions $succ(\theta)$ and $done$ respectively. The sort for a program P is $\mathcal{L}(P) = \{succ(\theta), done,\}$. By convention, overlined actions are for output or produced communications, while non–overbarred or co–actions are used for input or consumed communications. For example, $\overline{done}$ is generated by a terminating agent, while $done$ will be used by an agent which is awaiting the termination signal of some other agent. A coincident action and co–action pair results in a handshake or basic communication between two agents.

The θ argument in a $succ(\theta)$ action is a computed answer substitution. It takes the form of a set of variable–term pairs $Var \leftarrow term$, which represent variable bindings. A typical example is $\theta = \{\ X \leftarrow a(Y),\ Y \leftarrow c\ \}$. The θ notation may be subscripted when necessary. Empty answer substitutions are denoted by ϵ. It is often convenient to represent $succ(\theta)$ by just $succ$, and assume that θ is implicit. The precise structure and use of these binding substitutions is discussed in section 3.5.

A fundamental concept is the stream. A successful finite computation takes the form of a stream of $succ$ actions,

$$\overline{succ(\theta_1)}\ .\ \overline{succ(\theta_2)}\ .\ \cdots\ .\ \overline{succ(\theta_k)}\ .\ \overline{done}\ .\ \mathbf{0} \qquad (k \geq 1)$$

where the action $\overline{succ(\theta_i)}$ $(1 \leq i \leq k)$ represents a successful derivation returning answer substitution θ_i, action $\overline{done}$ represents the end of the computation, "." separates stream elements, and the null agent $\mathbf{0}$ represents the end of all activity. Finite failure is represented by termination with no success actions:

$$\overline{done}\ .\ \mathbf{0}$$

The $\overline{done}$ action is a termination convention which is eventually communicated for all terminating computations. All agents to be used in the semantics are defined to be *well–terminating*, which means that they respect this termination protocol. Infinite streams have the form,

$$\overline{succ(\theta_1)}\ .\ \overline{succ(\theta_2)}\ .\ \cdots$$

Here, no $\overline{done}$ is ever generated.

$$
\begin{aligned}
[f] &\equiv [\, succ'/succ,\ done'/done \,] \\
F &\equiv \{\ succ',\ done'\ \}
\end{aligned}
$$

$$
\begin{aligned}
Done &\stackrel{\text{def}}{=} \overline{done}.\mathbf{0} \\
True &\stackrel{\text{def}}{=} \overline{succ(\epsilon)}.Done \\
False &\stackrel{\text{def}}{=} Done
\end{aligned}
$$

Figure 3.2: Some basic definitions

Some basic semantic definitions used throughout the thesis are in figure 3.2, while the basic semantics of control are in figure 3.3. Three types of equality are used in the figures. Syntactic substitutivity is denoted by $\equiv$, and is used to make semantic definitions more concise. Semantic equivalence is denoted $=$, and is used for defining the translation function $\mathcal{M}[\![\ \ ]\!]$ in figure 3.3. Lastly, CCS constant definitions, which define agents, use $\stackrel{\text{def}}{=}$.

In figure 3.2, the $[f]$ expression is a shorthand notation for the relabelling function which decorates the *succ* and *done* actions. The F notation denotes this set of decorated actions. The agent $Done$ denotes finite failure . The $True$ agent represents logical true, and results in a success action with an empty answer substitution, followed by termination. Conversely, $False$ represents finite failure.

The semantics of Prolog control is in figure 3.3. A function $\mathcal{M}[\![\ \ ]\!]$ converts Prolog program constructs to their CCS meanings. The CCS translations are automatically compiled from Prolog source programs. Throughout the figure, atoms are represented by capital letters, and tuples of terms are denoted $\tilde{t}$.

Prolog predicates define OR agents in (i). Here, a predicate is considered to be an ordered list of clauses of the same name and arity. The operational task of an OR agent is to linearly sequence its child AND agents which define the clauses for the predicate. This is done via the sequencing operator in (iv). P is executed concurrently with $b.Q$. Because all agents are well–terminating, P will generate a $\overline{done}$ action when it terminates. This action is relabelled to b, and is further restricted to make it local to this expression. Q waits for this b action to be generated; when it is seen, the $\overline{b}$ and b will handshake, and Q will resume execution. The net effect of this is for

(i) Predicates (OR agents)

$$\mathcal{M}[\![\ [P_1, P_2, \cdots, P_k]\ ]\!] = \quad P(\tilde{X}) \stackrel{\text{def}}{=} P_1(\tilde{X}) \,\mathring{;}\, P_2(\tilde{X}) \,\mathring{;}\, \cdots \,\mathring{;}\, P_k(\tilde{X})$$

(ii) Clauses (AND agents)

$$\begin{aligned} \mathcal{M}[\![\ P_i(\tilde{t}).\ ]\!] = &\quad P_i(\tilde{X}) \stackrel{\text{def}}{=} (\tilde{X} = \tilde{t}) \\ \mathcal{M}[\![\ P_i(\tilde{t}) :- A,\ B, \cdots, Z.\ ]\!] = &\quad P_i(\tilde{X}) \stackrel{\text{def}}{=} (\tilde{X} = \tilde{t}) \rhd \mathcal{M}[\![\ A\ ]\!] \rhd \mathcal{M}[\![\ B\ ]\!] \rhd \\ &\qquad\qquad \cdots \rhd \mathcal{M}[\![\ Z\ ]\!] \end{aligned}$$

(iii) Program queries

$$\mathcal{M}[\![\ :- A,\ B, \cdots, Z.\ ]\!] = \quad \mathcal{M}[\![\ A\ ]\!] \rhd \mathcal{M}[\![\ B\ ]\!] \rhd \cdots \rhd \mathcal{M}[\![\ Z\ ]\!]$$

(iv) Sequencing operator

$$P \,\mathring{;}\, Q \stackrel{\text{def}}{=} (\ P[b/done] \mid b.Q\ ) \setminus b$$

(v) Goal backtracking operator

$$\begin{aligned} P \rhd Q &\stackrel{\text{def}}{=} \quad (P[f] \mid NextGoal_i) \setminus F \\ NextGoal_i &\stackrel{\text{def}}{=} \quad succ'(\theta).(Q\theta \,\mathring{;}\, NextGoal_i) + done'.Done \end{aligned}$$

(vi) Single goal calls

$$\mathcal{M}[\![\ G(\tilde{t})\ ]\!] \stackrel{\text{def}}{=} \begin{cases} G(\tilde{t}) & : G \ is \ a \ defined \ predicate \\ Done & : G \ not \ defined \\ \overline{succ(\theta)}.Done + Done & : G \ is \ a \ builtin \ atom \end{cases}$$

Figure 3.3: CCS semantics of Prolog control

Q to execute if and only if P has terminated. Should P never terminate, Q will be suspended indefinitely. Getting back to the OR agent in (i), this sequencing strategy is employed between all the clauses in the predicate, and results in the execution of clauses using their textual order in the program. Each clause (AND agent) reference in the OR agent definition is uniquely labelled via subscripts.

AND agents are defined for individual clauses in (ii). Unification is treated as an explicit call to a unification algorithm. Argument terms are factored out of clause headers as follows. The clause

$$p(X, b(Y)) \ :- \ a(X), \ ...$$

is equivalent to

$$p(A, B) \ :- \ (A, B) = (X, b(Y)), \ a(X), \ ...$$

where $=$ is a builtin unifier. This clause is represented in CCS as

$$p_i(A, B) \stackrel{\text{def}}{=} ((A, B) = (X, b(Y)) \rhd a(X) \rhd \cdots$$

The definition of the unification agent is:

$$\tilde{t}_1 = \tilde{t}_2 \stackrel{\text{def}}{=} \overline{succ(\theta)}.Done + Done$$

It returns either $\overline{succ(\theta)}.Done$ where θ is the most general unifier of $\tilde{t}_1$ and $\tilde{t}_2$, or $Done$ if $\tilde{t}_1$ and $\tilde{t}_2$ do not unify.

AND agents for assertions have a single call to the unification agent. The operational task for AND agents with backtracking is to sequentially process the goals in the clause body using left–to–right exhaustive backtracking. This is done with the $\rhd$ backtracking operator in (v). In $P \rhd Q$, P is executed autonomously. Using $[f]$, all the output of P is relabelled, and then restricted by "$\backslash F$". The effect of this is to limit the scope of P's output to this expression, which will allow this expressions to be freely nested within other expressions without causing interference. The $NextGoal_i$ loop then processes all the output generated by P. When P computes a result, Q is invoked. When Q terminates, the loop calls itself recursively to process the next action of P. This continues until P terminates (signalled via the $done'$ action). Notation is abused somewhat in the definition of $NextGoal_i$. $NextGoal_i$ is uniquely distinguished by the "i" index for each unique instance of backtracked goal pairs in the program.

However, $NextGoal_i$ is more properly defined as an agent operator over Q:

$$
\begin{aligned}
P \rhd Q &\stackrel{\text{def}}{=} (P[f] \mid NextGoal(Q)) \setminus F \\
NextGoal(X) &\stackrel{\text{def}}{=} succ'(\theta).(X \mathbin{\hat{;}} NextGoal(X)) + done'.Done
\end{aligned}
$$

The $NextGoal_i$ notation has been chosen over this for conciseness, since the loop's context within expressions is usually known.

The semantic meaning of each goal in the AND agent of (ii) is defined in (vi). If a goal refers to a defined predicate, the OR agent for that predicate is used. Should the goal be undefined in the program database, then $Done$ or $False$ is used. If the goal refers to a builtin atom (eg. "="), then that agent will be called. It is assumed that builtin agents like = either return a result and terminate, or just terminate.

Finally, backtracked goal expressions in program queries (iii) have the same semantics as AND agent bodies in (ii).

$$
\begin{array}{l}
a(X,Y) :- \ p(X,Z),\ q(Z,Y). \\
a(X,Y) :- \ r(Y),\ a(Y,X). \\
p(f,g). \\
p(f,h). \\
q(h,i). \\
t(X,Y).
\end{array}
\quad \Longleftrightarrow \quad
\begin{array}{rcl}
a(X,Y) & \stackrel{\text{def}}{=} & a_1(X,Y) \mathbin{\hat{;}} a_2(X,Y) \\
a_1(X,Y) & \stackrel{\text{def}}{=} & p(X,Z) \rhd q(Z,Y) \\
a_2(X,Y) & \stackrel{\text{def}}{=} & Done \rhd a(Y,X) \\
\\
p(X,Y) & \stackrel{\text{def}}{=} & p_1(X,Y) \mathbin{\hat{;}} p_2(X,Y) \\
p_1(X,Y) & \stackrel{\text{def}}{=} & (X,Y) = (f,g) \\
p_2(X,Y) & \stackrel{\text{def}}{=} & (X,Y) = (f,h) \\
\\
q(X,Y) & \stackrel{\text{def}}{=} & q_1(X,Y) \\
q_1(X,Y) & \stackrel{\text{def}}{=} & (X,Y) = (h,i) \\
\\
t(X,Y) & \stackrel{\text{def}}{=} & t_1(X,Y) \\
t_1(X,Y) & \stackrel{\text{def}}{=} & True
\end{array}
$$

Figure 3.4: Logic program and CCS translation

An example translation is in figure 3.4. Note that because r is undefined, the goal $r(Y)$ is replaced with $Done$ or finite failure. Also note that there is a declarative interpretation of the CCS translation, where " $\hat{;}$ " is read as logical OR and "$\rhd$" as logical AND, and the atomic agent references are relations. The basic CCS rules of transition in figure 2.2 are applicable to this semantic translation and the definitions of $\fatsemi$ and $\triangleright$. This permits symbolic computation and other analyses to be performed on the logic program semantics.

Incidentally, the *or* (;) operator used in most Prolog implementations is simply modelled using $\fatsemi$:

$$\mathcal{M}[\![\, P_i : -A, (B;C), D.\,]\!] = P_i \stackrel{\text{def}}{=} A \triangleright (B \fatsemi C) \triangleright D$$

3.3.2 High level bisimilarities

$$\begin{array}{lrcl}
\textbf{Seq}-\textbf{1}: & Done \fatsemi P & \approx & P \\
\textbf{Seq}-\textbf{2}: & (\alpha\ .\ P) \fatsemi Q & \approx & \alpha\ .\ (P \fatsemi Q) \qquad (\alpha \neq \overline{done}) \\
\textbf{Back}-\textbf{1}: & (\overline{succ(\theta)}\ .\ P) \triangleright Q & \approx & P \trianglerighteq Q\theta \\
\textbf{Back}-\textbf{2}: & Done \triangleright Q & \approx & Done \\
\textbf{Back}-\textbf{3}: & P \trianglerighteq \overline{succ(\theta)}\ .\ Q & \approx & \overline{succ(\theta)}\ .\ (P \trianglerighteq Q) \\
\textbf{Back}-\textbf{4}: & P \trianglerighteq Done & \approx & P \triangleright Q \quad (\dagger) \\
\textbf{Back}-\textbf{5}: & (\overline{succ(\theta)}\ .\ P) \triangleright Q & \approx & Q\theta \fatsemi (P \triangleright Q)
\end{array}$$

Note: In (†) and elsewhere, the expression $X \trianglerighteq Y$ is an abbreviation for $X \trianglerighteq (Y, Y')$, where Y' is an original state of the right–hand side (explained in text).

Figure 3.5: High–level bisimilarities for standard control

A *bisimilarity* in CCS is an observable behavioural equivalence between agent expressions. It was discussed in section 2.2 that *weak bisimilarity* or *weak observational equivalence*, denoted in CCS by "$\approx$", is an adequate theory of behavioural equality for this thesis. Since the provision described in that section will be followed throughout the semantics[2], $\approx$ can be treated as an equivalence relation.

The bisimilarities in figure 3.5 define behaviour of the semantic operators at a higher–level than the basic CCS definitions of the control operators (new symbols used

[2] Substitutions within summation expressions are never done.

in them are defined below). These rules represent various states of the sequencing and backtracking mechanisms.

The **Seq** rules sequences agents so that the previous agent first issues $\overline{done}$ before the next agent proceeds. Because the **Seq** bisimilarities are very simple, the application of either of the two will simply be labelled as a **Seq** transition. The derivation of these **Seq** rules follow.

Theorem 3.3.1

$$\begin{array}{lrcll} \mathbf{Seq-1}: & Done \fatsemi P & \approx & P & \\ \mathbf{Seq-2}: & (\alpha.P) \fatsemi Q & \approx & \alpha.(P \fatsemi Q) & (\alpha \neq \overline{done}) \end{array}$$

Proof: *These bisimilarities are derived using the definition of* $\fatsemi$ *and the inference rules of the basic CCS operators.*[3]

$$\begin{array}{llcll} \mathbf{Seq-1}: & Done \fatsemi P & \approx & (\overline{done}.\mathbf{0}) \fatsemi P & : \mathbf{Con}\ \ Done \\ & & \approx & ((\overline{done}.\mathbf{0})[b/done] \mid b.P) \setminus b & : \mathbf{Con}\ \ \fatsemi \\ & & \approx & (\overline{b}.\mathbf{0} \mid b.P) \setminus b & : \mathbf{Rel} \\ & & \approx & (\mathbf{0} \mid P) \setminus b & : expansion \\ & & \approx & P \setminus b & : (\mathbf{0}|P) \approx P \\ & & \approx & P & : b \notin \mathcal{L}(P)\ \ (by\ \ defn\ \ \fatsemi) \end{array}$$

$$\begin{array}{llcll} \mathbf{Seq-2}: & (\alpha.P) \fatsemi Q & \approx & (\alpha.P)[b/done] \mid b.Q) \setminus b & : \mathbf{Con}\ \ \fatsemi \\ & & \approx & ((\alpha.(P[b/done])) \mid b.Q) \setminus b & : \mathbf{Rel},\ \ \alpha \neq \overline{done} \\ & & \approx & \alpha.(P[b/done] \mid b.Q) \setminus b & : expansion \\ & & \approx & \alpha.(P \fatsemi Q) & : defn.\ \ \fatsemi \end{array}$$

□

An OR agent's exhaustive search of sequenced clauses means that multiple solutions are obtained for a particular goal invocation. These solutions take the form of streams of computed answer substitutions, and allow these streams to be processed in a manner which models Prolog control. However, rather than using the CCS expansion theorem, it is convenient to use the **Back** bisimilarities for stream processing. Their net effect is to take each result from the left hand side, and apply it to the right hand side, until the result stream of the left hand side is exhausted. A new operator is used:

$$P \ \rhd_i Q \stackrel{\text{def}}{=} (P[f] \mid (Q \fatsemi NextGoal_i)) \setminus F$$

The expression $A \rhd\ B$ represents the state of the backtracking mechanism between agents A and B when the computation of B is active. The "i" index refers to the

[3] Henceforth, the right–hand column of derivations will justify the inference step just applied to derive the current expression. The CCS rules of figure 2.2 will be freely referred to.

$NextGoal_i$ loop, which is defined within $P \rhd Q$. This is a convenient way of circumventing the need to carry an extra parameter in $\unrhd$, such as in:

$$P \ \unrhd \ (Q, Q') \stackrel{\text{def}}{=} (P[f] \mid (Q' \,\S\, NextGoal(Q))) \setminus F$$

where $NextGoal(Q)$ loops over the agent expression Q. However, the indexed version is syntactically simpler, and the "i" index can be ignored, since the context of $\unrhd$ within expressions is usually known. The **Back** derivations follow. Each **Back** rule is justified by using the expansion theorem along with the definitions of $\rhd$ and $\unrhd$.

Theorem 3.3.2 *Let P and Q be agents that generate well–terminating streams. Then the following bisimilarities hold:*

$$\begin{array}{lrcl}
\textbf{Back}-\textbf{1}: & (\overline{succ(\theta)}\ .\ P) \rhd Q & \approx & P \unrhd Q\theta \\
\textbf{Back}-\textbf{2}: & Done \rhd Q & \approx & Done \\
\textbf{Back}-\textbf{3}: & P \unrhd \overline{succ(\theta)}\ .\ Q & \approx & \overline{succ(\theta)}\ .\ (P \unrhd Q) \\
\textbf{Back}-\textbf{4}: & P \unrhd Done & \approx & P \rhd Q \\
\textbf{Back}-\textbf{5}: & (\overline{succ(\theta)}\ .\ P) \rhd Q & \approx & Q\theta \,\S\, (P \rhd Q)
\end{array}$$

Proof: *The derivations take the same form as those for* **Seq**. *Note that the form of P is only relevant so far as it conforms to those given in the bisimilarities; nonterminating and looping P do not affect the integrity of the bisimilarities. (Also note the comment in figure 3.5, which explains the form of* **Back-4**.*)*

$$\begin{array}{lll}
\multicolumn{2}{l}{\textbf{Back}-\textbf{1}:} & \\
\multicolumn{2}{l}{(\overline{succ(\theta)}.P) \rhd Q} & \\
\approx & ((\overline{succ(\theta)}.P)[f] \mid succ(\theta)'.(Q\theta \,\S\, NextGoal_i) + done'.Done) \setminus F & : \textbf{Con} \rhd \\
\approx & (\overline{succ(\theta)}'.(P[f]) \mid succ(\theta)'.(Q\theta \,\S\, NextGoal_i) + done'.Done) \setminus F & : \textbf{Rel}\ f \\
\approx & (P[f] \mid (Q\theta \,\S\, NextGoal_i) + done'.Done) \setminus F & : expansion \\
\approx & P \unrhd Q\theta & : defn.\ \unrhd
\end{array}$$

The notation $Q\theta$ denotes the application of the binding computed by P onto Q.

$$\begin{array}{lll}
\multicolumn{2}{l}{\textbf{Back}-\textbf{4}:} & \\
\multicolumn{2}{l}{P \unrhd Done} & \\
\approx & (P[f] \mid (Done \,\S\, NextGoal_i)) \setminus F & : \textbf{Con} \unrhd \\
\approx & (P[f] \mid (Done \,\S\, NextGoal(Q))) \setminus F & : defn\ \ NextGoal_i \\
\approx & (P[f] \mid NextGoal(Q)) \setminus F & : \textbf{Seq} \\
\approx & (P[f] \mid succ(\theta)'.(Q\theta \,\S\, NextGoal(Q)) + done'.Done) \setminus F & : \textbf{Con}\ \ NextGoal(Q) \\
\approx & (P[f] \mid succ(\theta)'.(Q\theta \,\S\, NextGoal_i)) + done'.Done) \setminus F & : defn.\ \ NextGoal_i \\
\approx & P \rhd Q & : defn. \rhd
\end{array}$$

The derivations of **Back-2** *and* **Back-3** *are similar to* **Back-1** *and* **Back-4**, *and are* *omitted. The derivation of* **Back-5** *is more involved. Firstly, by applying* **Back-1**, *the LHS can be rewritten as,*

$$(\overline{succ(\theta)}.P) \rhd Q \;\approx\; P \mathbin{\dot{\rhd}} Q\theta \qquad (\mathbf{1})$$

The bisimilarity is now shown by structural induction on the size of the stream generated by Q in $(\mathbf{1})$ *and the RHS.*

<u>Base case</u>: Substituting $Q\theta \approx Done$ in both expressions:

$$\begin{array}{lrcll}
(\mathbf{1}): & P \mathbin{\dot{\rhd}} Q\theta & \approx & P \mathbin{\dot{\rhd}} Done & : subst.\ Q\theta \\
 & & \approx & P \rhd Q & : \mathbf{Back-4} \\[1em]
RHS: & Q\theta \mathbin{\hat{;}} (P \rhd Q) & \approx & Done \mathbin{\hat{;}} (P \rhd Q) & : subst.\ Q \\
 & & \approx & P \rhd Q & : \mathbf{Seq}
\end{array}$$

<u>Inductive case</u>: Let $Q\theta \approx succ(\gamma).Q'$:

$$\begin{array}{lrclcl}
(\mathbf{1}): & P \mathbin{\dot{\rhd}} Q\theta & \approx & P \mathbin{\dot{\rhd}} \overline{succ(\gamma)}.Q' & & : subst.\ Q \\
 & & \approx & \overline{succ(\gamma)}.(P \mathbin{\dot{\rhd}} Q') & (\mathbf{2}) & : \mathbf{Back-3} \\[1em]
RHS: & Q\theta \mathbin{\hat{;}} (P \rhd Q) & \approx & (\overline{succ(\gamma)}.Q') \mathbin{\hat{;}} (P \rhd Q) & & : subst.\ Q\theta \\
 & & \approx & \overline{succ(\gamma)}.(Q' \mathbin{\hat{;}} (P \rhd Q)) & (\mathbf{3}) & : expansion
\end{array}$$

Now, $(\mathbf{2})$ *and* $(\mathbf{3})$ *are bisimilar by the induction hypothesis, since Q' generates a smaller stream than Q. The bisimilarity holds for terminating Q.*

When $Q\theta$ loops, then the expressions are bisimilar, since the LHS yields

$$P \mathbin{\dot{\rhd}} \bot \xrightarrow{\epsilon} P \mathbin{\dot{\rhd}} \bot$$

and the RHS gives

$$\bot \mathbin{\hat{;}} (P \rhd Q) \xrightarrow{\epsilon} \bot \mathbin{\hat{;}} (P \rhd Q)$$

These expressions are therefore bisimilar. (The ϵ notation means that application of the expansion theorem yields no action sequence.) □

The fact that an agent is nonterminating is irrelevant for the **Seq** and **Back-1** through **Back-4** bisimilarities, since they are only concerned with the immediate actions of their argument expressions.

Some normal forms for an expression with backtracking are identifiable, and are normal in the sense that any backtracking expression reduces to one of them. Note that the canonical normal form of any non–looping semantic expression is, of course, a stream of zero or more computed answer substitutions, possibly terminated by *Done*. However, the normal forms identified here will pertain to the intermediate expressions obtained during backtracking.

Theorem 3.3.3 *The normal forms for an expression $P \rhd Q$ are*

$$\begin{array}{l} P \rhd Q \\ P \triangleright Q \\ \overline{succ}.(P \triangleright Q) \\ Done \end{array}$$

Proof: *Agent behaviours are denoted by terminating or non–terminating action sequences composed of succ and done actions. When expressions denoting these agent behaviours are placed within expressions with $\rhd$ and $\triangleright$, a finite number of expression forms are produced. These expressions are the normal forms.*

The proof proceeds by applying the expansion theorem to the expression $A \rhd B$, and showing that all possible derivatives of this expression are normal forms. Three possible behaviours of agents A and B are: (i) $A \overset{\overline{succ}}{\rightarrow} A'$; (ii) $A \overset{\overline{done}}{\rightarrow} \mathbf{0}$; and (iii) $B \overset{\alpha}{\rightarrow} B'$. Letting, A and B generate all these behaviours, by applying the expansion theorem to $A \rhd B$:

$$\begin{aligned} A \rhd B \ &\approx\ \underbrace{(A'[f] \mid (B \fatsemi NextGoal_i)\backslash F}_{using\ (i)} \ + \ \underbrace{(\mathbf{0} \mid Done)\backslash F}_{using\ (ii)} \\ &\approx\ A' \triangleright B \ + \ Done \qquad (\mathbf{1}) \end{aligned}$$

The behaviour of B in (iii) has no effect in the above, as "$\backslash F$" suppresses it. Now $A' \triangleright B$ is similarly expanded. The possible behaviours of A' and B are: (iv) $A' \overset{\overline{succ}}{\rightarrow} A''$; (v) $A' \overset{\overline{done}}{\rightarrow} \mathbf{0}$; (vi) $B \overset{\overline{succ}}{\rightarrow} B'$; and (vii) $B \overset{\overline{done}}{\rightarrow} \mathbf{0}$. Using these cases, the expansion theorem is applied to $A' \triangleright B$ in $(\mathbf{1})$:

$$\begin{aligned} A' \triangleright B \ &\approx\ \underbrace{\overline{succ}.(A'[f] \mid (B' \fatsemi NextGoal_i)\backslash F}_{using\ (vi)} \ + \ \underbrace{(A'[f] \mid (NextGoal_i)\backslash F}_{using\ (vii)} \\ &\approx\ \overline{succ}.(A' \triangleright B') \ + \ A' \triangleright B \qquad (\mathbf{2}) \end{aligned}$$

Behaviours (iv) and (v) have no effect, as they are restricted.

In $(\mathbf{1})$ and $(\mathbf{2})$ above, the derivatives are finite in number and form, and are therefore normal forms. $\square$

The next theorem shows that **Back-1** through **Back-4** are operationally complete, in the sense that they account for all the possible transitions between the normal forms of theorem 3.3.3.

Theorem 3.3.4 *The* **Back-1**, **Back-2**, **Back-3**, *and* **Back-4** *bisimilarities completely account for all the possible transitions between normal forms.*

Proof: *The proof follows theorem 3.3.3, except that a* **Back** *bisimilarity is identified with each normal form derived. Three possible behaviours of agents A and B are: (i)* $A \stackrel{\overline{succ}}{\rightarrow} A'$*; (ii)* $A \stackrel{\overline{done}}{\rightarrow} \mathbf{0}$*; and (iii)* $B \stackrel{\alpha}{\rightarrow} B'$*. Applying the expansion theorem to* $A \rhd B$*:*

$$A \rhd B \approx \underbrace{(A'[f] \mid (B \,\hat{;}\, NextGoal_i)\backslash F}_{using\ (i)} + \underbrace{(\mathbf{0} \mid Done)\backslash F}_{using\ (ii)}$$

$$\approx \underbrace{A' \rhd B}_{(i)\ \mathbf{Back-1}} + \underbrace{Done}_{(ii)\ \mathbf{Back-2}} \qquad (\mathbf{1})$$

This represents the use of **Back-1** *and* **Back-2**. *Now* $A' \rhd B$ *is similarly expanded. The possible behaviours of* A' *and* B *are: (iv)* $A' \stackrel{\overline{succ}}{\rightarrow} A''$*; (v)* $A' \stackrel{\overline{done}}{\rightarrow} \mathbf{0}$*; (vi)* $B \stackrel{\overline{succ}}{\rightarrow} B'$*; and (vii)* $B \stackrel{\overline{done}}{\rightarrow} \mathbf{0}$. *Using these cases, the expansion theorem is applied to* $A' \rhd B$ *in* **(1)***:*

$$A' \rhd B \approx \underbrace{\overline{succ}.(A'[f] \mid (B' \,\hat{;}\, NextGoal_i)\backslash F}_{using\ (vi)} + \underbrace{(A'[f] \mid (NextGoal_i)\backslash F}_{using\ (vii)}$$

$$\approx \underbrace{\overline{succ}.(A' \rhd B')}_{(vi)\ \mathbf{Back-3}} + \underbrace{A' \rhd B}_{(vii)\ \mathbf{Back-4}} \qquad (\mathbf{2})$$

The use of **Back-3** *and* **Back-4** *are shown here.*

In **(1)** *and* **(2)** *above, each normal form is derived by applying one of the backtracking bisimilarities. Thus the* **Back** *rules are complete.* □

A result of this theorem is that **Back-5** is actually redundant. It is still useful within proofs.

Another useful bisimilarity is the **Resol** rule of figure 3.6. This rule defines the behaviour of a single resolution step, and permits a higher–level abstraction of agent invocation. The proof of **Resol** follows.

Resol :

$$P_i(\tilde{t}) \approx \begin{cases} (i)\ Done & : \tilde{t} \text{ and } \tilde{t}_i \text{ do not unify} \\ (ii)\ \overline{succ(\theta)}.Done & : \theta = mgu(\tilde{t}, \tilde{t}_i), \text{ and } P_i(\tilde{x}) \stackrel{\text{def}}{=} (\tilde{x} = \tilde{t}_i) \\ (iii)\ Q\theta & : \theta = mgu(\tilde{t}, \tilde{t}_i), \text{ and } P_i(\tilde{x}) \stackrel{\text{def}}{=} (\tilde{x} = \tilde{t}_i) \rhd Q \end{cases}$$

Figure 3.6: Resolution rule

Theorem 3.3.5

Resol :

$$P_i(\tilde{t}) \approx \begin{cases} (i)\ Done & : \tilde{t} \text{ and } \tilde{t}_i \text{ do not unify} \\ (ii)\ \overline{succ(\theta)}.Done & : \theta = mgu(\tilde{t}, \tilde{t}_i), \text{ and } P_i(\tilde{x}) \stackrel{\text{def}}{=} (\tilde{x} = \tilde{t}_i) \\ (iii)\ Q\theta & : \theta = mgu(\tilde{t}, \tilde{t}_i), \text{ and } P_i(\tilde{x}) \stackrel{\text{def}}{=} (\tilde{x} = \tilde{t}_i) \rhd Q \end{cases}$$

Proof: *Each case is proved separately.*

Case (i): Non–unifiability. This means that the call to the unification agent is bisimilar to Done. Therefore, if P_i is an assertion, then $P_i(\tilde{t}) \approx (\tilde{t} = \tilde{t}_i) \approx Done$. If P is a rule, then by using **Back-2**, $P_i(\tilde{t}) \approx (\tilde{t} = \tilde{t}_i) \rhd Q \approx Done \rhd Q \approx Done$.

Case (ii): Successful unification with an assertion. Like case (i), substituting the equivalence $\overline{succ(\theta)}.Done$ for the unification call gives the desired result, where θ is the mgu.

Case (iii): Successful unification with a rule. Let $P_i(\tilde{X}) \stackrel{\text{def}}{=} (\tilde{X} = \tilde{t}_i) \rhd Q$, and let the call be $P_i(\tilde{t})$. Then $(\tilde{t} = \tilde{t}_i) \approx \overline{succ(\theta)}.Done$ for a mgu θ. Expanding P_i:

$$\begin{array}{rcll} P_i(\tilde{t}) & \approx & \underline{(\tilde{t} = \tilde{t}_i)} \rhd Q & : \textbf{Con } P_i \\ & \approx & \overline{succ(\theta)}.Done \rhd Q & : subst.\ unification \\ & \approx & Q\theta \,⨟\, (Done \rhd Q) & : \textbf{Back} - \textbf{5} \\ & \approx & Q\theta \,⨟\, Done & : \textbf{Back} - \textbf{2} \\ & \approx & Q\theta & : P \,⨟\, Done \approx P \end{array}$$

□

3.3.3 Example symbolic computation

The control bisimilarities model computational behaviour identical to that of Prolog. An example symbolic computation using them is now given. Given the program and CCS translation in figure 3.4, the following is a symbolic evaluation of the query

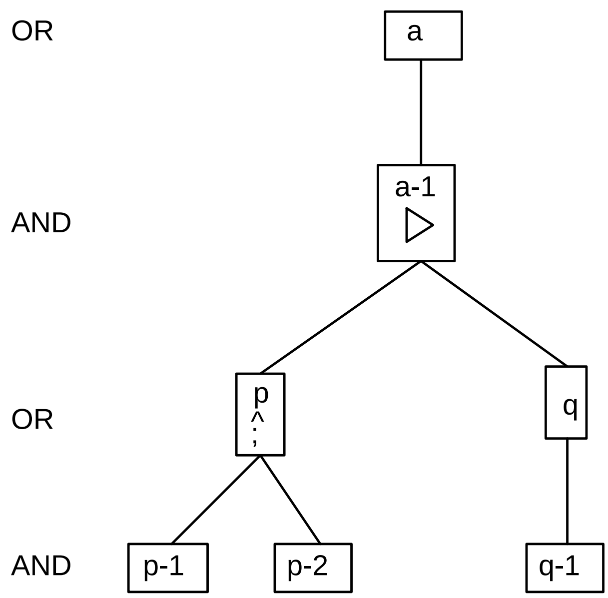

Figure 3.7: AND/OR process tree

"$?-\ a(X,Y)$". First, the stream generated by a general call to p is determined:

$$
\begin{array}{ll}
p(X,Z) & \\
\approx\ p_1(X,Z)\ \hat{;}\ p_2(X,Z) & : \textbf{Con}\ \ p \\
\approx\ (\overline{succ(\theta_1)}.Done)\ \hat{;}\ (\overline{succ(\theta_2)}.Done) & : \textbf{Resol},\ \ \theta_1 = \{X \leftarrow f, Z \leftarrow g\}, \\
 & \ \ \theta_2 = \{X \leftarrow f, Z \leftarrow h\} \\
\approx\ \overline{succ(\theta_1)}.\overline{succ(\theta_2)}.Done & : \textbf{Seq}
\end{array}
$$

The call to $a(X,Y)$ is now derived using the above stream:

$$
\begin{array}{lll}
 & a(X,Y) & \\
\approx & a_1(X,Y) \fatsemi a_2(X,Y) & : \textbf{Con } a \\
\approx & (p(X,Z) \triangleright \underline{q(Z,Y))} \fatsemi a_2(X,Y) & : \textbf{Con } a_1 \\
\approx & ((\overline{succ(\theta_1)}.\overline{succ(\theta_2)}.Done) \triangleright q(Z,Y)) \fatsemi a_2(X,Y) & : \textit{subst. } p(X,Z) \textit{ above} \\
\approx & ((\overline{succ(\theta_2)}.Done) \triangleright q(g,Y)) \fatsemi a_2(X,Y) & : \textbf{Back} - \textbf{1} \\
\approx & ((\overline{succ(\theta_2)}.Done) \triangleright Done) \fatsemi a_2(X,Y) & : \textbf{Con } q, \ \textbf{Resol } q_1 \\
\approx & ((\overline{succ(\theta_2)}.Done) \triangleright q(Z,Y)) \fatsemi a_2(X,Y) & : \textbf{Back} - \textbf{4} \\
\approx & (Done \triangleright \underline{q(h,Y))} \fatsemi a_2(X,Y) & : \textbf{Back} - \textbf{1} \\
\approx & (Done \triangleright \overline{succ(\theta_3)}.Done) \fatsemi a_2(X,Y) & : \textbf{Con } q, \ \textbf{Resol } q_1, \\
 & & \ \theta_3 = \{Y \leftarrow i\} \\
\approx & \overline{succ(\theta_3)}.(Done \triangleright Done) \fatsemi a_2(X,Y) & : \textbf{Back} - \textbf{3} \\
\approx & \overline{succ(\theta_3)}.(Done \triangleright q(Z,Y)) \fatsemi a_2(X,Y) & : \textbf{Back} - \textbf{4} \\
\approx & \overline{succ(\theta_3)}.Done \fatsemi a_2(X,Y) & : \textbf{Back} - \textbf{2} \\
\approx & \overline{succ(\theta_3)}.a_2(X,Y) & : \textbf{Seq} \\
\approx & \overline{succ(\theta_3)}.(Done \triangleright a(Y,X)) & : \textbf{Con } a_2 \\
\approx & \overline{succ(\theta_3)}.Done & : \textbf{Back} - \textbf{2}
\end{array}
$$

An AND/OR process tree representation of the expression "$(p(X,Z) \triangleright q(Z,Y)) \fatsemi q_2(X,Y)$" in the above derivation is in figure 3.7. Alternatively, if steps are performed in unison, and if the **Back-5** bisimilarity is used, a more concise derivation is possible:

$$
\begin{array}{lll}
 & a(X,Y) & \\
\approx & (p(X,Z) \triangleright \underline{q(Z,Y))} \fatsemi a_2(X,Y) & : \textbf{Con } a, \ a_1 \\
\approx & ((\overline{succ(\theta_1)}.\overline{succ(\theta_2)}.Done) \triangleright q(Z,Y)) \fatsemi a_2(X,Y) & : \textit{subst. } p(X,Z) \\
\approx & q(g,Y) \fatsemi ((\overline{succ(\theta_2)}.Done) \triangleright q(Z,Y)) \fatsemi a_2(X,Y) & : \textbf{Back} - \textbf{5} \\
\approx & q(g,Y) \fatsemi q(h,Y) \fatsemi (Done \triangleright q(Z,Y)) \fatsemi a_2(X,Y) & : \textbf{Back} - \textbf{5} \\
\approx & q(g,Y) \fatsemi q(h,Y) \fatsemi Done \fatsemi a_2(X,Y) & : \textbf{Back} - \textbf{2} \\
\approx & \underline{q(h,Y)} \fatsemi Done \fatsemi a_2(X,Y) & : \textbf{Resol } q, \ \textbf{Seq} \\
\approx & \overline{succ(\theta_3)}.Done \fatsemi a_2(X,Y) & : \textbf{Resol } q, \ \textbf{Seq} \\
\approx & \overline{succ(\theta_3)}.a_2(X,Y) & : \textbf{Seq} \\
\approx & \overline{succ(\theta_3)}.Done & : \textbf{Con } a_2, \ \textbf{Back} - \textbf{2}
\end{array}
$$

Throughout the above derivations, there is an implicit dataflow used which reflects what occurs within Prolog computations. For example, variables are treated as logical variables, logical variables are automatically renamed when necessary, variables are universally quantified within AND agents, and variables are existentially quantified between sequenced AND agent calls within OR agents. Section 3.5 discusses this in more depth.

3.4 The cut

One of sequential Prolog's less desirable features is the cut, " ! ". The cut affects the operational semantics of a program, and is used to allow a measure of user control over Prolog's standard left–to–right depth–first control strategy. Used judiciously, it is an effective means by which the search tree can be pruned during a program's computation. Although the cuts in a program are usually ignored when ascertaining a program's declarative semantics, the use of a cut would imply that the program is possibly not written declaratively, since declarative programs should have as little operational information encoded within them as possible.

Consider the predicate,

$$\begin{array}{lll} p & :- & a,\ b,\ !,\ d. \\ p & :- & e,\ f. \\ p & :- & g,\ h. \end{array}$$

The operational behaviour of the cut is as follows. Goals a and b are first resolved. Should a or b fail, then control passes to the second clause of p. If a and b succeed, then the cut is activated, and d is then resolved. However, activation of the cut means that the choicepoints for a and b are discarded: a and b are not backtrackable any more. In addition, the clauses following the activated cut are not searched. The net effect of the cut is to commit the search to the clause containing the cut, and not backtrack to goals resolved prior to the cut.

3.4.1 Semantics of the cut

A general schema for translating Prolog programs with cuts into CCS is outlined in figure 3.8. In addition to $\hat{;}$ and $\triangleright$ from before, three new operators, $\mathring{;}$, $\triangleright\!\!|$, and $\triangleright\!\!|_\ell$ are used. By convention, $\triangleright$ binds tighter than $\triangleright\!\!|$, and $\triangleright\!\!|$ binds tighter than $\triangleright\!\!|_\ell$. These conventions affect the semantics of CCS expressions. Some example translations are in figure 3.9, with bindings explicitly represented with parentheses.

Figure 3.10 defines some operators which define the operational semantics of Prolog's cut. The cut operators $\triangleright\!\!|$ and $\triangleright\!\!|_\ell$ correspond syntactically to the cuts within a clause. In the definition of $\triangleright\!\!|$,

$$A \triangleright\!\!| B \stackrel{\text{def}}{=} (A[f] \mid (succ'(\theta).B\theta \ + \ done'.Done)) \setminus F$$

only the first solution is obtained from A, after which B is invoked and A is ignored.

$$
\begin{array}{l}
Let\ \ predicate\ \ P\ \ be\ \ defined\ \ by\ \ clauses\ \ [P_1, ..., P_k],\\
and\ \ P_i :-\ Goals.\ \ be\ \ a\ \ clause\ \ of\ \ P.
\end{array}
$$

$$
\mathcal{M}[\![\ [P_1, \cdots, P_i, P_{i+1}, \cdots, P_k]\ ]\!] = \begin{cases} P \stackrel{\mathrm{def}}{=} \cdots P_i \overset{\circ}{;} P_{i+1} \cdots & : cut \in P_i \\ P \stackrel{\mathrm{def}}{=} \cdots P_i \overset{\wedge}{;} P_{i+1} \cdots & : otherwise \end{cases}
$$

$$
\mathcal{M}[\![\ P_i :-\ Goals.\ ]\!] = \begin{cases} P_i \stackrel{\mathrm{def}}{=} \mathcal{M}[\![\ Goals\ ]\!] & : cut \in Goals,\ \ P_i\ \ not\ \ last\ \ clause \\ P_i \stackrel{\mathrm{def}}{=} \mathcal{M}\llbracket\ Goals\ \rrbracket & : otherwise \end{cases}
$$

$$
\mathcal{M}[\![\ Goals\ ]\!] = \begin{cases} True & : Goals\ = !. \\ \mathcal{M}\llbracket\ Goals'\ \rrbracket & : Goals\ = !, Goals'. \\ a \rhd|_\ell\ \ \mathcal{M}\llbracket\ Goals'\ \rrbracket & : Goals\ =\ a, !, Goals' \\ a \rhd \mathcal{M}[\![\ Goals'\ ]\!] & : Goals\ =\ a, Goals' \end{cases}
$$

$$
\mathcal{M}\llbracket\ Goals\ \rrbracket = \begin{cases} True & : Goals\ =\ null \\ a & : Goals\ =\ a \\ a \rhd| \ \mathcal{M}\llbracket\ Goals'\ \rrbracket & : Goals\ =\ a, !, Goals' \\ a \rhd \mathcal{M}\llbracket\ Goals'\ \rrbracket & : Goals\ =\ a, Goals' \end{cases}
$$

Figure 3.8: Translating predicates with cuts into CCS

$$
\begin{aligned}
\mathcal{M}[\![\ P_i\ \ :-\ !.\ ]\!] &= P_i \stackrel{\mathrm{def}}{=} True \\
\mathcal{M}[\![\ P_i\ \ :-\ a, b, c, !.\ ]\!] &= P_i \stackrel{\mathrm{def}}{=} (a \rhd b \rhd c) \rhd|_\ell\ \ True \\
\mathcal{M}[\![\ P_i\ \ :-\ !, a, b, c.\ ]\!] &= P_i \stackrel{\mathrm{def}}{=} a \rhd b \rhd c \\
\mathcal{M}[\![\ P_i\ \ :-\ a, b, !, c, d.\ ]\!] &= P_i \stackrel{\mathrm{def}}{=} (a \rhd b) \rhd|_\ell\ \ (c \rhd d) \\
\mathcal{M}[\![\ P_i\ \ :-\ a, b, !, c, d, !, e.\ ]\!] &= P_i \stackrel{\mathrm{def}}{=} (a \rhd b) \rhd|_\ell\ \ ((c \rhd d) \rhd|\ e)
\end{aligned}
$$

$$
where\ \ P \stackrel{\mathrm{def}}{=} \cdots P_i \overset{\circ}{;} P_{i+1} \cdots
$$

Figure 3.9: Example translations of clauses with cuts

$$
\begin{aligned}
A \rhd\!| \; B &\overset{\text{def}}{=} (A[f] \mid (succ'(\theta).B\theta \; + \; done'.Done)) \setminus F \\
A \rhd\!|_{\ell} \; B &\overset{\text{def}}{=} (A[f] \mid (succ'(\theta).B\theta \; + \; done'.Done[\ell/done])) \setminus F \\
P \overset{\circ}{;} Q &\overset{\text{def}}{=} (P \mid \ell.Q) \setminus \ell \\
where \;\; \ell &\notin \mathcal{L}(A) \cup \mathcal{L}(B) \cup \mathcal{L}(Q)
\end{aligned}
$$

Figure 3.10: Operator definitions for cut

This differs from the backtracking operator $\rhd$:

$$A \rhd B \overset{\text{def}}{=} (A[f] \mid (succ'(\theta).(B\theta \hat{;} NextGoal_i) + done'.Done)) \setminus F$$

where successive solutions from A are retrieved via the $NextGoal_i$ loop. The $\rhd\!|_{\ell}$ operator differs from $\rhd\!|$ in that it is used when the search of following clauses is dependent upon whether this cut (the first) succeeds or fails. Consequently, $\rhd\!|_{\ell}$ uses two possible termination signals, ℓ when the cut has not been activated and search should continue in following clauses, and *done* when the cut has been activated and search is pruned. The sequencing operator $\overset{\circ}{;}$ is almost identical to $\hat{;}$:

$$
\begin{aligned}
P \overset{\circ}{;} Q &\overset{\text{def}}{=} (P \mid \ell.Q) \setminus \ell \\
P \hat{;} Q &\overset{\text{def}}{=} (P \; [\ell/done] \mid \ell.Q) \setminus \ell
\end{aligned}
$$

The difference is that $\overset{\circ}{;}$ does not relabel the termination signal *done* from P. Instead, this relabelling is done elsewhere within the clause by $\rhd\!|_{\ell}$.

Two events happen when a cut is activated: (i) the choice points of the goals found prior to the cut in the clause are discarded; (ii) the clauses following the clause with the cut are not searched. The CCS representation of these events is done by suspending agents. This is performed in CCS by simply not communicating to the agents which are to be suspended – by forcing deadlock. When a cut is activated, $\rhd\!|$ and $\rhd\!|_{\ell}$ immediately force deadlock with the agent on the left–hand side. This represents the pruning of backtracking amongst previous goals in the clause. In addition, $\rhd\!|_{\ell}$ also changes the *done* signal so that clauses following the activated clause are deadlocked, which models the pruning of the search space. The $\overset{\circ}{;}$ is defined to deadlock subsequent clauses if the cut is activated, or to execute them otherwise. The deadlock mechanism is illustrated in the next theorem, which states that any agent whose immediate actions are restricted is bisimilar to $\mathbf{0}$, since it is deadlocked.

Theorem 3.4.1 *For all $\alpha \in \mathcal{L}(P)$,*

$$(\textstyle\sum \alpha.P) \setminus \alpha \;\approx\; \mathbf{0}$$

Proof: *The restriction of α in $\alpha.P$ creates deadlock, since restriction prevents the communication of any α or $\overline{\alpha}$ from occurring outside the expression. Applying the expansion theorem results in $\mathbf{0}$, since all immediate actions α are restricted.* □

Theorem 3.4.1 is used for suspending both OR and AND agents. With OR agents, the expression $P \overset{\circ}{;} Q$ means that the sequencing of Q is conditional upon the agent P renaming its termination signal to the one expected by $\overset{\circ}{;}$ (ie. ℓ). This conditional sequencing is performed by the use of $\rhd\!\!|_{\ell}$ within the clause with the cut. The following theorems illustrate how agents are suspended using this scheme.

Theorem 3.4.2 *Let P be a terminating OR agent which produces the action sequence s, and then quits with termination signal $\overline{\ell}$. So $P \approx s.\overline{\ell}.\mathbf{0}$, and $P \overset{\circ}{;} Q \approx s.Q$. Otherwise, if P terminates with some termination signal $\alpha \neq \ell$, and $P \approx s.\alpha.\mathbf{0}$, then $P \overset{\circ}{;} Q \approx P$.*

Proof: *Let P be an agent which produces the action sequence s terminated by the termination signal ℓ, ie. $P \approx s.\overline{\ell}.\mathbf{0}$. Then,*

$$\begin{array}{lll}
P \overset{\circ}{;} Q & \approx (P \mid \ell.Q) \setminus \ell & : \mathbf{Con}\ \overset{\circ}{;} \\
& \approx (s.\overline{\ell}.\mathbf{0} \mid \ell.Q) \setminus \ell & : subst.\ P \\
& \approx s.(\overline{\ell}.\mathbf{0} \mid \ell.Q) \setminus \ell & : expansion \\
& \approx s.(\mathbf{0} \mid Q) \setminus \ell & : expansion \\
& \approx s.Q & : simplify\ \ (\mathbf{0} \mid Q \approx Q,\ \ \ell \notin \mathcal{L}(Q))
\end{array}$$

However, should P not *do this relabelling, then $\ell \notin \mathcal{L}(P)$, and P terminates with a different termination signal not seen by $\overset{\circ}{;}$. Let $P \approx s.\overline{done}.\mathbf{0}$:*

$$\begin{array}{lll}
P \overset{\circ}{;} Q & \approx s.\overline{done}.\mathbf{0} \overset{\circ}{;} Q & : subst.\ P \\
& \approx s.\overline{done}.(\mathbf{0} \mid \ell.Q) \setminus \ell & : \mathbf{Con}\ \overset{\circ}{;},\ expansion \\
& \approx s.\overline{done}.(\ell.Q) \setminus \ell & : simplify \\
& \approx s.\overline{done}.\mathbf{0} & : theorem\ \ 3.4.1 \\
& \approx P & : subst.\ P
\end{array}$$

□

Theorem 3.4.3 *Let A be an AND agent that produces a solution, ie. $A \approx \overline{succ(\theta)}.A'$. Then $A \rhd\!\!| B \approx B\theta$.*

Proof: *Recall $B\theta$ represents the execution of B with all appropriate bindings applied to its arguments. Expanding $A \rhd\!\!| B$,*

$$
\begin{array}{rcll}
A \rhd\!\!| B & \approx & (A[f] \mid (succ'(\theta).B + done'.Done)) \setminus F & : \mathbf{Con} \;\; \rhd\!\!| \\
& \approx & (\overline{succ(\theta)}.A'[f] \mid (succ'(\theta).B + done'.Done)) \setminus F & : subst. \;\; A \\
& \approx & (A'[f] \mid B\theta) \setminus F & : expansion \\
& \approx & (A'[f]) \setminus F \mid B\theta & : \mathbf{Res}, \;\; B\theta \backslash F \approx B\theta \\
& \approx & \mathbf{0} \mid B\theta & : theorem \;\; 3.4.1 \\
& \approx & B\theta & : simplify
\end{array}
$$

□

$$
\begin{array}{rrcl}
\mathbf{Cut-1:} & ((\overline{succ(\theta)}.P) \rhd\!\!|_{\ell} \; Q) \overset{\circ}{;} C & \approx & Q\theta \\
\mathbf{Cut-2:} & ((\overline{done}.P) \rhd\!\!|_{\ell} \; Q) \overset{\circ}{;} C & \approx & C \\
\mathbf{Cut-3:} & ((\overline{succ(\theta)}.P) \rhd\!\!| \; Q) \overset{\circ}{;} C & \approx & Q\theta \\
\mathbf{Cut-4:} & ((\overline{done}.P) \rhd\!\!| \; Q) \overset{\circ}{;} C & \approx & Done
\end{array}
$$

Figure 3.11: Bisimilarities for cut

Theorems 3.4.2 and 3.4.3 show how OR and AND agents are suspended respectively. Together, they are used to effect a program cut. When a cut is activated, all previous goal choicepoints are removed (theorem 3.4.3), and the search is terminated throughout remaining clauses (theorem 3.4.2). The bisimilarities in figure 3.11 model the cut.

Theorem 3.4.4

$$
\begin{array}{rrcl}
\mathbf{Cut-1:} & ((\overline{succ(\theta)}.P) \rhd\!\!|_{\ell} \; Q) \overset{\circ}{;} C & \approx & Q\theta \\
\mathbf{Cut-2:} & ((\overline{done}.P) \rhd\!\!|_{\ell} \; Q) \overset{\circ}{;} C & \approx & C \\
\mathbf{Cut-3:} & ((\overline{succ(\theta)}.P) \rhd\!\!| \; Q) \overset{\circ}{;} C & \approx & Q\theta \\
\mathbf{Cut-4:} & ((\overline{done}.P) \rhd\!\!| \; Q) \overset{\circ}{;} C & \approx & Done
\end{array}
$$

Proof: *Apply theorems 3.4.2 and 3.4.3.* □

3.4.2 Negation as failure, pruning operators, and if–then–else

A variety of other control devices can be modelled using concepts from the semantics of the cut. Negation as failure is implemented within Prolog as:

$$\begin{array}{l} not\ \ P :-\ call(P),\ !,\ fail. \\ not\ \ P. \end{array}$$

A semantic Not operator is based on the operational effects of this Prolog definition:

$$Not\ \ P \stackrel{\text{def}}{=} (\ P[f] \mid succ'.Done + done'.\overline{succ}.Done\ ) \setminus F$$

This definition explicitly causes P to deadlock once an answer has been obtained from it. Note that safe negation as failure would require P to be ground before invocation.

Two pruning operators – soft cuts and one–solution operators (Hill *et al.* 1990) – can also be modelled in CCS. The standard cut as described previously has been referred to as a *hard cut*, which is distinguished from a *soft cut* or *snip*. Given the predicate fragment,

$$\begin{array}{l} P_1 :-\ a, b, @, c, d. \\ P_2 :-\ e. \end{array}$$

the soft cut @, when activated, discards the subtrees for P_2 and any other clauses which may follow. This is analogous to the *commit* operator used in concurrent logic programming. The soft cut operator is modelled in CCS as

$$\begin{array}{rcl} A\ @\ B & \stackrel{\text{def}}{=} & (A[f] \mid NextSoft_i) \setminus F \\ NextSoft_i & \stackrel{\text{def}}{=} & succ'(\theta).(B\theta \overset{\circ}{;} NextSoft_i[\ell/done]) + done'.Done \end{array}$$

This is almost identical to the definition of $\rhd$, except that, when @ is activated, the *done* signal is renamed in the manner done in $\rhd\!|_\ell$. In addition, the clause following the one with a soft cut is then sequenced with $\overset{\circ}{;}$,

$$\begin{array}{l} P_1 :-\ A, @, B. \\ P_2 :-\ E. \end{array} \quad \Longrightarrow \quad \begin{array}{l} P \stackrel{\text{def}}{=} P_1 \overset{\circ}{;} P_2 \\ P_1 \stackrel{\text{def}}{=} A\ @\ B \\ P_2 \stackrel{\text{def}}{=} E. \end{array}$$

Of course, a soft cut in the last clause of a predicate has no effect, and so standard backtracking with $\rhd$ would be used.

A one–solution operator is as follows. In the program

$$\begin{array}{l} P_1 :-\ a, b, \&, c, d. \\ P_2 :-\ e. \end{array}$$

the one–solution operator &, when activated, discards the choice points for goals a and b. When goals c and d fail, search continues unaffected at P_2. This therefore acts as a local commit for a clause's goals. The CCS semantics of & is a simplification of the hard cut semantics:

$$\mathcal{M}[\![\, a, \&, b \,]\!] = a \rhd\!\!\mid b$$

In this translation, the pruning of the search done by $\mathring{;}$ and $\rhd\!\!\mid_\ell$ is not introduced. An example translation is the following:

$$\mathcal{M}[\![\, P_i \ :- \ a, b, \&, c, d, \&, e. \,]\!] \quad = \quad P_i \stackrel{\text{def}}{=} (a \rhd b) \rhd\!\!\mid (c \rhd d) \rhd\!\!\mid e$$

where $P \stackrel{\text{def}}{=} \ \cdots P_i \,\hat{;}\, P_{i+1} \cdots$.

Finally, the if–then–else or implication control construct is modelled as follows. In Prolog, implication is implemented as,

$$\begin{array}{lcl} P :- C \rightarrow Q; R. & \Rightarrow & P :- C,\ !,\ Q. \\ & & P :- R. \end{array}$$

The equivalent CCS semantics is therefore

$$\mathcal{M}[\![\, (C \rightarrow Q; R) \,]\!] = \ (C \rhd\!\!\mid_\ell\ Q) \mathring{;} R$$

When the *else* component is empty, the expression used is

$$\mathcal{M}[\![\, (C \rightarrow Q) \,]\!] = \ (C \rhd\!\!\mid Q)$$

These expressions are inserted within AND agent bodies which use the implication control construct.

3.4.3 Example symbolic computation

The semantics of cut is best illustrated by an example computation. Consider the Prolog predicate and its CCS translation in figure 3.12. A symbolic computation of the goal "$?-\ \ p(b, c)$" follows:

$$
\begin{array}{l}
p(X,Y)\ :-\ h(X),\ !,\ g(Y).\\
p(X,Y)\ :-\ p(Y,Y).\\
\\
g(b).\\
g(c) :-\ fail.\\
\\
h(c).\\
h(c) :-\ h(X).
\end{array}
\quad \Leftrightarrow \quad
\begin{array}{rcl}
p(X,Y) & \stackrel{\mathrm{def}}{=} & p_1(X,Y) \mathbin{\overset{\circ}{;}} p_2(X,Y)\\
p_1(X,Y) & \stackrel{\mathrm{def}}{=} & h(X) \rhd\!|_\ell\ g(Y)\\
p_2(X,Y) & \stackrel{\mathrm{def}}{=} & p(Y,Y)\\
\\
g(X) & \stackrel{\mathrm{def}}{=} & g_1(X) \mathbin{\hat{;}} g_2(X)\\
g_1(X) & \stackrel{\mathrm{def}}{=} & X = b\\
g_2(X) & \stackrel{\mathrm{def}}{=} & X = c \rhd \mathit{Done}\\
\\
h(X) & \stackrel{\mathrm{def}}{=} & h_1(X) \mathbin{\hat{;}} h_2(X)\\
h_1(X) & \stackrel{\mathrm{def}}{=} & X = c\\
h_2(X) & \stackrel{\mathrm{def}}{=} & X = c \rhd h(X)
\end{array}
$$

Figure 3.12: Prolog program and CCS translation

$$
\begin{array}{ll}
p(b,c) & \\
\approx\ p_1(b,c) \mathbin{\overset{\circ}{;}} p_2(b,c) & : \mathbf{Con}\ p\\
\approx\ (h(b) \rhd\!|_\ell\ g(c)) \mathbin{\overset{\circ}{;}} p_2(b,c) & : \mathbf{Con}\ p_1\\
\approx\ (\mathit{Done} \rhd\!|_\ell\ g(c)) \mathbin{\overset{\circ}{;}} p_2(b,c) & : \mathbf{Resol}\ h\ \mathit{twice}\\
\approx\ p_2(b,c) & : \mathbf{Cut-2}\\
\approx\ p_1(c,c) \mathbin{\overset{\circ}{;}} p_2(c,c) & : \mathbf{Con}\ p_2,\ p\\
\approx\ (h(c) \rhd\!|_\ell\ g(c)) \mathbin{\overset{\circ}{;}} p_2(c,c) & : \mathbf{Con}\ p_1\\
\approx\ ((h_1(c) \mathbin{\hat{;}} h_2(c)) \rhd\!|_\ell\ g(c)) \mathbin{\overset{\circ}{;}} p_2(c,c) & : \mathbf{Con}\ h\\
\approx\ ((\overline{succ(\epsilon)}.\mathit{Done} \mathbin{\hat{;}} h_2(c)) \rhd\!|_\ell\ g(c)) \mathbin{\overset{\circ}{;}} p_2(c,c) & : \mathbf{Resol}\ h_1\\
\approx\ g(c) & : \mathbf{Cut-1}\\
\approx\ \mathit{Done} & : \mathbf{Resol},\ \mathit{expansion}
\end{array}
$$

The activated cut pruned the infinite computations at h_2 and p_2.

3.5 Dataflow

The issue of the logic variable domain and dataflow within the CCS semantics is discussed. Some notation which permits dataflow to be hand–simulated during semantic analyses of programs is first introduced. Extensions to CCS which enable a Herbrand domain and logical variables to be supported are then described.

3.5.1 A semantic notation for dataflow

Since the main motivation of the CCS semantics is to model control, the details of the data domain are often kept abstract, and are implicitly assumed to function appropriately. Because the data given to logic programs determines the course of computations, some applications such as termination analyses require a more precise treatment of dataflow. Unless it is absolutely clear how the logical variables in CCS expressions are affected during the course of analyses, errors can easily be introduced. This will probably occur for all but the most trivial Prolog programs, given the dynamic nature of dataflow within computation trees. The notation of this section is intended to help alleviate this problem. This dataflow notation is a variation of that used by the functional semantics in (Baudinet 1988). Their implementation within CCS is discussed in the following section.

The form of a computed binding substitution is determined by the logical variables used in agent arguments, the idea being that only substitutions of interest are returned as results. For example, in the clause

$$p(X,Y) \;:-\; a(X,Z),\; b(Z,Y).$$

the only bindings which would be returned as a computed result from p are the ones for X and Y. A function Π_S is used to restrict answer substitutions to the variables in a set S. For example,

$$\Pi_{\{X,Y\}}\{X \leftarrow a, Y \leftarrow b, Z \leftarrow c\} = \{X \leftarrow a, Y \leftarrow b\}$$

Another function $vset$ returns the set of variables found in term $\tilde{t}$, for example,

$$vset(s(X,t(Y))) = \{X,Y\}$$

When a clause p_i is invoked with arguments $\tilde{t}$, the call should return an answer substitution referencing the variables in $\tilde{t}$, which is $vset(\tilde{t})$. This is denoted by $\Pi_{vset(\tilde{t})}\ p_i(\tilde{t})$. Within the CCS semantics, Π is initially applied when an OR agent is invoked. Given a call to a predicate $p(\tilde{t})$, the OR agent expression subsequently invoked is $p_1(\tilde{t}) \,\hat{;}\, \cdots \,\hat{;}\, p_k(\tilde{t})$. The argument $\tilde{t}$ is passed uniformly to each clause (AND agent). So,

$$\begin{aligned} p(\tilde{t}) &\approx \Pi_{vset(\tilde{t})}\ (p_1(\tilde{t}) \,\hat{;}\, \cdots\ p_k(\tilde{t})) \\ &\approx (\Pi_{vset(\tilde{t})}\ p_1(\tilde{t})) \,\hat{;}\, \cdots \,\hat{;}\, (\Pi_{vset(\tilde{t})}\ p_k(\tilde{t})) \end{aligned}$$

Each $p_i(\tilde{t})$ call reduces to a stream of $\overline{succ(\theta)}$ actions to which Π is eventually applied:

$$\Pi_S(\overline{succ(\theta_1)}\ .\ \cdots\ .\ \overline{succ(\theta_n)}\ .\ \cdots) \;=\; \overline{succ(\Pi_S\ \theta_1)}\ .\ \cdots\ .\ \overline{succ(\Pi_S\ \theta_n)} \cdots$$

Similarly, for a program query "? $-\ g_1(\tilde{t}_1),\ \ldots\ , g_k(\tilde{t}_k)$.",

$$\Pi_{vset((\tilde{t}_1,\ldots,\tilde{t}_k))} g_1(\tilde{t}_1) \rhd \cdots \rhd g_k(\tilde{t}_k)$$

The notation

$$\theta \circ (\overline{succ(\theta_1)}\ .\ \cdots\ .\ \overline{succ(\theta_k)}) \quad = \quad \overline{succ(\theta \circ \theta_1)}\ .\ \cdots\ .\ \overline{succ(\theta \circ \theta_k)}$$

accumulates answer substitutions during computations. Here, θ denotes the set of answer substitutions obtained thus far in the computation, say, for a previous set of backtracked goals.

The above dataflow mechanisms can be introduced into the **Back** and **Cut** bisimilarities. With regards to the backtracking operators $\rhd$ and $\trianglerighteq$, dataflow affects three of the equivalences in figure 3.5. **Back-1** is embellished with the following dataflow information:

$$(\overline{succ(\theta)}.P)\ \rhd Q(\tilde{t}) \quad \approx \quad \theta \circ\ (P \trianglerighteq Q(\tilde{t}\theta))$$

The answer substitution result from the left hand side contributes to the final answer substitution for the whole backtracking expression (" $\theta\ \circ$ "), and that P's result is applied to $Q(\tilde{t})$ before invoking it (" $Q(\tilde{t}\theta)$ "). The effect of this is that θ's will be accumulated and applied as goals are solved. The application of previously computed substitutions to newly computed ones occurs when **Back-3** is applied:

$$\begin{aligned} \theta \circ\ (P \trianglerighteq \overline{succ(\gamma)}.Q) \quad &\approx \theta \circ\ \overline{succ(\gamma)}.(P \trianglerighteq Q) \\ &\approx \overline{succ(\theta \circ \gamma)}.(P \trianglerighteq Q) \end{aligned}$$

Finally, **Back-4** is used to remove the application of previously obtained results from further results:

$$\theta \circ\ (P \trianglerighteq Done) \quad \approx \quad P \rhd Q$$

The **Cut** bisimilarities are similarly treated. For example, **Cut-1**:

$$((\overline{succ(\theta)}.P) \rhd\!|_{\ell}\ Q(\tilde{t}))\ \mathring{;}\ C \quad \approx \quad \theta \circ\ Q(\tilde{t}\theta)$$

3.5.2 A Herbrand extension to CCS

CCS does not directly support logical variables. However, CCS's value passing mechanism can be extended so that logical variables over a Herbrand universe is accommodated. This extension is similar to that done to handle value and agent arguments within CCS.

The data domain used by the semantics is the Herbrand universe for the program, U_P, as well as the domain of answer substitutions over U_P. Because basic CCS represents behaviour at the level of atomic actions, representing U_P requires enhancing the basic calculus to one with value passing over the Herbrand domain. (Milner 1989) translates agent expressions using value passing into basic ones without such values by creating enumerated agent expressions over the total space of domain values. Similarly, to handle U_P, all agent expressions defined for logic program constructs will implicitly denote basic CCS expressions enumerated over U_P. Given some agent constant $p'(X,Y)$ defined for the program predicate $p(A,B)$, the constant is equivalent to

$$p'(X,Y) \quad \equiv \quad \cup_{t_i,t_j \in U_p} \{p_{t_i,t_j} \stackrel{\text{def}}{=} \cdots \}$$

$p'(X,Y)$ represents a set of agent expressions in which each agent is devoted to a tuple of arguments from U_P. For example, given the AND agent $p_1(X) \stackrel{\text{def}}{=} q(a,b,t(X))$, the corresponding Herbrand universe is $U_P = \{\ a, b, t(a), t(b), t(t(a)), ...\ \}$. Then, using a suitable enumeration ordering, the agent p_1 represents the following family of agents:

$$\begin{array}{lcl}
p'_a & \stackrel{\text{def}}{=} & q_{a,b,t(a)} \\
p'_b & \stackrel{\text{def}}{=} & q_{a,b,t(b)} \\
p'_{t(a)} & \stackrel{\text{def}}{=} & q_{a,b,t(t(a))} \\
p'_{t(b)} & \stackrel{\text{def}}{=} & q_{a,b,t(t(b))} \\
p'_{t(t(a))} & \stackrel{\text{def}}{=} & q_{a,b,t(t(t(a)))} \\
\ldots & &
\end{array}$$

supplemented by a similar enumeration of q. Using this implicit translation, all the transitional inference rules of CCS will be valid, since expressions are reducible to basic CCS expressions over atomic actions.

The value passing variables used by CCS must be enhanced in order to function as logical variables. CCS does not address the concept of *fresh variable renaming*, which is required in first–order predicate logic. To enable CCS value variables to be treated logically, a new **Con** rule must be used:

$$\textbf{Con}^* \qquad \frac{P^* \stackrel{\alpha}{\to} P'}{A \stackrel{\alpha}{\to} P'} \qquad (A \stackrel{\text{def}}{=} P)$$

where P^* is the definition of A with all the variables X_i in A uniquely renamed. This is exactly what is done in predicate logic: the variables in predicative formulae are freely renamed as necessary, as long as the renaming does not interfere with variable names used elsewhere. It is assumed that there is an infinite set of possible variable names which can be used when accessing a new agent definition, and that variable names will be freely and appropriately chosen from this set. The observational equivalence of agents holds up to a renaming of logical variables, since two expressions which are identical up to a renaming of variables represent the same family of agent expressions.

The answer substitutions found in the θ argument of $succ(\theta)$ actions are enumerated in a similar manner as agent definitions:

$$succ'(\{X \leftarrow A, Y \leftarrow B\}) \;\equiv\; \cup_{t_i, t_j \in U_P} \{\; succ_{X \leftarrow t_i, Y \leftarrow t_j} \;\}$$

This enumeration, however, uses logical variable names as part of the ordering. CCS does not permit such meta–reflection on variable names. To perform this enumeration, two new rules of inference are required:

$$\textbf{Unify}_1 \quad \frac{}{\tilde{t}_1 = \tilde{t}_2 \xrightarrow{\epsilon} \overline{succ(\theta)}.Done} \quad (\theta = mgu(\tilde{t}_1, \tilde{t}_2))$$

$$\textbf{Unify}_2 \quad \frac{}{\tilde{t}_1 = \tilde{t}_2 \xrightarrow{\epsilon} Done} \quad (\neg\exists\theta.\tilde{t}_1\theta = \tilde{t}_2\theta)$$

These two rules perform unification externally from the other CCS transition rules, and allow variable names to be included as part of the computed θ result in $\overline{succ(\theta)}$. These additional transition rules should not interfere with the integrity of the other CCS rules, since these transitions do not introduce any behavioural phenomena during their invocation (the empty ϵ transition is used).[4]

Another issue is the distribution of answer substitutions within semantic expressions. Firstly, it is assumed that the sequenced AND agent calls within OR agents use normal CCS value passing. For example, given the OR agent definition,

$$P(\tilde{x}) \stackrel{\text{def}}{=} P_1(\tilde{x}) \,\hat{;}\, P_2(\tilde{x}) \,\hat{;}\, \cdots \,\hat{;}\, P_k(\tilde{x})$$

$\tilde{x}$ is a tuple of CCS value passing variables. This allows the variable scoping in the P_i terms to be mutually exclusive. Computed results, however, are shared amongst goals in AND agents which use backtracking. Consider a clause,

$$p(\tilde{t}_0) \;:-\; g_1(\tilde{t}_1),\; \cdots \;, g_k(\tilde{t}_k).$$

[4] Note that in section 3.3.1 and the rest of the thesis the unification term = is conceptualised as a call to a unification agent.

This is implemented in CCS as,

$$p(\tilde{x}) \stackrel{\text{def}}{=} (\tilde{x} = \tilde{t}_0) \rhd g_1(\tilde{t}_1\theta_0) \rhd g_2(\tilde{t}_2\theta_0\theta_1) \rhd \cdots \rhd g_k(\tilde{t}_k\theta_0\theta_1 \cdots \theta_{k-1})$$

where θ_0 is returned from "=", and θ_i is from g_i $(1 \leq i \leq k)$. This represents the cumulative application of answer substitutions onto successive goals. Here, computed answer substitutions are automatically applied to goals in a backtracked expression. When some $\overline{succ(\theta)}$ is computed, this θ is instantly applied to the environment. As a consequence, goals which are subsequently invoked will have θ automatically applied to them.

Only the variables which are resident in a call to an agent are returned in the computed result for the agent, and computed substitutions for variables found elsewhere in the clause are pruned from this result. Given some clause agent,

$$P_i(\tilde{x}) \stackrel{\text{def}}{=} \tilde{x} = \tilde{t} \rhd G_1(\tilde{t}_1) \rhd \cdots \rhd G_k(\tilde{t}_k)$$

the mechanism which prunes answer substitutions computed by P_i is:

$$P_i(\tilde{x}) \stackrel{\text{def}}{=} (\tilde{x} = \tilde{t} \rhd G_1(\tilde{t}_1) \rhd \cdots \rhd G_k(\tilde{t}_k))[f] \mid VarLoop(\theta_0 \circ \cdots \circ \theta_{k-1}, vars(\tilde{x})) \setminus F$$

where

$$VarLoop(\theta, D) \stackrel{\text{def}}{=} succ'(\gamma).\overline{succ(\Pi_D(\theta \circ \gamma))}.VarLoop(\theta, D) \ + \ done'.Done$$

The $vars$ and Π operators have the same usage as in section 3.5.1, and are meta–operators which inspect the syntax of CCS expressions to determine the form of variables. Therefore, as with the definition of = above, $vars$ and Π are implemented as CCS transition rules. The general idea is that each computed answer substitution is intercepted and pruned to contain only the variables of interest, which are those in D – the variables in the invoking call argument.

Using the above methods for (i) extending the Herbrand universe, (ii) automatically rename logical variables whenever necessary, (iii) automatically distributing and applying answer substitutions within backtracked expressions, and (iv) pruning computed results, the logical domain of Prolog programs is modelled within the CCS semantics. This modelling of Prolog's data domain is done implicitly; it is assumed throughout the thesis that the above mechanisms exist behind the scenes.

3.6 Conclusion

3.6.1 Discussion

This chapter presented a new CCS semantics for sequential Prolog. Two main semantic operators, $\fatsemi$ and $\triangleright$, model Prolog's standard control strategy. Control is further described by high–level bisimilarities, which take the form of rewriting rules. The semantics of the cut, snip, and negation by failure were also described. Some example symbolic computations were performed. Some contributions of the semantics are:

- The operational semantics of Prolog is compositional. Prolog programs are mappable to semantic expressions.

- The stream of answer substitution domain is not overly abstracted from the declarative domain of logic programs. Symbolic execution of the control operators result in the generation of stream output. This differs with the handling of streams within denotational semantics, in which streams may be a component of a complex abstract computational domain.

- Dataflow is conveniently abstracted from the control component of the semantics, which permits a concise semantics of control.

- The cut is elegantly modelled. The semantics of the cut uses some additional operators which intercompose directly with the existing semantics of standard control, while preserving the same stream domain.

This semantics of Prolog can be classified as a structured operational semantics (Plotkin 1981) (Hennessy 1990). This is so because CCS itself is defined by structured algebraic transition rules (figure 2.2), and these transitions are applicable to CCS translations of Prolog programs. The semantics retains a strong conceptual relationship with CCS. This relationship with structured algebraic semantics notwithstanding, a contribution of this semantics with respect to other semantics of Prolog is its conceptual foundation as a sequential AND/OR process model of Prolog.

One particular process model of Prolog control was suggested, which admittedly might not be the simplest or most lucid design possible. CCS was chosen because it is a well–accepted, solidly founded formalism. Other process algebras such as CSP (Hoare 1985*a*) might also prove suitable. Process algebras like CCS are well–suited towards modelling sequential Prolog. The AND/OR process tree paradigm is an ideal way of describing Prolog control, and encoding this characterisation in CCS results in an effective formalisation. This contrasts to the use of semantic formalisms commonly used for modelling imperative languages, whose characterisation of computations as transitions over states of memory values is irrelevant for logic program computations. Another advantage of the process semantic approach is that the temporal nature of streams (traces) provides a framework for modelling the sequential logic program resolution. In addition, the compositional nature of process semantics corresponds to the compositional structure of logic programs, and the algebraic foundation of process algebras is closely related to first–order predicate logic.

A criticism of this semantics might be that a system like CCS, which is designed to model concurrency, is too unwieldy for modelling sequential computations. Much of the theoretical results of CCS, especially the various classifications of computational equivalence, are not useful in a sequential context. Therefore, another possible approach would have been to define a stream semantics of Prolog directly on the language using structured transition rules as in (Hennessy 1990). The conceptual basis afforded by the use of CCS, however, is instrumental in the design of the semantics. CCS already contains the necessary formal concepts for modelling sequential Prolog, and a direct result of this is the concise account of control presented here. Reformulating the semantics of processes would have added to the design effort.

The semantics of Prolog defined at the level of the $\rhd$ and $\hat{;}$ operators is too low–level to be practical. At this basic level of control, the semantics is encumbered with communication details which detract from any intuitive modelling of control over the computation tree. The introduction of bisimilarities does capture the essence of Prolog control at a higher level. The use of bisimilarities should not detract from the very real semantic information afforded by the operator definitions: the definitions of $\rhd$ and $\hat{;}$ permitted the derivation of the bisimilarities, and will also be used for deriving various other semantic properties.

Some example semantic translations were verified using the Concurrency Workbench (CWB) system, a semi–automated CCS verification system (Cleaveland *et al.* 1989). The CWB allows different analyses such as tracing and bisimulations to be performed on CCS expressions. The current incarnation of the CWB is written to handle basic CCS expressions without parameter passing, and cannot handle infinite state spaces. These shortcomings naturally limit its applicability towards testing the CCS expressions used here. However, some AND/OR process trees for basic Prolog programs were constructed and loaded into the CWB, and the resulting traces obtained conformed to Prolog's expected control.

With respect to CCS, the extensions outlined in section 3.5 to handle logical variables warrant more study. Rather than extend the existing basic CCS formalism, a new incarnation of CCS that directly supports logical variables should be developed. The theoretical ramifications of logical variables in a process algebra needs attention.

3.6.2 Comparison to related research

Much attention has been devoted to the semantics of pure logic programs executed under a generalised search rule and computation rule (van Emden and Kowalski 1976) (Clark 1979) (Apt and van Emden 1982) (Lloyd 1984) (Hogger 1990). These approaches are not especially concerned with the particular idiosyncracies of Prolog control. In applications require consideration of the control strategy, the semantics of control is only informally handled.

The semantics in (Baudinet 1988) is similar in style and purpose to this one (see section 4.5 for a technical comparison). She proves termination properties of Prolog programs using a functional semantics of the language. Her approach is similar to this one in that her semantics maps directly to program components, the semantic domain is streams of answer substitutions, and her semantic functions $\bowtie$ and $\sqcup$ which describe the results of program backtracking and sequencing are similar in functionality to the $\rhd$ and $\fatsemi$ operators respectively. The CCS approach differs fundamentally from Baudinet's in that it describes the operational semantics of Prolog directly, whereas Baudinet defines the final results of executing a logic program assuming Prolog's search and computation rules. The CCS semantics is essentially a rational reconstruction of Baudinet's semantics. Given the expression $A \bowtie B$, the $\bowtie$ operator is defined to apply the stream of answer substitutions from A to the stream generated by B, which indirectly reflects the behaviour of Prolog's backtracking. Nontermination and looping is explicitly handled by cases dependent upon the form of the streams given to $\bowtie$. The CCS semantics deduces program behaviour directly from the semantics of Prolog control as defined by the $\triangleright$ and $\hat{;}$ operators. Given the expression $A \triangleright B$, the final result of this expression is derived using the operational semantics of $\triangleright$ itself. Repeated expansion of this expression yields the same result as $\bowtie$, and the expansion itself can be performed inductively, rather than explicitly. The advantage of this approach is that a wide variety of control strategies are definable in CCS. The semantics properties of new control schemes are then deducible from their low–level CCS definitions. In addition, looping phenomena are deduced from the CCS semantics, but are defined axiomatically in Baudinet's approach.

The semantics of cut are quite different. Baudinet treats the cut by using special cut flags within computation traces. These markers are used by the operators for modifying traces to simulate the effect of cuts. CCS models the cut using the $\triangleright\!\!|$ and $\overset{\circ}{;}$ operators. The effect of the cut on computation traces falls directly from the CCS definitions, without corrupting the semantics of streams.

Stylistically, Baudinet's semantics is designed towards concisely describing the nature of dataflow and the application of answer substitutions through streams. A cosmetic difference is that she uses function notation such as lambda expressions to express the distribution of dataflow, whereas we directly distribute answer substitutions throughout CCS expressions. However, both hers and our treatment of dataflow are essentially the same. Realistically, when using either of these semantics in an analytical application, one would probably make the dataflow implicit in order to lessen notational and semantic cumbersomeness.

Denotational semantics such as (Debray and Mishra 1988) account for Prolog's standard control strategy with cut (the Baudinet semantics can be considered to be a lean denotational semantics). The descriptive power of denotational semantics permits a semantics the entire Prolog language within one formalism. Features like the cut and database operations like "assert" are modelled using embellished domain spaces and continuations. But just because a semantic system can describe all the features of a language, it does not mean that it does so as lucidly or intuitively as possible. As with meta–program semantics below, programs lose their declarative interpretation within a denotational setting. Denotational semantics are more suitable for describing programming languages, as their complex abstract domain spaces is difficult to apply towards proving program properties (Ashcroft and Wadge 1982).

Another related semantics of control is in (Billaud 1985) (Billaud 1988) (Billaud 1990). Billaud defines an abstract interpreter for standard Prolog control with cut. Unlike meta–interpreters, Billaud's interpreter is defined in terms of an abstract Prolog machine encoded by algebraic rewrite equations. A strength of his approach is that Prolog code is directly mappable to rewrite equations, which permits program properties to be proven. In addition, his semantic equations have a functional fixpoint characterisation, which allows declarative properties of the semantics to be readily produced. His semantics differs from this one mostly in the underlying conceptual bases of the two formalisms – processes versus algebraic encoding of an abstract machine. For example, his equations explicitly encode stack structures of the underlying abstract machine. The handling of cuts requires manipulation of an algebraic expression which denotes an abstract data structure encoding the stack configuration required to prune search. The CCS semantics is a degree more abstract than this, and as a consequence, stacks are not explicitly denoted. Despite their different underlying formalisms, Billaud's semantics and this one are closely aligned with respect to their intention to model the control component of Prolog.

The meta–programming approach defines a logical meta–program semantics of Prolog. (Hill and Lloyd 1988) suggest an executable Horn clause semantics for various Prolog implementations, essentially deriving pure Prolog interpreters for Prolog. Depending on the complexity of the implementation being modelled, for example, whether extra–logical features like "var" or annotated variables are included, the declarative semantics is appropriately extended. Meta–program semantics offer concise axiomatisations of the operational semantics of languages, with the added advantage that such axiomatisations are implementable. Meta–interpreters are better applied towards describing general properties of a logic programming language, rather than towards proving properties of individual programs. The operational semantics of the meta–language must be considered in addition to the operational semantics of the source language being interpreted if an exact operational model of program behaviour is to be obtained. (The CCS semantics models a meta–interpreter in section 4.6.)

Proof theoretic semantics of Prolog control have been suggested (Andrews 1991). He shows the correspondence between an operational semantics for an abstract Prolog machine and a natural deduction theory. Prolog's control strategy is described by the proof rule axioms. The advantage this has over CCS is its clearer conceptual link with declarative logic programming, which would enable declarative properties of programs to be more naturally modelled. However, the semantic link with the declarative logical basis of logic programs is not really as strong as claimed. The operational semantics of logic program connectives differs from the semantics of the connectives used by the proof theory. For example, the logical AND used to delimit program goals differs from the logical AND used by the proof theory. With respect to proving properties of control, the CCS semantics of control might prove to be a more succinct formalism for studying operational behaviours of programs. Having a stream–based domain is advantageous for describing phenomena such as cuts, nontermination, and looping. It is also widely adaptable to other sequential and concurrent control strategies (chapter 8).

(Deransart 1988) uses attribute grammars to describe operational semantics. By decorating a representation of the computation computation tree, different computation rules can be modelled and analysed. Although it is primarily intended as a formal means for studying specific control strategies, it can also be applied towards analysing given program properties.

An application which uses CCS for describing an aspect of sequential Prolog control is in (Fung 1988). CCS–like expressions are used to describe novice programmer conceptualisations of logic program control. For example the expression

$$\begin{array}{llll}(SP1 \ (SP1RHS- & (+ & (SP1RHS & SP1-) \\ & & (FP1RHS & FP1-))))\end{array}$$

says that, to execute clause SP1, the body (SP1RHS) is contacted; if it is successful, SP1 succeeds, and if not (FP1RHS), it fails. In the above, "+" and "−" denote actions and co–actions, while the terms SP1 and others are actions indicating the success and failure of clause components (heads and bodies). Fung then applies the expansion theorem to the CCS representation for a program to obtain all the possible "directions" of control. Novice programmer errors in Prolog programming can then be compared with this expanded representation of control possibilities, which is done to check for control misconceptions commonly encountered by novices. Although her application is not intended as a formal semantics of the Prolog language, it does recognise the utility of the process algebra paradigm for describing aspects of Prolog control.

Chapter 4

Properties of the semantics

Some properties of the semantics defined in chapter 3 are derived. These properties describe different algebraic and computational behaviours which are useful within applications. In addition, this chapter establishes the correctness and semantic completeness of the semantics. Partial correctness means that any answer substitution computed by the semantics is consistent with the declarative semantics of the program – computed results are sound. The completeness of the semantics for a programming language does not necessarily refer to the same notion of completeness as is used when saying that an inference system is complete. Completeness means that the semantics adequately and faithfully models the programming language. Completeness is typically proven by showing the correspondence with the semantic formalism in question with some other semantics or abstract machine for the language which is deemed to be correct.

Section 4.1 derives some termination properties. Algebraic properties of the control operators, such associativity, distributivity, and non–commutativity, are derived in section 4.2. Section 4.3 shows that all the agents defined by the semantics are well–terminating, which is required for termination analyses. Section 4.4 illustrates how the semantics models Prolog's computation and search rules, by showing the correspondence between the semantics and both SLD–resolution and the immediate consequence operator T_P. The semantics is next compared to a functional semantics by (Baudinet 1988) in section 4.5. Lastly, the correspondence between it and the CCS semantics of a Prological meta–interpreter for Prolog is shown in section 4.6.

4.1 Termination properties

The termination of a computation is not an observable phenomena, since termination can be thought of as a permanent lack of observable activity(Hehner *et al.* 1986) . What is required is a means for establishing when an agent terminates – a *termination convention*. In section 4.3, both OR and AND agents are shown to be *well–terminating*, which means that the action *done* is always generated by an agent before and only before it terminates:

$$P \stackrel{s}{\Rightarrow} \stackrel{\overline{done}}{\rightarrow} \mathbf{0}$$

where s is a stream of actions $|s| \geq 0$, and $\mathbf{0}$ is the null or inactive agent. When an agent generates $\overline{done}$, it is understood that the agent has terminated. The *done* action is a termination convention. If $\overline{done}$ is not seen, then the agent is still active, and may still generate a finite or infinite number of actions, or perhaps none if it is looping. Having well–terminating agents means that the semantics is well–behaved with respect to adhering to a protocol of termination. Such predictability of behaviour is required when formally analysing termination.

There are three basic behaviours of AND and OR agents.

1. *Finite computations.* A finite computation is represented as a finite sequence of zero or more answer substitutions:

   $$\overline{succ(\theta_1)} \;.\; \cdots \;.\; \overline{succ(\theta_k)} \;.\; Done \quad (\; k \geq 0 \;)$$

   This is denoted as $\overline{succ(\theta_i)}\,_{i=1}^{k}.Done$ or just $\overline{succ(\theta_i)}^{\,k}.Done$ when $k > 0$, or $Done$ when $k = 0$.

2. *Infinite productive computations*: This occurs when a non–terminating agent generates an infinite stream of answer substitutions:

   $$\overline{succ(\theta_1)} \;.\; \cdots \;.\; \overline{succ(\theta_k)} \;.\; \cdots$$

   It is denoted $\overline{succ(\theta_i)}\,_{i=1}^{\omega}$ or just $\overline{succ(\theta_i)}^{\,\omega}$.

3. *Looping computations*: Looping computations are distinguished from infinite productive computations in that they produce no output whatsoever. We denote looping by "$\perp$", and define it in CCS as:

   $$\perp \stackrel{\text{def}}{=} \perp$$

This $\perp$ agent is defined by a recursive call to itself, which is the simplest CCS expression that exhibits looping behaviour. A looping agent is in a state where it produces no actions whatsoever: $\neg\exists\alpha : S \xrightarrow{\alpha} S'$. Looping is also known as *livelock*, and its difference with a terminated agent $\mathbf{0}$ is subtle. A terminated agent also has no observable behaviour. By the definition of well–termination, any terminated agent must generate $\overline{done}$ before terminating. A looping agent has no such termination signal, nor any actions for that matter. Therefore, for any looping agent P, we have $P \approx \perp$.

The next two theorems show the behaviour of clause sequencing and goal backtracking with respect to different combinations of finite and infinite answer substitution streams (looping is looked at later). Theorem 4.1.1 shows how streams are sequentially composed.

Theorem 4.1.1 *Let α and β represent answer substitutions, and let A and B generate the following combinations of sequences:*

$$\begin{array}{llll} (i) & A \approx \alpha^j.Done & \text{and} \quad B \approx \beta^k.Done & (j \geq 0,\ k \geq 0) \\ (ii) & A \approx \alpha^{\,\omega} & \text{and} \quad B \approx (anything) & \\ (iii) & A \approx \alpha^j.Done & \text{and} \quad B \approx \beta^{\,\omega} & (j \geq 0) \end{array}$$

Then $A \,\hat{;}\, B$ generates the following for the above cases:

$$A \,\hat{;}\, B \;\approx\; \begin{cases} (i) & \alpha^j.\beta^k.Done \\ (ii) & \alpha^{\,\omega} \\ (iii) & \alpha^j.\beta^{\,\omega} \end{cases}$$

Proof: *If CCS's expansion theorem is applied to each of these cases, the behaviour observed is bisimilar or observationally equivalent to the generated behaviours above.*

For example, in case (i), apply induction on the size of the stream generated by A. When $j = 0$, then $Done \,\hat{;}\, B \approx B$ by **Seq**. *For $j = n + 1$,*

$$\begin{array}{lll} & \alpha^{n+1}.Done \,\hat{;}\, B & \\ \approx & \alpha.\alpha^n.Done \,\hat{;}\, B & : notation \\ \approx & \alpha.(\alpha^n.Done \,\hat{;}\, B) & : expansion \\ \approx & \alpha.(\alpha^n.\beta^k.Done) & : inductive\ \ hypothesis \\ \approx & \alpha^{n+1}.\beta^k.Done & : notation \end{array}$$

For case (ii), the equivalence $\alpha^\omega \,\hat{;}\, B \approx \alpha^\omega$ holds by induction on the stream from A (details omitted). Likewise, in (iii), by induction on j, the expansion theorem *and* **Seq** *are applied to* $\alpha^j \,\hat{;}\, \beta^\omega$ *to generate* $\alpha^j.\beta^\omega$. $\square$

In the following, predicates a and c shows case (i) behaviour:

$a(X) :- c(X).$
$a(4).$
$c(1).$
$c(2).$
$c(3).$

The stream generated for "$?-\ a(Z).$" takes the form

$$\overline{succ(\{X \leftarrow 1\})}\ .\ \overline{succ(\{X \leftarrow 2\})}\ .\ \overline{succ(\{X \leftarrow 3\})}\ .\ \overline{succ(\{X \leftarrow 4\})}\ .\ Done$$

In

$a(4).$
$a(X) :- a(X).$
$a(1).$

case (iii) applies between the first two clauses, and case (ii) applies between the last two clauses, as a has the form $a_1(X) \,\hat{;}\, a_2(X) \,\hat{;}\, a_3(X)$, and $a_2(X) \approx \overline{succ(\{X \leftarrow 4\})}^{\,\omega}$.

The next theorem states that, if the stream generated by the left–hand–side of backtracked agents is finite, then the resulting stream may be finite or infinite. Otherwise, if the left–hand stream is infinite, nonterminating or looping behaviour will occur.

Theorem 4.1.2 *Let α and β represent answer substitution results, and let θ be the answer substitution environment which includes the result last computed by A.*
(i) Let $A \approx \alpha^n.Done$ $(n > 0)$. Then

$$A \rhd B \approx \begin{cases} \beta^k.Done & : if\ \ B\theta \approx \beta^{j_i}.Done\ \ (i = 1, ..., n)\ \ for\ \ every\ \ \theta\ \ from\ \ A, \\ & \quad j_1 + j_2 + ... + j_n = k \\ \beta^{\,\omega} & : if\ \ B\theta \approx \beta^\omega\ \ for\ \ any\ \ \theta \end{cases}$$

(ii) Let $A \approx \alpha^\omega$ and no derivative of B loops. Then $A \rhd B \approx \beta^{\,\omega}$ or $A \rhd B \approx \beta^i.\bot$ $(i \geq 0)$.

Proof: *For case (i), each α_i in $\alpha^n.Done$ $(1 \leq i \leq n)$ results in a new θ environment in which B is executed. The first condition is for $B\theta$ to produce a finite stream $\beta^{\,j_i}.Done$ $(j_i \geq 0)$ for each of these α_i.*

$$
\begin{array}{ll}
A \rhd B & \\
\approx\ \alpha^n.Done \rhd B & : subst.\ \ A \\
\approx\ \alpha_1.\alpha^{\,n-1}.Done \rhd B & : notation \\
\approx\ \alpha^{\,n-1}.Done \rhd B\theta & : \mathbf{Back-1} \\
\approx\ \alpha^{\,n-1}.Done \rhd \beta^{\,j_1}.Done & : subst.\ \ B\theta \\
\approx\ \beta^{\,j_1}.(\alpha^{\,n-1}.Done \rhd Done) & : expansion \\
\approx\ \beta^{\,j_1}.(\alpha^{\,n-1}.Done \rhd B) & : \mathbf{Back-4}
\end{array}
$$

Repeating this for all α_i, we get $\beta^{\,j_1}.\beta^{\,j_2}.\ \cdots\ .\beta^{\,j_n}.Done$. This is a finite stream of size $j_1 + j_2 + ... + j_n = k$.

If $B\theta$ generates an infinite stream for some θ from A (say, for α_1), then

$$
\begin{array}{ll}
A \rhd B & \\
\approx\ \alpha_1.A' \rhd B & : subst.\ \ A \\
\approx\ A' \rhd B\theta & : \mathbf{Back-1} \\
\approx\ A' \rhd \beta^{\,\omega} & : subst.\ \ B \\
\approx\ \beta^{\,\omega} &
\end{array}
$$

The last step is a result of the obvious bisimulation between $A' \rhd \gamma.\beta^{\,\omega} \approx \gamma.(A' \rhd \beta^{\,\omega})$ and $\gamma.\beta^{\,\omega}$.

Case (ii) states that a terminating stream is impossible when A is nonterminating. This can be seen by considering that there is no derivative for $\alpha^{\,\omega} \rhd B$ which generates the action $\overline{done}$, since by **Back-2**, *$P \rhd Q \approx Done$ only when $P \approx Done$, and the condition here states that no derivative A' of A is bisimilar to Done.* □

For theorem 4.1.2 to be useful, the precise effect that the left–hand goal has on the right–hand one needs to be ascertained. The stream generated by $A \rhd B$ is solely dependent upon what effect A's answer substitutions have on B's computation, and whether A itself terminates or not. This reflects how backtracking introduces a high measure of computation tree variability into computations. Program examples for this theorem follow. In

$$
\begin{array}{l}
p :-\ a(X),\ b(X). \\
q :-\ a(X),\ c(X). \\
a(1). \\
a(2). \\
b(X). \\
c(X). \\
c(X) :-\ c(X).
\end{array}
$$

the body of p conforms to the first part of case (i), since each stream generated by b is finite. q's goals exhibit the other part of case (i), as the call to c returns the infinite stream $\overline{succ(\epsilon)}^{\omega}$. Lastly,

$r :- f(X), h(X).$
$f(1).$
$f(X) :- f(X).$
$h(X).$

is a case (ii) example, as the stream for r will be infinite no matter what the form is of h.

The next three theorems describe looping. Recall that $\rightarrow$ denotes a single action transition, and $\Rightarrow$ denotes a multiple action transition. Theorem 4.1.3 shows how looping behaviour can be expanded out of agent expressions, while theorem 4.1.4 shows how looping inhibits agent sequencing.

Theorem 4.1.3 *If P generates a looping derivative, $P \stackrel{s}{\Rightarrow} \perp$, then $P \approx s.\perp$.*

Proof: *Apply expansion theorem.* □

Theorem 4.1.4 $\perp \,⨟\, P \;\approx\; \perp$

Proof: *Bisimilarity of two expressions is proven by showing that, for each α–derivative of one expression, the α–derivative of the other expression is bisimilar. Letting ϵ represent a null action, the left–hand side has only one derivative, $\perp ⨟ P \stackrel{\epsilon}{\rightarrow} \perp ⨟ P$. Similarly, for the right hand side, $\perp \stackrel{\epsilon}{\rightarrow} \perp$. Both these expressions yield states which are bisimilar.* □

Theorem 4.1.5 shows how looping inhibits backtracking. Recall that the $\triangleright$ operator represents the state of backtracking when the right–hand–side is processing.

Theorem 4.1.5

$$\begin{array}{lccc} (i) & \perp \triangleright B & \approx & \perp \\ (ii) & A \triangleright \perp & \approx & \perp \end{array}$$

Proof: *Similar to theorem 4.1.4.* □

An example of looping behaviour is the following:

$a(X) :- a(X).$
$a(1).$
$b(X) :- a(X),\ c(X).$
$c(2).$
$d(X) :- c(X),\ a(X).$

Theorem 4.1.4 is applicable to the clauses for a, while cases (i) and (ii) of theorem 4.1.5 apply to the goals of b and d respectively.

The looping behaviour described in theorems 4.1.3, 4.1.4, and 4.1.5 can be incorporated with the stream composition theorems 4.1.1 and 4.1.2. Doing so means that all Prolog program behaviour is represented. In $A \fatsemi B$ and $A \rhd B$, agents A and B can generate finite streams, infinite streams, or loop, and these theorems characterise the form of the resulting behaviour of these expressions.

4.2 Compositional properties

The compositional properties of $\fatsemi$ and $\rhd$ derived in this section show the effect of intercomposing $\fatsemi$ and $\rhd$ with themselves and each other. Some relationships between Prolog's control mechanism and the stream of answer substitutions derived during the computation is also shown, including the asymmetry inherent within Prolog computations.

The $\fatsemi$ operator is first shown to be associative.

Theorem 4.2.1 *Associativity of sequential composition*

$$(P \fatsemi Q) \fatsemi R \;\approx\; P \fatsemi (Q \fatsemi R)$$

Proof: *Let d and e be unique action labels, and let $[d] \equiv [d/done]$ and $[e] \equiv [e/done]$. The equivalence is proved using the following static laws. For consistency, the numbering found in (Milner 1989, pages 80 - 81) is retained.*

$$\begin{array}{rlll}
8(2): & P|(Q|R) \;=\; (P|Q)|R & & \\
9(1): & P\backslash L \;=\; P & if & \mathcal{L}(P) \cap (L \cup \overline{L}) = \emptyset \\
9(4): & (P|Q)\backslash L \;=\; P\backslash L \mid Q\backslash L & if & \mathcal{L}(P) \cap \overline{\mathcal{L}(Q)} \cap (L \cup \overline{L}) = \emptyset \\
11(1): & P[b/a] \;=\; P & if & a, \overline{a} \notin \mathcal{L}(P) \\
11(3): & P\backslash a[b/c] \;=\; P[b/c]\backslash a & if & b, c \neq a
\end{array}$$

The derivation of the equivalence follows:

$$
\begin{array}{lll}
 & (P \fatsemi Q) \fatsemi R & \\
\approx & (((P[d] \mid d.Q) \setminus d)[e] \mid e.R) \setminus e & : \textbf{Con} \fatsemi \\
\approx & (((P[d] \mid d.Q)[e] \setminus d) \mid e.R) \setminus e & : 11(3) \\
\approx & (((P[d] \mid d.Q)[e] \setminus d) \mid e.R \setminus d) \setminus e & : 9(1) \\
\approx & (((P[d] \mid d.Q)[e]) \mid e.R) \setminus e \setminus d & : 9(4) \\
\approx & (((P[d][e] \mid d.Q[e])) \mid e.R) \setminus e \setminus d & : 11(1) \\
\approx & ((P[d][e] \mid d.Q[e]) \mid e.R)) \setminus e \setminus d & : 8(2) \\
\approx & ((P[d] \mid d.Q[e]) \mid e.R)) \setminus e \setminus d & : 11(1) \\
\approx & ((P[d] \mid d.Q[e] \mid e.R) \setminus e) \setminus d & : 9(1), 9(4) \\
\approx & ((P[d] \mid d.Q \fatsemi R) \setminus d & : defn. \fatsemi \\
\approx & P \fatsemi (Q \fatsemi R) & : defn. \fatsemi
\end{array}
$$
□

Some distributivity results are now shown.

Theorem 4.2.2 *Right–distributivity:*

$$(A \fatsemi B) \triangleright D \quad \approx \quad (A \triangleright D) \fatsemi (B \triangleright D)$$

Proof: *First consider terminating agents. The proof proceeds by induction on the length of the sequence s generated in $A \fatsemi B \overset{s}{\Rightarrow} Done$.*

Base case $|s| = 0$: Here $s = \epsilon$ and $A \fatsemi B \approx Done$. So $A \approx Done$ and $B \approx Done$. Then

$$
\begin{array}{lrlll}
LHS: & (A \fatsemi B) \triangleright D \approx & (Done \fatsemi Done) \triangleright D & : subst.\ A, B \\
 & \approx & Done \triangleright D & : \textbf{Seq} \\
 & \approx & Done & : \textbf{Back}-\textbf{3} \\
 & & & \\
RHS: & (A \triangleright D) \fatsemi (B \triangleright D) \approx & (Done \triangleright D) \fatsemi (Done \triangleright D) & : subst.\ A, B \\
 & \approx & Done \fatsemi Done & : \textbf{Back}-\textbf{3}\ twice \\
 & \approx & Done & : \textbf{Seq}
\end{array}
$$

Inductive case $|s| = k+1$: There are two cases:

(i) Let $A \approx Done$. Then

$$
\begin{array}{lrlll}
LHS: & (A \fatsemi B) \triangleright D & \approx & (Done \fatsemi B) \triangleright D & : subst.\ A \\
 & & \approx & B \triangleright D & : \textbf{Seq} \\
 & & & & \\
RHS: & (A \triangleright D) \fatsemi (B \triangleright D) & \approx & (Done \triangleright D) \fatsemi (B \triangleright D) & : subst.\ A \\
 & & \approx & Done \fatsemi (B \triangleright D) & : \textbf{Back}-\textbf{3} \\
 & & \approx & B \triangleright D & : \textbf{Seq}
\end{array}
$$

(ii) Let $A \fatsemi B \overset{\overline{succ(\theta)}}{\rightarrow} \overset{t}{\Rightarrow} Done$, where $s = \overline{succ(\theta)}.t$ and $|t| = k$. Expanding the expression:

$$\begin{array}{lrcll}
LHS: & (A \mathbin{;} B) \rhd D & \approx & (\overline{succ(\theta)}.A' \mathbin{;} B) \rhd D & : subst.\ A \\
 & & \approx & D\theta \mathbin{;} ((A' \mathbin{;} B) \rhd D) & : \mathbf{Back-5}
\end{array}$$

$$\begin{array}{lrcll}
RHS: & (A \rhd D) \mathbin{;} (B \rhd D) & \approx & (\overline{succ(\theta)}.A' \rhd D) \mathbin{;} (B \rhd D) & : subst.\ A \\
 & & \approx & D\theta \mathbin{;} (A' \rhd D) \mathbin{;} (B \rhd D) & : \mathbf{Back-5}
\end{array}$$

By applying the induction hypothesis on the smaller stream generated by $A' \mathbin{;} B$, these expressions are bisimilar.

If A loops ($A \approx \bot$), then the following results:

$$\begin{array}{lrcll}
LHS: & (A \mathbin{;} B) \rhd D & \approx & (\bot \mathbin{;} B) \rhd D & : subst.\ A \\
 & & \approx & \bot \rhd D & : Theorem\ 4.1.4 \\
 & & \approx & \bot & : Theorem\ 4.1.5
\end{array}$$

$$\begin{array}{lrcll}
RHS: & (A \rhd D) \mathbin{;} (B \rhd D) & \approx & (\bot \rhd D) \mathbin{;} (B \rhd D) & : subst.\ A \\
 & & \approx & \bot \mathbin{;} (B \rhd D) & : Theorem\ 4.1.5 \\
 & & \approx & \bot & : Theorem\ 4.1.4
\end{array}$$

The bisimilarity is valid. □

Theorem 4.2.3 $\rhd$ *is right–distributive over* $\mathbin{;}$ *:*

$$(A_1 \mathbin{;} A_2 \mathbin{;} \cdots \mathbin{;} A_k) \rhd B \ = \ (A_1 \rhd B) \mathbin{;} (A_2 \rhd B) \mathbin{;} \cdots \mathbin{;} (A_k \rhd B)$$

Proof: *Induction on the length of sequenced agent list.*

<u>*Base case*</u>: $(A_1 \mathbin{;} A_2) \rhd B \ \approx \ (A_1 \rhd B) \mathbin{;} (A_2 \rhd B)$ *holds by theorem 4.2.2.*

<u>*Inductive case k+1 sequenced agents:*</u>

$$\begin{array}{ll}
(A_1 \mathbin{;} A_2 \mathbin{;} \cdots \mathbin{;} A_{k+1}) \rhd B & \\
\approx\ (A_1 \mathbin{;} (A_2 \mathbin{;} \cdots \mathbin{;} A_{k+1})) \rhd B & : assoc.\ (Theorem\ 4.2.1) \\
\approx\ (A_1 \rhd B) \mathbin{;} ((A_2 \mathbin{;} \cdots \mathbin{;} A_{k+1}) \rhd B) & : inductive\ \ hypothesis \\
\approx\ (A_1 \rhd B) \mathbin{;} (A_2 \rhd B) \mathbin{;} \cdots \mathbin{;} (A_{k+1} \rhd B) & : inductive\ \ hypothesis
\end{array}$$

□

Theorem 4.2.4 $\rhd$ *is not left–distributive through* $\mathbin{;}$ *:*

$$D \rhd (A \mathbin{;} B) \ \not\approx \ (D \rhd A) \mathbin{;} (D \rhd B)$$

Proof: *Let A and B both terminate. So $A \approx s.Done$ and $B \approx t.Done$. Let $D \approx \overline{succ(\theta)}.D'$. Then*

$$
\begin{array}{rlll}
LHS: & D \rhd (A \,\S\, B) & & \\
& \approx \overline{succ(\theta)}.D' \rhd (A \,\S\, B) & & : subst.\ D \\
& \approx D' \unrhd (A \,\S\, B)\theta & & : \textbf{Back}-\textbf{1} \\
& \approx D' \unrhd (s.Done \,\S\, t.Done) & & : subst.\ A, B \\
& \approx s.t.(D' \unrhd Done) & & : \textbf{Back}-\textbf{3}\ \ repeated \\
& \approx s.t.(D' \rhd (A \,\S\, B)) & (\mathbf{1}) & : \textbf{Back}-\textbf{4}
\end{array}
$$

$$
\begin{array}{rlll}
RHS: & (D \rhd A) \,\S\, (D \rhd B) & & \\
& \approx (\overline{succ(\theta)}.D' \rhd A) \,\S\, (\overline{succ(\theta)}.D' \rhd B) & & : subst.\ D \\
& \approx (D' \unrhd A\theta) \,\S\, (D' \unrhd B\theta) & & : \textbf{Back}-\textbf{1} \\
& \approx (D' \unrhd s.Done) \,\S\, (D' \unrhd t.Done) & & : subst.\ A, B \\
& \approx s.(D' \unrhd Done) \,\S\, t.(D' \unrhd Done) & & : \textbf{Back}-\textbf{3}\ \ repeated \\
& \approx s.(D' \rhd A) \,\S\, t.(D' \rhd B) & (\mathbf{2}) & : \textbf{Back}-\textbf{4}
\end{array}
$$

However, **(1)** *and* **(2)** *are not bisimilar, since the sequence $s.t$ found in* **(1)** *is separated by the sequence generated by $D' \rhd A$ in* **(2)**. □

The associativity of $\rhd$ is now proven.

Theorem 4.2.5 *Associativity of backtracking*

$$(A \rhd B) \rhd C \quad \approx \quad A \rhd (B \rhd C)$$

Proof: *The proof uses structural induction over the form of the stream generated by A. Consider first a terminating stream from A.*

<u>*Base case*</u>*: Let $A \approx Done$. Then,*

$$
\begin{array}{rlll}
LHS: & (A \rhd B) \rhd C & \approx (Done \rhd B) \rhd C & : subst.\ A \\
& & \approx Done \rhd C & : \textbf{Back}-\textbf{2} \\
& & \approx Done & : \textbf{Back}-\textbf{2}
\end{array}
$$

$$
\begin{array}{rlll}
RHS: & A \rhd (B \rhd C) & \approx Done \rhd (B \rhd C) & : subst.\ A \\
& & \approx Done & : \textbf{Back}-\textbf{2}
\end{array}
$$

<u>*Inductive case*</u>*: Let $A \approx \overline{succ(\theta)}.A'$. Then,*

$$
\begin{array}{rlll}
LHS: & (A \rhd B) \rhd C & \approx (\overline{succ(\theta)}.A' \rhd B) \rhd C & : subst.\ A \\
& & \approx (B\theta \,\S\, (A' \rhd B)) \rhd C & : \textbf{Back}-\textbf{5} \\
& & \approx (B\theta \rhd C) \,\S\, ((A' \rhd B) \rhd C) & : right\ \ distr.
\end{array}
$$

$$
\begin{array}{rlll}
RHS: & A \rhd (B \rhd C) & \approx (\overline{succ(\theta)}.A' \rhd (B \rhd C) & : subst.\ A \\
& & \approx (B \rhd C)\theta \,\S\, (A' \rhd (B \rhd C)) & : \textbf{Back}-\textbf{5}
\end{array}
$$

Now, $B\theta \rhd C = (B \rhd C)\theta$, because θ is implicitly distributed to C in the first term. The *other term in both expressions is bisimilar by the induction hypothesis over the smaller stream generated by A'. Both expressions are bisimilar.* □

Finally, an inherent asymmetry in Prolog control is shown.

Theorem 4.2.6 *Both $\fatsemi$ and $\rhd$ are non–commutative:*

$$\begin{array}{rcl} P \fatsemi Q & \not\approx & Q \fatsemi P \\ P \rhd Q & \not\approx & Q \rhd P \end{array}$$

Proof: *For $P \fatsemi Q$, let $P \approx s.Done$ and $Q \approx t.Done$, where $s \neq t$. Then by theorem 4.1.1, $P \fatsemi Q \ \approx\ s.t.Done$. But similarly, $Q \fatsemi P \ \approx\ t.s.Done$, and $s.t.Done \neq t.s.Done$. A similar argument holds for $P \rhd Q$.* □

Theorem 4.2.6 is a direct result of interpreting Prolog computations using CCS's domain of answer substitution streams. Commutativity is especially relevant when considering predicates with modes, and looping and nontermination properties. In order to derive a more conventional model–theoretic interpretation, the stream domain must be relaxed, and the notion of *sets of computed answer substitutions* must be used. For example, if some program P computes $P \stackrel{s}{\Rightarrow}\stackrel{\overline{done}}{\rightarrow} \mathbf{0}$, then one must treat the atomic components of stream $s = \alpha_1.\cdots.\alpha_k$ as a set $\cup_{i=1}^{k}\{\alpha_i\}$. Note that in a set interpretation of sequences, one loses both the order and multiplicity of computed results.

4.3 Well–termination

An agent P is *well–terminating* if (i) for every derivative P' or P, $P' \stackrel{done}{\rightarrow}$ is impossible, and (ii) if $P' \stackrel{\overline{done}}{\rightarrow}$ then $P' \approx \overline{done}.\mathbf{0}$ (Milner 1989)[1]. Well–terminating does not insist that P must terminate, but only that it generates $\overline{done}$ iff it does terminate. Showing that agents are well–terminating means that they are in some sense computationally well–behaved, which is required for analysing properties of program termination. Different semantic expressions are now shown to be well–terminating.

Theorem 4.3.1 *If P, Q are well–terminating, then so is $P \fatsemi Q$.*

Proof: *(i) Because P and Q are well–terminating, it neither possible for $P \stackrel{s}{\Rightarrow}\stackrel{done}{\rightarrow}$, nor for $Q \stackrel{t}{\Rightarrow}\stackrel{done}{\rightarrow}$. Furthermore, inspecting the definition of $\fatsemi$,*

$$P \fatsemi Q \ \approx\ (P[b/done] \mid b.Q) \setminus b$$

[1] Even though the actual definition uses the stronger $\sim$ equivalence, the $\approx$ equivalence holds also.

the action done cannot be produced. Therefore, $P \mathbin{\hat{;}} Q \stackrel{s}{\Rightarrow}\stackrel{t}{\Rightarrow}\stackrel{done}{\rightarrow}$ *is impossible.*

(ii) If $P \mathbin{\hat{;}} Q \stackrel{s}{\Rightarrow}\stackrel{\overline{done}}{\rightarrow} R$*, then* $\overline{done}$ *is not from* P*, because* $\hat{;}$ *uses it. So* $P \mathbin{\hat{;}} Q \stackrel{t}{\Rightarrow} Q \stackrel{w}{\Rightarrow}\stackrel{\overline{done}}{\rightarrow} R$ *for some* $|t| \geq 0$ *and* $|w| \geq 0$*. Because* Q *is well–terminating,* $Q \stackrel{w}{\Rightarrow}\stackrel{\overline{done}}{\rightarrow} \mathbf{0}$*. Therefore* $R \approx \mathbf{0}$*.*

From (i) and (ii), $P \mathbin{\hat{;}} Q$ *is well–terminating.* □

Theorem 4.3.2 *If* P, Q *are well–terminating, then so is* $P \rhd Q$*.*

Proof: *(i) From the definition of* $\rhd$*,*

$$\begin{array}{rcl} P \rhd Q & \stackrel{\text{def}}{=} & (P[f] \mid NextGoal_i) \setminus F \\ NextGoal_i & \stackrel{\text{def}}{=} & succ'.(Q \mathbin{\hat{;}} NextGoal_i) + done'.Done \end{array}$$

$P \rhd Q \stackrel{s}{\Rightarrow}\stackrel{done}{\rightarrow}$ *is impossible, as the action done is not generated by well–terminating* P *and* Q*, and is not generated elsewhere in the above expressions.*

(ii) Given $P \rhd Q \stackrel{s}{\Rightarrow}\stackrel{\overline{done}}{\rightarrow} R$ *and the definition of* $\rhd$ *in (i) above, the only term which can generate* $\overline{done}$ *is the term Done in* $NextGoal_i$*. Because* $Done \approx \overline{done}.\mathbf{0}$ *by definition, then* $R \approx \mathbf{0}$*.*

From (i) and (ii), $P \rhd Q$ *is well–terminating.* □

Theorem 4.3.3 *All agents defined by the semantics of basic Prolog control are well–terminating.*

Proof: *By structural induction on the composition of agent expressions for a program. The base case are terminating AND agents having the forms:*

$$\begin{array}{lll} P_i \stackrel{\text{def}}{=} Done & : undefined\ \ goal \\ P_i \stackrel{\text{def}}{=} \overline{succ(\theta)}.Done + Done & : builtin\ \ atom\ \ (unification) \end{array}$$

Both these forms are well–terminating. The inductive case is over the the composition of $\hat{;}$ *and* $\rhd$ *expressions. By associativity,*

$$\begin{array}{rcll} P_1 \mathbin{\hat{;}} P_2 \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} P_k & \stackrel{\text{def}}{=} & P_1 \mathbin{\hat{;}} (P_2 \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} P_k) & : Theorem\ \ 4.2.1 \\ G_1 \rhd G_2 \rhd \cdots \rhd G_k & \stackrel{\text{def}}{=} & G_1 \rhd (G_2 \rhd \cdots \rhd P_k) & : Theorem\ \ 4.2.5 \end{array}$$

Applying the inductive hypothesis to both expressions, both are well–terminating using theorems 4.3.1 and 4.3.2. Therefore the semantics define well–terminating agents. □

Some comments about the semantics of cuts are worth mentioning. Expressions with $\rhd\!\!|_\ell$ are not well–terminating, since the termination signal $\overline{\ell}$ can be substituted for $\overline{done}$. However, because $\rhd\!\!|_\ell$ is always used in unison with $\mathring{;}$, expressions are well–terminating. Informally, this can be shown by considering the expression for a predicate P with cuts,

$$P_1 \overset{\circ}{;} \cdots \overset{\circ}{;} P_k \overset{\circ}{;} P_{k+1}$$

where P_{k+1} is the last clause of P. Even though P_1 through P_k are not well–terminating, termination of the whole expression occurs only if P_{k+1} terminates. The whole expression is well–terminating because P_{k+1} is well–terminating, since $\triangleright\!\!\!|_\ell$ cannot be used within it.

4.4 Correspondence with SLD resolution

There is a correspondence between the semantics and SLD–resolution described in section 2.1. In particular, there is a direct mapping between the AND/OR trees denoted by semantic expressions and SLD–trees used to model SLD–derivations. The soundness of the semantics then follows from the soundness of SLD–resolution. When the soundness of SLD–resolution is referred to, it is assumed that the unification algorithm uses an occurs check.

Section 3.2 discusses how the semantics denotes AND/OR trees. A useful measure of an AND/OR tree's complexity is its depth. An *n–depth* AND/OR tree is a finite AND/OR tree with at most n levels of nested AND nodes along any of its branches. For example, a 1–depth AND/OR tree is one with at most one AND node on a branch. A 0–depth tree is an empty tree. With respect to a CCS representation of an AND/OR tree, the depth is a measure of the greatest number of nested AND agent invocations used to resolve a set of goals.

The next lemma shows how the semantics uses Prolog's left–to–right computation rule.

Lemma 4.4.1 *Consider the CCS translation $G_1 \triangleright \cdots \triangleright G_k$ for some goal $G = ?- G_1, ..., G_k$. The semantics uses a computation rule which selects the first goal G_1.*

Proof: *Assume each goal has the form $G_i \approx \overline{\alpha_i}.G'_i$ where $\alpha_i \in \{\overline{succ}, \overline{done}\}$. G is therefore*

$$\alpha_1.G'_1 \ \triangleright\ \alpha_2.G'_2 \ \triangleright \cdots \triangleright\ \alpha_k.G'_k$$

Applying the expansion theorem to this expression results in either

$$G'_1 \ \triangleright\ (\alpha_2.G'_2 \ \triangleright \cdots \triangleright\ \alpha_k.G'_k)$$

if $\alpha_1 = \overline{succ}$*, or Done if* $\alpha_1 = \overline{done}$*. This represents the selection of* G_1 *from amongst the goals.* □

The mapping of the semantics with SLD–resolution is now illustrated by showing how AND/OR trees denoted by semantic expressions correspond to SLD–trees.

Theorem 4.4.2 *Given a program* P *and query* G*, the AND/OR tree derived from the semantics corresponds to the SLD–tree for* $P \cup \{G\}$ *using Prolog's left–to–right computation rule. In particular, the expressions* $\overline{succ(\theta)}.Done$ *and Done map to success and failure leafs of the SLD–tree respectively, non–leaf nodes have corresponding backtracked expressions in the semantics, and sibling descendents of a node in the SLD tree map are delimited using* $\,\hat{;}\,$ *in semantic expressions.*

Proof: *The proof uses induction on the depth of the AND/OR trees produced by the semantics.*

Base case: 0–depth AND/OR tree. There are two subcases.

(i) Success leaf. Let goal G *be a single literal which calls* $H \stackrel{\text{def}}{=} H_1 \,\hat{;}\, \cdots \,\hat{;}\, H_n$ $(n \geq 1)$*. Then*

$$\begin{array}{lll} G & \approx H_1 \,\hat{;}\, \cdots \,\hat{;}\, \underline{H_i} \,\hat{;}\, \cdots \,\hat{;}\, H_n & (i \leq i \leq n) \\ & \approx H_1 \,\hat{;}\, \cdots \,\hat{;}\, \overline{succ(\theta)}.Done \,\hat{;}\, \cdots \,\hat{;}\, H_n & : \textbf{Resol} \end{array}$$

The goal G *above maps to the root of the SLD–tree, and the* H_i *term with its associated mgu* θ *corresponds to a success leaf of the SLD–tree for the clause* H_i*. The other* $H_{j \neq i}$ *are treated similarly (when case (ii) is also considered).*

(ii) Failure. Let goal $G = G_1 \triangleright \cdots \triangleright G_n$ $(n \geq 1)$*. By lemma 4.4.1, the goal selected is* G_1*. However, to have a 1–depth tree, no clause resolves with* G_1*:*

$$\begin{array}{lll} G & \approx G_1 \triangleright \cdots \triangleright G_n & \\ & \approx (H_1 \,\hat{;}\, \cdots \,\hat{;}\, H_m) \triangleright G_2 \triangleright \cdots \triangleright G_n & (m \geq 1) \\ & \approx Done \,\hat{;}\, \cdots \,\hat{;}\, Done) \triangleright G_2 \triangleright \cdots \triangleright G_n & : all \;\; H_i \approx Done \\ & \approx Done \triangleright G_2 \triangleright \cdots \triangleright G_n & : simplify \\ & \approx Done & : \textbf{Back}-\textbf{2} \end{array}$$

The goal G *again maps to the root of the SLD–tree, but because there are no resolvents for* G_1*, this root is a failure leaf. This is denoted by the expression Done above.*

Inductive case: Assume that $k-1$ *depth AND/OR trees have corresponding SLD–trees. Consider a* k *depth AND/OR tree, which derives either* $\overline{succ(\theta)}$ *or Done. Let the root node of the AND/OR tree be the goal expression* $G_1 \triangleright \cdots \triangleright G_n$*. Without* *any loss of generality, assume that the selected goal G_1 has depth k. Let G_1 refer to the equations,*

$$
\begin{array}{l}
H \stackrel{\text{def}}{=} H_1 \,\hat{;}\, \cdots \,\hat{;}\, H_m \\
H_1 \stackrel{\text{def}}{=} Body_1 \\
\ldots \\
H_m \stackrel{\text{def}}{=} Body_m
\end{array}
$$

where $Body_i$ is a single call to $=$ if the clause is an assertion. Then

$$
\begin{array}{lll}
 & G_1 \rhd \cdots \rhd G_n & \\
\approx & (H_1 \,\hat{;}\, \cdots \,\hat{;}\, H_m) \rhd \cdots \rhd G_n & : subst.\ G_1 \\
\approx & (H_1 \rhd \cdots \rhd G_n) \,\hat{;}\, \cdots \,\hat{;}\, (H_m \rhd \cdots \rhd G_n) & : left - distributivity
\end{array}
$$

Now, when **Resol** *is applied and the H_i are suitable replaced, the expression reduces to*

$$(Body_i \rhd \cdots \rhd G_n)\theta_i \,\hat{;}\, \cdots \,\hat{;}\, (Body_j \rhd \cdots \rhd G_n)\theta_j \quad (1 \le i \le j \le m) \qquad (\dagger)$$

because terms denoting clauses which did not resolve fall out of the expression. This expression represents one AND–level of the AND/OR tree for G. All AND/OR subtrees denoted by further derivations of terms in this expression will necessarily be of depth $< k$. This expression maps to the top of the SLD–tree for goal G: G corresponds with the root of the SLD–tree, and each term delimited by $\hat{;}$ maps to one descendent branch of the SLD–tree for $?-\ G_1, ..., G_n$, where G_1 is the selected goal. Thus, because $(\dagger)$ maps to the top of the SLD–tree for goal G, and since the subterms have corresponding SLD–trees by the induction hypothesis, then the whole AND/OR tree for G corresponds to the SLD–tree for the query created using the same left–to–right computation rule. □

The particular SLD–tree used by a computation depends upon the goal selection or computation rule used. Theorem 4.4.2 therefore affirms that that the CCS semantics models the same SLD–tree as Prolog, by virtue of the fact that the same left–to–right computation rule is used.

The soundness of the semantics can now be stated.

Theorem 4.4.3 *The CCS semantics for a program P derives computed results which are sound with respect to P's standard declarative semantics.*

Proof: *SLD–resolution is sound (Lloyd 1984).* □

The above can be made clearer by showing the correspondence of the CCS semantics with the immediate consequence operator T_P commonly used to link the declarative and operational semantics of logic programs (see section 2.1). This permits a more direct comparison with the minimal model MM_P from a program's declarative semantics to be performed. Because CCS expressions naturally map to AND/OR trees, the relationship between computed results and T_P is direct and intuitive. The following notation from (Lloyd 1984) will be used. If A is an atom, $[A] = \{A' \in B_P : A' = A\theta,\ \mathit{for\ some\ substitution}\ \theta\}$. $[A]$ is the set of all ground instances of A.

Theorem 4.4.4 *For each answer substitution $\overline{succ(\theta)}$ computed by the semantic denotation of a single literal goal G and program P,*

$$[G\theta] \ \subseteq \ MM_P$$

Proof: *There is a direct correspondence between computed results and the sets of atoms generated by the T_P operator. This can be shown by structural induction on the depth of AND/OR trees denoted by semantic expressions.*

<u>Base case</u>: *Consider a single goal G denoting an AND/OR tree of depth 1. Then G resolves with agents representing assertions. Computed results from this tree are contained in those computed by $T_P^1(\emptyset)$, the atoms of B_P represented by all program assertions. If $G \approx \overline{succ(\theta_i)}$ $(i \geq 1)$, then*

$$\cup_i\ [G\theta_i] \ \subseteq \ [G] \ \ (G\ \ an\ \ assertion) \ \ = \ \ T_P^1(\emptyset) \ \subseteq \ MM_P$$

<u>Inductive case</u>: *Assume that the results computed by semantic expressions denoting AND/OR trees of depth $k-1$ are contained in $T_P^{k-1}(\emptyset)$, and are therefore in MM_P. Consider a goal $G(\tilde{t})$ which denotes a tree of depth k. The OR agent definition with which $G(\tilde{t})$ matches has the form*

$$G(\tilde{X}) \ \stackrel{\text{def}}{=} \ G_1(\tilde{X}) \ \hat{;} \ \cdots \ \hat{;}\ G_n(\tilde{X}) \quad (n \geq 1).$$

There exists at least one G_j in $G_1(\tilde{t})\ \hat{;}\ \cdots\ \hat{;}\ G_n(\tilde{t})$ which denotes a tree of depth k. Assume there is only one such G_j, and let it be

$$G_j(\tilde{X}) \stackrel{\text{def}}{=} (\tilde{X} = \tilde{t}_j) \rhd Body \quad\ (1 \leq j \leq n).$$

(The computed results from the other $G_{i \neq j}$ hold by the induction hypothesis, since they all denote trees of depth $< k$.) Letting $\phi = mgu(\tilde{t}, \tilde{t}_j)$, then after applying **Resol**, *$G_j(\tilde{t}) \approx Body\ \phi$. Because the goal $G(\tilde{t})$ is an AND agent and contributes one level to*

the depth of the tree, the subtree for Body ϕ must necessarily be of depth $k-1$. Thus, by the definition of T_P, $G_j(\tilde{t})$ computes a result contained in $T_P^k(\emptyset)$, and

$$\cup_i \; [G\theta_i] \;\; \subseteq \;\; T_P^k(\emptyset) \;\; \subseteq \;\; MM_P$$

for all computed $\overline{succ(\theta_i)}$ from Body ϕ. □

The above easily generalises to multiple literal goals (details omitted).

To show that the semantics uses the same search rule as Prolog, computed solutions from semantic expressions must be shown to correspond to the results computed by using Prolog's search rule on the corresponding SLD–tree – a tree which has already been shown to be equivalent in theorem 4.4.2. (Hogger 1990) describes Prolog's search strategy on SLD–trees as follows:

1. The SLD–tree is constructed using Prolog's left–to–right computation rule.

2. For each non–leaf node, it's descendents are given a priority, which is simply the textual ordering of the program clauses in some predicate which was used to resolve with the selected goal.

3. The search begins at the tree's root node.

4. A search step from a non–leaf node searches whichever one of the node's so-far-unsearched immediate descendents has the highest priority; the next search step is from that descendent.

5. A search step from a leaf node identifies that node's most recently searched ancestor, if any, having so–far–unsearched immediate descendents, and the next search step is from that ancestor. If there is no such ancestor, then the whole tree has been searched.

Step 1 constructs the SLD–tree according to Prolog's left–to–right computation rule. Steps 3 and 4 are responsible for the top–down, depth–first nature of Prolog's search, while step 5 is responsible for the exhaustive backtracking nature of the search.

The next theorem shows how the semantics uses Prolog's search strategy.

Theorem 4.4.5 *Prolog's search rule is modelled by the semantics.*

Proof: *The proof uses induction on the depth of the AND/OR tree in a similar way as theorem 4.4.2.*

Base case: 0–depth AND/OR tree. If goal G has more than one literal, then it must be a failure leaf in order for it to be of depth 1. Derivations of this trivially use an exhaustive depth–first search. Otherwise, consider a one literal goal G, which refers to an agent $H \stackrel{\text{def}}{=} H_1 \,\hat{;}\, \cdots \,\hat{;}\, H_n$. *To be of depth 1, the* H_i *are either assertions which resolve with G* ($H_i \approx \overline{succ(\theta)}.Done$), *or do not resolve with G* ($H_i \approx Done$). *In either case, when* **Seq** *is applied to the goal, these* H_i *contribute to the final solution to the program according to the sequential order of the clauses in the predicate. This corresponds to Prolog's searching the SLD–tree using the textual ordering of clauses. The search is trivially depth–first.*

Inductive case: Assume that $k-1$ *depth AND/OR trees use Prolog's search strategy, and consider a k–depth AND/OR tree. For simplicity, consider a goal* $G_1 \triangleright G_2$, *and assume that* G_1 *contributes k levels of depth to the derivation. Let* G_1 *invoke some* $H \stackrel{\text{def}}{=} H_1 \,\hat{;}\, \cdots \,\hat{;}\, H_n$, *where* $H_i \stackrel{\text{def}}{=} Body_i$ *and* $Body_i$ *is a single call to = in the case of assertions. Then,*

$$
\begin{array}{ll}
G_1 \triangleright G_2 & \\
\approx\ (H_1 \,\hat{;}\, \cdots \,\hat{;}\, H_n) \triangleright G_2 & : subst.\ G_1 \\
\approx ... & \\
\approx\ (Body_i \triangleright G_2)\theta_1 \,\hat{;}\, \cdots \,\hat{;}\, (Body_j \triangleright G_2)\theta_j & : left-dist.,\ apply\ \textbf{Resol},\ simplify
\end{array}
$$

Because each term has a depth $< k$, *then, by the induction hypothesis, Prolog's search strategy is employed on each. Now, by theorem 4.1.1, a term must execute to completion (generate 'Done') before the term following it can commence. This means that a depth–first search of each resolvent of the SLD–tree is being performed. In addition, the above uses the same textual ordering scheme as Prolog. Thus Prolog's search strategy is used.*
□

4.5 Equivalence with a functional semantics

This section shows the correspondence between the CCS semantics and a functional stream–based semantics by (Baudinet 1988). The semantics of the cut will not be addressed, but it could be similarly compared.

Baudinet presents a skeletal denotational semantics in which Prolog program constructs define functions over streams of answer substitutions. As with the CCS semantics, her domain of answer substitution streams has as its basic element an answer substitution, which is a set of replacement pairs $\{\ Var \leftarrow term\ \}$. Streams

$$< S_1 > \sqcup < S_2 > = \begin{cases} < S_1, S_2 > & : if \;\; S_1 \;\; is \;\; finite \;\; and \;\; proper \\ < S_1 > & : if \;\; S_1 \;\; is \;\; infinite \;\; or \;\; improper \end{cases}$$

$$S \bowtie f \;=\; \begin{cases} (\theta_1 \circ f(\theta_1)) \;\sqcup\; \cdots \;\sqcup\; (\theta_n \circ f(\theta_n)) & : if \;\; S =< \theta_1, ..., \theta_n > \\ (\theta_1 \circ f(\theta_1)) \;\sqcup\; \cdots \;\sqcup\; (\theta_n \circ f(\theta_n)) \sqcup < \perp > & : if \;\; S =< \theta_1, ..., \theta_n, \perp > \\ (\theta_1 \circ f(\theta_1)) \;\sqcup\; (\theta_2 \circ f(\theta_2)) \;\sqcup\; \cdots & : if \;\; S =< \theta_1, \theta_2, ... > \end{cases}$$

Figure 4.1: Semantic function definitions

have three forms:

(i) Finite streams: $< \theta_1, ..., \theta_n >$

(ii) Infinite streams: $< \theta_1, \theta_2, ... >$

(iii) Looping streams: $< \theta_1, ...\theta_k, \perp >$, where $\perp$ represents looping.

Streams without $\perp$ are termed *proper*, and those with $\perp$ are *improper*.

Two semantic functions used by Baudinet are defined in figure 4.1. The $\sqcup$ operator concatenates streams. The $\bowtie$ or join operator distributes a stream S on the left onto a functional expression f on the right by applying each answer substitution element in stream S with the right side expression. In addition, that element is used as an argument within the right side expression.

Some notation dealing with dataflow is used, which is similar to that in section 3.5.1. Π_D restricts a stream to variables found in set D. This D set is constructed using $vset(\tilde{t})$, which returns all the logical variables found in tuple $\tilde{t}$. For example, $vset(a(X), b(Y, f(Z))) \;=\; \{X, Y, Z\}$. Answer substitutions are accumulated using the usual notation, eg. $\theta \circ \gamma$.

Baudinet's functional semantics of Prolog is in figure 4.2. All these semantic expressions define functions which produce the same streams of answer substitutions as Prolog. In (1), a predicate p is a concatenation of all the streams produced by each of its component clauses. The functions in (2a) and (2b) define clauses. The function in (2a) represents facts, and returns a single element stream containing the mgu of the unification, or a null stream if no unification is possible. The function in (2b) returns

$$(1)\quad [\![\, p \,]\!](\tilde{t}) \;=\; [\![\, p_1 \,]\!](\tilde{t}) \sqcup \cdots \sqcup [\![\, p_l \,]\!](\tilde{t})$$

$$for\ \ predicate\ \ p,\ \ clauses\ \ p_1, ..., p_l.$$

$$(2a)\quad [\![\, p_i \,]\!](\tilde{t}) \;=\; \begin{cases} if\ \ (\theta = fail)\ \ then\ \ <> \\ else\ \ \Pi_{vset(\tilde{t})}\ (<\theta>) \end{cases}$$

$$where\ \ \theta = mgu(p(\tilde{t}), head_i),\ \ p_i \equiv\ \ head_i.$$

$$(2b)\quad [\![\, p_i \,]\!](\tilde{t}) \;=\; \begin{cases} if\ \ (\theta = fail)\ \ then\ \ <> \\ else\ \ \Pi_{vset(\tilde{t})}\ (\theta \circ [\![\, body_i\theta \,]\!]) \end{cases}$$

$$where\ \ \theta = mgu(p(\tilde{t}), head_i),\ \ p_i \equiv\ \ head_i :- body_i.$$

$$(3a)\quad [\![\, nil \,]\!] \;= <\{\ \}>$$

$$(3b)\quad [\![\, a(\tilde{t}_a), b(\tilde{t}_b), ..., z(\tilde{t}_z) \,]\!] \;=\; \Pi_D([\![\, a \,]\!](\tilde{t}_a)\ \ \bowtie\ \ \lambda\ \phi.[\![\, b(\tilde{t}_b\phi), ..., z(\tilde{t}_z\phi) \,]\!])$$

$$where\ \ m \geq 1,\ D = vset(\tilde{t}_a, ..., \tilde{t}_z).$$

Figure 4.2: A functional semantics of Prolog over streams

the result of applying the mgu to the meaning of the body of the clause; the mgu θ is included in the result (via " $\theta \circ$ "), as well as applied to the body. Finally, (3a) and (3b) define the semantics of goals. Null goal bodies result in an empty stream in (3a). The stream produced by non–null goal bodies in (3b) is defined by joining the stream produced by the first goal with the functional meaning of the rest of the goals. The "λ" notation represents the fact that the rest of the goal expression uses the elements from the left hand side as arguments (evident in the definition of $\bowtie$ in figure 4.1).

The remainder of this section shows the semantic equivalence between $\sqcup$ and $\hat{;}$, and between $\bowtie$ and $\triangleright$. To show equivalence, the following mapping of streams between the two systems is implicit:

$$\begin{array}{llcl}
 & \underline{Functional} & & \underline{CCS} \\
Null\ \ streams: & <\ > & \Leftrightarrow & Done \\
Finite\ \ streams: & <\theta_1, \theta_2 ..., \theta_n> & \Leftrightarrow & \overline{succ(\theta_1)}\ .\ \overline{succ(\theta_2)}\ . \cdots .\ \overline{succ(\theta_k)}\ .\ Done \\
Infinite\ \ streams: & <\theta_1, \theta_2, ...> & \Leftrightarrow & \overline{succ(\theta_1)}\ .\ \overline{succ(\theta_2)}\ . \cdots \\
Looping\ \ streams: & <\theta_1, ..., \theta_n, \perp> & \Leftrightarrow & \overline{succ(\theta_1)}\ . \cdots .\ \overline{succ(\theta_k)}\ .\ \perp
\end{array}$$

The $\sqcup$ operator corresponds to the $\mathbin{\raisebox{0.5ex}{$\scriptstyle ?$};}$ operator. Item (1) of figure 4.2 defines a predicate p with clauses $p_1, ..., p_l$ to be $[\![\ p\ ]\!](\tilde{t}) \;=\; [\![\ p_1\ ]\!](\tilde{t}) \sqcup \cdots \sqcup [\![\ p_l\ ]\!](\tilde{t})$. The CCS translation for a predicate is $p(\tilde{X}) \stackrel{\text{def}}{=} p_1(\tilde{X}) \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} p_l(\tilde{X})$. Invoking $p(\tilde{X})$ with arguments $\tilde{t}$ is therefore $p(\tilde{t}) \;\approx\; p_1(\tilde{t}) \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} p_l(\tilde{t})$. Theorem 4.5.1 states that these expressions are equivalent.

Theorem 4.5.1 $\quad$ *$\sqcup$ and $\hat{;}$ are functionally equivalent.*

Proof: *From the definition of $\sqcup$ in figure 4.1, the expression $A \sqcup B$ produces the following for different A:*

$$A \sqcup B \;=\; \begin{cases} (i) & < A, B > \quad : A\ \ is\ \ finite\ \ and\ \ proper \\ (ii) & < A > \quad : A\ \ is\ \ infinite \\ (iii) & < A > \quad : A\ \ loops \end{cases}$$

In CCS, the corresponding results are:

$$A \mathbin{\hat{;}} B \;\approx\; \begin{cases} (i) & \alpha_1 \,.\cdots.\, \alpha_n \,.\, B \quad : Theorem\ \ 4.1.1,\ \ A \approx \alpha_1.\cdots\alpha_n.Done \\ (ii) & A \quad : Theorem\ \ 4.1.1 \\ (iii) & A \quad : Theorem\ \ 4.1.4 \end{cases}$$

$\square$

The $\bowtie$ operator corresponds to the $\triangleright$ operator. Figure 4.2 defines the semantics of an expression with goals to be:

$$\begin{aligned} &[\![\ g_1(\tilde{t}_1), g_2(\tilde{t}_2), ..., g_k(\tilde{t}_k)\ ]\!] \\ &= \;[\![\ g_1\ ]\!](\tilde{t}_1) \bowtie \lambda\phi_1.([\![\ g_2\ ]\!](\tilde{t}_2\phi_1) \bowtie \lambda\phi_2.(\ \cdots\ \bowtie \lambda\phi_k.[\![\ g_k\ ]\!](\tilde{t}_2\phi_1\phi_2\cdots\phi_{k-1})\cdots)) \end{aligned}$$

This expression represents the cumulative left–to–right application of computed results through a backtracking expression. The equivalent expression in CCS is

$$g_1(\tilde{t}_1) \triangleright g_2(\tilde{t}_2\theta_1) \triangleright \cdots \triangleright g_k(\tilde{t}_k\theta_1\theta_2\cdots\theta_{k-1})$$

The $\bowtie$ function treats streams as single computed objects, whereas $\triangleright$ treats backtracking as an algebraic expression which constructs successive stream elements when the expansion law or the **Back** equivalences are applied to it. The following shows that the streams generated by $\bowtie$ and $\triangleright$ are the same.

Theorem 4.5.2 *The $\bowtie$ and $\triangleright$ operators compute equivalent streams of answer substitutions.*

Proof: *The operators are proven to be equivalent by induction over the number of goals in a backtracking expression. (For clarity, Π is ignored throughout).*

<u>Base case</u>: For one goal, the functional semantics generates:

$$
\begin{aligned}
& [\![\, G_1 \,]\!](\tilde{t}_1) \;\bowtie\; \lambda\phi.[\![\, nil \,]\!] \\
= \;& [\![\, G_1 \,]\!](\tilde{t}_1) \;\bowtie < \{\,\} > \\
= \;& [\![\, G_1 \,]\!](\tilde{t}_1)
\end{aligned}
$$

The CCS semantics generates similar streams, since the stream generated is equivalent to $G_1(\tilde{t}_1)$.

<u>Inductive case</u>: Consider n backtracked goals $G_1(\tilde{t}_1), ..., G_n(\tilde{t}_n)$. The result is shown for the three types of streams generated by $G_1(\tilde{t}_1)$.

(i) G_1 generates a finite stream. For streams of size 0, the functional semantics produces:

$$
\begin{aligned}
& [\![\, G_1 \,]\!](\tilde{t}_1) \;\bowtie\; \lambda\phi.[\![\, G_2(\tilde{t}_2), ..., G_n(\tilde{t}_n) \,]\!] && \\
= \;& < > \bowtie\; \lambda\phi.[\![\, G_2(\tilde{t}_2), ..., G_n(\tilde{t}_n) \,]\!] && : subst.\ [\![\, G_1 \,]\!](\tilde{t}_1) \\
= \;& < > && : apply\ \ \bowtie
\end{aligned}
$$

CCS generates:

$$
\begin{aligned}
& G_1(\tilde{t}_1) \rhd G_2(\tilde{t}_2) \rhd \cdots \rhd G_n(\tilde{t}_n) && \\
= \;& Done \rhd G_2(\tilde{t}_2) \rhd \cdots \rhd G_n(\tilde{t}_n) && : subst.\ G_1 \\
= \;& Done && : \mathbf{Back-4}
\end{aligned}
$$

For streams of size $k > 0$, the functional semantics produces:

$$
\begin{aligned}
& [\![\, G_1 \,]\!](\tilde{t}_1) \;\bowtie\; \lambda\phi.[\![\, G_2(\tilde{t}_2), ..., G_n(\tilde{t}_n) \,]\!] && \\
= \;& < \theta_1, ..., \theta_k > \bowtie\; \lambda\phi.[\![\, G_2(\tilde{t}_2), ..., G_n(\tilde{t}_n) \,]\!] && : subst.\ [\![\, G_1 \,]\!](\tilde{t}_1) \\
= \;& (\theta_1 \circ A(\theta_1)) \sqcup (\theta_2 \circ A(\theta_2)) \cdots \sqcup (\theta_k \circ A(\theta_k)) \quad (\mathbf{1}) && : apply\ \ \bowtie
\end{aligned}
$$

where $A(\theta) = [\![\, G_2(\tilde{t}_2\theta), ..., G_n(\tilde{t}_n\theta) \,]\!]$. CCS produces:

$$
\begin{aligned}
& G_1(\tilde{t}_1) \rhd G_2(\tilde{t}_2) \rhd \cdots \rhd G_n(\tilde{t}_n) && \\
\approx \;& (\overline{succ(\theta_1)} \,.\cdots.\, \overline{succ(\theta_k)} \,.\, Done) \rhd G_2(\tilde{t}_2) \rhd \cdots \rhd G_n(\tilde{t}_n) && : subst.\ G_1 \\
\approx \;& (\theta_1 \circ A'(\theta_1)) \mathbin{\hat{;}} ((\overline{succ(\theta_2)} \,.\cdots.\, \overline{succ(\theta_k)} \,.\, Done) \rhd && \\
& \qquad\qquad G_2(\tilde{t}_2\theta_1) \rhd \cdots \rhd G_n(\tilde{t}_n\theta_1)) \quad (\mathbf{2}) && : \mathbf{Back-5} \\
\approx \;& ... && \\
\approx \;& (\theta_1 \circ A'(\theta_1)) \mathbin{\hat{;}} (\theta_2 \circ A'(\theta_2)) \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} (\theta_k \circ A'(\theta_k)) && : \mathbf{Back-5}\ \ repeated
\end{aligned}
$$

where $A'(\theta) \equiv G_2(\tilde{t}_2\theta) \rhd \cdots \rhd G_n(\tilde{t}_n\theta)$. Now for $(1 \leq i \leq k)$, because $\theta_i \circ A(\theta_i)$ and $\theta_i \circ A'(\theta_i)$ each have $n-1$ goals, they are equivalent by the induction hypothesis. $(\mathbf{1})$ and $(\mathbf{2})$ are therefore equivalent by the equivalence of $\sqcup$ and $\hat{;}$ (theorem 4.5.1).

(ii) G_1 generates an infinite stream. Similarly to (i) above, the expressions formulated are

$$(\theta_1 \circ A(\theta_1)) \sqcup (\theta_2 \circ A(\theta_2)) \sqcup \cdots \qquad (\mathbf{3})$$

$$(\theta_1 \circ A'(\theta_1)) \mathbin{\hat{;}} ((\overline{succ(\theta_2)} \,.\, \cdots) \rhd A'(\theta_1)) \quad (\mathbf{4})$$

where $A(\theta) = [\![\; G_2(\tilde{t}_2\theta), ..., G_n(\tilde{t}_n\theta) \;]\!]$ *and* $A'(\theta) \equiv G_2(\tilde{t}_2\theta) \rhd \cdots \rhd G_n(\tilde{t}_n\theta)$. *By inspection, successive applications of* **Back-5** *to* (**4**) *are equivalent to corresponding terms in* (**3**), *and* $\theta_i \circ A(\theta_i)$ *and* $\theta_i \circ A'(\theta_i)$ *are equivalent by the induction hypothesis.*

(iii) G_1 *loops. Let* G_1 *generate* $< \theta_1, ..., \theta_k, \perp >$ *for some* $k \geq 0$. *The functional semantics generates:*

$$\begin{array}{l} [\![\; G_1 \;]\!](\tilde{t}_1) \;\bowtie\; \lambda\phi.[\![\; G_2(\tilde{t}_2), ..., G_n(\tilde{t}_n) \;]\!] \\ = \;\; (\theta_1 \circ A(\theta_1)) \sqcup \cdots \sqcup (\theta_k \circ A(\theta_k)) \sqcup < \perp > \qquad (\mathbf{5}) \end{array}$$

where $A(\theta) = [\![\; G_2(\tilde{t}_2\theta), ..., G_n(\tilde{t}_n\theta) \;]\!]$. *The CCS expression is:*

$$\begin{array}{lll} & G_1(\tilde{t}_1) \rhd G_2(\tilde{t}_2) \rhd \cdots \rhd G_n(\tilde{t}_n) & \\ \approx & (\overline{succ(\theta_1)} \,.\, \cdots \,.\, \overline{succ(\theta_k)} \,.\, \perp) \rhd G_2(\tilde{t}_2) \rhd \cdots \rhd G_n(\tilde{t}_n) & : subst.\;\; G_1 \\ \approx & ... & \\ \approx & (\theta_1 \circ A'(\theta_1)) \mathbin{\hat{;}} (\theta_2 \circ A'(\theta_2)) \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} \theta_k \circ A'(\theta_k)) \mathbin{\hat{;}} & \\ & \qquad (\perp \rhd G_2(\tilde{t}_2) \rhd \cdots \rhd G_n(\tilde{t}_n)) & : \mathbf{Back-5}\;\; repeated \\ \approx & (\theta_1 \circ A'(\theta_1)) \mathbin{\hat{;}} (\theta_2 \circ A'(\theta_2)) \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} (\theta_k \circ A'(\theta_k)) \mathbin{\hat{;}} \perp \quad (6) & : \perp \rhd A \approx \perp \\ & & (theorem\;\; 4.1.5) \end{array}$$

where $A'(\theta) \equiv G_2(\tilde{t}_2\theta) \rhd \cdots \rhd G_n(\tilde{t}_n\theta)$. *Because* $\theta_i \circ A(\theta_i)$ *and* $\theta_i \circ A'(\theta_i)$ *are equivalent by the induction hypothesis, then* (**5**) *and* (**6**) *are equivalent by theorem 4.5.1.*

The streams generated by $\bowtie$ *and* $\rhd$ *are equivalent.* □

In summary, Baudinet's functional semantics and the CCS semantics share some characteristics. Both use a stream domain over answer substitutions, which is significantly less abstract than what is used by other denotational semantics of Prolog (Debray and Mishra 1988). The semantic behaviour of Prolog programs is denoted at the program level, rather than at a meta–program level as in (Hill and Lloyd 1988). Both systems use semantic operators which map to a Prolog programs goals and clauses, and the semantic meaning of these operators is the same.

The two formalisms, however, differ in a number of respects. The most obvious difference is the mathematical machinery used. Baudinet's semantics is a skeletal denotational semantics, in which all the components of the semantics are functions. The meaning of a program – the stream of answer substitutions which it computes – is defined by the fixpoint solution of the functional equations for the program. The CCS semantics uses algebraic operators whose semantics in turn are defined using algebraic transitions. The meaning of a program is derived by reducing the algebraic expressions for a program using equivalence–preserving semantic substitutions. CCS does have a fixpoint interpretation, and applying it onto the CCS semantics of Prolog results in a semantic model similar to Baudinet's.

A major difference is that the functional semantics defines the final computed results of goals and clauses axiomatically, while the CCS semantics defines the operational semantics of them from which computed results are deduced. For example, the $\bowtie$ operator defines the final stream result of two backtracked expressions, while the $\rhd$ operator models backtracking by successive applications of the expansion theorem or bisimilarities to it. The $\bowtie$ produces the results of backtracking in one fell swoop, whereas $\rhd$ *constructs* the stream with iterative (inductive) expansion. The CCS semantics can be regarded as a rational reconstruction of Baudinet's semantics, as it constructs the final stream results, and reasons directly about nontermination and looping, while Baudinet defines the semantics axiomatically.

4.6 Equivalence with a Prolog meta–interpreter

$$
\begin{array}{l}
solve(true). \\
solve((X,Y)) :- solve(X),\ solve(Y). \\
solve(not(X)) :- not\ \ solve(X). \\
solve(X) :- clause(\ X :- Y\ ),\ solve(Y).
\end{array}
$$

Figure 4.3: Prolog meta-interpreter

Meta–interpreters are a powerful way of prototyping and implementing logic programming languages and tools. For example, the basic Prolog meta–interpreter in figure 4.3 models Prolog's standard control strategy with negation as failure.
(Hill and Lloyd 1988) study some semantic issues for this and other Prolog meta–interpreters. They treat such meta–interpreters as pure Horn clause programs (figure 4.4), and discuss how the declarative and operational semantics of meta–interpreters

$$\begin{array}{rcl} solve(true) & & \\ \forall\, X,Y \;\; solve((X,Y)) & \leftarrow & solve(X) \;\wedge\; solve(Y) \\ \forall\, X \;\; solve(not(X)) & \leftarrow & \neg\; solve(X) \\ \forall\, X,Y \;\; solve(X) & \leftarrow & clause(X:-Y) \;\wedge\; solve(Y) \end{array}$$

Figure 4.4: Logical semantics of meta-interpreter

correspond to those of the programs they execute.

By virtue of their ability to treat programs as data, meta–interpreters can easily encode various control strategies for logic programming languages. However, meta–interpretive semantics are not intended to be used as programming calculi for proving program properties "in the small". The operational semantics of the meta–language must be accounted for when modelling the operational semantics of the object language. For example, in the meta–interpreter in figure 4.3, the meta–language's backtracking mechanism is used in the second and fourth clauses, and negation as failure is used in the third clause.

The intention of this section is to ascribe a CCS semantics of control onto the basic meta–interpreter shown in figure 4.3, and then demonstrate how the meta–interpreter's CCS semantics corresponds to the CCS semantics for the object programs which it executes. This exercise will present a new semantics of Prolog meta–interpreters, and also verify the semantic completeness of the CCS semantics itself. Dataflow is not treated here, and a logic variable domain is assumed throughout.

$$\begin{array}{rcl} solve(X) & \stackrel{\text{def}}{=} & solve_1(X) \,\hat{;}\, solve_2(X) \,\hat{;}\, solve_3(X) \,\hat{;}\, solve_4(X) \\ solve_1(X) & \stackrel{\text{def}}{=} & X = true \\ solve_2(X) & \stackrel{\text{def}}{=} & (X = (A,B)) \rhd solve(A) \rhd solve(B) \\ solve_3(X) & \stackrel{\text{def}}{=} & (X = not(A)) \rhd Not\;\; solve(A) \\ solve_4(X) & \stackrel{\text{def}}{=} & Clause(X,Y) \rhd solve(Y) \end{array}$$

Figure 4.5: CCS semantics of meta-interpreter

A CCS semantics of the meta–interpreter of figure 4.3 is in figure 4.5. The $\rhd$, $\hat{;}$

and Not operators are as before. An agent $Clause$ is new, and needs some elaboration. A first reaction to Prolog's *clause* utility is to consider it to be a messy extra–logical utility similar to the likes of *assert* and *retract*. However, (Hill and Lloyd 1988) suggest a logical interpretation of *clause*. Consider a predicate defined by the clauses

$$\begin{array}{l} P_1(\tilde{t}_1) :- B_1. \\ P_2(\tilde{t}_2) :- B_2. \\ \ldots \\ P_k(\tilde{t}_k) :- B_k. \end{array}$$

where B_i is either a list of goals $G_1, ..., G_n$, or the atom $true$ in the case that P_i is an assertion. The declarative semantics of *clause* is

$$\begin{array}{l} \forall\ clause(P(\tilde{t}_1), B_1) \\ \forall\ clause(P(\tilde{t}_2), B_2) \\ \ldots \\ \forall\ clause(P(\tilde{t}_k), B_k) \end{array}$$

where *clause* is a relation over the syntactic components of the predicates in the program. The corresponding operational meaning of $clause(P(\tilde{X}), Q)$, when $\tilde{X}$ and Q are uninstantiated, is to return each of the defined bodies Q for predicate header P, with the arguments $\tilde{X}$ bound appropriately. Of course, logical variables differ in meaning depending upon whether they range over program code as Q does in $clause(P, Q)$, or whether they are part of an argument within the program code itself as do the variables in $\tilde{X}$ (see (Hill and Lloyd 1988)).

A CCS treatment of *clause* is as follows. The domain of the *clause* relation is atoms from the Herbrand universe, embellished with operators for goal conjunction and negation. Likewise, the *Clause* agent in figure 4.5 will use value arguments over similar constructs. *Clause* itself is defined in figure 4.6. A call $clause(P, Q)$ returns a stream of $\overline{succ(\theta)}$ actions, each θ binding Q to a clause body whose head unifies with P, and with this unifying binding applied through it. The *Clauses* agent has encoded in it the clauses for all the predicates in the program database:

$$\overline{clause(p(\tilde{X}), C_1)}\ .\ \overline{clause(p(\tilde{X}), C_2)}\ .\ \cdots\ .\ \overline{clause(p(\tilde{X}), C_k)}\ .\ Done$$

where C_i are either the goals of rules, or $true$ in the case of assertions. The order of this sequence reflects the textual order of clauses in the program. *ClauseLoop* then unifies each clause to the predicate head in H; when a clause unifies, the body is returned with the unifying substitution applied to it. The behaviour of $Clause$ is stated in the following lemma.

$$
\begin{aligned}
&Let\ \ [f] \ \equiv\ [\ succ'/succ,\ done'/done\ ] \\
&\qquad F \ \equiv\ \{\ succ',\ done'\ \}
\end{aligned}
$$

$$Clause(H,B) \stackrel{\text{def}}{=} (\ Clauses\,[done'/done] \mid ClauseLoop(H,B)\ ) \setminus \{done', clause\}$$

$$Clauses \stackrel{\text{def}}{=} \overline{clause(P(\tilde{t}_1), B_1)}\ .\ \cdots\ .\ \overline{clause(P(\tilde{t}_k), B_k)}\ .\ Done$$

$$
\begin{aligned}
ClauseLoop(H,B) \stackrel{\text{def}}{=}\ & clause(P,Q)\ .\ GetBody(H,B,P,Q)\ ;\ ClauseLoop(H,B) \\
& + done'.Done
\end{aligned}
$$

$$
\begin{aligned}
GetBody(H,B,P,Q) \stackrel{\text{def}}{=}\ & (\ (H = P)[f] \mid (succ'(\theta)\ .\ \overline{succ(\{B \leftarrow Q\theta\})}\ .\ Done \\
& + done'.Done)\ ) \setminus F
\end{aligned}
$$

Figure 4.6: Clause agent

Lemma 4.6.1 *Given a goal $G = P(\tilde{t})$, and a predicate P composed of n clauses:*

$$
\begin{aligned}
&P(\tilde{t}_1) :- C_1. \\
&\ldots \\
&P(\tilde{t}_n) :- C_n.
\end{aligned}
$$

where C_i are lists of goals, or true if P_i is an assertion. If at least one clause resolves with G, then

$$Clause(P(\tilde{t}), Y) \ \approx\ \overline{succ(\theta_1)}\ .\ \cdots\ .\ \overline{succ(\theta_k)}\ .\ Done \qquad (1 \le k \le n)$$

where each $\theta_i = \{Y \leftarrow C_j\theta_j\}$ and $\theta_j = mgu(\tilde{t}, \tilde{t}_j)$, for $(1 \le i \le k)$ and $(1 \le j \le n)$. (k can be less than n in cases where some clauses do not unify.) Otherwise, if no clauses unify with G, then $Clause(P(\tilde{t}), Y) \ \approx\ Done$.

Proof: *Induction on number of clauses given to Clauses.* □

The main proof of semantic correspondence is now given.

Theorem 4.6.2 *Given a program P and query Q, and assume no negation by failure is being used. The computed result executed by the meta–interpreter is the same as what is generated by the operational semantics for P and Q directly, ie. $solve(Q) \approx Q$.*

Proof: *It will be shown that the CCS semantics of the meta–interpreter in figure 4.5* *with respect to query solve(Q) is bisimilar to the CCS semantics of the query Q itself:*

$$\mathcal{M}_{ccs}[\![\ solve(Q)\ ]\!] \ \approx\ \mathcal{M}_{ccs}[\![\ Q\ ]\!]$$

This is done by showing that solve is bisimilar to another agent solve′ which is semantically equivalent to the CCS semantics of source programs themselves. In particular, solve is applied to each predicate in the program database using its most general call, ie. $solve(P(\tilde{X}))$. *It is shown that the semantics of this general query is bisimilar to another agent* $solve'_P(\tilde{X})$, *which is just a syntactic variation of the CCS semantics of program* $P(\tilde{X})$ *itself.*

Consider a general predicate P composed of n clauses, each of which has the form "$P(\tilde{t_i}) :- B_i.$*", where* B_i *is a list of goals if the* i^{th} *clause is a rule, or "true" if an assertion. The expansion of the general meta–interpreter call to P is:*

$$\begin{array}{lll} & solve(P(\tilde{X})) & \\ \approx & solve_4(P(\tilde{X})) & : \textbf{Con}\ \ solve,\ \ simplify \\ \approx & \overline{Clause(P(\tilde{X}), Y)} \rhd solve(Y) & : \textbf{Con}\ \ solve_4 \\ \approx & (\overline{succ(\theta_1)}\ .\ \cdots\ .\ \overline{succ(\theta_n)}\ .\ Done) \rhd solve(Y) & : Lemma\ \ 4.6.1 \end{array}$$

In the last step, all the clauses unify with the general goal, and therefore there are n bindings, each having the form $\theta_i = \{Y \leftarrow B_i\phi_i\}$, *where* ϕ_i *is a binding of the header terms with the general* $\tilde{X}$ *argument. If we apply* **Back – 5** *to each* $\overline{succ(\theta_i)}$ *in the stream on the left, this expression is bisimilar to*

$$solve(B_1\phi_1)\ \hat{;}\ \cdots\ \hat{;}\ solve(B_n\phi_n) \qquad \mathbf{(1)}$$

Consider a term $solve(B_i\phi_i)$ $(1 \le i \le n)$ *from this expression. The clause body* B_i *has one of two forms. If it is an assertion, then*

$$\begin{array}{llll} & solve(B_i\phi_i) & & \\ \approx & solve(true\ \phi_i) & & : B_i\ \ is\ \ an\ \ assertion \\ \approx & solve_1(true\ \phi_i) & & : \textbf{Con}\ \ solve,\ \ simplify \\ \approx & \overline{succ(\phi_i)}.Done & \mathbf{(2)} & : \textbf{Resol}\ \ solve_1 \end{array}$$

However, if it is a rule with k goals, then

$$
\begin{array}{ll}
solve(B_i\phi_i) & \\
\approx\ solve((G_1(\tilde{t}_1), ..., G_k(\tilde{t}_k))\phi_i) & : B_i \ \ is \ \ a \ \ rule \\
\approx\ solve_2((G_1(\tilde{t}_1), ..., G_k(\tilde{t}_k))\phi_i) & : \textbf{Con} \ \ solve, \ \ simplify \\
\approx\ solve(G_1(\tilde{t}_1)\phi_i) \rhd solve((G_2(\tilde{t}_2), ..., G_k(\tilde{t}_k))\phi_i) & : \textbf{Resol} \ \ solve_2 \\
\approx ... & \\
\approx\ solve(G_1(\tilde{t}_1)\phi_i) \rhd solve(G_2(\tilde{t}_2)\phi_i) \rhd & \\
\qquad\qquad \cdots \rhd solve(G_k(\tilde{t}_k)\phi_i) \qquad (\mathbf{3}) & : \textbf{Resol} \ \ solve_2 \ \ repeated
\end{array}
$$

It will now be shown that expressions $(\mathbf{1})$, $(\mathbf{2})$, *and* $(\mathbf{3})$ *are bisimilar to the CCS semantics of the source program being executed. Firstly, in* $(\mathbf{1})$,

$$solve(P(\tilde{X})) \ \approx \ solve(B_1\phi_1) \,\hat{;}\, \cdots \,\hat{;}\, solve(B_n\phi_n)$$

$solve(P(\tilde{X}))$ *is bisimilar to a new agent* $solve'_P(\tilde{X})$, *which is defined as*

$$solve'_P(\tilde{X}) \stackrel{\text{def}}{=} solve'_{P_1}(\tilde{X}) \,\hat{;}\, \cdots \,\hat{;}\, solve'_{P_n}(\tilde{X}) \qquad (\mathbf{4})$$

These $solve'_{P_i}$ *agents are defined from* $(\mathbf{2})$ *and* $(\mathbf{3})$ *as follows. Each* $solve(B_i\phi_i)$ $(1 \leq i \leq n)$ *from* $(\mathbf{1})$ *is bisimilar to* $solve'_{P_i}(\tilde{X})$ *in* $(\mathbf{4})$. *This* $solve'_{P_i}(\tilde{X})$ *is defined according to whether case* $(\mathbf{2})$ *or* $(\mathbf{3})$ *holds:*

$$solve'_{P_i}(\tilde{X}) \stackrel{\text{def}}{=} \begin{cases} \tilde{X} = \tilde{t}_i & for \ \ (\mathbf{2}) \\ \\ (\tilde{X} = \tilde{t}_i) \rhd solve(G_1(\tilde{t}_1)) \rhd \cdots \rhd solve(G_k(\tilde{t}_k)) & for \ \ (\mathbf{3}) \end{cases}$$

where $\phi = mgu(\tilde{X}, \tilde{t}_i)$. *This conversion is performed on all the predicates in the program; consequently, all the goals in the new* $solve'_{P_i}$ *agents are made to refer to these newly constructed agents:*

$$solve'_{P_i}(\tilde{X}) \stackrel{\text{def}}{=} (\tilde{X} = \tilde{t}_i) \rhd solve'_{G_1}(\tilde{t}_1) \rhd \cdots \rhd solve'_{G_k}(\tilde{t}_k)$$

Once this has been done, the following correspondence is noted between the $solve'$ *agent for a particular predicate* P,

$$solve'_P(\tilde{X}) \stackrel{\text{def}}{=} solve'_{P_1}(\tilde{X}) \,\hat{;}\, \cdots \,\hat{;}\, solve'_{P_n}(\tilde{X})$$
$$solve'_{P_i}(\tilde{X}) \stackrel{\text{def}}{=} \begin{cases} \tilde{X} = \tilde{t}_i \\ \\ (\tilde{X} = \tilde{t}_i) \rhd solve'_{G_1}(\tilde{t}_1) \rhd \cdots \rhd solve'_{G_k}(\tilde{t}_k) \end{cases}$$

and the CCS semantics of that same predicate P:

$$P(\tilde{X}) \stackrel{\text{def}}{=} P_1(\tilde{X}) \,\mathbin{;}\, \cdots \,\mathbin{;}\, P_n(\tilde{X})$$

$$P_i(\tilde{X}) \stackrel{\text{def}}{=} \begin{cases} \tilde{X} = \tilde{t}_i \\ \\ (\tilde{X} = \tilde{t}_i) \rhd G_1(\tilde{t}_1) \rhd \cdots \rhd G_k(\tilde{t}_k) \end{cases}$$

These agent definitions are bisimilar, because they both define the same recursive agent expression, differing only by their syntactic relabelling of agent names. Because the solutions to recursive expressions are unique in CCS, they must be equivalent (Milner 1989, pages 56-58). □

For programs with negation by failure, the semantic translation of the meta–interpreter call for a negated goal G is "$Not\ \ G$", which is the same as the semantics of negated goals in CCS. Hence the translation is the same.

Theorem 4.6.2 shows how the operational semantics of Prolog in which the meta–interpreter executes under is exploited when executing Prolog programs. This is a significant exercise, since the computational behaviour of such interpretations is crucially dependent upon the operational behaviour of the meta–interpreter when executed with Prolog's standard control. This is reflected in the specific order of goals in the second and fourth clauses of the meta–interpreter, which execute properly with Prolog's left–to–right computation rule. This semantics of the meta–interpreter could therefore be used within analyses, such as proving termination properties. However, as shown above, the semantics of the meta–interpretation of the source program is equivalent to that of the program itself. In the context of proving program properties, performing analyses on the semantics of programs is more practical than doing so on a meta–program which uses programs as data.

4.7 Conclusion

This chapter has derived some algebraic properties of the semantics, as well as verified the correctness of the semantics. The termination properties give a stream characterisation of Prolog computations, and allow the semantic modelling of non–termination and looping. Properties such as associativity, right–distributivity, and non–commutativity define the compositional characteristics of the operators, and permit semantic expressions to be manipulated algebraically. Finally, some well–termination properties were derived, which establish an aspect of the integrity of semantic expressions.

Together, the termination and compositional properties contribute to a programming calculus of Prolog control. The termination properties define equivalence substitutions of expressions when the stream behaviours of component sub–expressions are known. The compositional properties define sound algebraic manipulations of semantic expressions. These properties will be usefully applied later within program analyses.

Chapter 5

Program Termination

Initial work in logic program termination assumed that a general fair search strategy was to be used, and proving program termination reduced to proving the existence of solutions for given programs. However, with the use of unfair search strategies such as Prolog's in which the search can easily follow nonterminating and looping branches of the computation tree, existential proofs of termination are not adequate. In addition, Prolog's depth–first–left–first control means that a program's behaviour is crucially dependent upon goal and clause order, and programs are usually written with the effect of ordering in mind. Because a Prolog program's behaviour can be subtlely dependent upon goal and clause order, termination characteristics are often difficult for programmers to intuit. This presses the need for formal methods of proving program termination with respect to particular control strategies.

The technique proposed in this chapter uses the CCS semantics of Prolog control within termination analyses. The type of termination studied is that which occurs with finite and infinite SLD–derivations; the nontermination which arise in unification algorithms which do not use an occurs check is not addressed. The intention here is not to suggest a new automatic termination proof technique, but rather, to propose a framework which explicitly uses a semantics of control within termination proofs. By using the semantics of Prolog control, including the termination properties derived in chapter 4, a more precise semantic characterisation of Prolog program behaviour is possible. CCS is especially useful in termination applications, since finite, infinite, and looping computations are straight–forwardly modelled. The proofs illustrate how a stream–based semantic characterisation of control can be exploited when proving termination properties.

Section 5.1 reviews the termination problem in conventional programming and logic programming. A strategy for proving Prolog termination using the semantics is outlined in section 5.2. Some example termination analyses are presented in section 5.3. A discussion concludes the chapter in section 5.4.

5.1 Review

5.1.1 The termination problem

Formally verifying that a computer program will terminate is an important issue in software reliability (see (Loeckx and Sieber 1984) or (Berg *et al.* 1982) for detailed treatments). To say a program terminates is to imply that the computation is finite, and that the computation ends in a finite amount of time. The issue is immediately complicated by the Halting Problem, which states that general program termination is undecidable (Boolos and Jeffrey 1980). Most practical programs fortunately fall in a class that have tractable termination characteristics, and therefore have feasible termination proofs.

The *state* of an imperative computation is the set of variable values found in the memory, and is denoted σ. A deterministic imperative computation is characterised by the transition sequence over program states: $\sigma_i \rightarrow \sigma_{i+1} \rightarrow \cdots \sigma_k$. A program P is a partial function over the state. It is partial because some computations might not terminate for particular σ. The intention is for $P : \sigma_i \rightarrow \sigma_f$ to compute from some initial state θ_i a final state θ_f. Although various formal programming methodologies differ in philosophy and mathematical tools used, they all use similar means to prove termination, which is required for establishing program completeness. A computation is terminating if there are a finite number of state transitions during execution:

$$\sigma_i \rightarrow \sigma_2 \rightarrow \cdots \rightarrow \sigma_k$$

One mathematical means for proving termination is by using a *variant function* (Gries 1981). A variant is a function over the state σ such that, (i) for each program transition $\sigma_j \rightarrow \sigma_{j+1}$, then $f(\sigma_{j+1}) \prec f(\sigma_j)$ for some well–founded ordering relation $\prec$, and (ii) $f(\sigma_f) = 0$ for any final state σ_f. If a variant is definable for a program, then the program state complies with a well–founded ordering, which is one in which there is no infinitely decreasing sequence,

$$\cdots \prec f(\sigma_3) \prec f(\sigma_2) \prec f(\sigma_i)$$

The variant is designed so that each successive state transition decreases the variant function value towards 0. Termination is assured because the ordering is well–founded: the function cannot decrease forever. Variant definitions are problem domain specific, and their derivation is often non–trivial. Although some methodologies do not explicitly use variants, they all use the same fundamental technique of demonstrating that state transitions adhere to a well–founded ordering.

5.1.2 Logic program termination

Logic program correctness is conceptually divided into partial correctness and completeness (Hogger 1984). A program is partially correct if the values it computes are consistent with the specification. Logic program completeness is a more involved. Logic program computations can return multiple (possible infinite) solutions to queries. Because the particular set of solutions computed by a program is dependent upon the control strategy used to explore the search space, completeness requires consideration of the control. A logic program is complete with respect to some control strategy and specification when (i) it computes all the values that the specification says it should, and the search space explored is finite, and (ii) the control scheme used is exhaustive.

Logic program computation is modelled by SLD–resolution, which in turn can be denoted by an SLD–tree (see chapter 2). Logic program termination is directly denoted by SLD–trees. Given an SLD–tree for a program, each leaf node denotes a terminating computation. Empty leaf nodes denote successful computations, while non–empty leafs denote finitely failed computations. Some branches of the tree may not terminate; there is no leaf node terminating the branch. These branches denote non–terminating computations – ones which do not compute a result. Finite and infinite computations are characterised by finite and infinite SLD–trees respectively. An infinite SLD–tree may have no infinite branches, but instead have an infinite number of empty leaf nodes. This means that there are an infinite number of solutions computed[1].

The effect of the control strategy on an SLD–tree is as follows. The computation rule – the order in which goals are selected for resolution – determines the *existence*

[1] All the solutions may possibly be identical.

of infinite computation tree branches for a computation. The *independence of the computation rule* (Lloyd 1984) states that any computation selection rule will result in the same set of successful derivations. However, the computation rule determines which SLD tree is used for the computation. The search rule – the order in which clauses are selected for resolution – determines how the particular SLD tree being used will be searched, and in particular, the order in which branches within the SLD trees are searched. A search rule is *fair* if all computable solutions are produced in a finite amount of time, and is *unfair* if computable solutions may never be produced. An example of a fair search rule is breadth–first search, while an unfair search rule is the depth–first rule used by standard Prolog.

The computed results generated in an SLD tree can be characterised by *streams* of computed answer substitutions if one associates each empty goal (node) with the corresponding answer substitution used to compute it. This stream characterisation of SLD tree computations allows the morphology of the tree to be abstracted away to obtain just the computed results which are computed within it. Therefore, instead of reasoning about finite and infinite branches of a tree, streams permit a more direct modelling of the essence of logic program termination: a program terminates if the stream of computed results is finite, and a program is nonterminating if the stream is infinite.

The seminal work in (Clark 1979) (see also (Apt and van Emden 1982) (Hogger 1984)) treats logic program termination by proving the *existence* of finite successful SLD trees for particular input. Proving the termination of a program amounts to formally proving that the following holds:

$$\forall \tilde{x}[I(\tilde{x}) \rightarrow \exists \tilde{y} P(\tilde{x} \star \tilde{y})]$$

In this formula, $\tilde{x}$ and $\tilde{y}$ are tuples of logical variables, $\tilde{x} \star \tilde{y}$ is a tuple representing a permutation of $\tilde{x}$ and $\tilde{y}$, $I(\tilde{x})$ is an *input relation* asserting some criteria on the input arguments, and $P(\tilde{x} \star \tilde{y})$ is a computed relation for program P. The formula asserts that, whenever the input satisfies some condition I, then there exists a computed result P. Given that general SLD resolution is refutation complete, proving that a program terminates under this formulation reduces to showing that there exists a finite successful SLD tree for some goal query. The proof methodology implicitly uses a fair inference strategy, and specific control schemes are not formally accounted for within proofs. However, particular control schemes can be informally handled by keeping the intended control scheme in mind during the proof, and applying the verification formula and proof steps appropriately. For example, consider the declarative representation of a goal query:

$$?-\ g_1(t_1),\ g_2(t_2),\ g_3(t_3).\ \Leftrightarrow\ \forall(\ g_1(t_1)\ \wedge\ g_2(t_2)\ \wedge\ g_3(t_3))$$

Prolog's left–to–right control is handled by treating the conjunction on the right with the ordering of goals in the original query in mind. Because logical *and* ($\wedge$) is reflexive and insensitive to the order of arguments, any enforced orderings with it are informal.

(Hogger 1990) suggests a termination proof technique. Nontermination only occurs within recursive programs[2] Consequently, given a program P and a computation rule R, Hogger identifies *potentially–recursive reductions* or PRR's, which are the recursive calling structures within P using R. After finding them, he applies a well–founded ordering proof to the arguments to prove termination. A mapping μ is derived between the argument terms of a PRR and a well–founded set, which is any structure $[W, \prec]$ in which W is a set, $\prec$ is a strict partial order, and W is well–founded wrt $\prec$: for $w_i \in W$, there exists no infinite path

$$\cdots \prec w_3 \prec w_2' \prec w_1.$$

For example, given the following clauses,

$$\begin{array}{l} A(\tilde{t_a}) :-\ B(\tilde{t_1}),\ \mathbf{others_1}... \\ B(\tilde{t_b}) :-\ C(\tilde{t_2}),\ D(\tilde{t_3}),\ \mathbf{others_2}... \\ D(\tilde{t_c}) :-\ E(\tilde{t_4}),\ A(\tilde{t_5}),\ \mathbf{others_3}... \end{array}$$

To show the recursive behaviour of A given Prolog's left–to–right computation rule, one unfolds the call to B,

$$A(\tilde{t_a}) :-\ C(\tilde{t_2})\theta_1,\ D(\tilde{t_3})\theta_1,\ \mathbf{others^*\theta_1}...$$

and then unfolds the call to D,

$$A(\tilde{t_a}) :-\ C(\tilde{t_2})\theta_1,\ E(\tilde{t_4})\theta_1\theta_2,\ A(\tilde{t_5})\theta_1\theta_2,\ \mathbf{others^*\theta_1\theta_2}...$$

[2] Note that the non–termination which arises during the unification of terms when there is no occurs check is not addressed, as it is assumed that the unifications performed are sound, and therefore use an occurs check.

where θ_1 and θ_2 are the mgu's of the unfolding of B and D respectively, and **others*** are accumulated $\textbf{others}_\mathbf{i}$ goals (which are ignored here). This left–to–right unfolding reflects Prolog's computation rule. It must be shown that (i) $A(\tilde{t}_a)$ is derivable, and $\mu(A(\tilde{t}_a)) \in W$, and (ii) $\mu(A(\tilde{t}_5)\theta_1\theta_2) \prec \mu(A(\tilde{t}_a))$. Condition (i) assures that $A(\tilde{t}_a)$ represents an invocation of interest, ie. any invocation which might conceivably occur within the program. Condition (ii) states that nontermination will occur, and is similar to the conventional approaches mentioned in section 5.1.1.

Hogger's termination analysis is generalisable to any computation rule by appropriately changing the unfolding procedure. However, this generality means that there are a multitude of PRR's which may conceivably be analysed, many of which may never realistically occur within the computation. For example, in the example above, should $C(\tilde{t}_2)\theta_1$ loop, then the well–foundedness of the PRR for A is irrelevant. In addition, this technique does not explicitly handle the search rule or backtracking, so the unfolding must be duplicated for each potentially unifiable clause within the program. In summary, Hogger proposes a formal characterisation of termination for Prolog programs, while recognising the effect of the computation rule on termination. However, like (Clark 1979), precisely accounting for specific control strategies such as Prolog's standard control requires appropriate adaption of the proof procedure, which is informally undertaken.

Non–termination can be characterised using concepts from term rewriting system technology (Dershowitz 1987). The idea is that terms being rewritten must in some sense become syntactically simpler if termination is to result. Non–termination arises when terms increase in syntactic complexity. The following from (Dershowitz 1987) makes this concept more concrete. The *homeomorphic embedding relation* $\trianglerighteq$ on a set T of terms is defined recursively as follows:

$$s = f(s_1, s_2, ..., s_m) \trianglerighteq g(t_1, t_2, ..., t_n) = t$$

if either there exists some s_i $(1 \leq i \leq m)$ such that $s_i \trianglerighteq t$, or $f = g$ and $s_{i_j} \trianglerighteq t_j$ for all $j = 1, ..., n$, where $1 \leq i_1 < i_2 < \cdots < i_n \leq m$. The $\trianglerighteq$ denotes the notion of syntactic simplicity: $s \trianglerighteq t$ means that t is syntactically simpler than s, and can be obtained from s by deletion of selected term components. A derivation $t_1 \Rightarrow t_2 \Rightarrow \cdots \Rightarrow t_j \Rightarrow \cdots \Rightarrow t_k \Rightarrow \cdots$ is *self–embedding* if $t_k \trianglerighteq t_j$ for some $j < k$. A rewrite system is *self–embedding* if it allows a self–embedding derivation.

Theorem 5.1.1 *If a finite rewrite system is non–terminating, then it is self–embedding.*

Proof: *If a system R does not terminate, then there exists at least one infinite derivation $t_1 \Rightarrow t_2 \Rightarrow \cdots$. Since there can be only a finite number of function symbols appearing in the derivation (those in t_1 and in R), then $t_k \trianglerighteq t_j$ for some $j < k$.* $\square$

Logic programming can be characterised by term rewriting systems (Dershowitz and Plaisted 1985). Similarly, the CCS semantics is characterisable as a term rewriting system, where program equations and bisimilarities are considered to be rewrite rules. The nature of termination and non–termination of semantic symbolic computations parallels term rewriting derivations. Theorem 5.1.1 will therefore be referred to in the thesis when divergence is encountered.

5.1.3 Prolog program termination

Research in proving Prolog program termination has been approached from two different perspectives. One approach looks at the termination problem for general Prolog programs. Generalised Prolog termination is necessarily complicated by the undecidability of termination, and is further compounded by the complexity that arises in nondeterministic logic program computations. The other approach is to study the termination characteristics of particular classes of programs which have decidable and desirable termination characteristics. As mentioned above, logic program termination analyses like those in (Clark 1979) (Hogger 1984) (Hogger 1990) can be applied towards Prolog control, the control component being informally considered. Recent approaches have been suggested which attempt to formalise the effects of Prolog's left–to–right depth–first control on program termination. A few of these approaches are now surveyed.

(Francez *et al.* 1985) introduce two methods for proving Prolog program termination. One technique is to derive a parametric variant function over the computation tree, which complies with a well–founded ordering over the data domain. This duplicates the style of variant functions used in verifying imperative programs. This method is exceedingly complex, as the variant function must account for backtracking amongst different clauses. The other technique is compositional and treats logic program components as stream generators. This seems to be a clearer approach, but it still uses complicated arithmetic variants.

(Vasak and Potter 1986) characterise program termination using an inductive technique reminiscent of abstract interpretation. They derive sets of goals which share termination characteristics. In addition, they do not explicitly account for any search strategy, which means that features such as Prolog's cut cannot be considered.

(Bezem 1989) (Apt and Bezem 1990) analyse the termination properties of recurrent and acyclic logic programs. They use a refined well–founded ordering called a *level mapping*, which is a function mapping variable–free atoms to $\mathcal{N}$, and is denoted $|A|$. Should an atom have logical variables, the *largest* valued level mapping possible for *all* possible ground instances of the atom is used. Given some clause "$P\ :-\ A_1,\ \cdots\ ,A_n.$", the head and each goal has derived for it a level mapping. A program P is *acyclic* if the level mapping of each clause's head is greater than the level mappings of its goals (negated literals have the same mapping as their non–negated equivalents). *Recurrent* programs are acyclic programs without negation. To prove termination, one treats the level mappings of the goals in a clause as a multiset over $\mathcal{N}$, and then define for it a well–founded ordering. Some results of their research is that acyclic programs terminate for all queries, acyclic programs can compute all total recursive functions, and pragmatically speaking, well–founded multiset orderings are a convenient tool with which to prove the termination properties of logic programs. Because the termination of acyclic programs are insensitive to the inference strategy, their approach is relevant to Prolog only as far as a Prolog program can be determined to be acyclic, which is theoretically undecidable, and impractical to expect of real Prolog programs.

(Apt and Pedreschi 1990) extend the above by redefining the notion of recurrency for Prolog's left–to–right computation rule. Level mappings for atoms must reduce for only a prefix of goals in a clause (a given initial sequence of the goals), rather than all the goals as is required for recurrent programs. This prefix is defined to be the set of goals which have finite SLD trees for some model I. Then, as with recurrent and acyclic programs, termination proofs involve finding level mappings for the program statements, under the provision that only given prefixes of goals have to satisfy the mappings.

(Baudinet 1988) analyses Prolog program termination by deriving a functional semantics of Prolog (see sections 3.6 and 4.5 for details). The meaning of a program is the meaning of the set of functional equations for it. Two styles of proofs of program termination are possible. One approach uses induction over the structure of streams or predicate arguments. Alternatively, functional fixpoint proofs can be used.

Recent research has focussed on automating termination proofs for Prolog programs (Plumer 1990) (Verschaetse and Schreye 1991). Because termination is undecidable in general, the class of programs successfully handled by these techniques is necessarily restricted. Many programs can be shown to terminate by showing that the recursive calls within them subscribe to syntactic simplification orderings, which is an instance of the well–founded ordering technique for termination proofs discussed above. Such proofs can be derived automatically when the arguments of recursive calls use terms which are syntactically simpler than the header's corresponding arguments. However, more attention is required if local variables are used within clause bodies, as it is not possible to syntactically determine the ordering relations between variable arguments without first knowing what is computed by them. Plümer's methodology handles ground queries, and applies dataflow analysis and ideas from rewriting system technology to automatically derive ordering relations for such programs, under the assumption that these programs are well–moded, normalised, and do not have mutually recursive calls. Verschaetse and De Schreye generalise Plümer's approach to handle any goal satisfying given mode criteria. They convert a Prolog program into an abstract set of relations which relate the sizes of arguments with one another. Termination is proven if this abstract relation is solvable.

5.2 A technique for analysing Prolog termination

The CCS semantics of Prolog control, along with the termination properties of section 4.1, contribute to a calculus of Prolog control which can be used towards proving program termination properties. Termination analyses use the semantics of Prolog control – the CCS bisimilarities for clause sequencing, backtracking, resolution, and "cut" – along with the termination and algebraic properties of chapter 4. The stream characterisation of Prolog computation afforded by this semantics is useful in termination proofs. Note that the nontermination which can arise in unification algorithms not using the occurs check is not addressed here (discussed further in section 5.4).

All termination proofs of Prolog programs, including the ones given here, share a fundamental flavour. Termination of a (Prolog) program can be shown if the recursive calls in it are computationally smaller or simpler than the original. This notion of "simpler" is formally demonstrated by establishing that the arguments to recursive predicates adhere to a well–founded ordering. The general technique used is:

1. A given Prolog program has a corresponding CCS translation derived for it.

2. Only a recursive structure (what Hogger calls a PRR in section 5.1.2) can conceivably be nonterminating, and non–recursive structures are assumed to terminate. Using the semantics, clause and query goals are symbolically expanded to generate PRR's.

3. To prove termination of a PRR, a well–founded set $[W, \prec]$ is defined, for set W and well–founded ordering $\prec$. The recursive arguments are demonstrated to conform to this well–founded set. Ideally, structural induction over the terms of the arguments may suffice. Complex argument domains may require more sophisticated ordering relations.

4. Whenever the form of a goal or clause stream can be deduced, termination properties can be applied. Structural induction over the size of streams themselves may also be useful.

The above needs some clarification. First, PRR's are not guaranteed to be the recursive calls which will occur during a program's computation, but are possible recursions which might occur. For example, in the following clauses,

$$\begin{array}{l} P :- \ A,\ T. \\ T :- \ Q,\ P. \end{array}$$

expanding T will determine that P is a PRR. However, during program execution, Q might always fail (perhaps it is undefined), so this PRR will never execute. The actual recursions that will occur during a program's execution are difficult – if not impossible – to predetermine from the program's syntax; information about the program's declarative semantics is required.

Different well–founded orderings are possible for programs, and the particular ordering chosen is program dependent. For example, many simple programs can use an ordering relation over the size of argument terms, which means that structural induction can be used. Multiset orderings can also be convenient (Apt and Bezem 1990). In any case, to prove termination, the ordering used should be well–founded, which means that there is no infinitely decreasing ordered sequence. The well–founded ordering should also be a *strict partial order.* A strict partial order over a set S is a relation $\prec$ over $S \times S$ such that, $\forall x, y, z \in S$, the relation is:

$$\begin{array}{ll} asymmetric: & \neg(x \prec y \wedge y \prec x) \\ irreflexive: & \neg(x \prec x) \\ transitive: & (x \prec z) \;\; if \;\; (x \prec y) \wedge (y \prec z) \end{array}$$

Asymmetry and irreflexivity introduce strictness to the ordering, which prevents looping recursive structures, for example,

$$p(X) :- p(x).$$

Terminating programs are those whose recursive predicates have arguments which can be mapped to well–founded orderings. Nonterminating programs are ones in which there is no well–founded ordering possible for its recursive structures. The concepts from term rewriting system divergence outlined in section 5.1.2 will be applied when divergence is encountered in the proofs.

$$\begin{array}{l} a(\tilde{t_1}) :- b(\tilde{t_2}),\ \mathbf{others_1} \ldots \\ a(\tilde{t_3}) :- a(\tilde{t_4}),\ \mathbf{others_2} \ldots \\ \\ b(\tilde{t_5}) :- e(\tilde{t_4}),\ a(\tilde{t_6}),\ \mathbf{others_3} \ldots \\ b(\tilde{t_7}) :- a(\tilde{t_8}),\ \mathbf{others_4} \ldots \end{array}$$

$$\Updownarrow$$

$$\begin{array}{rcl} a(\tilde{X}) & \stackrel{\text{def}}{=} & a_1(\tilde{X}) \,\hat{;}\, a_2(\tilde{X}) \\ a_1(\tilde{X}) & \stackrel{\text{def}}{=} & (\tilde{X} = \tilde{t_1}) \rhd b(\tilde{t_2}) \rhd \mathbf{others_1} \cdots \\ a_2(\tilde{X}) & \stackrel{\text{def}}{=} & (\tilde{X} = \tilde{t_3}) \rhd a(\tilde{t_4}) \rhd \mathbf{others_2} \cdots \\ \\ b(\tilde{X}) & \stackrel{\text{def}}{=} & b_1(\tilde{X}) \,\hat{;}\, b_2(\tilde{X}) \\ b_1(\tilde{X}) & \stackrel{\text{def}}{=} & (\tilde{X} = \tilde{t_5}) \rhd e(\tilde{t_6}) \rhd a(\tilde{t_7}) \rhd \mathbf{others_3} \cdots \\ b_2(\tilde{X}) & \stackrel{\text{def}}{=} & (\tilde{X} = \tilde{t_8}) \rhd a(\tilde{t_9}) \rhd \mathbf{others_4} \cdots \end{array}$$

Figure 5.1: Program and CCS translation

A general example of how the semantics is used in termination proofs is now illustrated. Consider the program and translation in figure 5.1. In proving the termination of a query "$? - \ a(\tilde{t})$.", the arguments in $\tilde{t}$ can be partially or wholly instantiated. Let such a query have uninstantiated arguments $\tilde{X}$. The proof proceeds by expanding the CCS expression for the query:

$$\begin{array}{ll}
a(\tilde{X}) & : initial \ \ query \\
\approx \ a_1(\tilde{X}) \,\hat{;}\, a_2(\tilde{X}) & : \mathbf{Con} \ \ a \\
\approx \ (\tilde{X} = \tilde{t_1}) \rhd b(\tilde{t_2}) \rhd \mathbf{others_1} \cdots \,\hat{;}\, a_2(\tilde{X}) & : \mathbf{Con} \ \ a_1
\end{array}$$

At this point, **Resol** is applied, the resulting binding distributed appropriately (letting the binding be θ_1), and the expansion is continued:

$$\begin{array}{ll}
\approx \ b(\tilde{t}_2\theta_1) \rhd \mathbf{others_1}\theta_1 \cdots \,\hat{;}\, a_2(\tilde{X}) & : \mathbf{Resol} \\
\approx \ (b_1(\tilde{t}_2\theta_1) \,\hat{;}\, b_2(\tilde{t}_2\theta_1)) \rhd \mathbf{others_1}\theta_1 \cdots \,\hat{;}\, a_2(\tilde{X}) & : \mathbf{Con} \ \ b \\
\approx ... & \\
\approx \ ((e(\tilde{t}_6\theta_2) \rhd a(\tilde{t}_7\theta_2) \rhd \mathbf{others}^*) & \\
\qquad \hat{;}\, (a(\tilde{t}_9\theta_3) \rhd \mathbf{others}^*)) \rhd \mathbf{others_1}\theta_\mathbf{i} \,\hat{;}\, a_2(\tilde{X}) & : (as \ \ above)
\end{array}$$

Here, θ_2 and θ_3 are the effects of resolving b_1 and b_2 respectively, and $\mathbf{others}^*$ are accumulated goals not being analysed. Two PRR's have been found so far. The first PRR is preceded by a call to predicate e. If e contains a PRR, then its behaviour will first have to be ascertained. Otherwise, it can be considered to be terminating. In either case, theorem 4.1.2 must be used to show the behavioural interaction between e and a. Once done, a well–founded set for the arguments in $a(\tilde{t}_7\theta_2\theta)$ (θ_e being the effect of each successful inference of e) is next discovered, using a similar expansion as just done. The simplest situation is when structural induction on argument terms is possible. The other PRR represented by $a(\tilde{t}_9\theta_3)$ is treated similarly. Note that, if need be, the stream results of b_1 and b_2 may be coalesced by theorem 4.1.1. Once this first clause of a has been analysed, the second call to a_2 is similarly dealt with.

5.3 Example termination proofs

5.3.1 Non–terminating productive program

This example applies the semantics to the Prolog program in figure 5.2. The dataflow notation from section 3.5.1 is used for doing necessary bookkeeping. Consider the query "$? - \ p(U, V)$". The following behaviour can be derived:

$$
\begin{array}{l}
p(a,b). \\
p(b,c). \\
\\
q(a). \\
q(Z) :- q(Z). \\
q(b) :- q(Y),\ p(Y,W).
\end{array}
\quad \Longleftrightarrow \quad
\begin{array}{rcl}
p(X,Y) & \stackrel{\text{def}}{=} & p_1(X,Y) \,\hat{;}\, p_2(X,Y) \\
p_1(X,Y) & \stackrel{\text{def}}{=} & (X,Y) = (a,b) \\
p_2(X,Y) & \stackrel{\text{def}}{=} & (X,Y) = (b,c) \\
\\
q(Z) & \stackrel{\text{def}}{=} & q_1(Z) \,\hat{;}\, q_2(Z) \,\hat{;}\, q_3(Z) \\
q_1(Z) & \stackrel{\text{def}}{=} & Z = a \\
q_2(Z) & \stackrel{\text{def}}{=} & q(Z) \\
q_3(Z) & \stackrel{\text{def}}{=} & Z = b \rhd q(Y) \rhd p(Y,W)
\end{array}
$$

Figure 5.2: Logic program and CCS translation

$$
\begin{array}{ll}
p(U,V) & \\
\approx \Pi_S\ \theta_1 \circ (p_1(X,Y) \,\hat{;}\, p_2(X,Y)) & : \textbf{Con}\ p,\ \ \theta_1 = \{U \leftarrow X, V \leftarrow Y\}, \\
& \ \ S = \{U,V\} \\
\approx \Pi_S\ \theta_1 \circ (\overline{succ(\theta_2)}\ .\ p_2(X,Y)) & : \textbf{Resol}\ p_1,\ \ \textbf{Seq}, \\
& \ \ \theta_2 = \{X \leftarrow a, Y \leftarrow b\}, \\
\approx \Pi_S\ \theta_1 \circ (\overline{succ(\theta_2)}\ .\ \overline{succ(\theta_3)}.Done) & : \textbf{Resol}\ p_2, \\
& \ \ \theta_3 = \{X \leftarrow b, Y \leftarrow c\} \\
\approx \Pi_S\ (\overline{succ(\theta_1 \circ \theta_2)}\ .\ \overline{succ(\theta_1 \circ \theta_3)}\ .\ Done) & : apply\ \ \theta_1 \\
\approx \overline{succ(\theta_4)}\ .\ \overline{succ(\theta_5)}\ .\ Done & : apply\ \ \Pi,\ \ \theta_4 = \{U \leftarrow a, V \leftarrow b\}, \\
& \ \ \theta_5 = \{U \leftarrow b, V \leftarrow c\}
\end{array}
$$

The above shows how the results generated by clauses are sequentially composed. Alternatively, theorem 4.1.1 could be used to join the two finite sequences from the clauses. Next, consider the query "$? -\ \ q(a).$":

$$
\begin{array}{lll}
q(a) & \approx q_1(a) \,\hat{;}\, q_2(a) \,\hat{;}\, q_3(a) & : \textbf{Con}\ \ q \\
& \approx \overline{succ(\epsilon)}.(q_2(a) \,\hat{;}\, q_3(a)) & : \textbf{Resol}\ \ q_1,\ \ \textbf{Seq} \\
& \approx \overline{succ(\epsilon)}.(q(a) \,\hat{;}\, q_3(a)) & : \textbf{Con}\ \ q_2 \\
& \approx \overline{succ(\epsilon)}.(q(a) \,\hat{;}\, Done) & : \textbf{Resol}\ \ q_3 \\
& \approx \overline{succ(\epsilon)}.q(a) & : P \,\hat{;}\, Done \approx P \\
& \approx \overline{succ(\epsilon)}.(\overline{succ(\epsilon)}.q(a)) & : repeat\ \ above \\
& \approx \overline{succ(\epsilon)}^{\,\omega} &
\end{array}
$$

The bisimilar substitution in the last step holds because of the uniqueness of recursive expressions Finally, consider the query "$? -\ \ p(U,V), q(U).$":

$$
\begin{array}{ll}
p(U,V) \rhd q(U) & \\
\approx \overline{succ(\theta_1)}\ .\ \overline{succ(\theta_2)}\ .\ Done\ \rhd q(U) & :(from\ \ above)\ \ \theta_1 = \{U \leftarrow a, V \leftarrow b\}, \\
 & \ \ \theta_2 = \{U \leftarrow b, V \leftarrow c\} \\
\approx \theta_1 \circ (\overline{succ(\theta_2)}\ .\ Done\ \rhd q(a)) & : \textbf{Back} - \textbf{2},\ \ apply\ \ \theta_1\ \ to\ \ q(U) \\
\approx \theta_1 \circ (\overline{succ(\theta_2)}\ .\ Done\ \rhd \overline{succ(\epsilon)}^{\,\omega}) & :(from\ \ above)\ \ q(a) \approx \overline{succ(\epsilon)}^{\,\omega} \\
\approx \theta_1 \circ \overline{succ(\epsilon)}^{\,\omega} & : theorem\ \ 4.1.2,\ \ P \rhd \alpha^{\,\omega} \approx \alpha^{\,\omega} \\
\approx \overline{succ(\theta_1)}^{\,\omega} & : apply\ \ \theta_1
\end{array}
$$

The infinite stream from q causes the backtracked expression to be infinite also.

5.3.2 Looping program

$$
\begin{array}{lcl}
 & & p(X) \stackrel{\text{def}}{=} p_1(X)\ \hat{;}\ p_2(X) \\
 & & p_1(X) \stackrel{\text{def}}{=} a(X) \rhd b \\
p(X) :- a(X), b. & & p_2(X) \stackrel{\text{def}}{=} p(X) \\
p(X) :- p(X). & & \\
 & & a(X) \stackrel{\text{def}}{=} a_1(X)\ \hat{;}\ a_2(X) \\
a(1). & \Longleftrightarrow & a_1(X) \stackrel{\text{def}}{=} X = 1 \\
a(2) :- a(2). & & a_2(X) \stackrel{\text{def}}{=} a(X) \\
 & & \\
b. & & b \stackrel{\text{def}}{=} b_1 \\
 & & b_1 \stackrel{\text{def}}{=} True
\end{array}
$$

Figure 5.3: Looping program

This example illustrates the handling of looping within the semantics. Consider the program in figure 5.3. The call $a(2)$ is first analysed.

$$
\begin{array}{lll}
a(2) & \approx a_1(2)\ \hat{;}\ a_2(2) & : \textbf{Con}\ \ a \\
 & \approx Done\ \hat{;}\ a_2(2) & : \textbf{Resol}\ \ a_1 \\
 & \approx a_2(2) & : \textbf{Seq} \\
 & \approx a(2) & : \textbf{Con}\ \ a_2 \\
 & \approx \bot & : Theorem\ \ 5.1.1
\end{array}
$$

Next, the behaviour of the call $a(X)$ is derived.

$$
\begin{aligned}
a(X) &\approx a_1(X) \mathbin{\dot{;}} a_2(X) && : \textbf{Con}\ a \\
&\approx \overline{succ(\{X \leftarrow 1\})}\ .\ a_2(X) && : \textbf{Resol}\ a_1,\ \textbf{Seq} \\
&\approx \overline{succ(\{X \leftarrow 1\})}\ .\ a(2) && : \textbf{Resol}\ a_2 \\
&\approx \overline{succ(\{X \leftarrow 1\})}\ .\ \bot && : \mathit{from\ above}
\end{aligned}
$$

One solution is generated, and then the computation loops. Now, given the query "$?-\ p(X).$":

$$
\begin{aligned}
p(X) &\approx (a(X) \rhd b) \mathbin{\dot{;}} p_2(X) && : \textbf{Con}\ p, p_1 \\
&\approx ((\overline{succ(\theta_1)}\ .\ \bot) \rhd b) \mathbin{\dot{;}} p_2(X) && : (\mathit{from\ above})\ \ \theta_1 = \{X \leftarrow 1\} \\
&\approx\ ...\ \approx \overline{succ(\theta_1)}\ .\ (\bot \rhd b) \mathbin{\dot{;}} p_2(X) && : \mathit{expansion} \\
&\approx \overline{succ(\theta_1)}\ .\ (\bot \mathbin{\dot{;}} p_2(X)) && : \mathit{theorem}\ 4.1.5 \\
&\approx \overline{succ(\theta_1)}\ .\ \bot && : \mathit{theorem}\ 4.1.4
\end{aligned}
$$

The query $p(X)$ therefore infers $\{X \leftarrow 1\}$, and then loops.

5.3.3 Adding cuts to force termination

$p(X) :-\ a(X), !, b.$ $p(X) :-\ p(X).$ $a(1).$ $a(2) :-\ a(2).$ $b.$	$\Longleftrightarrow$	$p(X) \stackrel{\text{def}}{=} p_1(X) \mathbin{\mathring{;}} p_2(X)$ $p_1(X) \stackrel{\text{def}}{=} a(X) \rhd\!\!\mid_\ell b$ $p_2(X) \stackrel{\text{def}}{=} p(X)$ $a(X) \stackrel{\text{def}}{=} a_1(X) \mathbin{\dot{;}} a_2(X)$ $a_1(X) \stackrel{\text{def}}{=} X = 1$ $a_2(X) \stackrel{\text{def}}{=} a(X)$ $b \stackrel{\text{def}}{=} b_1$ $b_1 \stackrel{\text{def}}{=} \mathit{True}$

Figure 5.4: Program with cut

This example shows the effect of cuts on looping. Cuts are commonly used to force nonterminating programs to terminate. The program in figure 5.4 is the program of figure 5.3 with a cut inserted into it to force termination.

The manner in which the cut prunes away goals which cause looping when backtracking is first shown. Executing the query "$?-\ p(X).$":

$$
\begin{array}{lll}
p(X) & \approx p_1(X) \overset{\circ}{;} p_2(X) & : \textbf{Con } p \\
 & \approx (a(X) \rhd\!|_\ell\ b) \overset{\circ}{;} p_2(X) & : \textbf{Con } p_1 \\
 & \approx ((\overline{succ(\theta_1)}\ .\ a_2(X)) \rhd\!|_\ell\ b) \overset{\circ}{;} p_2(X) & : \textbf{Resol } a_1,\ \ \theta_1 = \epsilon \\
 & \approx b\theta_1 & : \textbf{Cut} - \textbf{1} \\
 & \approx \overline{succ(\epsilon)}.Done & : \textbf{Resol } b_1
\end{array}
$$

The infinite subtree at $a_2(X)$ is discarded by the cut, and the computation terminates.

A cut's pruning of search through a looping clause is now illustrated. Consider the goal "$? -\ p(1)$." in the looping program of figure 5.3:

$$
\begin{array}{lll}
p(1) & \approx (a(1) \rhd \underline{b) \,\hat{;}\, p_2(1)} & : \textbf{Con } p, p_1 \\
 & \approx \ \ ...\ \ \approx \overline{succ(\epsilon)}\ .\ p_2(1) & : expansion \\
 & \approx \overline{succ(\epsilon)}\ .\ p(1) & : \textbf{Resol } p_2 \\
 & \approx \overline{succ(\epsilon)}^{\,\omega} & : p(1) \approx \alpha^{\,\omega}
\end{array}
$$

Executing the terminating version of figure 5.4:

$$
\begin{array}{lll}
p(1) & \approx (a(1) \rhd\!|_\ell\ b) \overset{\circ}{;} p_2(1) & : \textbf{Con } p, p_1 \\
 & \approx b & : \textbf{Resol } a,\ \ \textbf{Cut} - \textbf{1} \\
 & \approx \overline{succ(\epsilon)}.Done & : \textbf{Resol } b
\end{array}
$$

The infinite subtree at $p_2(1)$ is discarded by the cut, and the computation terminates.

Cuts are commonly used to improve efficiency by pruning unwanted multiple solutions. Consider a clause

$$P : -\ A.$$

in which A generates a stream of solutions $\overline{succ(\theta_i)}^{\,n}.Done$. If only one solution is required, a cut can be inserted at the end of the clause

$$P : -\ A,\ !.$$

The semantics of this new behaviour is:

$$
\begin{array}{lll}
P & \approx A \rhd\!| \ True & : \textbf{Con } P \\
 & \approx \overline{succ(\theta_1)}.\overline{succ(\theta_i)}^{\,n}_{\,i=2}.Done \rhd\!| \ True & : subst\ \ A \\
 & \approx \theta_1 \circ True & : \textbf{Cut} - \textbf{1} \\
 & \approx \overline{succ(\theta_1)}\ .\ Done & : \textbf{Resol } True
\end{array}
$$

The cut takes the first solution from A as a final result.

$$
\begin{array}{l}
d(A,[A|X],X).\\
d(A,[B|X],[B|Y]) :- d(A,X,Y).\\
\\
p([\,],[\,]).\\
p(X,[A|Y]) :- d(A,X,Z),\ p(Z,Y).
\end{array}
$$

$$\Updownarrow$$

$$
\begin{array}{rcl}
d(X,Y) & \stackrel{\text{def}}{=} & d_1(Q,P,R) \mathbin{\S} d_2(Q,P,R)\\
d_1(Q,P,R) & \stackrel{\text{def}}{=} & (Q,P,R) = (A,[A|X],X)\\
d_2(Q,P,R) & \stackrel{\text{def}}{=} & (Q,P,R) = (A,[B|X],[B|Y]) \ \triangleright d(A,X,Y)\\
\\
p(Q,R) & \stackrel{\text{def}}{=} & p_1(Q,R) \mathbin{\S} p_2(Q,R)\\
p_1(Q,R) & \stackrel{\text{def}}{=} & (Q,R) = ([\,],[\,])\\
p_2(Q,R) & \stackrel{\text{def}}{=} & (Q,R) = (X,[A|Y]) \ \triangleright d(A,X,Z) \triangleright p(Z,Y)
\end{array}
$$

Figure 5.5: Terminating program

5.3.4 Terminating program

The next example is from (Baudinet 1988). It will be proven that the query "$?-$ $p([v_1,...,v_k],V)$" produces a non–looping computation generating a stream of answer substitutions of length $k!$. First, a result for predicate d is needed.

Lemma: The query "$?-\ d(U,[v_1,...,v_l],W)$." results in a non–looping computation which generates a finite sequence of l answer substitutions, each answer substitution containing the instance $W \leftarrow [v'_1,...,v'_{l-1}]$, where $[v'_1,...,v'_{l-1}]$ is a sublist of $[v_1,...,v_l]$ of length $l-1$.

Proof: By induction on the length of list $[v_1,...,v_l]$.

<u>Base case $l = 0$</u>: A list of length 0 is $[\,]$, and the query is "$?-\ d(U,[\,],W)$.".

$$
\begin{array}{lll}
d(U,[\,],W) & \approx d_1(U,[\,],W) \mathbin{\S} d_2(U,[\,],W) & : \textbf{Con}\ d\\
 & \approx \mathit{Done} & : \textbf{Resol}\ d_1,d_2,\ \ \textbf{Seq}
\end{array}
$$

The result holds trivially.

<u>Inductive case $l > 0$</u>: Consider a list $[v_1,...,v_l]$ of length l.

$$
\begin{array}{ll}
d(U,[v_1,...,v_l],W) & \\
\approx \Pi_S\ d_1(U,[v_1,...,v_l],W) \mathbin{\hat{;}} d_2(U,[v_1,...,v_l],W) & : \textbf{Con}\ \ d,\ \ S=\{U,W\} \\
\approx \overline{succ(\theta_1)}\ .\ \Pi_S\ d_2(U,[v_1,...,v_l],W) & : \textbf{Resol}\ \ d_1, \\
 & \ \ \theta_1=\{U\leftarrow v_1, W\leftarrow [v_2,...,v_l]\} \\
\approx \overline{succ(\theta_1)}\ .\ \Pi_S\ \theta_2 \circ\ d(U,[v_2,...,v_l],Y) & : \textbf{Resol}\ \ d_2,\ \ \theta_2=\{U\leftarrow A, B\leftarrow v_1, \\
 & \ \ X\leftarrow [v_2,...,v_l],\ \ W\leftarrow [B|Y]\}
\end{array}
$$

By the induction hypothesis, since $[v_2,...,v_l]$ is of length $l-1\ <\ l$, then $d(U,[v_2,...,v_l],Y)$ returns a non–looping stream of $l-1$ answer substitutions, where Y in each is a sublist of $[v_2,...,v_l]$ of length $l-2$, which is denoted L_{l-2}. Ignoring U, this result is represented as $\overline{succ(\{Y\leftarrow L_{l-2}\})}^{\,l-1}.Done$. Continuing the derivation,

$$
\begin{array}{ll}
\approx \overline{succ(\theta_1)}\ .\ \Pi_S\ \theta_2 \circ\ \overline{succ(\{Y\leftarrow L_{l-2}\})}^{\,l-1}.Done & : substituting\ \ d \\
\approx \overline{succ(\theta_1)}\ .\ \overline{succ(\theta_3)}^{\,l-1}.Done & : apply\ \ \Pi\ \ and\ \ \theta_2, \\
 & \ \ \theta_3=\{W\leftarrow [v_1|L_{l-2}], U\leftarrow A\}
\end{array}
$$

Thus, the query "$?-\ d(U,[v_1,...,v_l],W)$." generates a stream of l answer substitutions, and each substitution unifies W to a sublist of $[v_1,...,v_l]$ of length $l-1$. □

The original problem is now proven. The query "$?-\ p([v_1,...,v_l],V)$." produces a non–looping computation generating a stream of answer substitutions of length $l!$.

Proof: By induction on the length l of the list $[u_1,...,u_l]$, and using the result for predicate d from (1) above.

Base case $l=0$: A list of length 0 is [ ], and the query is "$?-\ p([\,],V)$.".

$$
\begin{array}{ll}
p([\,],V) & \\
\approx \Pi_{\{V\}}\ p_1([\,],V) \mathbin{\hat{;}} p_2([\,],V) & : \textbf{Con}\ \ p \\
\approx \overline{succ(\theta_1)}.(\Pi_{\{V\}}\ p_2([\,],V)) & : \textbf{Resol}\ \ p_1,\ \ \textbf{Seq},\ \ \theta_1=\{V\leftarrow [\,]\} \\
\approx \overline{succ(\theta_1)}.(\Pi_{\{V\}}\theta_2 \circ\ (\ d(A,[\,],Z) \rhd p(Z,Y)\ )\ & : \textbf{Resol}\ \ p_2,\ \ \theta_2=\{X\leftarrow [\,], \\
 & \ \ V\leftarrow [A|Y]\} \\
\approx \overline{succ(\theta_1)}.(\Pi_{\{V\}}\theta_2 \circ\ (\ Done \rhd p(Z,Y)\ )\ & : from\ \ (1),\ \ d(A,[\,],Z)\approx Done \\
\approx \overline{succ(\theta_1)}\ .\ Done & : \textbf{Back}-\textbf{2}
\end{array}
$$

The stream generated is of length $l! = 0! = 1$.

Inductive case $l>0$: Consider a list $[v_1,...,v_l]$ of length l.

$$
\begin{array}{ll}
p([v_1,...,v_l],V) & \\
\approx \Pi_{\{V\}}\ p_1([v_1,...,v_l],U) \mathbin{;} p_2([v_1,...,v_l],U) & : \textbf{Con}\ \ p \\
\approx \Pi_{\{V\}}\ p_2([v_1,...,v_l],U) & : \textbf{Resol}\ \ p_1,\ \ \textbf{Seq} \\
\approx \Pi_{\{V\}}\ \theta_3 \circ\ (d(A,[v_1,...,v_l],Z) \rhd p(Z,Y)) & : \textbf{Resol}\ \ p_2, \\
& \ \ \theta_3 = \{X \leftarrow [v_1,...,v_l], V \leftarrow [A|Y]\} \\
\approx \Pi_{\{V\}}\ \theta_3 \circ\ (\overline{succ(\gamma_i)}^{\,l}.Done \rhd p(Z,Y)) & : \mathit{from}\ \ (1),\ \ \mathit{substituting}\ \ d, \\
& \ \ \gamma_i = \{A \leftarrow v_j, Z \leftarrow L_{l-1}\}, \\
& \ \ L_{l-1} = [v_1,...,v_l] - v_j\ \ (1 \leq j \leq l\}
\end{array}
$$

Since each list L_{l-1} from d is of length $l-1$, then by the inductive hypothesis, $p(L_{l-1},Y)$ generates a stream of $(l-1)!$ lists, each list denoted L'_{l-1}:

$$
\begin{array}{ll}
\overline{succ(\gamma_i)}^{\,l}.Done \rhd p(Z,Y) & \\
\approx \gamma_1 \circ\ (\overline{succ(\gamma_i)}^{\,l-1}.Done \rhd p(L_{l-1},Y)) & : \textbf{Back} - \textbf{1} \\
\approx \gamma_1 \circ\ (\overline{succ(\gamma_i)}^{\,l-1}.Done \rhd \overline{succ(\delta_i)}^{\,(l-1)!}.Done) & : \mathit{induction}\ \ \mathit{hypothesis}, \\
& \ \ \delta_i = \{Y \leftarrow L'_{l-1}\} \\
\approx \gamma_1 \circ\ \overline{succ(\delta_i)}^{\,(l-1)!}.\ (\overline{succ(\gamma_i)}^{\,l-1}.Done \rhd Done) & : \textbf{Back} - \textbf{3}\ \ \mathit{repeated} \\
\approx \overline{succ(\gamma_1 \circ \delta_i)}^{\,(l-1)!}.\ (\overline{succ(\gamma_i)}^{\,l-1}.Done \rhd p(Z,Y)) & : \mathit{apply}\ \ \gamma_1,\ \ \textbf{Back} - \textbf{4}
\end{array}
$$

This occurs for each of the l answers from d, and results in a stream of size $l(l-1)! = l!$. The simplified final stream is

$$
p([v_1,...,v_l],V) \quad \approx \quad \overline{succ(\{V \leftarrow [v_j|L'_{l-1}]\})}^{\,l!}.Done
$$

This is a finite non–looping stream of size $l!$. □

5.3.5 A program using a numerical well–founded ordering

$$
\begin{array}{l}
q(X,Y) :-\ X > 100,\ Y\ is\ X - 10. \\
q(X,Y) :-\ X \leq 100,\ U\ is\ X + 11,\ q(U,Y),\ q(Y,Z).
\end{array}
$$

$$\Updownarrow$$

$$
\begin{array}{rcl}
q(X,Y) & \stackrel{\text{def}}{=} & q_1(X,Y) \mathbin{;} q_2(X,Y) \\
q_1(X,Y) & \stackrel{\text{def}}{=} & X > 100 \rhd (Z\ \ is\ \ X - 10) \\
q_2(X,Y) & \stackrel{\text{def}}{=} & X \leq 100 \rhd (U\ is\ X + 11) \rhd q(U,Y) \rhd q(Y,Z)
\end{array}
$$

Figure 5.6: 91 program

A final example is the 91 program in figure 5.6. The program computes $n = 91$ if $m \leq 100$, and $n = m - 10$ otherwise. After the termination proof of this program is given, a variation of it will be analysed.

Proof: The proof follows one in (Hogger 1990). It is assumed that queries are well–moded, so that the first argument is an integer, and the second argument is uninstantiated. The CCS semantics uses some builtin agents. The definition of $\leq$ is

$$A \leq B \ \stackrel{\text{def}}{=}\ \overline{succ(\epsilon)}.Done \ + \ Done$$

where the first term is returned if $A \leq B$ for numerical values A and B. The $>$ agent is similar. The is agent is defined as

$$N \ is \ E \ \stackrel{\text{def}}{=}\ \overline{succ(\theta)}.Done$$

where $\theta = \{N \leftarrow n\}$ if E evaluates to value n. Note that the expression E must fully evaluate; there is no facility for error handling.

The proof begins by expanding a general call to q for some integer argument n:

$$\begin{array}{ll} q(n, Y) & \\ \approx\ q_1(n, Y) \mathbin{\hat{;}} q_2(n, Y) & : \textbf{Con}\ q \\ \approx\ (n > 100 \rhd Z\ \ is\ \ n - 10) \mathbin{\hat{;}} q_2(n, Y) & : \textbf{Con}\ q_1 \end{array}$$

Note that q_1 is non–recursive and always terminates. The expansion continues on the second clause:

$$\begin{array}{l} q_2(n, Y) \\ \approx\ n \leq 100 \rhd (U\ is\ n + 11) \rhd q(U, Y) \rhd q(Y, Z) \end{array}$$

This expression has two PRR's corresponding to the two recursive calls to q. For the first recursive call, the mapping μ maps the first integer arguments of q to the ordering

$$U \prec X \quad iff \quad X < U \leq 111$$

where 111 is the minimal element in the ordering (the first argument of q is driven towards 111). Because the calls to $\leq$ and is have been solved when this call to q is reached, it can be deduced that $n \leq 100$ and $U = n + 11$, which implies that $n < U \leq 111$.

The second recursive q has its first arguments mapped to the ordering

$$U \prec X \quad iff \quad X < U \leq 101$$

where 101 is the minimal element. To apply this, it is assumed that the first three agents in q_2 terminate. The execution of the call $q(U, Y)$ means that some declarative information about q can be used, ie. $Y = 91$ if $U \leq 100$ otherwise $Y = U - 10$. The wfo follows when the effect of these three goals is combined.

Variation: The above proof does not use many useful properties of control, other than the left–to–right ordering of goals. More varied behaviour results if the order of goals in the second clause is permuted as follows,

$$q_2'(X,Y) \stackrel{\text{def}}{=} (U \ \ is \ \ X+11) \rhd q(U,Y) \rhd (X \leq 100) \rhd q(Y,Z)$$

Here, non–termination will result. Consider the call $q(n, Y)$ for some $n > 100$:

$$\begin{array}{ll}
q(n,Y) & \\
\approx \underline{q_1(n,Y)} \ \hat{;} \ q_2'(n,Y) & : \textbf{Con} \ \ q \\
\approx \overline{succ(\theta_1)}.q_2'(n,Y) & : expand \ \ q_1, \\
 & \ \ \theta_1 = \{Y \leftarrow n-10\} \\
\approx \overline{succ(\theta_1)}.(U \ \ is \ \ n+11 \rhd q(U,Y) \rhd n \leq 100 \rhd q(Y,Z)) & : \textbf{Con} \ \ q_2' \\
\approx \overline{succ(\theta_1)}.(q(m,Y) \rhd n \leq 100 \rhd q(Y,Z)) & : simplify, \ \ m = n+11 \\
\approx \overline{succ(\theta_1)}.(q(m,Y) \rhd Done) & : simplify, \ \ \textbf{Back}-\textbf{2}
\end{array}$$

The last simplification uses the fact that $n > 100$, and so $n \leq 100 \rhd q(Y,Z) \approx Done \rhd q(Y,Z) \approx Done$. The expression $q(m,Y) \rhd Done$ cannot generate any $\overline{succ(\theta)}$ actions, because the $Done$ acts as a finite failure for the backtracking. The only way this expression can terminate is if it finitely fails, and this can only occur if $q(m, Y)$ terminates. However, the call $q(m, Y)$ is a variation of $q(n, Y)$, m being larger than n. Continued expansion of q results in the nested behaviour,

$$\begin{array}{l}
\approx \ \ldots \\
\approx \overline{succ(\theta_1)}.(q(m,Y) \rhd Done) \\
\approx \overline{succ(\theta_1)}.((q(m+11,Y) \rhd Done) \rhd Done) \\
\approx \overline{succ(\theta_1)}.(((q(m+22,Y) \rhd Done) \rhd Done) \rhd Done) \\
\approx \overline{succ(\theta_1)}.((((q(m+33,Y) \rhd Done) \rhd Done) \rhd Done) \rhd Done
\end{array}$$

This looping behaviour is therefore bisimilar to $\perp$, and there is no well–founded ordering possible. The recursive calls do not permit a well–founded mapping (theorem 5.1.1). This is because there is no minimal value for the ordering, as the argument term becomes self–embedding. The behaviour of the original call is therefore $q(n,Y) \approx \overline{succ(\theta_1)}.\perp$. □

5.4 Conclusion

This chapter applied the CCS semantics of Prolog towards proving termination properties of Prolog programs. Using a well–founded orderings and some termination properties derived in chapter 4, some example termination proofs were given. The main contributions of this technique are:

- A semantic characterisation of Prolog control is formally accounted for within termination proofs. Pragmatically speaking, goal and clause order are always maintained, backtracking is explicitly represented, and dataflow can be recorded.

- A stream–based domain is ideal for representing the universe of possible behaviours of logic program computations. Finite failure, finite streams, infinite productive streams, and looping can be modelled in CCS. This permits nonterminating computations to be reasoned about, and not just those which universally terminate.

- Prolog–specific termination properties give some tactical guidance for proofs. For example, knowing how looping affects the interaction between program clauses and goals can be exploited. The CCS semantics is a general calculus which allows a variety of proof styles and techniques – a necessity given the variety of program behaviours possible.

- Extra–logical control features like "cut" are easily handled. Proofs using cuts often result in the discarding of stream components.

The scope of what is being done in this chapter needs to be reiterated. It is not the intention here to suggest a novel or automated proof technique for proving Prolog program termination. Rather, a semantic framework for reasoning about program termination given an explicit semantics of the control scheme has been proposed. A semantics of Prolog control with its stream–based characterisation of program behaviour is a useful means for reasoning about termination properties.

It can be argued that maintaining goal and clause order within termination proofs is simple to handle manually and informally within termination proofs. This might seem true for simple Prolog programs. However, formal methodologies require a rigorous formal account of *all* relevant phenomena which affect computations, and this therefore includes goal and clause order. Such a semantics will be especially necessary when more complex control schemes are being used. The semantics additionally accounts for the stream characterisation of termination, nontermination, and looping, and gives an effective semantics of the cut.

It is difficult to ignore dataflow within termination proofs. A major drawback of this is that termination proofs of even the smallest logic programs quickly become encumbered with bookkeeping regarding variable bindings and the computation state. This is symptomatic of other analyses of Prolog program termination. When the data domain is maintained "by hand", errors may occur. An alternative with the CCS semantics is to formally represent the data domain with the dataflow devices. The solution is to use a semi–automated theorem proving environment to do this record keeping.

This research supplements the generalised termination proofs used by (Clark 1979) and (Hogger 1984) (Hogger 1990) by including a semantics of the control scheme within termination analyses. The CCS semantics of control justifies much of the informal proof tactics used by Clark and Hogger which simulate the effect of Prolog's computation rule. In addition, the semantics supports a complete characterisation of Prolog behaviour, and as a result, the effects of backtracking, clause sequencing, as well as looping and nontermination, are completely modelled.

Logic program termination research has not given much attention to the nontermination which can arise with the use of unification algorithms that have no occurs check. Without the occurs check, it is possible for terms with self–embedded variables to never terminate when unified. For example, when queried with the goal "$? -\ p(X, X)$.", the predicate

$$p(X, f(X)) :- p(X, X).$$

will cause an infinite loop during unification, due to the circular binding. The semantics could be enhanced to model this phenomenon by adding the following transition rule to the two listed in section 3.5.2:

$$\textbf{Unify}_\mathbf{3} \qquad \frac{}{\tilde{t}_1 = \tilde{t}_2 \xrightarrow{\epsilon} \perp} \qquad when\ \ unification\ \ loops$$

The condition for this transition is satisfied when the particular unification algorithm used loops, which can occur when no occurs check is performed.

This chapter rationally reconstructs the functional semantics in (Baudinet 1988). Baudinet's semantics is closely related to this one, and the style of termination analyses are accordingly similar in spirit. However, the technical differences between her semantics and this one discussed in sections 3.6.2 and 4.5 result in some stylistic differences in program proofs. The major difference stems from the different mathematical tools used: Baudinet uses a functional semantics of Prolog, and this thesis uses an algebraic semantics. It is debatable as to which formal tool is more naturally suited to analysing Prolog computations; some of the following points might argue in favour of CCS. Baudinet's semantic operators, being functions over the domain of streams, directly manipulate and generate finite, infinite, or looping streams. The CCS control operators construct computed streams using the expansion theorem and other bisimilar equivalences, and streams are best deduced inductively. Baudinet requires fixpoint theory to prove that program components generate particular behaviours, especially in the case of looping computations, whereas the CCS proofs use the expansion theorem and bisimilar substitutions to accomplish the same effect. Baudinet's semantics of the cut is more awkward than the CCS treatment. However, despite these stylistic differences, the general style of termination analysis done by both formalisms is very similar. Both would benefit with the incorporation of better well–founded ordering strategies, for example, level mappings.

Recent research has suggested (semi-) automated techniques for proving the termination of classes of Prolog programs (Plumer 1990) (Verschaetse and Schreye 1991). These termination proof methodologies are considerably more ambitious than the manual termination analyses done here. Since these techniques informally assume Prolog's control scheme, incorporating a semantic account of Prolog control within them is worth investigation. Such a semantics of control is especially necessary if the cut and other more involved control strategies are to be considered. If the semantics is treated as a term rewriting system, technology for proving the termination of term rewriting (which (Plumer 1990) (Verschaetse and Schreye 1991) employ) would be applicable, with the advantage that general control schemes are accounted for.

Chapter 6

Transformations Using Cuts

This chapter applies the CCS semantics of Prolog towards verifying the correctness of a selection of source–to–source Prolog program transformations which use cuts. The notion of transformational validity used here is a strong one: two programs are valid if they generate the same streams of computed answer substitutions, which means they produce the same computed results in the same order. The example transformations in this section all use cuts. It is assumed throughout the proofs that programs always terminate. Extending the proofs to handle nonterminating or looping computations would not be difficult, and would entail the application of the ideas of chapter 5.

Section 6.1 reviews Prolog program transformation. Some preliminary notions are discussed in section 6.2. Section 6.3 uses the semantics to verify the soundness of some source–to–source transformations. A conclusion and comparison to other research concludes the chapter in section 6.4.

6.1 Review

Declarative logic programs are particularly suitable for transformation, as their logical foundation permits logically sound transformations to be performed on them without altering their declarative meanings. First–order predicate calculus is therefore a powerful tool in logic program transformation. However, when unfair control strategies such as that of Prolog are being used, logic itself is not a sufficient means with which to formalise program transformations. Instead, some sort of formal account of the inference scheme must be used to ensure the soundness of transformations.

Program transformations exist which introduce cuts to prune unwanted search. Cuts are inserted into programs to increase their computational efficiency by pruning unwanted computation subtrees, which is often necessary given the constraints of Prolog's rigid control strategy. For example, cuts are effective when a solution has been found, and continued execution of the program is known to be fruitless. Such a cut is called a *green cut*. This is opposed to *red cuts*, which will prune possible solutions from the computation, and therefore tend to be less transparent in functionality (Sterling and Shapiro 1986).

A *determinate* predicate is one in which at most one clause computes a solution when the predicate is called, and it never again succeeds when it is backtracked. When a predicate is known to be determinate, the insertion of cuts into the clauses prevents unnecessary computation. (Sawamura and Takeshima 1985) look at this issue. The determination whether predicates are determinate is generally undecidable. However, as with termination and other computational phenomena, practical programs usually have tractable characteristics of determinism. They identify some forms of determinism in logic programs, and suggest source–to–source transformations which exploit them. One of these transformations will be studied later in this chapter.

(Debray and Mishra 1988) approach the issue of Prolog transformations from a different perspective. They derive a denotational semantics of Prolog with cut. Then, using the semantics, they verify some Prolog transformations which use cuts. The basis of these transformations parallels those by Sawamura and Takeshima, as the basic premise is that determinate program components can have cuts inserted in them to make the computation more efficient, without altering the program's correctness.

6.2 Program transformation properties

A goal is determinate if it computes at most one solution. Likewise, clauses, predicates, and programs can similarly be determinate. A CCS agent P is determinate if the following holds:

$$P \quad \approx \quad \overline{succ}.Done + Done$$

Here, P has two choices of behaviour – either generate one solution and quit, or just quit.

The following theorems illustrate how determinacy is inherited throughout agent expressions. The first two theorems shows how the backtracking operator preserves determinism.

Theorem 1 *If a and b are determinate, then so is $a \rhd b$.*

Proof: *By the definition of determinacy, $a \approx \overline{succ}.Done+Done$ and $b \approx \overline{succ}.Done+Done$. Then*

$$\begin{array}{rll} a \rhd b \approx & (a[f] \mid (succ'.(b \,\mathring{;}\, NextGoal_a) + done'.Done)) \setminus F & : \textbf{Con} \ \rhd \\ \approx & ((\overline{succ}.Done + Done)[f] \mid & \\ & \quad (succ'.((\overline{succ}.Done + Done) \,\mathring{;}\, NextGoal_a) & \\ & \qquad +done'.Done)) \setminus F & : subst. \ a, \ b \\ \approx & \overline{succ}.Done + Done & : expansion \end{array}$$

Therefore $a \rhd b$ is determinate. □

Corollary 2 *If a_i $(1 \leq i \leq n)$ are determinate, so is $a_1 \rhd a_2 \rhd \cdots \rhd a_n$.*

Proof: *By induction on the number of goals, using theorem 1 and the associativity of $\rhd$.* □

The next theorem shows the effect that the sequencing operator $\mathring{;}$ has on determinate predicates.

Theorem 3 *Given $P \stackrel{\text{def}}{=} P_1 \,\mathring{;}\, \cdots \,\mathring{;}\, P_k$, and P is determinate, then either (i) all $P_i \approx Done$, or (ii) if $P_j \approx \overline{succ}.Done$ $(1 \leq j \leq k)$, then $P_i \approx Done$ $(i \neq j, \ 1 \leq i \leq k)$.*

Proof: *(i) When $P \approx Done$, then all $P_i \approx Done$. If this were not the case, then $P \not\approx Done$. This is verified by induction over the length of the list of P_i's (details omitted).*

(ii) Assume more than one $P_i \approx \overline{succ}.Done$. Let $P_j \approx \overline{succ_j}.Done$ and $P_l \approx \overline{succ_l}.Done$ $(l \neq j)$, and all other $P_i \approx Done$. Then

$$\begin{array}{rcl} P & \approx & P_1 \,\mathring{;}\, \cdots \,\mathring{;}\, P_j \,\mathring{;}\, \cdots \,\mathring{;}\, P_l \,\mathring{;}\, \cdots \,\mathring{;}\, P_k \\ & \approx & \overline{succ_j}.\overline{succ_l}.Done \end{array}$$

by expansion. But then P is no longer determinate. So only one $P_i \approx \overline{succ}.Done$. □

The explicit treatment of logic variable bindings and dataflow is not addressed whenever possible, since logic program dataflow does not greatly affect these transformations. However, it is worth remembering what is occurring implicitly with logic variables in the semantics. The notation $P\theta$ represents the application of all relevant bindings to the arguments of agent P. For example, in

$$\overline{succ}.A \rhd\!\mid B \ \approx \ B\theta$$

the binding returned by the left hand $\overline{succ}$ is applied to B, along with all other computed binding substitutions in the environment. Worth noting is the effect of executing $True$, which is defined $True \stackrel{\text{def}}{=} \overline{succ(\epsilon)}.Done$. For example, in the following

$$\overline{succ}.A \rhd\!\!| \; True \;\;\approx\;\; True \qquad (\textbf{Cut})$$

executing $True$ has the effect of returning the binding obtained from the left hand side. This is because, as described in section 3.5.1, the union of all the bindings computed by the goals in a clause are returned as a result. $True$ simply adds the empty binding ϵ to this net result.

6.3 Example transformation proofs

6.3.1 Non–folding of clauses with cuts

This first example shows how the syntactic unfolding of clauses with cuts is generally invalid. Unfolding is the process of replacing a goal in a clause by a prospective resolvent. For example, the following schemata shows an unfolding of a goal B in a predicate A:

$$\begin{array}{lcl} A :- B,\ C. & & A' :- W,\ X,\ C. \\ A :- D. & \Longrightarrow & A' :- D. \\ B :- W,\ X. & & B :- W,\ X. \end{array}$$

However, the unfolding is invalid if B contains a cut:

$$\begin{array}{lcl} A :- B,\ C. & & A'' :- W,\ !,\ X,\ C. \\ A :- D. & \not\Rightarrow & A'' :- D. \\ B :- W,\ !,\ X. & & B :- W,\ !,\ X. \end{array}$$

Proof: Let $W \approx \overline{succ}.W'$, so that the cut becomes activated. Then the following results.

$$\begin{array}{rcll} A & \approx & A_1 \mathbin{\hat{;}} A_2 & : \textbf{Con}\ \ A \\ & \approx & (B \rhd C) \mathbin{\hat{;}} A_2 & : \textbf{Con}\ \ A_1 \\ & \approx & ((W \rhd\!\!| \; X) \rhd C) \mathbin{\hat{;}} A_2 & : \textbf{Con}\ \ B \\ & \approx & ((\overline{succ}.W' \rhd\!\!| \; X) \rhd C) \mathbin{\hat{;}} A_2 & : \textit{subst.}\ \ W \\ & \approx & (X \rhd C) \mathbin{\hat{;}} A_2 & : \textbf{Cut} \end{array}$$

$$\begin{array}{rcll} A'' & \approx & A''_1 \mathbin{\overset{\circ}{;}} A''_2 & : \textbf{Con}\ \ A \\ & \approx & (W \rhd\!\!|_{\ell} \ X \rhd C) \mathbin{\overset{\circ}{;}} A'_2 & : \textbf{Con}\ \ A''_1 \\ & \approx & (\overline{succ}.W' \rhd\!\!|_{\ell} \ X \rhd C) \mathbin{\overset{\circ}{;}} A'_2 & : \textit{subst.}\ \ W \\ & \approx & X \rhd C & : \textbf{Cut} \end{array}$$

Therefore $A \not\approx A''$. The expressions are not bisimilar because the syntactic unfolding of the cut in A'' affects search in A''_2. $\square$

6.3.2 Distributing cuts through clauses

This example exploits the algebraic nature of the semantics to its fullest. The unfolding of goals before cuts is valid:

$$\begin{array}{l} P_i :- A, B, !, C. \\ A_1 :- H_1. \\ A_2 :- H_2. \end{array} \quad \Longrightarrow \quad \begin{array}{l} P'_i :- H_1, B, !, C. \\ P'_{i+1} :- H_2, B, !, C. \end{array}$$

where P'_i and P'_{i+1} are new clauses, and A, B, C, H_i represent lists of goals without cuts.

Proof: The unification of A with A_1 or A_2 is treated as an explicit call to the unification agent $=$, which is an initial goal in H_i. The CCS representation of this transformation is therefore

$$(A \rhd B) \rhd\!\!| \; C \;\approx\; ((H_1 \rhd B) \rhd\!\!|_\ell \; C) \stackrel{\circ}{;} ((H_2 \rhd B) \rhd\!\!| \; C)$$

The LHS can be rewritten:

$$\begin{array}{lll} (A \rhd B) \rhd\!\!| \; C & \approx \; ((H_1 \stackrel{\cdot}{;} H_2) \rhd B) \rhd\!\!| \; C & : \textbf{Con } A \\ & \approx \; ((H_1 \rhd B) \stackrel{\cdot}{;} (H_2 \rhd B)) \rhd\!\!| \; C & : right-distributivity \end{array}$$

The transformation is therefore

$$((H_1 \rhd B) \stackrel{\cdot}{;} (H_2 \rhd B)) \rhd\!\!| \; C \;\approx\; ((H_1 \rhd B) \rhd\!\!|_\ell \; C) \stackrel{\circ}{;} ((H_2 \rhd B) \rhd\!\!| \; C)$$

or more concisely,

$$(X \stackrel{\cdot}{;} Y) \rhd\!\!| \; C \;\approx\; (X \rhd\!\!|_\ell \; C) \stackrel{\circ}{;} (Y \rhd\!\!| \; C)$$

The behaviour of X is $X \approx Done + \overline{succ}.X'$. The bisimilarity is proven for each of these two possible behaviours of X.

(i) $X \approx Done$:

$$\begin{array}{llll} LHS: & (X \stackrel{\cdot}{;} Y) \rhd\!\!| \; C & \approx \; (Done \stackrel{\cdot}{;} Y) \rhd\!\!| \; C & : subst.\ X \\ & & \approx \; Y \rhd\!\!| \; C & : \textbf{Seq} \\ & & & \\ RHS: & (X \rhd\!\!|_\ell \; C) \stackrel{\circ}{;} (Y \rhd\!\!| \; C) & \approx \; (Done \rhd\!\!|_\ell \; C) \stackrel{\circ}{;} (Y \rhd\!\!| \; C) & : subst.\ X \\ & & \approx \; Y \rhd\!\!| \; C & : \textbf{Cut}-\textbf{2} \end{array}$$

(ii) $X \approx \overline{succ}.X'$:

$$
\begin{array}{lrcll}
LHS: & (X \fatsemi Y) \rhd\!| \; C & \approx & (\overline{succ}.X' \fatsemi Y) \rhd\!| \; C & : subst.\ X \\
 & & \approx & C\,\theta & : \mathbf{Cut-1}
\end{array}
$$

$$
\begin{array}{lrcll}
RHS: & (X \rhd\!|_\ell \; C) \fatsemi (Y \rhd\!| \; C) & \approx & (\overline{succ}.X' \rhd\!|_\ell \; C) \fatsemi (Y \rhd\!| \; C) & : subst.\ X \\
 & & \approx & C\,\theta & : \mathbf{Cut-1}
\end{array}
$$

The activation of the cut in both expressions yields the same computational result. □

6.3.3 Deleting an unnecessary cut

This example is from (Debray and Mishra 1988). The last clause of predicate P,

$$P_k :- \ g_1, \cdots, g_n.$$

where all g_i are determinate, can be replaced by

$$P'_k :- \ g_1, \cdots, g_n, !.$$

Proof: The CCS representation for P'_k is

$$P_k \stackrel{\text{def}}{=} g_1 \rhd \cdots \rhd g_n \rhd\!| \; True$$

By corollary 2, $g_1 \rhd \cdots \rhd g_n$ is determinate. Call this expression G. We need to show the equivalence between $P_k \stackrel{\text{def}}{=} G$ and $P'_k \stackrel{\text{def}}{=} G \rhd\!| \; True$. Assume that $G \approx Done + \overline{succ}.Done$.

(i) Let $G \approx Done$.

$$
\begin{array}{rcll}
P_k & \approx & Done & : \mathbf{Con}\ P_k,\ \ subst.\ G \\[1ex]
P'_k & \approx & Done \rhd\!| \; True & : \mathbf{Con}\ P'_k,\ \ subst.\ G \\
 & \approx & Done & : \mathbf{Cut-2}
\end{array}
$$

(ii) Let $G \approx \overline{succ}.Done$.

$$
\begin{array}{rcll}
P_k & \approx & \overline{succ}.Done & : \mathbf{Con}\ P_k,\ \ subst.\ G \\[1ex]
P'_k & \approx & \overline{succ}.Done \rhd\!| \; True & : \mathbf{Con}\ P'_k,\ \ subst.\ G \\
 & \approx & True\,\theta & : \mathbf{Cut-1} \\
 & \approx & \overline{succ}.Done & : \mathbf{Con}\ True
\end{array}
$$

This equivalence holds because, implicit in the dataflow, $True\,\theta$ effects the same final substitution as is returned by just G. □

6.3.4 Adding cuts to ends of determinate clauses

This example is from (Debray and Mishra 1988). It shows how determinate predicates can be made more efficient with the addition of cuts at the ends of their clause bodies. The following transformation of predicate P into predicate P' is valid

$$\begin{array}{lcl} P_1 :- g_{1,1}, \cdots, g_{1,l}. & & P'_1 :- g_{1,1}, \cdots, g_{1,l}, !. \\ P_2 :- g_{2,1}, \cdots, g_{2,m}. & \Rightarrow & P'_2 :- g_{2,1}, \cdots, g_{2,m}, !. \\ \cdots & & \cdots \\ P_k :- h_1, \cdots, h_n. & & P_k :- h_1, \cdots, h_n. \end{array}$$

if all $g_{i,j}$ and h_i are determinate.

Proof: By corollary 2, the CCS representation for P is equivalent to

$$\begin{array}{l} P_1 \stackrel{\text{def}}{=} G_1 \\ P_2 \stackrel{\text{def}}{=} G_2 \\ \cdots \\ P_{k-1} \stackrel{\text{def}}{=} G_{k-1} \\ P_k \stackrel{\text{def}}{=} H \end{array}$$

where $G_i \approx g_{i,1} \triangleright \cdots \triangleright g_{i,l} \approx \overline{succ}.Done + Done$, for all $P_i \quad (1 \leq i \leq k-1)$. The proof proceeds by induction over the number of clauses.

<u>Base case</u>: Consider 2 clauses:

$$\begin{array}{lcl} P \stackrel{\text{def}}{=} P_1 \,\hat{;}\, P_2 & & P' \stackrel{\text{def}}{=} P'_1 \,\mathring{;}\, P'_2 \\ P_1 \stackrel{\text{def}}{=} G_1 & \Longrightarrow & P'_1 \stackrel{\text{def}}{=} G_1 \triangleright\!|_\ell \; True \\ P_2 \stackrel{\text{def}}{=} G_2 & & P'_2 \stackrel{\text{def}}{=} G_2 \end{array}$$

To show $P \approx P'$, we consider the two behaviours of G_1.

(i) Let $G_1 \approx Done$.

$$\begin{array}{rll} P \approx & G_1 \,\hat{;}\, P_2 & : \textbf{Con } \; P, \;\; P_1 \\ \approx & Done \,\hat{;}\, P_2 & : subst. \;\; G_1 \\ \approx & P_2 & : \textbf{Seq} \end{array}$$

$$\begin{array}{rll} P' \approx & G_1 \triangleright\!|_\ell \; True \,\mathring{;}\, P_2 & : \textbf{Con } \; P, \;\; P_1 \\ \approx & Done \triangleright\!|_\ell \; True \,\mathring{;}\, P_2 & : subst. \;\; G_1 \\ \approx & P_2 & : \textbf{Cut} - \textbf{2} \end{array}$$

(ii) Let $G_1 \approx \overline{succ}.Done$.

$$
\begin{aligned}
P &\approx G_1 \mathbin{\hat{;}} P_2 && : \textbf{Con}\ \ P,\ \ P_1\\
&\approx \overline{succ}.Done \mathbin{\hat{;}} P_2 && : subst.\ \ G_1\\
&\approx \overline{succ}.P_2 && : expansion, \textbf{Seq}\\
&\approx \overline{succ}.Done && : Theorem\ \ 3
\end{aligned}
$$

$$
\begin{aligned}
P' &\approx G_1 \rhd\!|_{\ell}\ True \mathbin{\mathring{;}} P_2 && : \textbf{Con}\ \ P,\ \ P_1\\
&\approx \overline{succ}.Done \rhd\!|_{\ell}\ True \mathbin{\mathring{;}} P_2 && : subst.\ \ G_1\\
&\approx True && : \textbf{Cut} - \textbf{1}\\
&\approx \overline{succ}.Done && : \textbf{Con}\ \ True
\end{aligned}
$$

<u>Inductive case</u>: For $k+1$ clauses,

$$
\begin{array}{lcl}
P \stackrel{\text{def}}{=} P_1 \mathbin{\hat{;}} P_2 \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} P_k \mathbin{\hat{;}} P_{k+1} & & P' \stackrel{\text{def}}{=} P_1' \mathbin{\mathring{;}} P_2' \mathbin{\mathring{;}} \cdots \mathbin{\mathring{;}} P_k' \mathbin{\mathring{;}} P_{k+1}\\
P_1 \stackrel{\text{def}}{=} G_1 & & P_1 \stackrel{\text{def}}{=} G_1 \rhd\!|_{\ell}\ True\\
P_2 \stackrel{\text{def}}{=} G_2 & \Longrightarrow & P_2 \stackrel{\text{def}}{=} G_2 \rhd\!|_{\ell}\ True\\
\cdots & & \cdots\\
P_k \stackrel{\text{def}}{=} G_k & & P_k \stackrel{\text{def}}{=} G_k \rhd\!|_{\ell}\ True\\
P_{k+1} \stackrel{\text{def}}{=} H & & P_{k+1} \stackrel{\text{def}}{=} H
\end{array}
$$

Again, we show $P \approx P'$ for each form of G_1.

(i) Let $G_1 \approx Done$.

$$
\begin{aligned}
P &\approx G_1 \mathbin{\hat{;}} P_2 \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} P_k \mathbin{\hat{;}} P_{k+1} && : \textbf{Con}\ \ P,\ \ P_1\\
&\approx Done \mathbin{\hat{;}} P_2 \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} P_k \mathbin{\hat{;}} P_{k+1} && : subst.\ \ G_1\\
&\approx P_2 \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} P_k \mathbin{\hat{;}} P_{k+1} && : \textbf{Seq}
\end{aligned}
$$

$$
\begin{aligned}
P' &\approx (G_1 \rhd\!|_{\ell}\ True) \mathbin{\mathring{;}} P_2' \mathbin{\mathring{;}} \cdots \mathbin{\mathring{;}} P_k' \mathbin{\mathring{;}} P_{k+1} && : \textbf{Con}\ \ P,\ \ P_1\\
&\approx (Done \rhd\!|_{\ell}\ True) \mathbin{\mathring{;}} P_2' \mathbin{\mathring{;}} \cdots \mathbin{\mathring{;}} P_k' \mathbin{\mathring{;}} P_{k+1} && : subst.\ \ G_1\\
&\approx P_2' \mathbin{\mathring{;}} \cdots \mathbin{\mathring{;}} P_k' \mathbin{\mathring{;}} P_{k+1} && : \textbf{Cut} - \textbf{2}
\end{aligned}
$$

Both are equivalent by the inductive hypothesis.

(ii) Let $G_1 \approx \overline{succ}.Done$.

$$
\begin{aligned}
P &\approx G_1 \mathbin{\hat{;}} P_2 \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} P_k \mathbin{\hat{;}} P_{k+1} && : \textbf{Con}\ \ P,\ \ P_1\\
&\approx \overline{succ}.Done \mathbin{\hat{;}} P_2 \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} P_k \mathbin{\hat{;}} P_{k+1} && : subst.\ \ G_1\\
&\approx \overline{succ}.(P_2 \mathbin{\hat{;}} \cdots \mathbin{\hat{;}} P_k \mathbin{\hat{;}} P_{k+1}) && : expansion,\ \ \textbf{Seq}\\
&\approx \overline{succ}.Done && : Theorem\ \ 3
\end{aligned}
$$

$$
\begin{aligned}
P' &\approx (\overline{succ}.Done \rhd\!|_{\ell}\ True) \mathbin{\mathring{;}} P_2' \mathbin{\mathring{;}} \cdots \mathbin{\mathring{;}} P_k' \mathbin{\mathring{;}} P_{k+1} && : \textbf{Con}\ \ P,\ \ P_1,\ \ subst.\ \ G_1\\
&\approx True && : \textbf{Cut} - \textbf{1}\\
&\approx \overline{succ}.Done && : \textbf{Con}\ \ True
\end{aligned}
$$

Therefore $P \approx P'$ for any number of clauses. $\square$

6.3.5 Inserting cut within a determinate clause

This example is from (Sawamura and Takeshima 1985). The following transformation is valid,

$$\begin{array}{lcl} P_1 :- S_1. & & P_1 :- S_1. \\ \cdots & & \cdots \\ P_i :- S_{i,1},\ S_{i,2}. & \Rightarrow & P_i :- S_{i,1},\ !,\ S_{i,2}. \\ \cdots & & \cdots \\ P_n :- S_n. & & P_n :- S_n. \end{array}$$

where the S's represents lists of goals $g_1, \cdots, g_k \quad (k \geq 0)$, and one of the following conditions hold:

1. If no cut is in $S_{i,1}$, then P_i is the last clause, and all goals in $S_{i,1}$ are determinate.
2. If there are cuts in $S_{i,1}$, then every goal right of the rightmost cut is determinate.

Proof: Condition 1: The CCS representation for this transformation is:

$$\begin{array}{lcl} P_1 \stackrel{\text{def}}{=} S_1. & & P_1 \stackrel{\text{def}}{=} S_1. \\ \cdots & \Rightarrow & \cdots \\ P_i \stackrel{\text{def}}{=} S_{i,1} \rhd S_{i,2} & & P_i' \stackrel{\text{def}}{=} S_{i,1} \rhd\!\!| \ S_{i,2}. \end{array}$$

Using corollary 2, $S_{i,1}$ can be treated as a determinate agent. We therefore need to show

$$S_{i,1} \rhd S_{i,2} \ \approx \ S_{i,1} \rhd\!\!| \ S_{i,2}$$

(i) If $S_{i,1} \approx Done$, expanding both the expressions results in $Done$.

(ii) Let $S_{i,1} \approx \overline{succ}.Done$, and $S_{i,2}\,\theta \approx s.Done$ for some s where $|s| \geq 1$.

$$\begin{array}{rlll} S_{i,1} \rhd S_{i,2} \approx & \overline{succ}.Done \rhd S_{i,2} & : \mathit{subst.}\ S_{i,1} \\ \approx & Done \unrhd S_{i,2}\,\theta & : \mathbf{Back-2} \\ \approx & s.(Done \rhd S_{i,2}) & : \mathbf{Back-4}\ \mathit{repeated},\ \ \mathbf{Back-5} \\ \approx & s.Done & : \mathbf{Back-3} \end{array}$$

$$\begin{array}{rlll} S_{i,1} \rhd\!\!| \ S_{i,2} \approx & \overline{succ}.Done \rhd\!\!| \ S_{i,2} & : \mathit{subst.}\ S_{i,1} \\ \approx & S_{i,2}\,\theta & : \mathbf{Cut} \\ \approx & s.Done & : \mathit{subst.}\ S_{i,2}\,\theta \end{array}$$

Condition 2: The clause to be transformed has the form

$$P_i :- A,\ !,\ B,\ S_{i,2}.$$

where $S_{i,1} \equiv A, !, B$, and B is a sequence of determinate goals with no cuts. The CCS semantics of the transformation is

$$\begin{array}{lcl} P_i \stackrel{\text{def}}{=} A \rhd\!|_\ell \; (B \rhd S_{i,2}) & & P'_i \stackrel{\text{def}}{=} A \rhd\!|_\ell \; (B \rhd\!| \; S_{i,2}) \\ \cdots & \Rightarrow & \cdots \\ P \stackrel{\text{def}}{=} \cdots P_i \,\mathring{;}\, P_{i+1} \cdots & & P' \stackrel{\text{def}}{=} \cdots P'_i \,\mathring{;}\, P_{i+1} \cdots \end{array}$$

We therefore need to show that $P \approx P'$, ie. $P_i \approx P'_i$, or

$$B \rhd S_{i,2} \;\approx\; B \rhd\!| \; S_{i,2}$$

Let $B \approx \overline{succ}.Done + Done$, and $S_{i,2}\theta \approx s.Done$ as before ($|s| \geq 1$).

(i) When $B \approx Done$:

$$\begin{array}{llll} LHS: & B \rhd S_{i,2} \approx & Done \rhd S_{i,2} & : subst.\;\; B \\ & \approx & Done & : \textbf{Back}-\textbf{2} \end{array}$$

$$\begin{array}{llll} RHS: & B \rhd\!| \; S_{i,2} \approx & Done \rhd\!| \; S_{i,2} & : subst.\;\; B \\ & \approx & Done & : \textbf{Cut}-\textbf{2} \end{array}$$

(ii) When $B \approx \overline{succ}.Done$:

$$\begin{array}{llll} LHS: & B \rhd S_{i,2} \approx & \overline{succ}.Done \rhd S_{i,2} & : subst.\;\; B \\ & \approx & Done \unrhd S_{i,2}\theta & : \textbf{Back}-\textbf{1} \\ & \approx & Done \unrhd s.Done & : subst.\;\; S_{i,2} \\ & \approx & s.(Done \unrhd Done) & : repeated\;\; expansion \\ & \approx & s.Done & : \textbf{Back}-\textbf{4},\;\; \textbf{Back}-\textbf{2} \end{array}$$

$$\begin{array}{llll} RHS: & B \rhd\!| \; S_{i,2} \approx & \overline{succ}.Done \rhd\!| \; S_{i,2} & : subst.\;\; B \\ & \approx & S_{i,2}\theta & : \textbf{Cut}-\textbf{1} \\ & \approx & s.Done & : subst.\;\; S_{i,2} \end{array}$$

□

6.4 Conclusion

This chapter applied the CCS semantics towards verifying the soundness of a selection of source–to–source Prolog transformations. Proving validity of a transformation entails showing the bisimilarity or observational equivalence between two CCS expressions representing the transformation in question. Advantages of this approach are:

- The CCS semantics very concisely represents the essence of control needed to prove the transformations, and allows dataflow to be effectively abstracted away.
- Proofs took the form of illustrating the equivalence of generated streams, and bisimulation is an effective proof technique for this application.

The CCS semantics of cut and the bisimulation technique performed in this chapter are well–suited to verifying the source–to–source program transformations using cuts. Bisimulations simply required a case–by–case analysis for each possible behaviour (*succ* or *done*) for the goal(s) before a cut. This type of proof is easily automated. In fact, the proof of section 6.3.2 was automatically verified using the Concurrency Workbench (CWB) system (Cleaveland *et al.* 1989). Both expressions were translated into the basic CCS expressions required by the CWB. Then the observational equivalence of these two expressions were automatically verified with the CWB. This is an interesting result, and hints at the usefulness that a more sophisticated semi-automatic support environment with a logic variable domain might have for validating logic program transformations.

The techniques of this chapter are generalisable towards proving the observational equivalence of *any* pair of logic programs or components thereof (see (Maher 1988) for a discussion of logic program equivalence). Bisimulation based upon observational equivalence allows the equivalence of programs with respect to the control used to execute them to be undertaken.

Very little work in formally proving source–to–source transformations using the cut has been done. (Debray and Mishra 1988) have proved a few of the transformations of this chapter using their denotational semantics of Prolog with cut. Although denotational semantics is sufficiently powerful to prove these and other types of Prolog program properties, the complex domain spaces used are not easy to conceptualise, and are very abstract in comparison to the stream domain of the CCS semantics. In addition, their proofs require fixpoints in order to solve equivalences between functions, which pull solutions from "out of the blue". The proofs in this chapter use straight–forward bisimulations to achieve the same purpose.

An early work in Prolog program transformation is (Tamaki and Sato 1983). They use existential proofs to verify the validity of fold and unfold transformations of pure Prolog programs. They assume that a fair control strategy is to be used, and do not consider standard Prolog control nor the cut. A semantics of control would permit a more formal treatment of the effects of control on fold and unfold transformations.

(Hill *et al.* 1990) and (Hill 1991) describe pruning operators which behave better than the cut during unfolding. It would be worth modelling these operators in CCS to formally illustrate program transformations using them.

Chapter 7

Partial Evaluation

This chapter presents a semantic characterisation of partial evaluation transformations of Prolog programs. A technique for partially evaluating programs by transforming semantic representations is suggested. Because the bisimilarities of Prolog control preserve behavioural equivalence, it is possibile to model the basic control component of the partial evaluation process. In addition, The CCS semantics can be used to analyse whether or not particular partial evaluation transformations preserve completeness.

Section 7.1 reviews the concept of partial evaluation. Then the semantics is applied towards partial evaluation transformations in section 7.2. Section 7.3 discusses the reasons why some partial evaluations do not preserve behavioural equivalence. A discussion and comparison to related research is in section 7.4.

7.1 Review

An active area of research is partial evaluation (Sestoft and Zamulin 1988). The concept was pioneered by (Futamura 1971), and was further expanded upon by (Ershov 1982). Besides being a powerful program transformation technique, partial evaluation lends theoretical insight into program language theory, as well as compiler theory and development.

Partial evaluation or mixed computation is a programming language characterisation of the concept of *projection* in functional mathematics (Rogers 1988). Given a program and part of its input, partial evaluation is the process that produces a residual program which, when executed with the remaining input of the original program, computes the same result that the original program would when given the complete input. For example, let L be a programming language, p be a program, and $< d_1, ..., d_n >$ be an n–tuple of data arguments such that each $d_i \in D$ for some domain D. Then the execution of p can be represented as

$$L(p)(< d_1, ..., d_n >) \rightarrow d$$

Here, L treats p and the d_i as data, and determines a semantic interpretation of p when executed with the d_i input. The partial evaluation process is represented by the following expression

$$L(mix)(p)(< x_1, ..., x_j, y_1, ..., y_k >) \ = \ L(r)(< y_1, ..., y_k >)$$

L is the target programming language, mix is the partial evaluation procedure, p is a program to be specialised for given input data $x_1, ..., x_j$, and $y_1, ..., y_k$ is the remaining input. The result of this procedure is the residual program r which, when executed on the y_i, produces the same result as the original program:

$$L(r)(< y_1, ..., y_k >) = L(p)(< x_1, ..., x_j, y_1, ..., y_k >)$$

Program r is specialised with respect to the x_i data, and may execute considerably more efficiently than the original program. A good discussion of the above is in (Fuller and Abramsky 1988).

Similarly, in logic programming, partial evaluation is the process of evaluating a goal at compile time with some of its arguments instantiated, and deriving a new residual logic program which produces the same output as the original program, except that it runs more efficiently. (Lloyd and Shepherdson 1987) present some soundness and completeness results of the partial evaluation of pure Prolog programs. They derive conditions which assure soundness and completeness, and also discuss the effects of unsafe negation as failure. Partial evaluation is sound and complete so long as some *closedness* conditions are observed, which assure that goals do not become too specialised so that viable solutions become uncomputable. They also discuss how the control strategy affects completeness. There are three stages which may use various control schemes:

(i) Source program control: the control used to execute the original program before partial evaluation

(ii) Partial evaluation control: the control employed by the partial evaluation procedure itself onto the original program

(iii) Residual control: the control used to execute the residual program

Ideally, if a fair computation strategy is used, the particular control strategies above are of no concern, since the completeness of fair inference schemes assures that the entire solution space is searched. However, given unfair strategies, the control used at each of these stages becomes important. The observed behaviour of the pre–partial evaluated source program and the residual program should be the identical, save for the added efficiency of the residual program. The possibility of using a different control strategy during the transformation procedure at stage (ii) means that this equivalence can be sacrificed. For example, an original program may finitely fail when executed with a left–to–right computation rule, while its residual counterpart created with a partial evaluation algorithm which uses a right–to–left computation rule may loop.

There are different proposals for partial evaluation algorithms, which describe effective procedures for deriving efficient residual programs which retain computational completeness (Fuller and Abramsky 1988) (Benkerimi and Lloyd 1990). One common characteristic of these algorithms is that some basic control component of SLD resolution is used by the partial evaluation algorithm (stage (ii) above). This basic control scheme is usually enhanced with analytical devices such as loop checking, closedness maintenance, and goal suspension. The simplest approach is to apply Prolog's standard depth–first left–to–right control as a basic control strategy for partially evaluating a program. This is a sensible approach, given that programs to be partially evaluated are written with this control scheme in mind, and partially evaluating such programs using a different control scheme may result in undesirable behaviour. Algorithms are not constrained to use this control strategy, and may employ other control schemes, such as breadth–first or heuristic search. Because Prolog easily treats Prolog code as data, Prolog partial evaluators traditionally take the form of meta–interpreters written in Prolog (Fujita and Furukawa 1988) (Venken and Demoen 1988) (Fuller and Abramsky 1988).

7.2 Semantically characterising partial evaluation

A fundamental tenet of programming languages and their formal semantics is that *mechanisms which are extensionally computationally equivalent are substitutive with one another*. This concept permits more efficient programs to supersede less efficient ones. Likewise, language semantics reflect this by the notion of syntactic substitutivity of equivalent expressions, which is a prerequisite for a formal system to be equationally useful. The concept of substitutivity is taken a step further in (Hehner 1984a) (Hehner 1984b). Hehner's *predicative programming* principle posits that program code and its semantic meaning are substitutive within semantic expressions. This allows a program's concrete syntax and the formal semantic language to be intercomposed in semantic expressions. This is useful in applications such as program synthesis and verification, as it supports the step–wise refinement between program code and the semantics language within one formal context. It also assures that translations back and forth between the programming language and the semantic representation are sound.

The partial evaluation paradigm fundamentally depends upon the principle of program equivalence. When a logic program is partially evaluated, a residual program is desired which is behaviourally equivalent to the source program, except that it might be more efficient. Computational equivalence means that both the source and residual programs compute the same success and failure sets. A stricter criteria, however, requires that the original program and the residual program have identical output behaviour. This implies that the order of solutions is the same, as well as non–terminating and looping behaviour. This strict characterisation of equivalence, helpful when considering unfair control strategies, could be relaxed if fairer schemes are being used.

The substitutivity of equivalent agent expressions is fundamental to CCS theory. Using the concept of equivalence relations, if two agents are equivalent in CCS, then their corresponding expressions are substitutive within the calculus (under conditions based upon the theory of equivalence being used). CCS's observational equivalence relation can be used to describe the desired final result of partial evaluation. Let P be a Prolog program, and Q be a residual program for P. Then the desired property which should hold between them is

$$P \;\approx\; Q$$

Here, P and Q are behaviourally equivalent with respect to their observed output, and therefore share the same nontermination and looping characteristics. Their internal state transitions (or computation trees) will differ, of course, if Q is a more efficient implementation.

Rudimentary partial evaluation can be modelled in CCS by applying behaviourally equivalent transformations to the semantic expression for a program. Given the semantics of a program for a particular control strategy, partial evaluation transformations must be applied which respect the integrity of this semantic meaning. Because the behaviour of logic programs ultimately depends upon the control strategy used during execution, partial evaluation must consider the control scheme if behavioural equivalence is to be retained by the residual program. A basic method for partial evaluating programs is:

(i) Translate a logic program P into CCS semantic meaning C which models a control strategy of interest. In the examples to follow, Prolog control is assumed.

(ii) Apply bisimilar transformations t_i to C, eventually generating a residual expression C':

$$C \vdash^{t_1} \cdots \vdash^{t_k} C'$$

(iii) Find a CCS expression R' bisimilar to C', and which is translatable to logic program code.

(iv) Translate R' to a residual Prolog program R.

In the above, the bisimilar transformations applied in step (ii) can vary in complexity and sophistication; the ones performed in this chapter use definitional equivalences and control operator bisimilarities.

The following examples use the above steps to perform basic partial evaluation directly on an object program's semantic representation. Given a program P and a query Q, bisimilarities for a given control strategy are applied to the semantic expression for P and Q to effect symbolic computation. Using Hehner's notion of predicative programming, the application of these bisimilarities to the program's semantic representation represent corresponding equivalence–preserving transformations of the source program. The criteria for freezing the expansion of terms in the semantic expression is decided upon at a meta–level. Eventually a CCS expression C' results. This expression is normalised into a set of agent definitions and calls bisimilar to terms within C', which are in turn translated into equivalent Prolog code. This translation process may require predicate/agent abstraction (creating new agent/predicate definitions).

$$
\begin{array}{l}
intersect(X,Y,M) :- member(M,X),\ member(M,Y).\\
member(A,[A|B]).\\
member(A,[H|T]) :- member(A,T).
\end{array}
$$

$$\Updownarrow$$

$$
\begin{array}{rcl}
intersect(X,Y,M) & \stackrel{\text{def}}{=} & intersect_1(X,Y,M)\\
intersect_1(X,Y,M) & \stackrel{\text{def}}{=} & member(M,X) \rhd member(M,Y)\\
\\
member(X,Y) & \stackrel{\text{def}}{=} & member_1(X,Y) \fatsemi member_2(X,Y)\\
member_1(X,Y) & \stackrel{\text{def}}{=} & (X,Y) = (A,[A|B])\\
member_2(X,Y) & \stackrel{\text{def}}{=} & (X,Y) = (A,[H|T]) \rhd member(A,T)
\end{array}
$$

Figure 7.1: Intersect program

The first example is the program and CCS translation in figure 7.1. *intersect(X,Y,M)* is satisfied if the item in M is a member of both lists X and Y. We would like to specialise $intersect$ for a two element list $X \leftarrow [a,b]$. First note how $member$ generates multiple solutions through backtracking, for example,

$$member(M,[a,b]) \ \approx\ \overline{succ(\theta_1)}\ .\ \overline{succ(\theta_2)}\ .\ Done$$

where $\theta_1 = \{M \leftarrow a\}$ and $\theta_2 = \{M \leftarrow b\}$. Now consider the partial evaluation of the goal "$?-\ intersect([a,b],Y,M).$", in which both Y and M are uninstantiated:

$$
\begin{array}{ll}
intersect([a,b],Y,M) & \\
\approx\ member(M,[a,b]) \rhd member(M,Y) & : \textbf{Con}\ \ intersect\\
\approx\ (\overline{succ(\theta_1)}.\overline{succ(\theta_2)}.Done) \rhd member(M,Y) & : subst\ \ member\ \ (above)\\
\approx\ member(a,Y) \fatsemi (\overline{succ(\theta_2)}.Done \rhd member(M,Y)) & : \textbf{Back}-\textbf{5}\\
\approx\ member(a,Y) \fatsemi member(b,Y) \fatsemi (Done \rhd member(M,Y)) & : \textbf{Back}-\textbf{5}\\
\approx\ member(a,Y) \fatsemi member(b,Y) \fatsemi Done & : \textbf{Back}-\textbf{2}\\
\approx\ member(a,Y) \fatsemi member(b,Y) & : P \fatsemi Done \approx P
\end{array}
$$

The call to $intersect([a,b],Y,M)$ is bisimilar to $intersect'([a,b],Y,M)$, where

$$
\begin{array}{l}
intersect'(A,B,C) \stackrel{\text{def}}{=} intersect'_1(A,B,C) \fatsemi intersect'_2(A,B,C)\\
intersect_1(A,B,C) \stackrel{\text{def}}{=} (A,B,C) = ([a,b],Y,a) \rhd member(a,Y)\\
intersect_2(A,B,C) \stackrel{\text{def}}{=} (A,B,C) = ([a,b],Y,b) \rhd member(b,Y)
\end{array}
$$

This represents the residual program

$$
\begin{array}{l}
intersect'([a,b],Y,a) :- member(a,Y).\\
intersect'([a,b],Y,b) :- member(b,Y).
\end{array}
$$

$$
\begin{array}{l}
satisfiable(true). \\
satisfiable(X \wedge Y) :- satisfiable(X),\ satisfiable(Y). \\
satisfiable(X \vee Y) :- satisfiable(X). \\
satisfiable(X \vee Y) :- satisfiable(Y). \\
satisfiable(\neg X) :- invalid(X). \\
\\
invalid(false). \\
invalid(X \vee Y) :- invalid(X),\ invalid(Y). \\
invalid(X \wedge Y) :- invalid(X). \\
invalid(X \wedge Y) :- invalid(Y). \\
invalid(\neg X) :- satisfiable(Y).
\end{array}
$$

Figure 7.2: Satisfiability program

The second example is the program in figure 7.2 (from (Sterling and Shapiro 1986)) and its CCS translation in figure 7.3. This program determines whether a expression of propositional logic is satisfiable or invalid. It can be used to generate satisfiable expressions (using a fair control scheme), or can verify the validity of expressions given to it. Consider the partial evaluation of the query

$$?-\ satisfiable(\neg(X \vee Y) \wedge \neg\neg(Y \vee Z)).$$

What is desired is a residual program specialised for the propositional formula "$\neg(X \vee Y) \wedge \neg\neg(Y \vee Z)$". Logic variables denote unknown expressions, and are intended to be instantiated at a later time. During partial evaluation, calls that have a single expression variable argument will be frozen.

The partial evaluation follows:

$$
\begin{array}{rcl}
satisfiable(E) & \stackrel{\text{def}}{=} & satisfiable_1(E) \fatsemi satisfiable_2(E) \fatsemi satisfiable_3(E) \\
 & & \quad \fatsemi satisfiable_4(E) \fatsemi satisfiable_5(E) \\
satisfiable_1(E) & \stackrel{\text{def}}{=} & E = true \\
satisfiable_2(E) & \stackrel{\text{def}}{=} & (E = X \wedge Y) \rhd satisfiable(X) \rhd satisfiable(Y) \\
satisfiable_3(E) & \stackrel{\text{def}}{=} & (E = X \vee Y) \rhd satisfiable(X) \\
satisfiable_4(E) & \stackrel{\text{def}}{=} & (E = X \vee Y) \rhd satisfiable(Y) \\
satisfiable_5(E) & \stackrel{\text{def}}{=} & (E = \neg X) \rhd invalid(X) \\
 & & \\
invalid(E) & \stackrel{\text{def}}{=} & invalid_1(E) \fatsemi invalid_2(E) \fatsemi invalid_3(E) \\
 & & \quad \fatsemi invalid_4(E) \fatsemi invalid_5(E) \\
invalid_1(E) & \stackrel{\text{def}}{=} & E = false \\
invalid_2(E) & \stackrel{\text{def}}{=} & (E = X \vee Y) \rhd invalid(X) \rhd invalid(Y) \\
invalid_3(E) & \stackrel{\text{def}}{=} & (E = X \wedge Y) \rhd invalid(X) \\
invalid_3(E) & \stackrel{\text{def}}{=} & (E = X \wedge Y) \rhd invalid(Y) \\
invalid_3(E) & \stackrel{\text{def}}{=} & (E = \neg X) \rhd satisfiable(X)
\end{array}
$$

Figure 7.3: CCS translation

$$
\begin{array}{ll}
satisfiable(\neg(X \vee Y) \wedge \neg\neg(Y \vee Z)) & \\
\approx \; satisfiable_1(E) \fatsemi satisfiable_2(E) \fatsemi satisfiable_3(E) \fatsemi & : \textbf{Con} \;\; satisfiable \\
\qquad satisfiable_4(E) \fatsemi satisfiable_5(E) & \;\; E = \neg(X \vee Y) \\
 & \qquad \wedge \neg\neg(Y \vee Z) \\
\approx \; satisfiable_2(\neg(X \vee Y) \wedge \neg\neg(Y \vee Z)) & : satisfiable_i \approx Done \\
 & \;\; (i \neq 2), \;\; simplify \\
\approx \; satisfiable(\neg(X \vee Y)) \rhd satisfiable(\neg\neg(Y \vee Z)) & : \textbf{Resol} \;\; satisfiable_2 \\
\approx \; satisfiable_5(\neg(X \vee Y)) \rhd satisfiable_5(\neg\neg(Y \vee Z)) & : satisfiable_i \approx Done \\
 & \;\; (i \neq 5) \;\; for \;\; both \\
\approx \; invalid(X \vee Y) \rhd invalid(\neg(Y \vee Z)) & : \textbf{Resol} \;\; twice \\
\approx \; invalid_2(X \vee Y) \rhd invalid_5(\neg(Y \vee Z)) & : (a) \; invalid_i(X \vee Y) \\
 & \qquad \approx Done \;\; (i \neq 2) \\
 & \;\; (b) \; invalid_i(\neg(Y \vee Z)) \\
 & \qquad \approx Done \;\; (i \neq 5) \\
\approx \; invalid(X) \rhd invalid(Y) \rhd satisfiable(Y \vee Z) & : \textbf{Resol} \;\; invalid_{2,5} \\
\approx \; invalid(X) \rhd invalid(Y) \rhd & \\
\qquad (satisfiable_3(Y \vee Z) \fatsemi satisfiable_4(Y \vee Z)) & : satisfiable_{1,2,5} \approx Done \\
\approx \; invalid(X) \rhd invalid(Y) \rhd & \\
\qquad (satisfiable(Y) \fatsemi satisfiable(Z)) & : \textbf{Resol} \;\; satisfiable_{3,4}
\end{array}
$$

This residual expression is bisimilar to

$$\begin{array}{rcl}
satisfiable'(E) & \stackrel{\text{def}}{=} & satisfiable'_1(E) \\
satisfiable'_1(E) & \stackrel{\text{def}}{=} & (E = \neg(X \vee Y) \wedge \neg\neg(Y \vee Z)) \rhd invalid(X) \rhd invalid(Y) \\
 & & \quad \rhd new(Y, Z) \\
new(Y, Z) & \stackrel{\text{def}}{=} & new_1(Y, Z) \,\mathring{;}\, new_2(Y, Z) \\
new_1(Y, Z) & \stackrel{\text{def}}{=} & satisfiable(Y) \\
new_1(Y, Z) & \stackrel{\text{def}}{=} & satisfiable(Z)
\end{array}$$

This represents the Prolog code:

$$\begin{array}{l}
satisfiable'(\neg(X \vee Y) \wedge \neg\neg(Y \vee Z)) :- invalid(X),\ invalid(Y),\ new(Y, Z). \\
new(Y, Z) :- satisfiable(Y). \\
new(Y, Z) :- satisfiable(Z).
\end{array}$$

which is supplemented with the original program from figure 7.2. This residual will efficiently determine the validity of propositional expressions matching this specific propositional formula .

Note that the above transformations are very straight–forward, and are well–suited to automation. The basic activity consists of the application of equivalence–preserving (bisimilar) transformations to a program's semantic representation. The only halting condition is a test which inspects the form of predicate arguments. Of course, this meta–control could be further enhanced if required.

7.3 Non–equivalence preserving transformations

The previous section used the semantics of a program as a basis for applying equivalence–preserving partial evaluation transformations. The semantics can also be used to show why particular types of transformations are invalid. In particular, some partial evaluation strategies produce residual programs which behave differently from the original program. This occurs when the control strategy used by the partial evaluation process differs from the control used to execute the original program. From a semantic perspective, this happens when non–bisimilar transformations are applied to a program's semantic representation.

An example of a transformation which does not preserve program behaviour under Prolog's standard left–to–right computation rule is the partial evaluation of goals using a right–to–left computation rule. In the program,

$p(X) :- a(X),\ b(X).$
$a(1) :- a(1).$
$a(X).$
$b(3).$

Using a left–to–right computation rule, then $p(X) \approx \perp$. Partially evaluating $p(X)$ using a right–to–left computation rule results in the predicate "$p(3)$.", which is not bisimilar to the original program's behaviour.

Incongruities result when non–bisimilar transformations are applied to a program. In the above example, the right–to–left computation rule (call it "$\lhd$") used during partial evaluation is not bisimilar to the left–to–right one used by standard Prolog. Consequently, the application of right–to–left bisimilarities to the program do not preserve behavioural equivalence with respect to the program's left–to–right semantic meaning. To illustrate this, note that $A \lhd B \approx B \rhd A$. It must therefore be shown that $A \rhd B \not\approx B \rhd A$. This inequality obviously holds due to the non–commutativity of $\rhd$ (theorem 4.2.6). For example, let $A \approx \perp$ and $B \approx \mathit{Done}$. Then, using theorem 4.1.5, $A \rhd B \approx \perp \rhd B \approx \perp$. Using **Back-2**, $B \rhd A \approx \mathit{Done} \rhd A \approx \mathit{Done}$. The right–to–left partial evaluation of P will not preserve behavioural equivalence with Prolog's standard left–to–right computation rule.

Another example of an unsound transformation is when clause order is altered, which results because the clause sequencing operator $\,\hat{;}\,$ is non–commutative (theorem 4.2.6). Consider a predicate $P \stackrel{\text{def}}{=} P_1 \,\hat{;}\, P_2$. If P_1 and P_2 partially evaluate to P_1' and P_2' respectively, then the residual program $P' \stackrel{\text{def}}{=} P_2' \,\hat{;}\, P_1'$ is not bisimilar (ignoring degenerate cases).

7.4 Conclusion

This chapter presents a semantic characterisation of the partial evaluation of logic programs. Some perspectives afforded by this approach are:

- Given that the control strategy used to execute logic programs determines their computational behaviour, a formal consideration of control helps justify the correctness of program transformations. This applies to partial evaluation transformations as well. The CCS semantics of Prolog can ensure that partial evaluation transformations on a program preserve the computational equivalence between the original program and the residual program.

- Using the predicative programming principle, semantic expressions and program code are inter–translatable between one another. Even though a semantic expression may translate to many possible programs, predicative programming assures that the behaviour of such programs is equivalent and unambiguous.

- The control bisimilarities for standard Prolog control can model the basic control component of partial evaluation. Because CCS can model many types of logic program control, including sequential and concurrent control, this semantic approach could be applied towards the partial evaluation of programs using other control strategies, including those which use a combination of different control schemes.

- The CCS semantics can be used to explain why some partial evaluations do not preserve behavioural equivalence. In particular, transformations which are not bisimilar to the control strategy under which the original program is executed will result in residual programs which behave differently from the original.

- The semantics permits partial evaluation of impure Prolog features such as the cut. It could also be applied towards modelling partial evaluation of programs that use better behaved pruning operators (Hill 1991).

A quite strict notion of behavioural equivalence is used here, which specifies that programs and their residual counterparts produce identical observable output. This equivalence can be relaxed if fair control strategies are to be studied. However, strict equivalences such as this one are helpful when unfair computation strategies like that Prolog's are being used.

The CCS semantics is well–suited towards partial evaluation transformations of programs. Having a concise semantics of the control component permits straight–forward partial evaluation. Other semantics of Prolog such as (Debray and Mishra 1988) could also be applied to this same purpose, so long as the semantics models the control used by the inference mechanism. The practicality of other semantic formalisms depends upon their conceptual complexity; to this ends, the usefulness of denotational semantics is suspect.

It should be stressed that the semantic approach proposed here models one aspect of the partial evaluation process – basic partial evaluation control. The bisimilarities for standard Prolog control are used to effect symbolic computation, which is a fundamental ingredient of partial evaluation transformations. The tactics used to apply these bisimilarities, such as when to freeze goals and create new predicates, are decided at a meta–level. Some of the partial evaluation algorithms suggested in the literature are considerably more sophisticated than the simple bisimulation transformations done here. Including the program semantics within these other frameworks would permit a formal means for justifying the validity of more complex transformation strategies.

(Lloyd and Shepherdson 1987) discuss partial evaluation and its effect on the declarative and procedural semantics of pure logic programs. They illustrate how the computation and search rules affect completeness results of residual programs, and also discuss the effects of using unsafe negation as failure. The failure behaviour of programs and their residual counterparts is not necessarily preserved when different computation rules are used during partial evaluation. In addition, they show how clause order must be maintained in residual programs to preserve behavioural equivalence. Using the CCS semantics, these phenomena are shown to result when non–bisimilar transformations are applied to a program. In particular, backtracking and clause sequencing are non–commutative, and program transformations which permute goal and clause order do not preserve equivalence.

A significant open problem is to study the circumstances under which alternative non–equivalence–preserving partial evaluation control strategies produce residual program with more desirable computational characteristics than the source program. Non–equivalence preserving program transformations can be more useful than ones which strictly preserve equivalence. For example, transforming programs to remove loops is a beneficial non–equivalence preserving transformation. This is essentially the problem of control in logic programming which is concerned with the issue of tailoring control to particular programs to effect more desirable computations. Partial evaluation transformations which do not preserve observational equivalence can likewise produce residual programs whose output behaviour is more desirable than the source programs.

Prolog partial evaluators are typically implemented as meta–interpreters. The operational semantics of the target programming language is usually implicitly accounted for in the structure of the meta–interpreter, and relies on the operational semantics of the language in which the meta–interpreter itself is implemented. A consequence of this is the need to prove the correctness and completeness of these meta–interpretive program transformation systems, which can be a non–trivial task (eg. see (Fuller and Abramsky 1988)). Semantically–sound transformations using a program's formal semantics affords a more verifiably sound transformation environment. The semantic approach given here could be implemented via a term rewriting system, in which the bisimilarities and program semantics represent rewrite rules.

An open research problem is to prove properties of the partial evaluation process such as termination. Some sort of meta–semantics of the partial evaluation algorithms would be useful in this regard. Work along this line has been done by (Hascoet 1988). He uses an inference system which uses inference rules defining the dynamic semantics of programming language interpreters, along with tactical inference rules which define partial evaluation strategies.

A similar approach to this one is taken in (Ross 1989) towards partially evaluating imperative programs. An imperative program's relational semantics is represented by a logic program. This logic program is then treated as a pure Prolog program, and is partially evaluated using a Prolog meta–interpreter. Transformations of the logical semantics reflect correctness–preserving transformations of the imperative program. The residual logic program is translated back into imperative code using the predicative program principle.

Chapter 8

Other Sequential Control Schemes

The usual depth–first left–to–right control of Prolog is just one possible inference strategy. This chapter gives examples of CCS semantics for other sequential control schemes. This will show CCS's robustness for describing logic program control. The control strategies studied in this chapter are:

- A breadth–first strategy which uses a fairer search rule is looked at in section 8.1. It uses a clause sequencing operator which interleaves the solutions found by the clauses of a predicate.
- The semantics of a breadth–first computation rule is derived in section 8.2. This strategy resolves goals in a rule from left–to–right, but performs one single resolution or clause unwinding for each goal, rather than depth–first as with standard Prolog.
- A full breadth–first strategy is modelled in section 8.3 which employs both breadth–first computation and search rules.
- The semantics of goal delaying mechanisms are investigated in section 8.4.
- Some nondeterministic sequential control schemes are suggested in section 8.5.

Then section 8.6 discusses some issues regarding the use of these semantics in program analysis applications. Section 8.7 discusses the possibility of intercomposing control operators to model new control schemes. A discussion concludes the chapter in section 8.8.

8.1 An interleaving search rule

Standard Prolog uses a search rule in which clauses are searched by their textual order in the program, and the solutions from each clause are exhaustively generated before the following clause is searched. A fairer search rule is one in which the solutions from the clauses comprising a predicate contribute their solutions equally to a computation. A simple strategy for doing this is, after using a solution from some clause P_i, using the solutions from the other clauses before taking the next solution from P_i.

$$
\begin{array}{ll}
\textit{Let} & [f] \equiv [\ succ'/succ,\ done'/done\ ] \\
 & F \equiv \{\ succ',\ done'\ \}
\end{array}
$$

$$
A \mid;\mid B \stackrel{\text{def}}{=} (\ A[f] \mid done'.B + succ'.\overline{succ}.(B \mid;\mid (A - succ))) \setminus F
$$

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\textit{Bisimilarities}:} \\
\mid;\mid -\mathbf{1} & (\overline{succ}.A) \mid;\mid B & \approx & \overline{succ}.(B \mid;\mid A) \\
\mid;\mid -\mathbf{2} & Done \mid;\mid B & \approx & B
\end{array}
$$

Figure 8.1: Sequential clause interleaving operator and bisimilarities

$$
\begin{array}{rcl}
P - \alpha & \stackrel{\text{def}}{=} & (P[f] \mid f(\alpha).Echo_i^{-1}) \setminus F \\
Echo_i^{-1} & \stackrel{\text{def}}{=} & \sum_\alpha f(\alpha).\overline{f^{-1}(\alpha)}.Echo_i^{-1}
\end{array}
$$

Figure 8.2: Next state operator

An approach to defining a fairer search rule is to interleave the solutions from different clauses. Given clauses for a predicate, they are searched using their textual order in the program, as is done in Prolog. However, only one solution is obtained from a clause, after which the search carries on to the following one. This means that the textual order of clauses is still used, but that a non–exhaustive search is done on each.

To effect this, a new clause sequencing operator $\mid;\mid$ is used. It is defined in figure 8.1, along with some higher–level bisimilarities. The interleaving works by waiting for A to produce a single solution, and once it does, generate it, and then invoke B recursively sequenced by the next state of A. With respect to goal sequencing, the same backtracking mechanism is used as in standard Prolog. The bisimilarities are derived through expanding the operator definitions for the two cases (proofs omitted).

This interleaved search strategy is fair for non–looping computations – those that generate finite and infinite streams. Even though clauses are initially searched according to their textual order within the predicate, when there is no looping, all finite subtrees will be explored. But should a clause loop, this control strategy is not fair, since a clause being searched which loops will never allow other clauses to be searched:

$$\perp \mid;\mid P \;\approx\; \perp$$

To help alleviate looping , the breadth–first computation rule in section 8.2 can be used.

An important property of $\mid;\mid$ is that it is *right–associative* only, rather than fully associative:

$$A \mid;\mid B \mid;\mid C \;\equiv\; A \mid;\mid (B \mid;\mid C)$$

The following theorem shows how the composition of expressions using $\mid;\mid$ affects observed results.

Theorem 1

$$(P \mid;\mid Q) \mid;\mid R \;\not\approx\; P \mid;\mid (Q \mid;\mid R)$$

Proof: *Let $A \approx a.A'$, $B \approx b.B'$, and $C \approx c.C'$, where a, b, and c are succ actions. Expanding both sides of the above inequation result in different streams.*

$$\begin{array}{ll}
(A \mid;\mid B) \mid;\mid C & \\
\approx (a.A' \mid;\mid b.B') \mid;\mid c.C' & : subst.\ A, B, C \\
\approx a.(b.B' \mid;\mid A') \mid;\mid c.C' & : \mid;\mid -\mathbf{1} \\
\approx a.(c.C' \mid;\mid (b.B' \mid;\mid A')) & : \mid;\mid -\mathbf{1} \\
\approx a.c.((b.B' \mid;\mid A') \mid;\mid C') & : \mid;\mid -\mathbf{1}
\end{array}$$

$$\begin{array}{ll}
A \mid;\mid (B \mid;\mid C) & \\
\approx a.A' \mid;\mid (b.B' \mid;\mid c.C') & : subst.\ A, B, C \\
\approx a.((b.B' \mid;\mid c.C') \mid;\mid A') & : \mid;\mid -\mathbf{1} \\
\approx a.(b.(c.C' \mid;\mid B') \mid;\mid A') & : \mid;\mid -\mathbf{1} \\
\approx a.b.(A' \mid;\mid (c.C' \mid;\mid B')) & : \mid;\mid -\mathbf{1}
\end{array}$$

□

The "$-$" operator used by $|;|$ is defined in figure 8.2. It represents the subsequent state of an agent after it generates a particular action. Given $P \xrightarrow{\alpha} P'$, then $P - \alpha$ invokes P, waits for α (which is relabelled using f described below), and once found, echoes the rest of P. The $Echo_i^{-1}$ agent is indexed to reflect the set of α handled. The following lemmas show the behaviour of $-$.

Lemma 2 *Let f and f^{-1} be relabelling functions which, for all $\alpha \in \mathcal{L}(A)$, $f(\alpha) = \alpha'$ and $f^{-1}(\alpha') = \alpha$. Let $A[f]$ be the application of f to A, and $F = \cup f(\alpha)$ for all $\alpha \in \mathcal{L}(\mathcal{A})$. Then $(A[f] \mid Echo_i^{-1}) \setminus F \;\approx\; A$.*

Proof:

$$
\begin{array}{ll}
(A[f] \mid Echo_i^{-1}) \setminus F & \\
\approx\; (A[f] \mid \sum_\alpha \alpha'.\overline{\alpha}.Echo_i^{-1}) \setminus F & : \textbf{Con}\;\; Echo_i^{-1} \\
\approx\; A & : \textit{repeated expansion}
\end{array}
$$

$Echo_i^{-1}$ simply undoes the relabelling done by $[f]$. □

Lemma 3 $\quad \overline{\alpha}.A - \alpha \;\approx\; A \qquad (\alpha \in \mathcal{L}(\mathcal{A}))$

Proof:

$$
\begin{array}{ll}
\overline{\alpha}.A - \alpha & \\
((\overline{\alpha}.A)\;[f] \mid f(\alpha).Echo_i^{-1}) \setminus F & : \textbf{Con}\;\; - \\
\approx\; (\overline{f(\alpha)}.A\;[f] \mid f(\alpha).Echo_i^{-1}) \setminus F & : \textit{apply}\;\; [f] \\
\approx\; (A\;[f] \mid Echo_i^{-1}) \setminus F & : \textit{expansion} \\
\approx\; A & : \textit{Lemma}\;\; 2
\end{array}
$$

□

$$
\begin{array}{lcl}
 & & p(X) \stackrel{\text{def}}{=} p_1(X)\ |;|\ p_2(X) \\
p(X) :- q(X). & & p_1(X) \stackrel{\text{def}}{=} q(X) \\
p(c). & & p_2(X) \stackrel{\text{def}}{=} X = c \\
 & \Longleftrightarrow & \\
q(a). & & q(X) \stackrel{\text{def}}{=} q_1(X)\ |;|\ q_2(X) \\
q(b). & & q_1(X) \stackrel{\text{def}}{=} X = a \\
 & & q_2(X) \stackrel{\text{def}}{=} X = b
\end{array}
$$

Figure 8.3: Program and CCS translation

$$
\begin{array}{ll}
p(X) & \\
\approx\ p_1(X)\ |;|\ p_2(X) & :\textbf{Con}\ p \\
\approx\ (q_1(X)\ |;|\ q_2(X))\ |;|\ p_2(X) & :\textbf{Con}\ p_1, q \\
\approx\ (\overline{succ(\theta_1)}\ .\ q_2(X))\ |;|\ p_2(X) & :\textbf{Resol}\ q_1,\ \ |;|\ -\mathbf{2},\ \theta_1 = \{X \leftarrow a\} \\
\approx\ \overline{succ(\theta_1)}\ .\ (p_2(X)\ |;|\ q_2(X)) & :\ |;|\ -\mathbf{1} \\
\approx\ \overline{succ(\theta_1)}\ .\ \overline{succ(\theta_2)}\ .\ q_2(X) & :\textbf{Resol}\ P_2,\ \ |;|\ -\mathbf{2},\ \theta_2 = \{X \leftarrow c\} \\
\approx\ \overline{succ(\theta_1)}\ .\ \overline{succ(\theta_2)}\ .\ \overline{succ(\theta_3)}\ .\ Done & :\textbf{Resol}\ q_2,\ \ \theta_3 = \{X \leftarrow b\} \\
\square &
\end{array}
$$

Figure 8.4: Symbolic computation of "$?-\ p(X).$"

Consider the logic program in figure 8.3 with its CCS translation. An example symbolic computation is given in figure 8.4, which uses the bisimilarities of figure 8.1. The search rule computed the ordered stream a, c, and b for X. Prolog's standard search rule computes a, b, and c.

8.2 Breadth–first computation rule

Prolog's computation rule selects goals from left–to–right, by using the order they are found within a clause or query. When combined with Prolog's depth–first search, a goal G_i will have to completely infer a result before the following goal G_{i+1} is resolved. If G_i loops, then G_{i+1} will never execute (theorem 4.1.5).

A breadth–first computation rule selects goals for resolution in a breadth–first fashion, so that the depth of resolution down the computation tree for all the goals in a query is the same. This contrasts with the depth–first search done in standard Prolog. Consider the query

$$?-\ G_1, G_2, ..., G_k.$$

A goal G_i is initially selected (let it be G_1) and resolved with a clause "$C :- B_1, ..., B_n.$" using the unifying substitution θ:

$$?-\ (B_1, ..., B_n, G_2, ..., G_k.)\theta$$

The computation rule next selects a goal from amongst the remaining goals $G_2, ..., G_n$ (say G_2), and resolves it one level deep:

$$?-\ (B_1, ..., B_n, C_1, ..., C_m, G_3, ..., G_k.)\theta\theta'$$

The rationale is that each goal in the query should have its resolvent computed one level deeper (its immediate children), and that the differences in resolution levels of the goals should never differ by more than one level. A goal which resolves with a fact falls out of the query. This procedure repeats for the remaining goals in $G_3, ..., G_k$, and it continues until an empty goal is obtained. Backtracking is similar to that done in Prolog: computed results from left–hand goals are exhaustively applied to right–hand goals, and new left–hand results are only applied after right–hand goals have completed execution.

Four basic actions are used by agents:

$$\mathcal{L}(P) = \{\ succ(\theta),\ res(\theta),\ done,\ doneclause\ \}$$

The *succ* and *done* actions have the same meaning as before. The action $res(\theta)$ represents the resolution of a goal with a clause using a unifying substitution θ. The θ argument will usually be omitted. res differs from *succ* in that it represents a successful single resolution step, whereas *succ* represents a completed inference or computation of a goal. *doneclause* is used to observe when a clause has exhaustively generated all its solutions. It will be generated by a clause sequencing operator to be discussed shortly. The above actions will be superscripted when necessary, for example, $succ'$ and $doneclause''$.

$$\begin{aligned}
\mathcal{M}[\![\ [P_1, P_2, ..., P_k]\ ]\!] &= \quad P \stackrel{\text{def}}{=} P_1 \ \ \dagger\ \ P_2 \ \ \dagger\ \cdots\ \dagger\ \ P_k \\
\mathcal{M}[\![\ P_i(\tilde{t}).\ ]\!] &= \quad P_i(\tilde{x}) \stackrel{\text{def}}{=} \tilde{x} = \tilde{t} \\
\mathcal{M}[\![\ P_i(\tilde{t})\ \ :-\ G_1,\ G_2,\ ...\ G_n.\ ]\!] &= \quad P_i(\tilde{x}) \stackrel{\text{def}}{=} (\tilde{x} = \tilde{t}) \trianglerighteq\ G_1 \rhd^{+} G_2 \rhd^{+} \cdots \rhd^{+} G_n \\
\mathcal{M}[\![\ ?-\ \ P,\ Q.\ ]\!] &= \quad (P \rhd^{+} Q) \setminus \{doneclause,\ res,\ doneclause''\}
\end{aligned}$$

Figure 8.5: Example translations

Some example translations using the breadth–first computation rule semantics are in figure 8.5. The translations are similar to those of Prolog programs used earlier, except that (i) a $\trianglerighteq$ operator separates the unification goal from the rest of the goals in the clause, (ii) clause sequencing uses a "loud" sequencing operator, which echoes $\overline{doneclause}$ at the end of every clause's execution, and (iii) query expressions have $doneclause$ and res restricted.

$$
\begin{aligned}
Let\ [g] &\equiv [\, succ'/succ,\ res'/res,\ done'/done,\\
&\qquad doneclause'/doneclause,\ doneclause'''/doneclause''\,]\\
[d] &\equiv [\, doneclause''/doneclause\,]\\
[d^{-1}] &\equiv [\, doneclause/doneclause''\,]\\
G &\equiv \{\, succ',\ res',\ done',\ doneclause',\ doneclause'''\,\}
\end{aligned}
$$

$$
\begin{aligned}
A \rhd^{+} B &\stackrel{\text{def}}{=} (\ A[g] \mid Gloop_i\ ) \setminus G\\
Gloop_i &\stackrel{\text{def}}{=} \ done'.Done\\
&\quad + succ'.B\ \dagger\ (Nextclause(A - succ) \rhd^{+} B)\\
&\quad + res'.(B \rhd^{+} (Clause(A - res)[d^{-1}]))\\
&\qquad \dagger\ (Nextclause(A - res) \rhd^{+} B)
\end{aligned}
$$

$$
\begin{aligned}
Clause(P) &\stackrel{\text{def}}{=} (\ P[g] \mid Cloop\ ) \setminus G\\
Cloop &\stackrel{\text{def}}{=} succ'.\overline{succ}.Cloop + res'.\overline{res}.Cloop + doneclause'''.\overline{doneclause''}\\
&\quad + doneclause'.Done + done'.Done
\end{aligned}
$$

$$Nextclause(P) \stackrel{\text{def}}{=} (\ P[g] \mid done'.Done\ +\ doneclause'.Echo_i^{-1}\ ) \setminus G$$

$$P\ \dagger\ Q \stackrel{\text{def}}{=} (\ P[b/done][d] \mid b.\overline{doneclause}.Q\ ) \setminus b \qquad (b \notin \mathcal{L}(P) \cup \mathcal{L}(Q))$$

$$P \rhd\ Q \stackrel{\text{def}}{=} (\ P[g] \mid succ'.\overline{res}.Q + done'.Done\ ) \setminus G$$

Figure 8.6: Breadth–first computation rule operators

The operators are defined in figure 8.6. The "$-$" operator used is from section 8.1. The $\rhd^{+}$ operator is a breadth–first backtracking operator. In $A \rhd^{+} B$, when A produces $\overline{succ}$, then B is executed. When $A \stackrel{\overline{res}}{\rightarrow} A'$, then $B \rhd^{+} A'$ is recursively invoked. The state A' is determined using agent $Clause$, which gives the behaviour of the current clause of A (discussed below). After this recursive call, the following clauses are processed recursively. The $Gloop_i$ mechanism is not a loop, but is an agent used for clarity's sake. It is uniquely named via the i index.

Theorem 4 *The $\rhd^{+}$ operator is associative.*

$$(A \rhd^{+} B) \rhd^{+} C\ \approx\ A \rhd^{+} (B \rhd^{+} C)$$

Proof: *The proof is similar to the theorem 4.2.5 for the associativity of $\rhd$, and is omitted.* $\square$

The definition for $\dagger$ is similar to normal clause sequencing, except that $\overline{doneclause}$ is generated at the termination of clauses. This action is observed by *Clause* and $\rhd^{+}$ to distinguish the streams produced by different clauses. It is also right–associative only:

$$P \dagger Q \dagger R \equiv P \dagger (Q \dagger R)$$

The $|\!\!\rhd$ operator models one resolution step between a goal and a clause. In $P \,|\!\!\rhd\, Q$, agent P is always the first goal of the clause, being the call to the unification agent $(=)$. Instead of returning $\overline{succ}$, or executing the rest of the clause body, $|\!\!\rhd$ returns $\overline{res}$. The res action is used by $\rhd^{+}$ to sequence goal resolutions.

$$\begin{array}{lrcll}
\dagger-\mathbf{1}: & (\alpha.A) \dagger B & \approx & \alpha.(A \dagger B) & (\alpha \notin \{done, doneclause\}) \\
\dagger-\mathbf{2}: & Done \dagger B & \approx & \overline{doneclause}.B & \\
 & & & & \\
\rhd^{+}-\mathbf{1}: & Done \rhd^{+} B & \approx & Done & \\
\rhd^{+}-\mathbf{2}: & (\overline{succ}.Done \dagger A) \rhd^{+} B & \approx & B \dagger (A \rhd^{+} B) & \\
\rhd^{+}-\mathbf{3}: & (\overline{res}.A \dagger A') \rhd^{+} B & \approx & (B \rhd^{+} A) \dagger (A' \rhd^{+} B) & \\
\rhd^{+}-\mathbf{4}: & \overline{succ}.Done \rhd^{+} B & \approx & B \dagger Done & \\
\rhd^{+}-\mathbf{5}: & \overline{res}.A \rhd^{+} B & \approx & (B \rhd^{+} A) \dagger Done & \\
 & & & & \\
|\!\!\rhd-\mathbf{1}: & \overline{succ}.A \,|\!\!\rhd\, B & \approx & \overline{res}.B & \\
|\!\!\rhd-\mathbf{2}: & Done \,|\!\!\rhd\, B & \approx & Done &
\end{array}$$

Figure 8.7: Breadth–first computation rule bisimilarities

The higher–level bisimilarities in figure 8.7 are now derived from the operator definitions in figure 8.6. The bisimilarities for $\dagger$ and $|\!\!\rhd$ are first derived.

Theorem 5

$$\begin{array}{lrcll}
\dagger-\mathbf{1}: & (\alpha.A) \dagger B & \approx & \alpha.(A \dagger B) & (\alpha \notin \{done, doneclause\}) \\
\dagger-\mathbf{2}: & Done \dagger B & \approx & \overline{doneclause}.B &
\end{array}$$

Proof:

$$\begin{array}{lll}
\dagger-\mathbf{1}: & (\alpha.A) \dagger B & \\
 & \approx (\alpha.A[b/done][d] \mid b.\overline{doneclause}.B) \setminus b & : \mathbf{Con} \ \dagger \\
 & \approx \alpha.(A[b/done][d] \mid b.\overline{doneclause}.B) \setminus b & : expansion \\
 & \approx \alpha.(A \dagger B) & : defn \ \dagger
\end{array}$$

$$\begin{aligned}
\fatsemi\mathbf{-2}: \quad & Done \fatsemi B & \\
\approx\ & (\overline{done}.\mathbf{0}[b/done] \mid b.\overline{doneclause}.B) \setminus b & : \mathbf{Con} \quad \fatsemi \\
\approx\ & (\overline{b}.\mathbf{0} \mid b.\overline{doneclause}.B) \setminus b & : \mathbf{Rel} \\
\approx\ & (\mathbf{0} \mid \overline{doneclause}.B) \setminus b & : expansion \\
\approx\ & \overline{doneclause}.B & : simplify
\end{aligned}$$

□

Theorem 6

$$\begin{aligned}
\rhd\mathbf{-1}: \quad \overline{succ}.A \rhd B &\approx \overline{res}.B \\
\rhd\mathbf{-2}: \quad Done \rhd B &\approx Done
\end{aligned}$$

Proof:

$$\begin{aligned}
\rhd\mathbf{-1}: \quad & \overline{succ}.A \rhd B & \\
\approx\ & (\overline{succ}.A[f] \mid succ'.\overline{res}.B + done'.Done) \setminus F & : \mathbf{Con} \quad \rhd \\
\approx\ & (A[f] \mid \overline{res}.B) \setminus F & : expansion \\
\approx\ & (A[f] \setminus F) \mid \overline{res}.B & : simplify \\
\approx\ & \mathbf{0} \mid \overline{res}.B & : simplify \\
\approx\ & \overline{res}.B & : simplify
\end{aligned}$$

$\rhd\mathbf{-2}$: *Similar to above.* □

The manner in which individual clause streams are observed and used is now shown. The clause sequencing operator $\fatsemi$ generates a $\overline{doneclause}$ action whenever the left–hand–side clause terminates and the right–hand–side clause is to commence execution. This action is used to distinguish the stream generated by a clause. However, in order for this *doneclause* action to be effective, any *doneclause* actions internal to the stream must be silenced. This is done using the $[d]$ relabelling within $\fatsemi$, which mutes internal *doneclause* actions by relabelling them to $doneclause''$. The $Nextclause$ and $Clause$ agents described below identify particular clause streams.

Lemma 7

$$\begin{aligned}
(i) \quad & Clause(A \fatsemi B) \approx A[d] \\
(ii) \quad & Clause(A) \approx A
\end{aligned}$$

Proof: *Case(i): Let $A \approx s.Done$, and t be the stream $s[d]$ ($\fatsemi$ relabells doneclause to $doneclause''$ via $[d]$). Expanding Clause:*

$$
\begin{array}{lll}
 & Clause(A \fatsemi B) & \\
\approx & (A \fatsemi B)[g] \mid Cloop) \setminus G & : \textbf{Con } Clause \\
\approx & (s.Done \fatsemi B)[g] \mid Cloop) \setminus G & : subst.\ A \\
\approx & t.((\underline{Done \fatsemi B})[g] \mid Cloop) \setminus G & : expansion \\
\approx & t.((\overline{doneclause}.B)[g] \mid Cloop) \setminus G & : \fatsemi -\mathbf{2} \\
\approx & t.(B[g] \mid Done) \setminus G & : expansion \\
\approx & t.(B[g]\setminus G \mid Done) & : simplify \\
\approx & t.(\mathbf{0} \mid Done) & : simplify \\
\approx & t.Done & : simplify \\
\approx & A[d] & : simplify
\end{array}
$$

This holds for non-terminating A in the usual way.

<u>Case (ii)</u>: When there is no $\fatsemi$ in expression, the proof is similar to case (i). Clause uses the action done in a manner similar to doneclause. □

Lemma 8

$$
\begin{array}{l}
(i)\quad Nextclause(A \fatsemi B) \approx \begin{cases} B & : if\ \ A\ \ is\ \ terminating \\ \perp & : if\ \ A \approx \alpha^{\omega}\ \ or\ \ A \approx \perp \end{cases} \\
(ii)\quad Nextclause(A) \approx Done
\end{array}
$$

Proof: *The same argument as in lemma 7 is used, except that all the output from A is ignored until $\overline{doneclause}$ is reached, upon which the rest of the stream is echoed using $Echo_i^{-1}$ from section 8.1.* □

Bisimilarities for $\rhd^{+}$ are now defined.

Theorem 9

$$
\begin{array}{lrcl}
\rhd^{+}\!-\mathbf{1}: & Done \rhd^{+} B & \approx & Done \\
\rhd^{+}\!-\mathbf{2}: & (\overline{succ}.Done \fatsemi A) \rhd^{+} B & \approx & B \fatsemi (A \rhd^{+} B) \\
\rhd^{+}\!-\mathbf{3}: & (\overline{res}.A \fatsemi A') \rhd^{+} B & \approx & (B \rhd^{+} A) \fatsemi (A' \rhd^{+} B) \\
\rhd^{+}\!-\mathbf{4}: & \overline{succ}.Done \rhd^{+} B & \approx & B \fatsemi Done \\
\rhd^{+}\!-\mathbf{5}: & \overline{res}.A \rhd^{+} B & \approx & (B \rhd^{+} A) \fatsemi Done
\end{array}
$$

Proof:

(i) $\rhd^{+}-\mathbf{1}$:*using expansion (omitted)*

(ii) $\rhd^{+}-\mathbf{2}$:

$$
\begin{array}{lll}
 & (\overline{succ}.Done \fatsemi A) \rhd^{+} B & \\
\approx & ((\overline{succ}.Done \fatsemi A)[g] \mid Gloop_i) \setminus G & : \textbf{Con } \rhd^{+} \\
\approx & ((Done \fatsemi A)[g] \mid B \fatsemi & \\
 & \quad (Nextclause(succ.Done \fatsemi A - succ) \rhd^{+} B)) \setminus G & : expansion \\
\approx & ((Done \fatsemi A)[g] \mid B \fatsemi (A \rhd^{+} B)) \setminus G & : Lemma\ 8 \\
\approx & B \fatsemi (A \rhd^{+} B) & : simplify
\end{array}
$$

The last simplification occurs because the restriction of G makes the left-hand term *drop out (since it is equivalent to* $\mathbf{0}$*), and it has no effect on the right–hand term.*

(iii) $\rhd^+ - \mathbf{3}$:

$$\begin{array}{lll} & (\overline{res}.A \ \dagger \ A') \rhd^+ B & \\ \approx & ((\overline{res}.A \ \dagger \ A')[g] \mid Gloop_i) \setminus G & : \mathbf{Con} \ \rhd^+ \\ \approx & ((A \ \dagger \ A')[g] \mid (B \rhd^+ (Clause(\overline{res}.Done \ \dagger \ A - res)[d^{-1}]) & \\ & \quad \dagger \ (Nextclause(res.A \ \dagger \ A' - succ) \rhd^+ B))) \setminus G & : expansion \\ \approx & ((A \ \dagger \ A')[g] \mid (B \rhd^+ (A[d][d^{-1}]) \ \dagger \ (A' \rhd^+ B)) \setminus G & : Lemmas\ 7,\ 8 \\ \approx & (B \rhd^+ A) \ \dagger \ (A' \rhd^+ B) & : simplify \end{array}$$

(iv) $\rhd^+ - \mathbf{4}$ *and (v)* $\rhd^+ - \mathbf{5}$: *The derivations are similar to* $\rhd^+ - 3$ *and* $\rhd^+ - 4$ *respectively.*

□

Using these bisimilarities, the basic control mechanism used in the semantics will be illustrated. Consider the backtracking expression:

$$(A_1 \ \dagger \ A_2 \ \dagger \ \cdots \ \dagger \ A_k) \rhd^+ B$$

Let $A_1 \approx \overline{res}.A_1'$. The expression is then bisimilar to:

$$(\overline{res}.A_1' \ \dagger \ A_2 \ \dagger \ \cdots \ \dagger \ A_k) \rhd^+ B$$

Applying $\rhd^+ - \mathbf{3}$ to this:

$$(B \rhd^+ A_1') \ \dagger \ ((A_2 \ \dagger \ \cdots \ \dagger \ A_k) \rhd^+ B)$$

Note that the search strategy is depth–first down the first clause A_1 of the original query. Now consider the first term of this sequence. Letting $B \approx B_1 \rhd^+ \cdots \rhd^+ B_n$, and using associativity:

$$((B_1 \rhd^+ \cdots \rhd^+ B_n) \rhd^+ A_1') \ \approx \ (B_1 \rhd^+ (B_2 \rhd^+ \cdots \rhd^+ B_n \rhd^+ A_1'))$$

Depending on the form of B_1, appropriate bisimilarities are applied to this term. For example, letting $B_1 \approx \overline{res}.B_1'$, and using $\rhd^+ - 5$:

$$\begin{array}{ll} & B_1 \rhd^+ (B_2 \rhd^+ \cdots \rhd^+ B_n \rhd^+ A_1') \\ \approx & (\overline{res}.B_1') \rhd^+ (B_2 \rhd^+ \cdots \rhd^+ B_n \rhd^+ A_1') \\ \approx & (B_2 \rhd^+ \cdots \rhd^+ B_n \rhd^+ A_1' \rhd^+ B_1') \ \dagger \ Done \end{array}$$

From this, it can be seen how the goals are being expanded breadth–first. The expression with $\rhd^+$–sequenced terms is acting like a queue in which successive resolvents from goals at the front are appended to the end.

$$
\begin{array}{lcl}
 & & a(X) \stackrel{\text{def}}{=} a_1(X) \mathbin{+\!\!\!\!;} a_2(X) \mathbin{+\!\!\!\!;} a_3(X) \\
a(x). & & a_1(X) \stackrel{\text{def}}{=} X = x \\
a(y) :- a(y). & & a_2(X) \stackrel{\text{def}}{=} X = y \unrhd a(y) \\
a(z). & \Longleftrightarrow & a_3(X) \stackrel{\text{def}}{=} X = z \\
 & & \\
b(z). & & b(X) \stackrel{\text{def}}{=} b_1(X) \\
 & & b_1(X) \stackrel{\text{def}}{=} X = z
\end{array}
$$

Figure 8.8: Program and CCS translation

$$
\begin{array}{ll}
a(X) \rhd^{+} b(X) & \\
\approx (a_1(X) \mathbin{+\!\!\!\!;} a_2(X) \mathbin{+\!\!\!\!;} a_3(X)) \rhd^{+} b(X) & : \textbf{Con}\ a \\
\approx ((\overline{succ(\theta_1)}.Done) \mathbin{+\!\!\!\!;} a_2(X) \mathbin{+\!\!\!\!;} a_3(X)) \rhd^{+} b(X) & : \textbf{Resol}\ a_1, \\
 & \quad \theta_1 = \{X \leftarrow x\} \\
\approx b(x) \mathbin{+\!\!\!\!;} ((a_2(X) \mathbin{+\!\!\!\!;} a_3(X)) \rhd^{+} b(X)) & : \rhd^{+}\!-\mathbf{2} \\
\approx (a_2(X) \mathbin{+\!\!\!\!;} a_3(X)) \rhd^{+} b(X) & : \textbf{Resol}\ b,\ \mathbin{+\!\!\!\!;} -\mathbf{2}^{\dagger} \\
\approx ((X = y \unrhd a(y)) \mathbin{+\!\!\!\!;} a_3(X)) \rhd^{+} b(X) & : \textbf{Con}\ a_2 \\
\approx (\overline{res(\theta_2)}.a(y) \mathbin{+\!\!\!\!;} a_3(X)) \rhd^{+} b(X) & : expansion,\ \unrhd -\mathbf{1}, \\
 & \quad \theta_2 = \{X \leftarrow y\} \\
\approx (b(y) \rhd^{+} a(y)) \mathbin{+\!\!\!\!;} (a_3(X) \rhd^{+} b(X)) & : \rhd^{+}\!-\mathbf{3} \\
\approx (Done \rhd^{+} a(y)) \mathbin{+\!\!\!\!;} (a_3(X) \rhd^{+} b(X)) & : \textbf{Resol}\ b \\
\approx a_3(X) \rhd^{+} b(X) & : \rhd^{+}\!-\mathbf{1},\ \mathbin{+\!\!\!\!;} -\mathbf{2}^{\dagger} \\
\approx (\overline{succ(\theta_3)}.Done) \rhd^{+} b(X) & : \textbf{Resol}\ a_3, \\
 & \quad \theta_3 = \{X \leftarrow z\} \\
\approx b(z) \mathbin{+\!\!\!\!;} Done & : \rhd^{+}\!-\mathbf{4} \\
\approx \overline{succ(\theta_3)}.Done & : \textbf{Resol}\ b,\ expansion^{\dagger} \\
\Box &
\end{array}
$$

Figure 8.9: Symbolic computation of ": $-$ $a(X),\ b(X)$."

Consider the logic program in figure 8.8 with its CCS translation. An example symbolic computation is given in figure 8.9, which uses the bisimilarities of figure 8.7. Each line of the computation has "$\backslash \ \{res, doneclause, doneclause''\}$" implicitly appended to it (omitted for clarity). The steps with † denote when this restriction is applied. The breadth–first computation rule computes the binding $X \leftarrow z$, whereas Prolog's depth–first left–to–right control loops at the second clause of a.

The breadth–first expansion of a set of goals does not guarantee that looping computations will not occur. In addition, the clause sequencing used is essentially the same as standard Prolog's, so that one clause will have to be totally computed before subsequent ones are searched. The final breadth–first strategy studied is completely fair, and uses both breadth–first search and computation rules.

8.3 Full breadth–first control

The breadth–first strategy studied in this section has a control scheme with characteristics similar to the ones discussed previously. Goals are expanded using a breadth–first computation rule like that of section 8.2. This is combined with an interleaved search throughout the clauses, similar to the one in section 8.1, except that the interleaving occurs between the resolution of clause descendents, and not just between computed solutions as before. Together, these two strategies result in control which searches the entire computation tree in a breadth–first fashion.

Let $D \equiv \{doneclause,\ next,\ res,\ doneclause''\}$

$$\begin{aligned}
\mathcal{M}[\![\ [P_1, P_2, ..., P_k]\ ]\!] &= P \stackrel{\text{def}}{=} P_1 \ \overset{*}{;}\ P_2 \ \overset{*}{;}\ \cdots \ \overset{*}{;}\ P_k \\
\mathcal{M}[\![\ P_i(\tilde{t}).\ ]\!] &= P_i(\tilde{x}) \stackrel{\text{def}}{=} \tilde{x} = \tilde{t} \\
\mathcal{M}[\![\ P_i(\tilde{t})\ :-\ G_1,\ G_2,\ ...\ G_n.\ ]\!] &= P_i(\tilde{x}) \stackrel{\text{def}}{=} \tilde{x} = \tilde{t} \rhd\ G_1 \rhd^{*} G_2 \rhd^{*} \cdots \rhd^{*} G_n \\
\mathcal{M}[\![\ ?-\ P,\ Q.\ ]\!] &= (P \rhd^{*} Q) \setminus D
\end{aligned}$$

Figure 8.10: Example translations

Some example translations for Prolog code are in figure 8.10. Figure 8.11 con-

$$
\begin{array}{rcl}
Let\ [h] & \equiv & [\ succ'/succ,\ res'/res,\ done'/done,\ next'/next, \\
 & & \quad doneclause'/doneclause,\ doneclause'''/doneclause''] \\
{[d]} & \equiv & [\ doneclause''/doneclause\ ] \\
{[d^{-1}]} & \equiv & [\ doneclause/doneclause''\ ] \\
H & \equiv & \{\ succ',\ res',\ done',\ doneclause',\ next'\ doneclause'''\ \}
\end{array}
$$

$$
\begin{array}{rcl}
A \rhd^{*} B & \stackrel{\mathrm{def}}{=} & \overline{next}.(\ A[h] \mid Hloop_i\ ) \setminus H \\
A \rhd\!\!\!^{*} B & \stackrel{\mathrm{def}}{=} & (\ A[h] \mid Hloop_i\ ) \setminus H \\
Hloop_i & \stackrel{\mathrm{def}}{=} & done'.Done\ +\ succ'.(B\ \overset{*}{;}\ (Nextclause(A - succ) \rhd\!\!\!^{*} B)) \\
 & & +\, res'.(B \rhd^{*} (Clause(A - res)[d^{-1}])) \\
 & & \quad \overset{*}{;}\ (Nextclause(A - res) \rhd\!\!\!^{*} B)
\end{array}
$$

$$
\begin{array}{rcl}
P\ \overset{*}{;}\ Q & \stackrel{\mathrm{def}}{=} & (P[done'/done, next'/next][d] \mid \\
 & & \quad (done'.\overline{doneclause}.Q\ +\ next'.(Q\ \overset{*}{;}\ (P - next))))\setminus H
\end{array}
$$

Figure 8.11: Full breadth–first control operators

$$
\begin{array}{lrcl}
\overset{*}{;}-\mathbf{1}: & (\overline{succ}.A)\ \overset{*}{;}\ B & \approx & \overline{succ}.(A\ \overset{*}{;}\ B) \\
\overset{*}{;}-\mathbf{2}: & (\overline{next}.A)\ \overset{*}{;}\ B & \approx & B\ \overset{*}{;}\ A \\
\overset{*}{;}-\mathbf{3}: & Done\ \overset{*}{;}\ B & \approx & \overline{doneclause}.B \\
 & & & \\
\rhd^{*}-\mathbf{1}: & Done \rhd^{*} B & \approx & \overline{next}.Done \\
\rhd^{*}-\mathbf{2}: & (\overline{succ}.Done\ \overset{*}{;}\ A) \rhd^{*} B & \approx & \overline{next}.(B\ \overset{*}{;}\ (A \rhd\!\!\!^{*} B)) \\
\rhd^{*}-\mathbf{3}: & (\overline{res}.A\ \overset{*}{;}\ A') \rhd^{*} B & \approx & \overline{next}.((B \rhd^{*} A)\ \overset{*}{;}\ (A' \rhd\!\!\!^{*} B)) \\
\rhd^{*}-\mathbf{4}: & \overline{succ}.Done \rhd^{*} B & \approx & \overline{next}.(B\ \overset{*}{;}\ Done) \\
\rhd^{*}-\mathbf{5}: & \overline{res}.A \rhd^{*} B & \approx & \overline{next}.((B \rhd^{*} A)\ \overset{*}{;}\ Done) \\
 & & & \\
\rhd\!\!\!^{*}-\mathbf{1}: & Done \rhd\!\!\!^{*} B & \approx & Done \\
\rhd\!\!\!^{*}-\mathbf{2}: & (\overline{succ}.Done\ \overset{*}{;}\ A) \rhd\!\!\!^{*} B & \approx & B\ \overset{*}{;}\ (A \rhd\!\!\!^{*} B) \\
\rhd\!\!\!^{*}-\mathbf{3}: & (\overline{res}.A\ \overset{*}{;}\ A') \rhd\!\!\!^{*} B & \approx & (B \rhd^{*} A)\ \overset{*}{;}\ (A' \rhd\!\!\!^{*} B) \\
\rhd\!\!\!^{*}-\mathbf{4}: & \overline{succ}.Done \rhd\!\!\!^{*} B & \approx & B\ \overset{*}{;}\ Done \\
\rhd\!\!\!^{*}-\mathbf{5}: & \overline{res}.A \rhd\!\!\!^{*} B & \approx & (B \rhd^{*} A)\ \overset{*}{;}\ Done
\end{array}
$$

Figure 8.12: Full breadth–first control bisimilarities

tains CCS definitions of the full breadth–first control operators. The sort of actions possible for agents P is:

$$\mathcal{L}(P) = \{\ succ(\theta),\ res(\theta),\ done,\ doneclause,\ next\ \}$$

The $succ$, res, $done$, and $doneclause$ actions are used in the same way as in the last section. The $next$ action represents when all the immediate descendents of one clause have been resolved, and that the computation is to continue at a new clause. As before, superscripts are sometimes used to relabel actions.

The $\mathrel{\rhd\!\!\!-}$ operator is the same as before. The $\rhd^*$ backtracking operator uses a breadth–first computation strategy identical to that of the $\rhd^+$ operator discussed earlier. There are three differences between $\rhd^*$ and $\rhd^+$. Firstly, the $\rhd^*$ operator immediately generates a $\overline{next}$ action, which is used by any external $\stackrel{*}{;}$ sequencing operator. Secondly, $\rhd^*$ is defined to be left–associative: $A \rhd^* B \rhd^* C \ \equiv\ (A \rhd^* B) \rhd^* C$. This asymmetry is introduced by the $next$ action. Lastly, $\rhd^*$ uses a $\dot{\rhd}^*$ operator for its recursive calls. The $\dot{\rhd}^*$ operator is identical to $\rhd^*$, except for the lack of a $\overline{next}$ action within it. The functionality of $\rhd^*$ and $\dot{\rhd}^*$ will be made clear later after higher–level bisimilarities are derived.

The sequencing operator $\stackrel{*}{;}$ combines characteristics of the $|;|$ and $\stackrel{+}{;}$ operators. The $\stackrel{*}{;}$ operator interleaves the search whenever a $next$ action is encountered, rather than with $succ$ actions as with $|;|$. Like $\stackrel{+}{;}$, the action $doneclause$ marks the termination of clauses. It is also right–associative only:

$$P \stackrel{*}{;} Q \stackrel{*}{;} R \equiv P \stackrel{*}{;} (Q \stackrel{*}{;} R)$$

Derivations of the high–level bisimilarities of figure 8.12 follow.

Theorem 10

$$\begin{array}{rrcl}
\stackrel{*}{;}-\mathbf{1}: & (\overline{succ}.A) \stackrel{*}{;} B & \approx & \overline{succ}.(A \stackrel{*}{;} B) \\
\stackrel{*}{;}-\mathbf{2}: & (\overline{next}.A) \stackrel{*}{;} B & \approx & B \stackrel{*}{;} A \\
\stackrel{*}{;}-\mathbf{3}: & Done \stackrel{*}{;} B & \approx & \overline{doneclause}.B
\end{array}$$

Proof:

$$\begin{array}{rll}
\stackrel{*}{;}-\mathbf{2}: & (\overline{next}.A) \stackrel{*}{;} B & \\
& \approx (\overline{next'}.A[h] \mid (B \stackrel{*}{;} (\overline{next}.A - next)) \setminus H & : \mathbf{Con}\ \stackrel{*}{;},\ \mathbf{Rel},\ expansion \\
& \approx A[h]\backslash H \mid (B \stackrel{*}{;} A) & : simplify \\
& \approx B \stackrel{*}{;} A & : simplify
\end{array}$$

$\stackrel{*}{;}-\mathbf{1}$ and $\stackrel{*}{;}-\mathbf{3}$ are similar. □

The derivations of the $\dot\triangleright^{*}$ bisimilarities are identical to to their $\triangleright^{*}$ counterparts, except for the presence of the $\overline{next}$ actions.

Theorem 11

$$
\begin{array}{lrcl}
\triangleright^{*}\!-\mathbf{1}: & Done \triangleright^{*} B & \approx & \overline{next}.Done \\
\triangleright^{*}\!-\mathbf{2}: & (\overline{succ}.Done \overset{*}{;} A) \triangleright^{*} B & \approx & \overline{next}.(B \overset{*}{;} (A \dot\triangleright^{*} B)) \\
\triangleright^{*}\!-\mathbf{3}: & (\overline{res}.A \overset{*}{;} A') \triangleright^{*} B & \approx & \overline{next}.((B \triangleright^{*} A) \overset{*}{;} (A' \dot\triangleright^{*} B)) \\
\triangleright^{*}\!-\mathbf{4}: & \overline{succ}.Done \triangleright^{*} B & \approx & \overline{next}.(B \overset{*}{;} Done) \\
\triangleright^{*}\!-\mathbf{5}: & \overline{res}.A \triangleright^{*} B & \approx & \overline{next}.((B \triangleright^{*} A) \overset{*}{;} Done)
\end{array}
$$

Proof:

(i) $\triangleright^{*}\!-\mathbf{1}$*: using expansion (omitted)*

$$
\begin{array}{ll}
(ii)\ \triangleright^{*}\!-\mathbf{2}: & \\
((\overline{succ}.Done) \overset{*}{;} A) \triangleright^{*} B & \\
\approx \overline{next}.(((\overline{succ}.Done) \overset{*}{;} A)[h] \mid Hloop_i) \setminus H & : \mathbf{Con}\ \triangleright^{*} \\
\approx \overline{next}.((Done \overset{*}{;} A)[h] \mid (B \overset{*}{;} & \\
\qquad (Nextclause((\overline{succ}.Done) \overset{*}{;} A - succ) \dot\triangleright^{*} B))) \setminus H & : expansion \\
\approx \overline{next}.((Done \overset{*}{;} A)[h] \mid (B \overset{*}{;} (A \dot\triangleright^{*} B))) \setminus H & : Lemma\ 8 \\
\approx \overline{next}.(B \overset{*}{;} (A \dot\triangleright^{*} B)) & : simplify
\end{array}
$$

$$
\begin{array}{ll}
(iii)\ \triangleright^{*}\!-\mathbf{3}: & \\
((\overline{res}.A) \overset{*}{;} A') \triangleright^{*} B & \\
\approx \overline{next}.(((\overline{res}.A) \overset{*}{;} A')[h] \mid Hloop_i) \setminus H & : \mathbf{Con}\ \triangleright^{*},\ expansion \\
\approx \overline{next}.((A' \overset{*}{;} A)[h] \mid & \\
\qquad (B \triangleright^{*} (Clause(\overline{res}.A \overset{*}{;} A' - res)[d^{-1}])) \overset{*}{;} & \\
\qquad (Nextclause((\overline{res}.A) \overset{*}{;} A' - res) \dot\triangleright^{*} B)) \setminus H & : expansion \\
\approx \overline{next}.((A' \overset{*}{;} A)[h] \mid (B \dot\triangleright^{*} (A[d][d^{-1}])) \overset{*}{;} (A' \dot\triangleright^{*} B)) \setminus H & : Lemmas\ 7,\ 8 \\
\approx \overline{next}.((B \dot\triangleright^{*} A) \overset{*}{;} (A' \dot\triangleright^{*} B)) & : simplify
\end{array}
$$

(iv) $\triangleright^{*}\!-\mathbf{4}$ *and* (v) $\triangleright^{*}\!-\mathbf{5}$: *Their derivations are similar to* $\triangleright^{*}\!-2$ *and* $\triangleright^{*}\!-3$ *respectively.*

□

The basic control mechanism is now illustrated. Let

$$D \equiv \{\ res,\ next,\ doneclause,\ doneclause''\ \}$$

A computation is initially a goal query, possibly with backtracking:

$$(P \triangleright^{*} Q) \setminus D$$

The general form of predicate P is

$$P_1 \overset{*}{;} P_2 \overset{*}{;} \cdots \overset{*}{;} P_k$$

Substituting this in the query gives:

$$((P_1 \overset{*}{;} P_2 \overset{*}{;} \cdots \overset{*}{;} P_k) \rhd^* Q) \setminus D$$

Expanding this yields:

$$(\overline{next}.(A_1 \overset{*}{;} A_2 \overset{*}{;} \cdots \overset{*}{;} A_k)) \setminus D$$

where each $A_i (1 \leq i \leq k)$ is either of the form $\overline{succ}.Done$, $Done$, or $\overline{res}.A_i'$, where A_i' is a descendent goal expression. Expanding this gives,

$$(A_1 \overset{*}{;} A_2 \overset{*}{;} \cdots \overset{*}{;} A_k) \setminus D$$

since the restriction of D on the query suppresses *next* actions. The next step is to process A_1. If $A_1 \approx \overline{succ}.Done$, then $\overset{*}{;}-\mathbf{1}$ will produce this $\overline{succ}$ as a result for the computation:

$$\overline{succ}.(A_2 \overset{*}{;} \cdots \overset{*}{;} A_k) \setminus D$$

If $A_1 \approx Done$, then $\overset{*}{;}-\mathbf{3}$ makes it drop out:

$$(A_2 \overset{*}{;} \cdots \overset{*}{;} A_k) \setminus D$$

Should A_1 be a backtracking construct, then it generates an expression akin to the one above for the query:

$$\begin{aligned} A_1 \quad &\approx \overline{res}.A_i' \\ &\approx \overline{next}.(B_1 \overset{*}{;} \cdots \overset{*}{;} B_n) \end{aligned}$$

where the B_i's have the same form as the A_i's above. Substituting this into the query gives:

$$\begin{aligned} &((\overline{next}.(B_1 \overset{*}{;} \cdots \overset{*}{;} B_n)) \overset{*}{;} A_2 \overset{*}{;} \cdots \overset{*}{;} A_k) \setminus D \\ \approx\ &(A_2 \overset{*}{;} \cdots \overset{*}{;} A_k \overset{*}{;} B_1 \overset{*}{;} \cdots \overset{*}{;} B_n) \setminus D \qquad : \overset{*}{;}-\mathbf{2} \end{aligned}$$

The queue of goal expressions grows in this manner whenever new backtracking constructs are executed, and the queue shrinks when either $\overline{succ}$ or *Done* terms are encountered.

Worth explanation is the way the $\overline{next}$ action appends goal sets onto the end of the queue. It was mentioned earlier that $\rhd^*$ is left–associative: $A \rhd^* B \rhd^* C \equiv (A \rhd^* B) \rhd^* C$. This left-associativity is used so that only one *next* action is generated every time a backtracking $\rhd^*$ sequence is encountered. In the definition of $\rhd^*$ in figure 8.11, the action $\overline{next}$ is immediately generated when the operator is encountered, which is evident within all the bisimilarities for $\rhd^*$ in figure 8.12. When an expression contains multiple instances of $\rhd^*$, then a right–associative structure will cause multiple instances of $\overline{next}$ to be generated, in fact, one for each $\rhd^*$ in the expression. When left–associativity is used, the other $\overline{next}$ actions are generated by agents on the left–hand sides of $\rhd^*$ expressions. By relabelling them with $[h]$, they are effectively suppressed during execution of $\rhd^*$, and only the right–most $\rhd^*$'s $\overline{next}$ action is generated. The $\unrhd^*$ operator is used by $\rhd^*$ so that extraneous $\overline{next}$'s are not produced in the recursive definition. The effect of all the above is the generation of one $\overline{next}$ per clause being executed. This represents one single resolution of all its descendents, given the breadth–first computation rule used within it.

$$
\begin{array}{lcl}
 & & p(X) \stackrel{\text{def}}{=} p_1(X) \\
 & & p_1(X) \stackrel{\text{def}}{=} True \unrhd\ a(X) \rhd^* b(X) \\
 & & \\
p(X) :-\ a(X),\ b(X). & & q(X) \stackrel{\text{def}}{=} q_1(X) \\
q(w). & & q_1(X) \stackrel{\text{def}}{=} X = w \\
 & & \\
a(x). & \Longleftrightarrow & a(X) \stackrel{\text{def}}{=} a_1(X)\ \overset{*}{;}\ a_2(X)\ \overset{*}{;}\ a_3(X) \\
a(y) :-\ a(y). & & a_1(X) \stackrel{\text{def}}{=} X = x \\
a(z). & & a_2(X) \stackrel{\text{def}}{=} X = y\ \unrhd\ \ a(y) \\
b(z). & & a_3(X) \stackrel{\text{def}}{=} X = z \\
 & & \\
 & & b(X) \stackrel{\text{def}}{=} b_1(X) \\
 & & b_1(X) \stackrel{\text{def}}{=} X = z
\end{array}
$$

Figure 8.13: Program and CCS translation

Figure 8.13 contains a program and CCS translation, and figures 8.14 and 8.15 contain symbolic computations. Steps tagged by † represent when the restriction of D is applied. The computation in figure 8.15 uses the stream from figure 8.14. The extraneous $\overline{doneclause}$'s are ignored by $\overset{*}{;}$.

$$
\begin{array}{ll}
a(X) \rhd^* b(X) & \\
\approx (a_1(X) \stackrel{*}{;} a_2(X) \stackrel{*}{;} a_3(X)) \rhd^* b(X) & : \textbf{Con } a \\
\approx (\overline{succ(\theta_1)}.Done \stackrel{*}{;} a_2(X) \stackrel{*}{;} a_3(X)) \rhd^* b(X) & : \textbf{Resol } a_1, \\
 & \quad \theta_1 = \{X \leftarrow x\} \\
\approx \overline{next}.(b(x) \stackrel{*}{;} ((a_2(X) \stackrel{*}{;} a_3(X)) \rhd^* b(X))) & : \rhd^* - \mathbf{2} \\
\approx \overline{next}.(b(x) \stackrel{*}{;} ((\overline{res(\theta_2)}.a(y) \stackrel{*}{;} a_3(X)) \rhd^* b(X))) & : expansion,\ \rhd -\mathbf{1}, \\
 & \quad \theta_2 = \{X \leftarrow y\} \\
\approx \overline{next}.(b(x) \stackrel{*}{;} (b(y) \rhd^* a(y)) \stackrel{*}{;} (a_3(X) \rhd^* b(X))) & : \rhd^* - \mathbf{3} \\
\approx \ \ldots & \\
\approx \overline{next}.(b(x) \stackrel{*}{;} (b(y) \rhd^* a(y)) \stackrel{*}{;} b(z) \stackrel{*}{;} Done) & : as\ \ above \\
\approx \overline{next}.(Done \stackrel{*}{;} (b(y) \rhd^* a(y)) \stackrel{*}{;} b(z) \stackrel{*}{;} Done) & : \textbf{Resol } b(x) \\
\approx \overline{next}.\overline{doneclause}.((b(y) \rhd^* a(y)) \stackrel{*}{;} b(z) \stackrel{*}{;} Done) & : \stackrel{*}{;} -\mathbf{1} \\
\approx \overline{next}.\overline{doneclause}.((Done \rhd^* a(y)) \stackrel{*}{;} b(z) \stackrel{*}{;} Done) & : \textbf{Resol } b(y) \\
\approx \overline{next}.\overline{doneclause}.(\overline{next}.Done \stackrel{*}{;} b(z) \stackrel{*}{;} Done) & : \rhd^* - \mathbf{1} \\
\approx \overline{next}.\overline{doneclause}.(b(z) \stackrel{*}{;} Done \stackrel{*}{;} Done) & : \stackrel{*}{;} -\mathbf{2} \\
\approx \overline{next}.\overline{doneclause}.\overline{succ(\theta_3)}.\overline{doneclause}.\overline{doneclause}.Done & : \textbf{Resol } b(z)^{\ddagger} \\
 & \quad \theta_3 = \{X \leftarrow Z\} \\
\Box &
\end{array}
$$

Figure 8.14: Simplifying $a(X) \rhd^* b(X)$

$$
\begin{array}{ll}
p(X) \rhd^* q(Y) & \\
\approx (True \rhd\ a(X) \rhd^* b(X)) \rhd^* q(Y) & : \textbf{Con } p,\ p_1 \\
\approx (\overline{res(\epsilon)}.(a(X) \rhd^* b(X))) \rhd^* q(Y) & : expansion,\ \ \rhd\ -\mathbf{1} \\
\approx q(Y) \rhd^* ((a(X) \rhd^* b(X)) \stackrel{*}{;} Done) & : \rhd^* - \mathbf{5}^{\dagger} \\
\approx \overline{succ(\theta_4)}.Done \rhd^* ((a(X) \rhd^* b(X)) \stackrel{*}{;} Done) & : \textbf{Resol } q(Y),\ \ \theta_4 = \{Y \leftarrow w\} \\
\approx ((a(X) \rhd^* b(X)) \stackrel{*}{;} Done) \stackrel{*}{;} Done & : \rhd^* - \mathbf{4},\ \ expansion^{\dagger} \\
\approx (\overline{next}.\overline{doneclause}.\overline{succ(\theta_3)}.\overline{doneclause}. & \\
\quad \overline{doneclause}.Done) \stackrel{*}{;} Done) \stackrel{*}{;} Done & : Figure\ \ 8.14 \\
\approx (Done \stackrel{*}{;} \overline{doneclause}.\overline{succ(\theta_3)}.\overline{doneclause}. & \\
\quad \overline{doneclause}.Done) \stackrel{*}{;} Done & : \ \stackrel{*}{;} -\mathbf{2} \\
\approx \overline{succ(\theta_5)}.Done & : expansion,^{\dagger} \\
\Box & \quad \theta_4 = \theta_3 \cup \theta_4 \\
 & \qquad = \{X \leftarrow z,\ Y \leftarrow w\}
\end{array}
$$

Figure 8.15: Symbolic computation of ": $-\ \ p(X),\ q(Y)$."

8.4 Freeze predicates

This section models a simple predicate freezing strategy similar to that of SICSTUS Prolog (Carlsson and Widen 1988). The semantics given here should be considered to be a sketch of how freezing mechanisms can be handled in CCS, rather than a complete semantics of a predicate freezing implementation.

Predicate freezing permits the dynamic alteration of program control during execution. Program goals are either (i) active or executable, or (ii) blocked or frozen. An active goal is one which is a candidate for resolution, and a blocked goal is one which is suspended pending further computation. The criteria for deciding when to freeze or awaken a blocked goal is typically the instantiation pattern or mode of a predicate. For example, a predicate may be designated to be frozen whenever its arguments are non–ground terms. Although predicate freezing is considered to be sequential by virtue of the fact that it is supported in some sequential implementations of Prolog, for example, SICSTUS and NU–Prolog (Thom and Zobel 1988), it begins to verge on concurrent paradigm of logic program control. In addition, predicate freezing is very implementation dependent, and different implementations have different control conventions.

In SICSTUS, a predicate is designated as freezable using a *wait* declaration, and such a predicate becomes blocked whenever its first argument is uninstantiated. For example, with the declaration

$$?-\ wait\ \ p.$$

the predicate p is frozen if, during execution, a call to p contains a non–instantiated first argument. Such a goal becomes blocked until either (i) the variable in the first argument becomes instantiated, or (ii) the goals following p in the computation all fail, and p therefore fails due to exhausted backtracking.

Some example translations are in figure 8.16. Predicates declared as freezable use the $Freeze$ operator, while others have sequential predicate definitions as in standard Prolog. A new backtracking operator $\rhd^f$ is used. The goal $True$ is appended to the end of all $\rhd^f$ expressions. Queries are handled by a $Query$ operator.

The CCS definitions for the operators are in figure 8.17. The sort of possible actions are $\mathcal{L} = \{succ, done, wait, kill, awake, donew, var, nonvar, floundered\}$, along

$$\mathcal{M}[\![\ [P_1, P_2, ..., P_k]\ ]\!] \ = \ \begin{cases} P \stackrel{\text{def}}{=} Freeze(X_1, (P_1 \fatsemi \cdots \fatsemi P_k)) & : P \ freezable, \\ & \ \ X_1 \ 1st \ arg. \ of \ P \\ P \stackrel{\text{def}}{=} P_1 \fatsemi \cdots \fatsemi P_k & : otherwise \end{cases}$$

$$\mathcal{M}[\![\ P_i(\tilde{t}).\ ]\!] \ = \ P_i(\tilde{x}) \stackrel{\text{def}}{=} \tilde{x} = \tilde{t} \rhd^f True$$

$$\mathcal{M}[\![\ P_i(\tilde{t}) \ :- G_1, \ ... \ G_n.\ ]\!] \ = \ P_i(\tilde{x}) \stackrel{\text{def}}{=} \tilde{x} = \tilde{t} \rhd^f G_1 \rhd^f \cdots \rhd^f G_n \rhd^f True$$

$$\mathcal{M}[\![\ ?- \ P, \ Q.\ ]\!] \ = \ Query(P \rhd^f Q \rhd^f True)$$

Figure 8.16: Example translations

$$\begin{aligned} Let \ \ & [e] \equiv \forall \alpha \ [\ \alpha' / \alpha\ ] \\ & [n^*] \equiv \forall \alpha \ [\ \alpha^n / \alpha\ ] \ \ for \ \ a \ \ unique \ \ n \\ & E \equiv \cup_\alpha \ \{\ \alpha'\ \} \\ & I \equiv \cup_{i,\alpha} \ \{\ \alpha^i\ \} \end{aligned}$$

$$\begin{aligned} A \rhd^f B & \stackrel{\text{def}}{=} \ (\ A[e] \mid Eloop_j\ ) \setminus E \\ Eloop_j & \stackrel{\text{def}}{=} \ done'.Done \\ & \quad + succ'.B \fatsemi ((A - succ) \rhd^f B) \\ & \quad + wait'.(B \fatsemi \overline{kill}.Done \mid A[n^*]) \setminus kill \\ & \quad + \textstyle\sum_i awake^i.(Echo(i) \rhd^f A \rhd^f B) \fatsemi Eloop_j \end{aligned}$$

$$Echo(i) \stackrel{\text{def}}{=} \alpha^i.\overline{\alpha}.Echo(i) \ + \ donew^i.Done$$

$$\begin{aligned} Freeze(X, P) & \stackrel{\text{def}}{=} \ (\ Var(X) \mid var.\overline{wait}.Wait_k(X) + nonvar.P\ ) \setminus \{var, nonvar\} \\ Wait_k(X) & \stackrel{\text{def}}{=} \ var.\overline{wait}.Wait_k(X) \\ & \quad + nonvar.\overline{awake}.P \fatsemi (\overline{donew}.Synchronise(X)) \fatsemi Wait_k(X) \\ & \quad + kill.Done \end{aligned}$$

$$\begin{aligned} Query(Q) & \stackrel{\text{def}}{=} (\ Q[e] \mid QLoop\ ) \setminus E \setminus I \\ QLoop & \stackrel{\text{def}}{=} succ'.(wait^i.\overline{floundered}.QLoop + \overline{succ}.QLoop) + done'.Done \end{aligned}$$

Figure 8.17: Operators for predicate freezing

with indexed α' and $\alpha^i (i \geq 0)$ of these $\mathcal{L}$ actions. In $A \rhd^f B$, the behaviour of the expression is identical to standard backtracking using $\rhd$ when no frozen predicates are involved, as is evident in the first two terms in $Eloop_j$[1]. However, if A becomes blocked and generates a *wait* action, the blocked A is then executed concurrently with B. The behaviour of this frozen A is discussed below. The expression $[n^*]$ denotes a unique relabelling of agent A using some new integer value n. The *kill* signal, local to this term, terminates the blocked A when B has exhaustively backtracked and terminated. The fourth term represents the case when some blocked goal has awoken, which is seen via the generation of an $awake^i$ action. When this occurs, the stream generated by this awoken goal, which is uniquely determined by the label index i, is introduced to the front of the current computation. In summary, the net effect of $\rhd^f$ is to (i) execute non–blocked goals in the usual sequential way; (ii) execute blocked goals autonomously with the current computation, killing them later as appropriate; and (iii) injecting awoken goals into the computation.

The Freeze operator uses a builtin system predicate $Var(X)$, which continually polls the variable X, generating *var* as long as X is not instantiated, and *nonvar* when it becomes instantiated. $Freeze$ itself will initially execute P directly if the first argument X is instantiated, or it goes into a polling mode via the $Wait_k$ loop if X is uninstantiated. Within the $Wait_k$ loop, the action *wait* is generated as long as X remains uninstantiated. When X becomes instantiated, the action $\overline{awake}$ is generated, and the predicate P itself is executed. A $\overline{donew}$ occurs when P has terminated. Then a builtin $Synchronise$ agent is invoked, and the loop is re-executed. $Synchronise$ waits for its variable argument to unbind and re-bind. This is done so that a blocked goal does not infinitely awaken for one binding of a variable, but instead waits for a new instantiation of the variable before awakening. A *kill* signal terminates the $Freeze$ agent.

The $Query$ operator generates $\overline{floundered}$ should there be any blocked actions at the end of an executed query. It also suppresses any blocked goal streams via " $\setminus I$".

A small program and its CCS equivalent are in figure 8.18. A symbolic computation will now be shown. Consider the query "$? -\ p(X), r(X).$". The derivation begins:

[1] As before, loops are uniquely identified by subscripts.

$$
\begin{array}{lcl}
?-\ \mathit{wait}\ \ p. & & p(X) \stackrel{\mathrm{def}}{=} \mathit{freeze}(X, p_1(X)) \\
 & & p_1(X) \stackrel{\mathrm{def}}{=} \mathit{True} \\
p(X). & \Longleftrightarrow & \\
 & & \\
r(a). & & r(X) \stackrel{\mathrm{def}}{=} r_1(X) \\
r(b). & & r_1(X) \stackrel{\mathrm{def}}{=} X = a \rhd^f \mathit{True}
\end{array}
$$

Figure 8.18: Program and CCS translation

$$
\begin{array}{lll}
 & p(X) \rhd^f r(X) \rhd^f \mathit{True} & \\
\approx & (\mathit{Freeze}(X, p_1(X))[e] \mid \mathit{Eloop}_j)\backslash E & : \textbf{Con}\ \rhd^f,\ p \\
\approx & (\underline{(r(X) \rhd^f \mathit{True}} \mathbin{\hat{;}} \overline{\mathit{kill}}.\mathit{Done} \mid \mathit{Freeze}'(X, p_1(X))[1^*])\backslash \mathit{kill}\ \ (\mathbf{1}) & : \mathit{simplify}
\end{array}
$$

The Freeze' term in the last line represents the state of the goal blocked by X. The underlined expression is now expanded:

$$
\begin{array}{lll}
 & r(X) \rhd^f \mathit{True} & \\
\approx & (\underline{r(X)[e]} \mid \mathit{Eloop}_k) \setminus E & : \textbf{Con}\ \rhd^f \\
\approx & (\overline{\mathit{succ}(\theta_1)}.\mathit{Done}[e] \mid \mathit{Eloop}_k) \setminus E \quad (\mathbf{2}) & : \textbf{Resol}\ r,\ \ \theta_1 = \{X \leftarrow a\}
\end{array}
$$

At this point, because X is bound, the blocked goal is awoken, and the Freeze' expression generates the stream:

$$
\mathit{Freeze}'(X, p_1(X))[1^*] \approx \mathit{awake}^1.\overline{\mathit{succ}(\epsilon)}^1.\mathit{Done}^1.\mathit{Freeze}''(X, p_1(X))
$$

This stream interrupts the execution at $(\mathbf{2})$:

$$
\begin{array}{lll}
 & (\overline{\mathit{succ}(\theta_1)}.\mathit{Done}[e] \mid \mathit{Eloop}_k) \setminus E & \\
\approx & (\overline{\mathit{succ}(\theta_1)}.\mathit{Done}[e] \mid (\overline{\mathit{succ}(\epsilon)}.\mathit{Done} \rhd^f \overline{\mathit{succ}(\theta_1)}.\mathit{Done} \rhd^f \mathit{True}) & \\
 & \quad \mathbin{\hat{;}} \mathit{Eloop}_k) \setminus E & : \mathit{expansion} \\
\approx & (\overline{\mathit{succ}(\epsilon)}.\mathit{Done} \rhd^f \overline{\mathit{succ}(\theta_1)}.\mathit{Done} \rhd^f \mathit{True}) \mathbin{\hat{;}} \mathit{Eloop}_k) \setminus E & : \mathit{simplify}
\end{array}
$$

This continues, and produces $\overline{\mathit{succ}(\theta_1)}.\mathit{Done}$ as a result. Then, Freeze'' stalls until X is unbound, and the expression at $(\mathbf{1})$ continues execution, producing one floundering result. When it terminates, the blocked agent is killed.

The semantics of a practical implementation of predicate freezing would differ from the above in the following respects:

- When two or more predicates are blocked, an order for unblocking them needs to be established. A simple strategy is to use the order in which they became blocked. The above treatment, however, nondeterministically chooses goals to unblock.

- The simple "Var" approach above can be enhanced to use more sophisticated tests.
- The above generates solutions only when there are no outstanding frozen goals. An alternative approach is to produce solutions supplemented by a message that floundering goals were present.

8.5 Sequential nondeterminism

Various forms of nondeterministic sequential control are readily modelled with CCS. The control schemes suggested here exploit CCS's modelling of nondeterminism normally used for concurrency. Consequently, these nondeterministic sequential strategies are very similar to concurrent ones. They differ from the previous breadth–first ones in that nondeterminism is used for choosing which goal or clause is to be used, while the breadth–first schemes use a strictly deterministic selection criteria, it being the static ordering within program definitions.

Nondeterminism can be introduced into both the computation and search rules. One crude form of nondeterministic search is to model all possible permutations of clause search orderings. For example, for a three clause predicate,

$$\begin{aligned}\mathcal{M}[\![\ [P_1, P_2, P_3]\ ]\!] = \quad & P_1 \mathbin{⨾} P_2 \mathbin{⨾} P_3 \ + \ P_1 \mathbin{⨾} P_3 \mathbin{⨾} P_2 \ + \ P_2 \mathbin{⨾} P_1 \mathbin{⨾} P_3 \\ & + P_2 \mathbin{⨾} P_3 \mathbin{⨾} P_1 \ + \ P_3 \mathbin{⨾} P_2 \mathbin{⨾} P_1 \ + \ P_3 \mathbin{⨾} P_1 \mathbin{⨾} P_2\end{aligned}$$

CCS's choice operator "+" delimits nondeterministic choices of clause ordering. This expression models *every* possible ordering for the three clauses. If the expansion theorem is applied to such an expression, the six streams produced by the six search strategies will be generated. However, the stream generated by each strategy is statically defined.

A more nondeterministic search is the following. Let $[g_i] \equiv [done^i/done]$. Then,

$$\begin{aligned}&\mathcal{M}[\![\ [P_1, P_2, ..., P_n]\ ]\!] = \\ &(\ P_1[g_1] \mid P_2[g_2] \mid \cdots \mid P_n[g_n] \mid done^1.done^2.\cdots.done^n.Done) \setminus \{done^1, ..., done^n\}\end{aligned}$$

This expression represents all possible interleavings of streams generated by the clauses P_i. Each time a solution is computed by a clause, it is generated as a result. The final term with the indexed $done^i$ actions collects all the termination signals from the clauses, and generates $\overline{done}$ when and only when they have all terminated. These termination signals are collected in an arbitrary order; other permutations of them are not seen because of restriction. It should be apparent that the nondeterministic search above is also a model of a simple form of OR–parallelism without clause commits (guards). All the clauses are conceptually executed concurrently, and the semantics models all possible interleavings of their generated streams. It is considered to be a sequential control strategy here because the streams are intended to be used by some sort of sequential computation rule elsewhere, for example, with the standard backtracking operator $\triangleright$. Each term in the summation therefore represents a nondeterministically chosen sequential search possibility. However, this does not preclude its use as a semantics of parallelism, and this shows the close relationship between logic program nondeterminism and parallel computations.

Nondeterministic computation rules are also possible. A crude nondeterministic computation rule for three goals is

$$\begin{aligned}\mathcal{M}[\![\ G_1, G_2, G_3\ ]\!] = \quad & G_1 \triangleright G_2 \triangleright G_3 \ + \ G_1 \triangleright G_3 \triangleright G_2 \ + \ G_2 \triangleright G_1 \triangleright G_3 \\ & + G_2 \triangleright G_3 \triangleright G_1 \ + \ G_3 \triangleright G_2 \triangleright G_1 \ + \ G_3 \triangleright G_1 \triangleright G_2\end{aligned}$$

As with the similar search strategy shown previously, this represents the six possible permutations of goal ordering, each permutation of which uses standard Prolog backtracking. Applying the expansion theorem to this expression yields the six possible behaviours which result from each ordering of goals.

A better nondeterministic computation rule strategy is possible. Let $[f_i] \equiv [succ^i/succ, done^i/done]$. Then

$$\begin{aligned}\mathcal{M}[\![\ P\ (\tilde{t}).\ ]\!] = \quad & P(\tilde{X}) \stackrel{\text{def}}{=} (\tilde{X} = \tilde{t}) \\ \mathcal{M}[\![\ P\ (\tilde{t}) : -\ Body.\ ]\!] = \quad & P(\tilde{X}) \stackrel{\text{def}}{=} (\tilde{X} = \tilde{t}) \triangleright \mathcal{M}[\![\![\ \{Body\}\ ]\!]\!]\end{aligned}$$

$$\begin{aligned}\mathcal{M}[\![\![\ \{G\}\ ]\!]\!] = \quad & G \\ \mathcal{M}[\![\![\ \{G_1, ..., G_k\}\ ]\!]\!] = \quad & \\ (\ G_1[f_1] \mid \cdots \mid G_k[f_k] \mid \quad & \textstyle\sum_{j=1}^{k} succ^{\,j}(\theta).(\mathcal{M}[\![\![\ \cup\{G_i\theta\}_{i\neq j}\ ]\!]\!] \\ & \qquad\qquad ;\ \mathcal{M}[\![\![\ \{G_j - succ(\theta)\} \cup \{G_i\}_{i\neq j}\ ]\!]\!]) \\ & + done^1.done^2.\cdots.done^k.Done\ ) \setminus \cup_{j=1}^{k} \{\ succ^{\,j}, done^{\,j}\ \}\end{aligned}$$

The notation $\cup\{G_i\}_{i\neq j}$ denotes the set of goals indexed by i but not including G_j. All the goals are initially executed concurrently. When one goal succeeds, then the rest of the goals are executed with the first goal's computed answer substitution applied to them. When they terminate, all the goals are re-executed, but with the initial goal in the next state $(G_j - succ(\theta))$. As with the nondeterministic clause search, the whole agent terminates when all the goals terminate. The net effect of this semantics is (i) goals are nondeterministically selected, and (ii) the semantics models *all* possible nondeterministic computations. As with nondeterministic search, this strategy is also very analogous to simple AND–parallelism.

8.6 Program analysis

Using the semantics given previously, termination and transformation properties of programs under various control schemes can be ascertained. In the case of program termination, proofs entail the derivation of a symbolic computation of some query in question, showing via induction and well–founded orderings that the streams generated are finite, infinite, or looping. The semantics of control affect proofs by determining the way in which the symbolic computation evolves for the program. When defined, high–level bisimilarities and other control–specific properties determine how streams are generated and shared amongst program components. It is therefore helpful to derive behavioural properties for the particular control strategy being used in order to simplify proofs.

$$
\begin{array}{l}
p(X) :- \, a(X), b. \\
p(X) :- \, p(X). \\
\\
a(1). \\
a(2) :- \, a(2). \\
\\
b.
\end{array}
\qquad \Longleftrightarrow \qquad
\begin{array}{l}
p(X) \stackrel{\text{def}}{=} p_1(X) \ \overset{+}{;} \ p_2(X) \\
p_1(X) \stackrel{\text{def}}{=} True \mathrel{\rhd\!\!\!\!-} \ a(X) \rhd^{+} b \\
p_2(X) \stackrel{\text{def}}{=} True \mathrel{\rhd\!\!\!\!-} \ p(X) \\
\\
a(X) \stackrel{\text{def}}{=} a_1(X) \ \overset{+}{;} \ a_2(X) \\
a_1(X) \stackrel{\text{def}}{=} X = 1 \\
a_2(X) \stackrel{\text{def}}{=} X = 2 \mathrel{\rhd\!\!\!\!-} \ a(2) \\
\\
b \stackrel{\text{def}}{=} b_1 \\
b_1 \stackrel{\text{def}}{=} True
\end{array}
$$

Figure 8.19: Looping program

Consider the looping program of figure 5.3 of section 5.3.2, which is reproduced in figure 8.19 with a breadth–first computation rule operator. The termination analysis to follow shows how looping can still appear with a breadth–first computation rule. Two termination properties for the breadth–first computation rule are the following:

$$\begin{array}{ll} \textbf{Property 1}: & \bot \mathbin{\dagger} P \approx \bot \\ \textbf{Property 2}: & \overline{res}^{\,\omega} \mathbin{\dagger} P \approx \bot \end{array}$$

Recall that $\overline{res}^{\,\omega}$ is an infinite stream of $\overline{res}$ actions.

The call to $a(X)$ is first derived:

$$\begin{array}{ll}
a(X) & \\
\approx a_1(X) \mathbin{\dagger} a_2(X) & : \textbf{Con } a \\
\approx \overline{succ(\theta_1)}.\overline{doneclause}.a_2(X) & : \textbf{Resol},\ \ expansion,\ \ \theta_1 = \{X \leftarrow 1\} \\
\approx \overline{succ(\theta_1)}.\overline{doneclause}.\overline{\overline{res(\theta_2)}.\overline{res(\epsilon)}}^{\,\omega} & : \unrhd \mathbf{-1}\ \ repeated,\ \ \theta_2 = \{X \leftarrow 2\}
\end{array}$$

At this level, the looping behaviour manifests itself as an infinite stream of res actions. This should not yet be replaced with $\bot$, because it is possible this stream might be used externally.

Next, the call to p is expanded. We assume that res and $doneclause$ are implicitly restricted at the query level.

$$\begin{array}{ll}
p(X) & \\
\approx p_1(X) \mathbin{\dagger} p_2(X) & : \textbf{Con } p \\
\approx (\overline{res(\epsilon)}.a(X) \rhd^{+} b) \mathbin{\dagger} p_2(X) & : \textbf{Con } p_1,\ \ \unrhd \mathbf{-1} \\
\approx (\overline{succ(\theta_1)}.\overline{doneclause}.\overline{\overline{res(\theta_2)}.\overline{res(\epsilon)}}^{\,\omega} \rhd^{+} b \mathbin{\dagger} p_2(X) & : subst.\ a(X) \\
\approx b \mathbin{\dagger} (\overline{\overline{res(\theta_2)}.\overline{res(\epsilon)}}^{\,\omega} \rhd^{+} b \mathbin{\dagger} p_2(X) & : \rhd^{+}\mathbf{-\,2} \\
\approx \overline{succ(\theta_1)}.(\overline{\overline{res(\theta_2)}.res(\epsilon)}^{\,\omega} \rhd^{+} b) \mathbin{\dagger} p_2(X) & : \textbf{Resol } b,\ \ \mathbin{\dagger}\mathbf{-1} \\
\approx \overline{succ(\theta_1)}.((\overline{res(\epsilon)}^{\,\omega} \rhd^{+} b) \mathbin{\dagger} Done) \mathbin{\dagger} p_2(X) & : \rhd^{+}\mathbf{-\,5} \\
\approx \overline{succ(\theta_1)}.(\overline{res(\epsilon)}^{\,\omega} \rhd^{+} b) \mathbin{\dagger} p_2(X) & : simplify \\
\approx \overline{succ(\theta_1)}.\overline{res(\epsilon)}^{\,\omega} \mathbin{\dagger} p_2(X) & : simplify \\
\approx \overline{succ(\theta_1)}.\bot \mathbin{\dagger} p_2(X) & : \textbf{Property 1} \\
\approx \overline{succ(\theta_1)}.\bot & : \textbf{Property 2}
\end{array}$$

The expression reduces to an infinite stream of res actions, which loops in the absence of any external goals linking with it.

Another example is the variation of the 91 program in figure 8.20, originally studied in section 5.3.5 using standard Prolog control. With standard Prolog's computation rule, it was determined that $q(n, Y) \approx \overline{succ(\{Y \leftarrow n - 10\})}.\bot$ for $n > 100$. The program is now analysed using a breadth–first computation rule. It can be seen that the first clause generates the same solution $\{Y \leftarrow n - 10\}$ and terminates. The second clause is now analysed. The restriction of the action set $\{res, doneclause, doneclause''\}$

$$
\begin{array}{l}
q(X,Y) :- \; X > 100, \; Z \; is \; X - 10. \\
q(X,Y) :- \; X \leq 100, \; U \; is \; X + 11, \; q(U,Y), \; q(Y,Z).
\end{array}
$$

$$\Updownarrow$$

$$
\begin{array}{rcl}
q(X,Y) & \stackrel{\text{def}}{=} & q_1(X,Y) \;\dagger\; q_2(X,Y) \\
q_1(X,Y) & \stackrel{\text{def}}{=} & True \trianglerighteq \; X > 100 \rhd^{+} (Z \;\; is \;\; X - 10) \\
q_2(X,Y) & \stackrel{\text{def}}{=} & True \trianglerighteq \; (U \;\; is \;\; X + 11) \rhd^{+} q(U,Y) \rhd^{+} (X \leq 100) \rhd^{+} q(Y,Z)
\end{array}
$$

Figure 8.20: 91 program

is implicit over the following expansion. Also, the syntactic equivalences $D \equiv (n \leq 100) \rhd^{+} q(Y,Z)$ and $E \equiv (m \leq 100) \rhd^{+} q(Y',Y)$ are used for convenience.

$$
\begin{array}{ll}
q_2(n,Y) & \\
\approx \; \overline{res}.(U \;\; is \;\; n+11) \rhd^{+} q(U,Y) \rhd^{+} D & : \mathbf{Con} \;\; q_2 \\
\approx \; q(m,Y) \rhd^{+} D & : simplify, \;\; \rhd^{+} - \mathbf{4}, \\
 & \;\; U \leftarrow m, \;\; m = n+11 \\
\approx \; (q_1(m,Y) \;\dagger\; q_2(m,Y)) \rhd^{+} D & : \mathbf{Con} \;\; q \\
\approx \; (\overline{\overline{res}.succ(\theta_2)}.Done \;\dagger\; q_2(m,Y)) \rhd^{+} D & : simplify \;\; q_1, \\
 & \;\; \theta_2 = \{Y \leftarrow l\}, l = n+1 \\
\approx \; (n \leq 100 \rhd^{+} q(Y,Z) \rhd^{+} \overline{succ(\theta_2)}.Done) & \\
\qquad \dagger\; q_2(m,Y) \rhd^{+} D & : \rhd^{+} - \mathbf{3} \\
\approx \; (Done \rhd^{+} q(Y,Z) \rhd^{+} \overline{succ(\theta_2)}.Done) & \\
\qquad \dagger\; q_2(m,Y) \rhd^{+} D & : subst. \;\; \leq \\
\approx \; Done \;\dagger\; q_2(m,Y) \rhd^{+} D & : \rhd^{+} - \mathbf{1} \\
\approx \; q_2(m,Y) \rhd^{+} D & : \;\dagger\; -\mathbf{2}, \;\; simplify \\
\approx \; (\overline{res}.U' \;\; is \;\; m+11 \rhd^{+} q(U',Y') \rhd^{+} E) \rhd^{+} D & : subst. \;\; q_1 \\
\approx \; (n \leq 100) \rhd^{+} q(Y,Z) \rhd^{+} U' \;\; is \;\; m+11 \rhd^{+} q(U',Y') \rhd^{+} E & : \rhd^{+} - \mathbf{5}, \;\; simplify \\
\approx \; Done \rhd^{+} q(Y,Z) \rhd^{+} U' \;\; is \;\; m+11 \rhd^{+} q(U',Y') \rhd^{+} E & : subst. \;\; \leq \\
\approx \; Done & : \rhd^{+} - \mathbf{1}
\end{array}
$$

A breadth–first computation rule therefore causes the query to terminate. This happens because finite failure caused by the numerical test $\leq$ can be ascertained, whereas standard left–to–right control searches down the infinite tree before this test can be applied.

8.7 Composition of control

A powerful feature of the CCS semantics is the relative ease of intercomposing operators to create new control schemes, since the underlying CCS formalism is uniform for all the various semantics. This is not necessarily as simple as putting different operators together in expressions. When combining different control mechanisms, a useful relationship between them must be first be established so that a suitable and usable control strategy results. In addition, different control schemes often use different protocols, and the creation of hybrid control strategies requires that a conventionalised strategy is defined between them. Therefore, a technical task is to standardise the interface between different operators so that useful and predictable behaviour is modelled.

Consider the modelling of a control strategy which merges standard Prolog's backtracking and full breadth–first control. A design issue for this new control scheme is to assure that the additional actions used by the breadth–first scheme do not interfere with the $\rhd$ and $\hat{;}$ operators. Because $\rhd$ and $\hat{;}$ only use the sort $F = \{succ, done\}$, they will naturally ignore the full breadth–first actions

$$L = \{res, doneclause, next, doneclause''\}.$$

However, what effect this ignoring of actions has upon the breadth–first search, as well as on other program components, must be decided upon. Because the breadth–first expressions generate the action $\overline{next}$ to begin the search down a new clause, the $\rhd$ operator should ignore this signal and others in order to localise the breadth–first search within its two backtracked goals. Let

$$\begin{array}{rcl} [g] & \equiv & [\,res'/res,\ next'/next,\ doneclause'/doneclause,\ doneclause'''/doneclause''] \\ G & \equiv & \{\,res',\ next',\ doneclause',\ doneclause'''\,\} \end{array}$$

and L as above. A modified $\rhd'$ which localises breadth–first search is:

$$P \rhd' Q \stackrel{\text{def}}{=} (P[g] \rhd Q) \setminus G \setminus L$$

The restriction of L assures that these actions, which can be produced by Q, do not affect control outside this expression.

To use this new control, consider the expression:

$$Q \stackrel{\text{def}}{=} (A_1 \rhd^* \cdots \rhd^* A_k) \rhd' (B_1 \rhd^* \cdots \rhd^* B_n)$$

The behaviour of Q is as follows:

1. The left–hand–side expression with A_i's is executed using a full breadth–first strategy. However, this search is localised within the scope of this A_i term. Any breadth–first search outside the LHS and Q will not be part of the breadth–first search within Q.

2. Whenever a $\overline{succ}$ is computed, the right–hand expression with B_i's is invoked. The B_i expression then exhaustively generates solutions, again using a localised breadth–first control. When it terminates, control passes back to the A_i expression.

3. When the A_i expression terminates, the whole expression terminates.

This behaviour is simply standard Prolog's backtracking between two goals, but these goals each use full breadth–first search within them.

It should be noted that the merging of control strategies such as the predicate freezing and nondeterministic ones with other schemes requires more effort. With predicate freezing, the unfreezing mechanism must be introduced to all the backtracking components. Such a conversion would be uniformly administered throughout the semantics. Because the nondeterministic strategies exploit CCS's summation operator, care must be taken regarding its affects on observational equivalence, since expressions are not always substitutive within summation expressions.

8.8 Conclusion

This chapter presented CCS models of other sequential logic programming control strategies. Three different breadth–first control schemes, a predicate freezing mechanism, and some nondeterministic sequential control strategies were modelled. The use of these semantics towards program analysis was discussed. The intercomposition of different control operators was also addressed. This exercise in modelling various control schemes in CCS proved that the process algebra paradigm is a good formalism for describing logic program control, which is a central hypothesis of this thesis.

The breadth–first semantics share the same advantages as the semantics for standard Prolog: all used the domain of streams of answer substitutions; all used control operators which correspond syntactically to program components; and with the exception of the predicate freezing semantics, all had higher–level bisimilarities defined for them. One problem is that, as with the standard Prolog semantics, the basic control operators definitions of the breadth–first schemes describe the operational semantics at too fine a level of detail. However, the high–level bisimilarities for the operators create an higher level of abstraction which is more conceptually useful.

CCS betrayed some drawbacks when deriving the semantics of various control schemes. A design motivation of CCS is that it is a minimalist model of concurrency in which a wide variety of concurrent behaviours can be described using a small set of basic CCS operators. This has both advantages and disadvantages from the standpoint of deriving programming language semantics. One obvious advantage is that the designer need only learn and master a small vocabulary of formal tools. A disadvantage is that complex language behaviour is not always easy to conceptualise using the primitive operators of kernel CCS. Describing some of the high–level behaviour of the breadth–first schemes in terms of low–level primitives was difficult. The solution to this was to derive higher–level operators in terms of the CCS kernel language (eg. "-", Clause, etc), a task often found to be nontrivial. The design process would have benefited by having a stock library of higher–level operators already supported by CCS, such as those available in Hoare's CSP (Hoare 1985a).

A powerful way of deriving new operators in CCS is via rules of inference. Milner defines the basic CCS operators using rules of inference. This approach also lends itself well for defining the control operators studied in this chapter. Doing so means that, instead of basic CCS definitions, control operators are defined directly in terms of the semantic expressions of which they are components. A technical requirement, however, is that such operators must be shown to preserve the theory of equality being used. For example, if an operator f is defined via rules of inference, and if $P \approx s$, then it must be shown that $f(P,Q) \approx f(s,Q)$. This proof must be shown for all possible forms of expressions P. Such definitions are possible for all the control operators given in this chapter, and would have been similar in descriptive content to the high–level bisimilarities. Proofs that observational equivalence was preserved would have then been required. Technically speaking, definition by inference would have alleviated the problem of distinguishing individual clause streams in the breadth–first control schemes. Instead of using $Clause$ and $NextClause$, clause streams would have been directly represented in inference rules by referring to the expressions between $\overset{\dagger}{;}$ and $\overset{*}{;}$ operators.

CCS is most suitable to modelling control strategies which can be characterised either by the relative positioning of goals or clauses, or by nondeterminism. Standard Prolog control, interleaved search, and all the breadth–first strategies can be described by the relative ordering of goals and clauses, while predicate freezing and the nondeterministic strategies exploit nondeterminism. Some control rules, however, are difficult to model in basic CCS, and their definitions would probably be involved. For example, a computation rule which uses the first positive or ground negative atom is not easily derived in CCS, because the control mechanism needs to examine whether atoms are negative, and if so, whether the arguments are ground. This information would have to be communicated within the control protocol, which is undesirable. This is a serious weakness if control schemes like safe negation as failure are to be modelled, which require that the groundness of negated atoms be considered when selecting goals. Another example of a difficult computation rule is one which prioritises particular atoms, for example, by using a priority ordering b, a, c for goals. Basic CCS definitions of such backtracking control either need to "hard wire" these atoms into the backtracking operator definition, either explicitly or by communicating the identity of atoms within the protocol, which again is very inelegant. The only practical solution to the definition of such control operators is to define them using rules of inference.

A common method of prototyping various logic program control strategies is via meta–interpreters. Figure 8.21 shows a simple breadth–first computation rule meta–interpreter[2]. The search and computation rules which occur with this interpreter are the ones modelled in section 8.2. The interpreter's behaviour is determined by the operational semantics of the meta–language. This semantic circularity can be overcome if a concrete semantics of control is introduced at some level, such as in section 4.6.

Other semantic formalisms could be used to describe logic program control. Denotational semantics like (Debray and Mishra 1988) are too mathematically unwieldy for analysing programs. The advantages of process algebraic semantics over Baudinet's style of functional semantics (Baudinet 1988) is especially apparent when

[2] Based on one by Lee Naish.

$$
\begin{array}{l}
solve\_bf(G) :- \\
\quad solve\_bf\_d(G - T, T). \\
\\
solve\_bf\_d(QH,\ QT)\ :- \\
\quad QH == QT, \\
\quad !. \\
solve\_bf\_d((A, B) - C,\ D - E)\ :- \\
\quad !, \\
\quad clause(A,\ D), \\
\quad solve\_bf\_d(B - C,\ E). \\
solve\_bf\_d(true - A,\ B)\ :- \\
\quad !, \\
\quad solve\_bf\_d(A,\ B). \\
solve\_bf\_d(A - B,\ C - D)\ :- \\
\quad clause(A,\ C), \\
\quad solve\_bf\_d(B,\ D).
\end{array}
$$

Figure 8.21: Breadth–first computation rule interpreter

alternate control schemes are to be modelled. Baudinet requires the final stream results to be axiomatically defined, whereas the CCS semantics defines the operational semantics directly, and the stream results are constructed from it.

Chapter 9

Conclusion

This chapter summarises and critiques the main results of the thesis. More specific technical evaluations are given in the discussions of previous chapters. Section 9.1 gives a summary and discussion of the thesis. Some future research directions are suggested in section 9.2. Section 9.3 discusses using CCS to handle concurrent logic programs.

9.1 Summary

This thesis presents a new algebraic semantics of standard Prolog, as well as for logic programming languages using other sequential control schemes. The semantics uses an AND/OR process interpretation of logic program computation. Unlike most other process interpretations modelling concurrency, the processes modelled here are sequential. The semantics is encoded as a process algebra using Milner's CCS.

The CCS semantics of Prolog belongs to the family of structured operational semantics of programming languages (Plotkin 1981) (Hennessy 1990), since CCS itself is defined using algebraic transitional rules of inference. The design and utility of this semantics is strongly influenced by the CCS formalism underlying it. CCS's basic treatment of communicating agents is an underlying means for modelling logic program computations. Goals and clauses of logic programs correspond semantically to AND and OR agents respectively. The semantics of the inference scheme used is modelled by appropriate definitions for these agents. During a computation, the dynamic AND/OR computation tree is denoted by a corresponding evolving CCS expression, which is essentially an algebraic representation for the tree.

The utility of a program language semantics is determined by many factors, including the mathematic formalism in which it is written, the nature of the language being modelled, and most importantly, its performance as a tool for high–level reasoning about programs. To prove properties of logic programs, a useful semantics should have the following features:

1. Semantic expressions should be mappable to program constructs (and vice versa). The semantics should reflect the modularity of logic programs.

2. The control scheme should be modelled as abstractly as possible. There should be few machine–oriented mechanisms encoded in the semantics (eg. stacks).

3. The semantics should operate over a domain not overly abstracted from a logic program's stream of answer substitutions.

The CCS semantics of Prolog satisfies all the above. Semantic equations are structurally isomorphic to their logic program counterparts. At the bisimilarity level, control is modelled at a high level of abstraction. Unlike the complex domains of denotational semantics, its stream domain is practical to apply to many types of logic program analyses.

A general principle in defining a useful programming language semantics is to find the right semantic level required for the intended language and application. This issue was encountered here. The semantics of the $\rhd$ and $\mathring{;}$ operators were first defined in basic CCS. However, this level is too primitive for reasoning about programs. As a result, these definitions were used to derive bisimilarities which model control at a higher level. This higher–level model is more suitable for program analysis applications, given its syntactic proximity to the syntax of Prolog programs.

The main feature of this semantics is that the control component of logic programming languages is concisely and rigorously represented. The semantics allows the details of data flow, such as the binding states and binding distributions, to be made abstract. This is convenient for applications in which data flow is of subsidiary importance. The semantics of Prolog's cut is particularly easy to model in CCS, as the pruning of computation tree branches which occurs when a cut is activated is modelled by forcing agents to deadlock. Once a semantics for standard Prolog control was derived, the descriptive robustness of CCS was tested by modelling various types of breadth–first control, as well as predicate freezing. As with the semantics of standard Prolog control, the semantics of the breadth–first schemes are hierarchical: control operators are defined using basic CCS, and high–level bisimilarities are defined on top of these definitions. Some nondeterministic sequential strategies are also modelled, which can be considered to be semantics of simple parallel control schemes. A powerful possibility is the intercomposing of control operators. This requires a coalescence of protocols between operators so that reasonable and predictable behaviour results.

The use of CCS in this thesis is somewhat contrary to its intended purpose. CCS is intended as theory for studying different concurrent phenomena. Its central motivation is that a wide variety of concurrent computations can be represented and studied using a small but powerful set of basic algebraic operators. In addition, CCS is not intended as a semantic formalism for verifying properties of individual programs – and especially not sequential ones. However, despite these differences in motivations, this thesis proves that CCS is well–suited as a calculus of sequential logic program control. There are many reasons why it is successful in this regard. As mentioned above, CCS's notion of process or agent is commonly used in logic programming semantics. Sequential mechanisms are easily described in CCS, and as a consequence, the sequential AND/OR process interpretation of Prolog is readily modelled. The dynamic nature of logic program computation trees is naturally represented by evolving CCS expressions. CCS's facility for defining hierarchical levels of semantics lends itself well to deriving high–level bisimilarities for control operators. Finally, the ability to reason about sequential logic programs using CCS streams is very useful.

CCS's notion of behavioural equivalence is useful in logic program analysis. The theory of observational equivalence used in this thesis dictates that two programs are equivalent only if the observed streams of computed answer substitutions generated by them are equivalent. This strict notion of equivalence is helpful when unfair control strategies such as Prolog's are being considered. When fairer control schemes are being studied, this equivalence could be relaxed, perhaps in favour of a set–based characterisation of computed results, which is what declarative logical semantics is intended to model in the first place.

The current scheme of CCS representation does not adequately model Prolog data flow. CCS's value–passing calculus is extended to handle the Herbrand universe and logic variables. This is done by adding some new transition rules to CCS, which enabled the syntax of expressions using logical variables to be analysed. Having to modify the CCS formalism this way is not ideal, and the ramifications of a logical variable domain in CCS is an interesting topic for further study. However, the central motivation of this thesis was not a rigorous representation of the data domain, but rather, a semantics of control. Research in (Ross n.d.) has applied the π–calculus (Milner *et al.* 1989) – a process algebra similar to CCS but with label passing – to the semantics of parallel Prolog. The π–calculus can directly model the Herbrand domain and logic variables, and should be applicable to the sequential semantics in this thesis.

The semantics is similar to the functional semantics of Prolog in (Baudinet 1988). The CCS semantics can be considered to be a rational reconstruction of Baudinet's semantics. Baudinet defines axiomatically the stream results of Prolog goals and clauses; the CCS semantics defines the operational behaviour of Prolog goals and clauses (via the $\hat{;}$ and $\triangleright$ operators), and the stream results are constructed with application of the expansion theorem. Termination, non–termination, and looping are analysed directly using CCS, rather than axiomatically as by Baudinet. The advantages of this approach over hers are: (i) a variety of logic program behaviours can be reasoned about; (ii) different sequential control schemes can be more easily modelled (including a cleaner treatment of the cut); (iii) the modelling of concurrency is a clear possibility; and (iv) within program analyses, algebraic transitions and induction are arguable more intuitive than fixpoint induction.

The utility of the semantics is tested by using it in program termination and program transformation applications. The main contribution of the semantics in these applications is its ability to concisely represent control, which aided reasoning about control within proofs. In the case of standard Prolog control, termination characteristics are fundamentally determined by the particulars of goal and clause order. The CCS semantics enforces a rigid ordering of program components, while its stream–based domain allowed different phenomena such as nontermination and looping to be modelled. Some basic termination properties of Prolog are derived with the semantics, which are used in termination analyses when showing that the CCS representations of programs conformed to well–founded orderings. The use of Baudinet's data flow notation is necessary if data flow is to be explicitly represented. Alternatively, a more conventional logical treatment using quantifiers is worth investigation.

Proofs of program transformations using cuts are straight–forwardly done with the semantics. The data component of the semantics can be entirely abstracted from proofs, which simplifies the resulting proofs. Proofs of different source–to–source transformations are done by showing that the CCS expressions for each type of transformation schema were bisimilar, which is an excellent application of the bisimulation proof technique.

The semantics is used to suggest a characterisation of sound partial evaluation transformations of Prolog programs. Correctness–preserving partial evaluation transformations are performed on a program by applying bisimilar transformations on the program's semantic representation. A simple example of such transformations are the high–level bisimilarities of the control operators, which is a way of formalising the rudimentary control strategy used by partial evaluation algorithms. Conversely, non–sound partial evaluation transformations are those which do not preserve the semantic integrity of the program's semantic representation.

9.2 Future directions

There are theoretical and practical extensions for this research. On the theoretical side, one possible project is to study in more detail different semantic interpretations of CCS when applied to modelling logic programming languages, and compare these interpretations with existing declarative and operational semantics of logic programs. For example, CCS has a fixpoint interpretation in which recursive agent definitions are uniquely interpreted by canonical fixpoint expressions. It is worth investigating the formal relationship between this and existing fixpoint theories of logic programs.

Another possibility is to characterise different types of nondeterminism inherent in logic program computations. The stream interpretation of logic programs can be relaxed to create a set of answer substitution interpretation. It might be possible to use a process semantics to study the different computational means by which a program can generate the same set of computed solutions. Of obvious utility in a concurrent context, this approach might also be applicable to sequential programs in order to determine the "degree of parallelism" inherent within them. CSP (Hoare 1985*a*) is worth investigating in this regard, since its characterisation of nondeterminism using sets of actions seems similar in nature to existing model–theoretic semantics of logic programs.

On the practical side, one possible topic is to extend the semantics to handle more features of sequential Prolog. One obvious feature readily modelled with streams is the input–output facility. More control schemes can be modelled in CCS. The breadth–first and predicate freezing mechanisms addressed here are just a few of many possible sequential control schemes, and the modelling of constraint logic programming languages is worth considering. (The next section addresses the issue of concurrency.) However, process algebras are by no means the most suitable tool for modelling some types of Prolog phenomena, for example, the database manipulations done by *assert* and *retract*. It is probably the case that, rather than aiming for a single semantic formalism which describes all the features of a language, different semantic tools should be chosen according to how effectively they model the phenomena being studied.

The CCS semantics is intended as an example of a programming calculus which is to be used "by hand". It has been argued, however, that formal software engineering is fundamentally flawed if one considers the complexities that are encountered with the formal proofs of even the smallest of programs (de Millo *et al.* 1979). If program proofs defy intuitive understanding, then there is not much hope for the formal methods approach. Computed–aided program development and analysis is the solution: computers are perfectly suited for keeping track of the complexities and bookkeeping which occur in formal proofs. A practical extension of this thesis is therefore the creation of a semi–automated support environment for logic program analysis. There are a number of possible approaches for such a system. One approach for such a system is to have a kernel support module for CCS, which might take the form of a subset of the Concurrency Work Bench used in CCS analysis (Cleaveland *et al.* 1989). An interface would then interact between this module and higher–level modules tailored for logic program analysis. These modules would allow useful interaction between the user, support routines, and the AND/OR process tree semantics of logic program computation as encoded in CCS. In addition, interactive modules encapsulating methodologies for proving particular program properties such as termination, transformations, and partial evaluation, could be designed. Such modules could be made semi–automated with the use of sophisticated proof techniques (Plumer 1990) and proof plans (Bundy *et al.* 1988).

Another approach is to encode the CCS semantics for Prolog control in terms of an algebraic rewrite system (Dershowitz and Plaisted 1985). A *term rewriting system* is a set of algebraic equations of the form $e(\tilde{t}) \Rightarrow e'(\tilde{t})$, where e and e' are expressions with variables $\tilde{t}$ ranging over terms in an algebra E. A rewrite rule is applied by matching an expression $x \in E$ with a left hand side e of one of the rules, and applying the rule to result in a *derivative* x'. The CCS semantics for Prolog can also be characterised as a rewrite system. The bisimilarities of figures 3.5 and 3.6 represent *conditional* rewrite rules having the form

$$C \supset (P \Rightarrow P')$$

where C is a condition, P is the term to be rewritten, and P' is the rewritten term. For example, the rewrite rule for **Seq** is

$$(\alpha \neq \overline{done}) \supset ((\alpha.P) \fatsemi Q \Rightarrow \alpha.(P \fatsemi Q))$$

The control bisimilarities, along with the agent definitions for the program, are straight–forwardly encoded as conditional rewrite rules. Higher–level modules tailored to Prolog program analysis can then be built on top of this rewrite system.

9.3 Extending the semantics for concurrency

A central hypothesis of this thesis is that process algebras are a suitable means for modelling both sequential and concurrent logic program computations in one uniform formalism. The utility of process algebras for modelling different sequential control schemes has been illustrated here. CCS has been applied elsewhere towards concurrent logic languages (Beckman *et al.* 1986) (Beckman 1986) (described shortly), albeit in quite a different manner than the style of semantics used here. In fact, the non-deterministic sequential strategies modelled in section 8.5 can be considered to be axiomatisations of very simple concurrent strategies. The next logical step is therefore to extend this style of semantics to handle a variety of concurrent control strategies.

The semantics of concurrent logic programming languages is an active research area, and CCS has been applied to describe concurrent logic program execution. The issue of control is especially relevant in this area, since that is the essential distinction between sequentiality and concurrency. Given two sequenced agents, one can commence only when the other has terminated. The mechanism for accomplishing this sequencing requires the use of a termination protocol between the agents. In CCS, this is defined by the expression

$$(P[b/done] \mid b.Q) \setminus b$$

The CCS expression

$$P \mid Q$$

models all possible interleavings of generated solutions of these two concurrent agents. Of course, most implementations of concurrent logic programming languages require additional control mechanisms between concurrently executing program components, and this simple representation would require additional control mechanisms.

The CCS semantics for concurrent Prolog in (Beckman *et al.* 1986) (Beckman 1986) is more concise than the sequential CCS semantics. The major difference is that Beckman maps predicates to actions, while this semantics maps predicates to agents, and uses specific actions to denote success and termination. Beckman's lack of specialised actions makes modelling aspects of concurrent and sequential control difficult. For example, when modelling a breadth–first computation rule, it is convenient to use a specialised action such as res for denoting phenomena such as a single clause resolution. Not using these special actions means that the inference rules of the basic CCS operators must necessarily be more specialised and complex. Beckman also modifies the definition of CCS's composition operator to account for unification and the distribution of bindings. This abstracts the operational semantics of concurrent control mechanisms such as guards and variable annotations. The CCS semantics of this thesis explicitly models unification using a unification agent, leaving CCS unaltered. For convenience, a higher–level resolution bisimilarity **Resol** is used. Finally, Beckman uses CCS value variables to directly model the Herbrand universe. This is possible because backtracking is never done, and so the full semantics of logical variables as found in sequential implementations need not be modelled. The semantics of this chapter must unbind logical variables during backtracking. Beckman's abstraction of logic variables (as well as the aforementioned control mechanisms) means that many properties of concurrent languages are not precisely modelled.

An obvious research topic is to model concurrent logic program languages such as GHC (Ueda 1986) or Andorra Prolog (Haridi and Brand 1988) using CCS in a style similar to that of the semantics in this thesis. A general goal is that concurrent and sequential control should be represented – and intercomposable – within one semantic framework. Besides the unifying perspective afforded by modelling these paradigms under one formalism, this also has practical repercussions. Unconstrained large–scale parallelism is not practical given current hardware and software technology. As a result, concurrent languages have to use many control mechanisms for harnessing parallelism, such as guards and variable annotations. Much of these control devices are sequential in nature. A semantics of sequential control mechanisms is therefore necessary if concurrent control is to be adequately understood.

Extending the sequential semantics to handle concurrent computations will require some enhancements. Replacing the sequential control operators with CCS's composition | operator is not sufficient for modelling concurrency. Given a sequentially backtracked expression $A \rhd B \rhd C$, a simple AND–parallel incarnation of this expression is $A \mid B \mid C$. One problem with this expression is that there are no means available for these goals to communicate with one another, other than by simply handshaking. This suggests the need for new concurrent control operators, similar in spirit to the sequential ones of this thesis. As with sequential control, concurrent control operators would model all the relevant aspects of concurrency without getting into too low–level of detail, and at a high enough level to permit reasoning about programs. The nondeterministic strategies given in section 8.5 is a first step in this direction.

Variable sharing between concurrently executing program components is a major issue in concurrent language design. A CCS semantics of shared memory might lend insight into implementation issues. A logic variable could be modelled by an autonomous agent whose state changes as substitutions are applied to it.

Applying a semantics for concurrent logic program control for proving program properties would benefit with the use of a semi–automated program verification environment. Concurrent programs require a support environment even more so than sequential ones. Enormously vast numbers of computational states arise within even the most trivial of concurrent programs. The corresponding semantic expressions for such computations are inherently massive in scale and complexity. When combined with the dynamic nature of the computation tree and data domain which arises with logic programs, this means that a support environment is crucial.

Bibliography

Andrews, J.H. (1991). Logic Programming: Operational Semantics and Proof Theory. PhD thesis. University of Edinburgh.

Apt, K.R. (1981). Ten years of Hoare's logic: a survey. *TOPLAS* **3**, 431–483.

Apt, K.R. and D. Pedreschi (1990). Studies in Pure Prolog: Termination. In: *Proceedings of the Symposium in Computational Logic* (J. Lloyd, Ed.). Springer-Verlag. Brussels. pp. 150–176.

Apt, K.R. and M. Bezem (1990). Acyclic Programs. In: *7th International Conference on Logic Programming.* MIT Press. Jerusalem, Israel.

Apt, K.R. and M.H. van Emden (1982). Contributions to the Theory of Logic Programming. *Journal of the ACM* **29**(3), 841–862.

Arbab, B. and D.M. Berry (1987). Operational and Denotational Semantics of Prolog. *Journal of Logic Programming* **4**(4), 309–329.

Ashcroft, E.A. and W.W. Wadge (1982). R for Semantics. *ACM Transactions on Programming Languages and Systems* **4**(2), 283–294.

Baudinet, M. (1988). Proving Termination Properties of Prolog Programs: A Semantic Approach. Technical report. Computer Science Department, Stanford U.

Beckman, L. (1986). Towards a formal semantics for concurrent logic programming languages. In: *3rd International Conference on Logic Programming, LNCS 225.* pp. 335–349.

Beckman, L., R. Gustavsson and A. Waern (1986). An algebraic model of parallel execution of logic programs. In: *Logic in Computer Science.* Cambridge, Mass.

Benkerimi, K. and J. Lloyd (1990). A Partial Evaluation Procedure for Logic Programs. In: *North American Conference on Logic Programming.* pp. 343–358.

Berg, H.K., W.E. Boebert, W.R. Franta and T.G. Moher (1982). *Formal Methods of Program Verification and Specification.* Prentice-Hall.

Bezem, M. (1989). Characterizing Termination of Logic Programs with Level Mappings. In: *Proceedings North American Conference on Logic Programming.* Cleveland, Ohio. pp. 69–90.

Billaud, M. (1985). Une Formalisation def Structures de Controle de Prolog. PhD thesis. Universite de Bordeaux. Bordeaux, France.

Billaud, M. (1988). Prolog Control Structures: a Formalization and its Applications. In: *Programming of Future Generation Computers* (K. Fuchi and M. Nivat, Eds.). pp. 57–73. Elsevier Science Publishers (North Holland).

Billaud, M. (1990). Simple Operational and Denotational Semantics for Prolog with Cut. *Theoretical Computer Science 71* pp. 193–208.

Boolos, G. and R. Jeffrey (1980). *Computability and Logic.* Cambridge University Press.

Bundy, A., F. van Harmelen, J. Hesketh and A. Smaill (1988). Experiments with Proof Plans for Induction. Technical report. University of Edinburgh.

Carlsson, M. and J. Widen (1988). *SICStus Prolog User's Manual.*

Clark, K.L. (1979). Predicate Logic as a Computational Formalism. Technical Report 79/59. Imperial College.

Cleaveland, R., J. Parrow and B. Steffen (1989). The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems. Technical Report ECS-LFCS-89-83. University of Edinburgh.

Clocksin, W.F. and C.S. Mellish (1981). *Programming in Prolog.* Springer-Verlag.

Conery, J.S. and D.F. Kibler (1985). AND Parallelism and Nondeterminism in Logic Programs. *New Generation Computing* **3**(1), 43–70.

de Millo, R.A., R.J. Lipton and A.J. Perlis (1979). Social Processes and Proofs of Theorems and Programs. *Communications of the ACM* **22**(5), 271–280.

Debray, S.K. and P. Mishra (1988). Denotational and Operational Semantics for Prolog. *Journal of Logic Programming* **5**, 61–91.

Deransart, P. (1988). On the Multiplicity of Operational Semantics for Logic Programming and Their Modelization by Attribute Grammars. Technical Report 916. INRIA.

Dershowitz, N. (1987). Termination of Rewriting. *Journal of Symbolic Computation* **3**, 69–116.

Dershowitz, N. and D.A. Plaisted (1985). Logic Programming cum Applicative Programming. In: *Proceedings of the Symposium on Logic Programming.* pp. 54–66.

Dijkstra, E.W. (1975). Guarded commands, nondeterminacy, and the formal derivation of programs. *Communications of the ACM* **18**, 453–457.

Drabent, W. (1987). Do Logic Programs Resemble Programs in Conventional Languages. In: *Proceedings of the Symposium on Logic Programming.* pp. 389–396.

Ershov, A.P. (1982). Mixed Computation: Potential Applications and Problems for Study. *Theoretical Computer Science 18* **18**, 41–67.

Fitting, M. (1985). A Deterministic Prolog Fixpoint Semantics. *Journal of Logic Programming* **2**, 111–118.

Francez, N., O. Grumberg, S. Katz and A. Pnueli (1985). Proving Termination of Logic Programs. In: *Logics of Programs conference, LNCS 193.* Springer-Verlag. Brooklyn.

Fujita, H. and K. Furukawa (1988). A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation. *New Generation Computing* **6**(2,3), 91–118.

Fuller, D.A. and S. Abramsky (1988). Mixed Computation of Prolog Programs. *New Generation Computing* **6**(2,3), 119–141.

Fung, P. (1988). A formalisation of novices' errors in Prolog programs. Technical Report 50. The Open University.

Futamura, Y. (1971). Partial evaluation of computation process – an approach to a compiler-compiler. *Systems – Comput. – Controls* **2**(5), 45–50.

Gries, D. (1981). *The Science of Programming.* Springer-Verlag.

Hallnas, L. and P. Schroeder-Heister (1988). A proof–theoretic approach to logic programming. Technical Report SICS R88005. Swedish Institute of Computer Science.

Harel, David (1980). And/Or Programs: A New Approach to Structured Programming. *ACM Transactions on Programming Languages and Systems* **2**(1), 1–17.

Haridi, S. and P. Brand (1988). Andorra Prolog: An integration of Prolog and commited choice languages. In: *Proceedings of the International Conference on Fifth Generation Computer Systems.* Tokyo, Japan. pp. 745–754.

Hascoet, L. (1988). Partial Evaluation with Inference Rules. *New Generation Computing* **6**(2,3), 187–209.

Hehner, E.C.R. (1984*a*). Predicative Programming Part I. *Communications of the ACM* **27**(2), 134–143.

Hehner, E.C.R. (1984*b*). Predicative Programming Part II. *Communications of the ACM.*

Hehner, E.C.R. (1984*c*). *The Logic of Programming.* Prentice-Hall.

Hehner, E.C.R., L.E. Gupta and A.J. Malton (1986). Predicative Methodology. *Acta Informatica* **23**, 487–505.

Hennessy, M. (1988). *Algebraic Theory of Processes.* MIT Press.

Hennessy, M. (1990). *The Semantics of Programming Languages – An Elementary Introduction Using Structural Operational Semantics.* John Wiley and Sons.

Hill, P.M. (1991). Pruning Operators for Partial Evaluation. Technical Report 91.30. University of Leeds.

Hill, P.M. and J.W. Lloyd (1988). Analysis of Meta-Programs. Technical Report CS-88-08. University of Bristol.

Hill, P.M., J.W. Lloyd and J.C. Shepherdson (1990). Properties of a Pruning Operator. *Journal of Logic and Computation* **1**(1), 99–143.

Hoare, C.A.R. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM* **12**(10), 576–583.

Hoare, C.A.R. (1985*a*). *Communicating Sequential Processes.* Prentice–Hall.

Hoare, C.A.R. (1985*b*). Programs are Predicates. In: *Mathematical Logic and Programming Languages* (C.A.R. Hoare and J.C. Shepherdson, Eds.). pp. 141–155. Prentice-Hall.

Hogger, C.J. (1984). *Introduction to Logic Programming.* Academic Press.

Hogger, C.J. (1990). *Essentials of Logic Programming.* Oxford University Press.

Jones, C.B. (1986). *Systematic Software Development Using VDM.* Prentice-Hall.

Jones, N.D. and A. Mycroft (1984). Stepwise development of operational and denotational semantics for Prolog. In: *Proceedings of the Symposium on Logic Programming.* Atlantic City. pp. 281–288.

Kowalski, R. (1979). Algorithm = Logic + Control. *Communications of the ACM* **22**(7), 424–436.

Kowalski, R. (1985). The relation between logic programming and logic specification. In: *Mathematical Logic and Programming Languages* (C.A.R. Hoare and J.C. Shepherdson, Eds.). pp. 11–27. Prentice-Hall.

Kowalski, R.A. (1974). Predicate logic as a programming language. In: *Proceedings IFIP-74.* North Holland. Amsterdam. pp. 569–574.

Lindstrom, G. and P. Panangaden (1984). Stream-based execution of logic programs. In: *Symposium on Logic Programming.* Atlantic City.

Lloyd, J.W. (1984). *Foundations of Logic Programming.* Springer-Verlag.

Lloyd, J.W. and J.C. Shepherdson (1987). Partial Evaluation in Logic Programming. Technical Report CS-87-09. University of Bristol.

Loeckx, J. and K. Sieber (1984). *The Foundations of Program Verification.* Wiley-Teubner.

Maher, M.J. (1988). Equivalences of Logic Programs. In: *Foundations of Deductive Databases and Logic Programming* (J. Minker, Ed.). pp. 627–658. Morgan Kaufmann Publishers.

Milner, R. (1989). *Communication and Concurrency.* Prentice Hall.

Milner, R., J. Parrow and D. Walker (1989). A Calculus of Mobile Processes, Part I. Technical Report ECS-LFCS-89-85. LFCS, U. of Edinburgh.

Nicholson, T. and N. Foo (1989). A Denotational Semantics for Prolog. *TOPLAS* **11**(4), 650–665.

North, N.D. (1986). Prolog, A denotational definition. Technical Report PS/141. National Physical Laboratory.

Plotkin, G. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19. Computer Science Department, Aarhus University.

Plumer, L. (1990). *Termination Proofs for Logic Programs.* Springer-Verlag.

Rogers, H. (1988). *Theory of Recursive Functions and Effective Computability.* MIT Press.

Ross, B.J. (1989). The Partial Evaluation of Imperative Programs Using Prolog. In: *Meta–programming in Logic Programming.* MIT Press.

Ross, B.J. (1991*a*). A Semantic Approach to Prolog Program Analysis. In: *Workshop on Constructing Logic Programs.* Paris, France.

Ross, B.J. (1991*b*). Semantics–based Partial Evaluation of Prolog Programs. In: *Workshop on Logic Program Synthesis and Transformation.* Manchester, England.

Ross, B.J. (1991*c*). Using Algebraic Semantics for Proving Prolog Termination and Transformation. In: *Proc. UK ALP 91.* Springer–Verlag. Edinburgh, Scotland.

Ross, B.J. (n.d.). A π–calculus Semantics of Parallel Prolog. Unpublished manuscript.

Ross, B.J. and A. Smaill (1991). An Algebraic Semantics of Prolog Program Termination. In: *Proc. Eighth International Conference on Logic Programming.* MIT Press. Paris, France. (also see DAI TR #510, U. of Edinburgh, 1990).

Sannella, D. (1988). A Survey of Formal Software Development Systems. Technical Report LFCS-88-56. LFCS, U. of Edinburgh.

Saraswat, V.J. (1989). Concurrent Constraint Programming Languages. PhD thesis. Carnegie Mellon.

Sawamura, H. and T. Takeshima (1985). Recursive Unsolvability of Determinacy, Solvable Cases of Determinacy and Their Applications to Prolog Optimization. In: *Proceedings of the Symposium on Logic Programming.* pp. 200–207.

Sestoft, P. and A.V. Zamulin (1988). Annotated Bibliography on Partial Evaluation and Mixed Computation. *New Generation Computing* **6**(2,3), 309–354.

Sterling, L. and E. Shapiro (1986). *The Art of Prolog.* Prentice–Hall.

Stoy, J. (1977). *Denotational Semantics.* MIT Press.

Tamaki, H. and T. Sato (1983). A Transformation System for Logic Programs which Preserves Equivalence. Technical Report TR-018. ICOT. Tokyo, Japan.

Thom, J.A. and J. Zobel (1988). *NU-Prolog Reference Manual Version 1.3.*

Ueda, K. (1986). Guarded Horn Clauses. PhD thesis. University of Tokyo.

van Emden, M.H. and R.A. Kowalski (1976). The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM* **23**(4), 733–742.

Vasak, T. and J. Potter (1986). Characterisation of Terminating Logic Programs. In: *Symposium on Logic Programming.*

Venken, R. and B. Demoen (1988). A Partial Evaluation System for Prolog: some Practical Considerations. *New Generation Computing* **6**(2,3), 279–290.

Verschaetse, K. and D. De Schreye (1991). Deriving Termination Proofs for Logic Programs, using Abstract Procedures. In: *Proc. Eighth International Conference on Logic Programming.* MIT Press. Paris, France.

Waern, A. (1986). Process models of logic programs: a comparison. Technical Report SICS R86007. Swedish Institute of Computer Science.