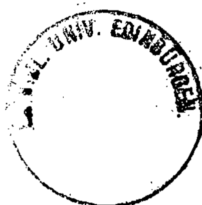
Transforming Imperative Programs

Martin Illsley

Doctor of Philosophy
University of Edinburgh
1988



ABSTRACT

This thesis describes methods for transforming imperative programs. These transformations are semantics preserving and therefore provide a means of producing a correct efficient program from an inefficient but clear program. Although imperative programming languages are more widely used than functional ones, much more transformation work has been done for the latter. This is mainly because of the more complex nature of imperative programming languages.

We extend the usual notion of transformation by introducing a transformation rule, in addition to axioms. This rule is strongly related to the fixed-point characterization rule for the **while** construct. Whereas transformation axioms have side conditions to restrict their instantiations, our transformation rule has a conclusion which is dependent upon another transformation being possible. That is, if A,B,C,D are programs, in addition to the axiom form of transformation, "A transforms to B", we introduce the rule form of transformation, "if A transforms to B then C transforms to D". We generalise this rule to be context-specific.

We have implemented our transformation system, and we give many examples. As a strong indication of the *power* of the system we prove that a subset of it is sufficient to derive the usual Hoare's logic. This involves setting up a correspondence between Hoare triples and semantic equivalences. We also discuss the relationship between our system and the Unfold/Fold (UF) system of Burstall and Darlington. We derive a subset of our system using UF via a translation system, but argue that to be a *fair* comparison we must include a notion of *relatively reasonable* translation.

ACKNOWLEDGEMENTS

I would like to thank my supervisors Rod Burstall and Ian Mason for their great insight and tolerance.

Many other members, past and present, of the Department of Computer Science have been helpful and encouraging. I would especially like to thank the following colleagues and friends: Furio Honsell, Alberto Pettorossi, Eugenio Moggi, Faron Moller, Murray Cole, Laurent Langlois, Marek Bednarczyk, Mads Dam, Andreas Knobel, Jordi Farres-Casals, Pawel Paczkowski and Mads Tofte.

I would also like to thank Bill Scherlis for his enthusiasm and ideas, and all in the Computer Science Department of Carnegie-Mellon University for providing me with such a pleasant atmosphere within which to work, during my time there. Thankyou to Tim Griffin of Cornell University for his expert tuition on using the synthesizer generator.

I am grateful to the SERC for providing finance for the majority of my research, and to Edinburgh University for providing the rest, via a Research Assistantship. Hewlett-Packard Limited, in the form of Robin Gallimore, have also provided much encouragement and support, for which I am very grateful.

Finally, I would like to acknowledge the personal support of my parents, whose encouragement has never wavered.

I dedicate this thesis to G, S, S & R.

Table of Contents

1. Introduction **7**

1.1 General Motivation 7

1.2 Short Overview and Specific Motivation 9

1.3 Outline 12

1.4 Notation 13

2. IF Language System **15**

2.1 Transformations 15

2.2 Normal Form 17

2.3 Completeness 18

2.4 Derivability 19

3. WHILE Language System **21**

3.1 Transformations 22

3.2 Examples 23

3.2.1 A Note on Linearization 24

3.2.2 Examples of Transformations 26

3.3 Context Dependent WHILE rules 33

3.4 Examples using Full System 35

4. Relative Completeness **80**

4.1 Mimicking Hoare's Logic 81

4.2 UF Transformation System 90

4.3 Derivation of W via UF 94

4.4 A Note on Translation 102

5. Implementation **106**

5.1 Motivation 106

5.2 Using the Synthesizer Generator 107

5.3 An Example Dialogue 108

5.4 Implementation Details 115

6. Summary and Further Work **120**

6.1 Summary 120

6.2 Further Work 121

6.2.1 Extending the Theory 121

6.2.2 Extending the Language 123

6.2.3 Extending the Implementation 123

A. Semantics **125**

B. Soundness and Completeness Proofs **133**

B.1 Soundness of $\Leftrightarrow_I$ 133

B.2 Reduction to Normal Form 136

B.3 Completeness of $\Leftrightarrow_N$ for **IF** 137

B.4 Soundness of $\Rightarrow_W$ 139

B.4.1 The IF System Extended 140

B.4.2 The **while** Axioms and Rules 141

C. List of Transformations **146**

C.1 Derived Transformations 149

Bibliography **158**

Chapter 1

Introduction

1.1 General Motivation

It is well known that there is a general demand for reliable software. The ever-increasing uses of critical software in such fields as nuclear power, finance and defence systems indicate the extent of this reliability problem. The root of the problem being, as Darlington sees it ([**Darlington 82**]), that the

> *..design, construction and maintenance of programs is still largely an unmechanised activity and regarded more as an art or a craft than a precise science ... only when the specification and design of programs has been formalised sufficiently to allow computers to assist in this process will adequate standards of accuracy and reliability be achieved.*

More specifically we list below some of the main interrelated problem areas:

- *Clear and Efficient Programming* : How do we write programs that are both clear and efficient? These aims are more often than not incompatible.
- *Specification and Correctness* : Once we have written a program how do we know that it is *correct*, i.e that it meets its specification? Indeed what is a specification?
- *Program Maintenance and Adaptation* : Most of programming time is spent modifying programs which do not meet their specifications, or for which the specifications have changed, and adapting old programs to new situations.

How can this vital process be done in a rigorous manner so that new programs are correct?

Program transformation aims to fulfil these needs in a different way from that of very high level languages, structured programming, classic verification methods, symbolic execution and constructive mathematics. The fundamental transformational methodology rests on the idea of providing a calculus or algebra of programs.

- *Clear and Efficient Programming* : Program transformation enables the programmer to write a clear program free of concerns about efficiency. He can then use transformations to give an efficient version. Provided these transformations are equivalence preserving the programmer is assured that both versions *do the same thing.*

- *Specification and Correctness* : The classical approach is to give a proof of correctness once a program has been written. By the previous point, transformation would require such proofs to be given only for simpler and clearer programs, which are easier to prove. Perhaps more significantly, transformation offers another very promising approach to this problem. The idea here is to generate, via transformation, a program from the formal specification. In fact it has been argued that this latter approach should replace the troublesome and rather unnatural classical methods, see **[Scherlis/Scott 83]**, **[Mason 86]**.

- *Program Maintenance and Adaptation* : The approach here is an extension of the previously mentioned uses of program transformation, and centres around the use of a *program derivation* or *evolution*. This is a *sequence* of insights required to derive an implementation from a straightforward specification. Transformational techniques provide the right kind of basis for program derivations, see **[Scherlis/Scott 83]**.

In short transformational programming covers most, if not all, phases of the classical software engineering life cycle.

1.2 Short Overview and Specific Motivation

As a consequence of there being such a multitude of motivations for transformation systems, there has been a great deal of work done (see **[Partsch/Steinbruggen 83]** and **[Goldberg 86]** for overviews). There are however basically only two distinct approaches to transformation systems.

- **Generative:** A few (powerful) basic transformations are given providing a basis for constructing new transformations. Examples of this approach are given by the unfold/fold system (UF) of **[Burstall/Darlington 77]**, the expression procedure system (EP) of **[Scherlis 80]**, the functional programming system (FP) of **[Backus 78]** (see also **[Backus 81a]**, **[Backus 81b]**), **[Arsac 79]** (see also **[Arsac 85]**) and **[Chusho 80]**. The systems UF and EP both work on a functional recursive equation language, while the system FP uses an applicative type language without variables, built partly with the purpose of its associated algebra being simple but powerful. The systems detailed in **[Arsac 79]** and **[Chusho 80]** both work on imperative type languages, the former on the RE_∞ control structure language, the latter on PASCAL.

- **Catalogue:** Typically a large number of transformations are given with little or no theoretical grounding and structure. Examples are given by **[Smith et al 85]**, **[Standish et al 76]**, **[Maher/Sleeman 83]**, **[Balzer 81]**, **[Loveman 77]** and **[Cheatham et al 81]**.

With generative systems we have the right sort of basis to prove theoretical results concerning the "power" of the basic transformations; catalogue systems have no *fixed* power since the set of transformations is expandable (although it may be argued that they have convergent power, see **[Barstow 85]**). In the practical use of a transformation system we need high-level transformations in addition to the low-level basic transformations given by the generative approach. However rather than proving <u>every</u> new high-level transformation correct, as in the catalogue approach,

it is better to have proven once and for all the foundational transformations and build the (catalogue of) higher level ones up from these. So if we can find a very powerful set of basic transformations then the two approaches will meet peacefully!

In comparison to functional transformation systems, there has been little work done on imperative systems because of the complex nature of imperative languages (see [**Backus 78**]). As Darlington comments (see [**Darlington 82**]),

> *At present one way we see transformation being applied aims at eventually producing an efficient program in a conventional high level language that can be compiled and run in the normal way. However as the nature of these languages preclude any significant transformations being performed after the translation from an applicative language it is important that as much work as possible is done within the applicative language.*

The use of an implicit store, and the subsequent sequentiality, only cause some of the problems. We shall not consider the *hard* problems of side-effects (see [**Mason 86**], [**Mycroft 81**]) and control transfers.

With generative systems it is natural to concentrate, at least initially, on obtaining a powerful set of basic transformations. In the literature few really powerful, yet simple, sets of basic transformations exist; UF and EP are perhaps the only ones. A great deal of work has gone into the necessary task of constructing *strategies* (or *higher level transformations*, see [**Pettorossi 84**], [**Pettorossi/Proietti 88**]). What distinguishes UF from other generative systems is its simplicity and the fact that it uses transformation *rules*, as well as axioms. Transformation rules allow a system to make transformations that are dependent on other transformations being possible. In UF the use of transformation rules is disguised by the inclusion of multiple definitions in the language (i.e duplicate, but consistent, left hand sides of the recursive equations).

Our basic concern is with theoretical results and the heart of any transformation system, its transformations. A starting point for the theory of any formal system is *soundness* and *completeness*. Some transformations have been proven

sound: **[Neilson 81]** proves the correctness of very simple imperative transformations using denotational semantics; **[Huet/Lang 78]** prove the correctness of functional to imperative schematic rewrite rules (see **[Darlington/Burstall 76]**) also using denotational semantics; **[Scherlis 80]** proves the correctness of EP transformations using an evaluator model. Completeness results, i.e sets of transformations that are complete in the sense that any pair of semantically equivalent programs can be transformed into one another using transformations from this set, are unsurprisingly rather thin on the ground. The following completeness results exist in the literature: for straight line code **[Aho/Ullman 72]**; for boolean and conditional expressions **[Bloom/Tindell 83]**, **[McCarthy 63]**, **[Guessarian 85]**; for syntactic transformations, i.e history of computation preserved **[Arsac 79]** (proof in **[Cousineau 77]**); for mixed computation, i.e partial evaluation **[Ershov 82]** (proof in **[Sabelfield 78]**). **[Hoare el al 85]** give an **IF** language completeness result, however the equivalence of normal forms is dependent upon an equivalence system for expressions and the **if** construct is merely pushed into the expressions so that the implied completeness result is really only a slight extension of **[Aho/Ullman 72]**. Because of the limitations of absolute completeness (see **[Kibler 78]**), it is important to study *relative completeness*, i.e to compare the *power* of transformation systems.

Other theoretical work has been done by **[Kott 78]** (see also **[Kott 85]**) on how to preserve termination when using the UF system, **[Pettorossi 84]** on tupling as communication, again in UF, and the use of memoing to extend this idea, and **[Koga 85]** on the efficient use of tupling and how to automate the technique. In the FP system, theoretical work has been more in the line of establishing theorems based on functional identities. **[Williams 82]** extends the expansion theorem, relating functional equations to concrete programs, to include some "non-linear" functions, and **[Harrison/Khoshnevisan 86]** give equivalent imperative programming language loops for a large class of "linear" recursive functions. Transformation systems using assertions (see for example **[Scherlis 80]**, **[Gerhart 75]**, **[van Diepen/de Roever 86]**) are difficult to reason about

because of the formal separation of logic and programs. **[Back 87]** shows how the introduction of context assertions and partial correctness are connected.

1.3 Outline

In this section we give an overview of this thesis, and an outline of the contents of each of the following chapters.

In this thesis we give a transformation system over imperative languages; the vast majority of "real world" programs are written in imperative languages. We choose to transform the **WHILE** language, as described later, because this represents a minimal language for imperative programs.[1] This is a language that most students and programmers can understand easily.

In **chapter 2** a set of transformations for a simple **IF** language is presented. This set is proven to be complete. The transformation system is then extended to cope with the addition of program variables. **Chapter 3** is the main chapter of the thesis. A transformation system for a simple **WHILE** language is introduced, W, extending the **IF** system and language. The usual notion of transformation is extended by introducing transformation rules in addition to axioms. That is, if A,B,C,D are programs, in addition to the axiom form of transformation, "A transforms to B", a rule form of transformation, "if A transforms to B then C transforms to D", is used. This rule form of transformation is generalised to be context-specific. Many examples of the system's use are given in this chapter. In **chapter 4** a subset of W is proven to be sufficient to derive the usual Hoare's logic. This involves setting up a correspondence between Hoare triples and semantic equivalences. This result gives a strong indication of the *power* of the system. A comparison of the W transformation system with UF (see **[Burstall/Darlington 77]**) is also considered, and a subset of W is proven to be derivable via UF. A

---

[1]Using the concepts defined, a program can be written to compute each partial recursive function.

set of transformations with the appropriate *translation property* is used to give a correspondence between the imperative and functional languages. The relationship between translation and transformation is discussed, and a definition of relatively reasonable translation, in order for a comparison of transformation systems to be fair, is proposed. In **chapter 5** the implemented system used for the previous examples is detailed. The implementation uses the synthesizer generator (**[Reps/Teitelbaum 85]**) as its base. An example dialogue is presented. **Chapter 6** is the final chapter, and it is devoted to a summary and areas of possible future research. The **appendices** are split into three. The first part gives a semantics for the simple **WHILE** language, and presents some simple lemmas about its properties. The second part presents the details of some soundness and completeness proofs referred to in the body of the thesis. The third part consists of a collective table of the full W transformation system, and derivations of some higher level transformations built into the implementation.

1.4 Notation

We let L denote a first-order language with equality. We use the letters x, y, z to denote the variables of L, the letters c, d, e to denote the terms (expressions) of L (R^n and f^n denote n-ary predicate and function symbols of terms, respectively), and the letter b to denote a quantifier-free formula (boolean expression) of L. Our imperative language, **WHILE**, is built on top of L in the obvious way, and is simply given by the following BNF definition:

$$w ::= \textbf{skip} \mid (x_1 := c_1 \,\&\ldots\&\, x_n := c_n) \mid w_0;\, w_1 \mid \textbf{if}(b, w_0, w_1) \mid \textbf{while}(b, w)$$

We use ma to denote multiple (simultaneous) assignments, e.g if $x = 0$ then after $(x := x + 1 \& y := x)$ y is 0; the assignments' left hand sides must be distinct. We use **IF** to denote the above language without the $\textbf{while}(b, w)$ clause. We use $\textbf{if}(b, w)$ to abbreviate $\textbf{if}(b, w, \textbf{skip})$, $\bar{x} := \bar{c}$ and $\bar{y} := \bar{d}$ as abbreviations for $(x_1 := c_1 \,\&\ldots\&\, x_n := c_n)$ and $(y_1 := d_1 \,\&\ldots\&\, y_m := d_m)$, and X_n and Y_m as

abbreviations for $\{x_1, \ldots, x_n\}$ and $\{y_1, \ldots, y_m\}$. We also use the usual notion of substitution, where $t[a/b]$ denotes t with all free occurences of b replaced by a.

Chapter 2

IF Language System

In this chapter we shall introduce a transformation system for **IF** programs. This system will be proven to be *complete* in a sense made more precise below. The results we obtain here are only minor extensions on existing ones.

2.1 Transformations

We introduce transformations via schematic axioms, defining a relation $\Leftrightarrow_I$ between **IF** programs. These are given in Fig.2–1. The (; $elim_1$) axiom is a generalised version (for multiple assignments) of the following simple axiom for merging assignments:

$$x := c;\ y := d \ \Leftrightarrow_I \begin{cases} y := d[c/x] & \text{if } x = y \\ x := c \& y := d[c/x] & \text{otherwise.} \end{cases}$$

In addition to the transformations given in Fig.2–1 we also use reflexivity, transitivity, monotonicity and symmetry (given in Fig.2–2). In general, these (type of) transformations will not be detailed again. We shall use the (named) axioms in Fig.2–1 by replacing the left hand sides with the right hand sides. Using the symmetry rule we can also do the reverse. When we do this we shall refer to the transformation by its name with *elim* (*intro*) replaced by *intro* (*elim*). If the name of the transformation does not contain *intro* or *elim* then we refer to the

$(;\ elim_1)$	$\bar{x} := \bar{c};\ \bar{y} := \bar{d}\ \Leftrightarrow_I\ \&_{x_j \in X_n - Y_m} x_j := c_j$ $\&_{y_j \in Y_m} y_j := d_j[c_1/x_1 \ldots c_n/x_n]$
$(;\ elim_2)$	$\bar{x} := \bar{c};\ \mathbf{if}(b, i_0, i_1)\ \Leftrightarrow_I\ \mathbf{if}(b[c_1/x_1 \ldots c_n/x_n], \bar{x} := \bar{c};\ i_0, \bar{x} := \bar{c};\ i_1)$
$(;\ elim_3)$	$\mathbf{if}(b, i_0, i_1);\ i_2\ \Leftrightarrow_I\ \mathbf{if}(b, i_0;\ i_2, i_1;\ i_2)$
$(\wedge\ intro)$	$\mathbf{if}(b_0, \mathbf{if}(b_1, i_0, i_1), i_2)\ \Leftrightarrow_I\ \mathbf{if}(b_0 \wedge b_1, i_0, \mathbf{if}(b_0, i_1, i_2))$
$(skip\ elim_1)$	$\mathbf{skip}\ \Leftrightarrow_I\ x := x$
$(if\ intro_1)$	$ma\ \Leftrightarrow_I\ \mathbf{if}(b, ma, ma)$
$(;\ assoc)$	$i_0;\ (i_1;\ i_2)\ \Leftrightarrow_I\ (i_0;\ i_1);\ i_2$

Figure 2–1: transformations for reduction to normal form

replacement of the left hand side by the right hand side by adding *intro* onto the name, and by adding *elim* onto the name for the reverse.

It is straightforward to prove that the relation $\Leftrightarrow_I$ preserves semantics, i.e $\Leftrightarrow_I$ is a relation between semantically equivalent programs, denoted by $\equiv$ and defined formally in Appendix A.

Theorem 2.1 *(Soundness of $\Leftrightarrow_I$)*
$\forall i_0, i_1 \in \mathbf{IF}$ *if* $i_0\ \Leftrightarrow_I\ i_1$ *then* $i_0 \equiv i_1$

Proof:
see Appendix B □

(*refl*)	$i \Leftrightarrow_I i$
(*trans*)	$\dfrac{i_0 \Leftrightarrow_I i_1,\ i_1 \Leftrightarrow_I i_2}{i_0 \Leftrightarrow_I i_2}$
($mono_1$)	$\dfrac{i_0 \Leftrightarrow_I i_0',\ i_1 \Leftrightarrow_I i_1'}{i_0; i_1 \Leftrightarrow_I i_0'; i_1'}$
($mono_2$)	$\dfrac{i_0 \Leftrightarrow_I i_0',\ i_1 \Leftrightarrow_I i_1'}{\mathbf{if}(b, i_0, i_1) \Leftrightarrow_I \mathbf{if}(b, i_0', i_1')}$
(*symm*)	$\dfrac{i_0 \Leftrightarrow_I i_1}{i_1 \Leftrightarrow_I i_0}$

Figure 2–2: reflexivity, transitivity, monotonicity and symmetry transformations

2.2 Normal Form

In proving completeness we shall use the following *normal* form:

Definition 2.1 *The set of normal form programs, NF, is the smallest set defined inductively as follows:*

- $\mathbf{if}(b, ma, ma') \in NF$
- *if* $n \in NF$ *then* $\mathbf{if}(b, ma, n) \in NF$

We justify the introduction of the above definition by proving that any **IF** program can be *transformed* into a semantically equivalent (by Theorem 2.1) normal form program. This is stated formally as follows,

Lemma 2.1 $\forall i \in \mathbf{IF} \quad \exists n \in NF \ s.t \ i \Leftrightarrow_I n$

Proof:

See Appendix B □

(& *assoc*)	$ma\&(ma'\&ma'') \Leftrightarrow_N (ma\&ma')\&ma''$
(& *ident*)	$ma \;\&\; y := y \Leftrightarrow_N ma$
(& *symm*)	$ma\&ma' \Leftrightarrow_N ma'\&ma$
(*if logic*$_1$)	$\mathbf{if}(b, n, n') \Leftrightarrow_N \mathbf{if}(\neg b, n', n)$
(*if logic*$_2$)	$\mathbf{if}(b_1, ma_1, \mathbf{if}(b_2, ma_2, n)) \Leftrightarrow_N \mathbf{if}(b_1, ma_1, \mathbf{if}(\neg b_1 \wedge b_2, ma_2, n))$
(*exp/bool*)	$\dfrac{b \Rightarrow (c_i \equiv d_i \quad i = 1, \ldots, n)}{\mathbf{if}(b, \bar{x} := \bar{c}, n) \Leftrightarrow_N \mathbf{if}(b, \bar{x} := \bar{d}, n)}$
(*bool*$_1$)	$\dfrac{b \equiv b'}{\mathbf{if}(b, ma, n) \Leftrightarrow_N \mathbf{if}(b', ma, n)}$

Figure 2–3: transformations for equivalence of normal forms

2.3 Completeness

We now define the relation $\Leftrightarrow_N$, over NF, given in Fig.2–3. We again use reflexivity, transitivity, monotonicity and symmetry in addition to the axioms given in Fig.2–3. We overload the symbol $\equiv$ in (*exp/bool*) and (*bool*$_1$) by using it to additionally denote expression equivalence and boolean equivalence respectively. We use $\Rightarrow$ in (*exp/bool*) to denote logical implication. The (*bool*$_1$) axiom enables us to incorporate a (complete) transformation system for the booleans, and (*exp/bool*) enables us to incorporate a transformation system for expressions (under the conditions of a boolean). Soundness is again straightforward, however this time we can also state the following:

Theorem 2.2 *(Completeness of* $\Leftrightarrow_N$*)*
$\forall n_0, n_1 \in NF$ *if* $n_0 \equiv n_1$ *then* $n_0 \Leftrightarrow_N n_1$

Proof:

See Appendix B □

Consider the relation $\Leftrightarrow_T \triangleq \Leftrightarrow_I \cup \Leftrightarrow_N$. We can now give the main result of this section which states that the previously given axioms form a complete set of transformations for **IF**, that is they enable us to transform any **IF** program into any other semantically equivalent **IF** program.

Theorem 2.3 *(Completeness of $\Leftrightarrow_T$)*

$\forall i_0, i_1 \in \mathbf{IF}$ *if* $i_0 \equiv i_1$ *then* $i_0 \Leftrightarrow_T i_1$

Proof:

The proof follows as a consequence of Lemma 2.1, Theorem 2.2 and symmetry □

2.4 Derivability

Consider the following axiom:

$$\mathbf{if}(b, \mathbf{if}(b, i_0, i_1), i_2) \Leftrightarrow_T \mathbf{if}(b, i_0, i_2)$$

This is not one of our primitive axioms, and although we can derive all ground instances of this axiom, since we have completeness, we cannot derive this actual schematic axiom. To solve this problem we want more than a transformation system, we want a system that can generate higher level transformations. How then do we express the desire to do this? What do we need to add to enable our system to do it?

Syntactically we need to add program variables to our language, giving $\mathbf{IF}^{\circ}$. Note that previously we were able to write such axioms by using metavariables. We extend $\Leftrightarrow_T$ with the transformations given in Fig.2–4. The rather obvious axioms (*if elim*$_2$) and (*if elim*$_3$) were not needed previously because of (*exp*/*bool*),

(*skip elim*$_2$)	**skip**; $p \Leftrightarrow_T p$
(*skip elim*$_3$)	p; **skip** $\Leftrightarrow_T p$
(*if elim*$_2$)	**if**$(true, p, p') \Leftrightarrow_T p$
(*if elim*$_3$)	**if**$(false, p, p') \Leftrightarrow_T p'$

Figure 2–4: transformations to deal with program variables

and (*skip elim*$_2$), (*skip elim*$_3$) were not needed as we could always use (*skip elim*$_1$) and propagate the assignment.

Extending $\Leftrightarrow_T$ as in Fig.2–4 we have a normal form lemma and completeness theorem for the **IF**^{o} language similar to those in the previous section.

Chapter 3

WHILE Language System

This is the main chapter of the thesis. Here we introduce our imperative transformation system for the **WHILE** language. It is obviously based on the **IF** language system, but also on the general idea of the functional UF system of [**Burstall/Darlington 77**]. Using the **IF** transformations alone will not allow us to do anything significant. Nontrivial **WHILE** programs can be seen as infinitely nested **if** statements, and to transform these infinite objects we need some induction. We therefore include in this chapter a transformation rule which is directly based on fixpoint induction. As stated previously, our approach is not that of a haphazard catalogue style, but rather to use the *minimum* of rules to give *maximum* power. Since a **while** is a possibly infinite nesting of **if** statements, intuitively all we need to add to the **IF** language system are transformations for *unfolding* and *folding* this nesting, hence the analogy with UF.

We shall gradually introduce the new aspects of the full system, as compared to the **IF** system. We shall extend the usual notion of transformation by introducing a transformation rule, in addition to axioms. Such a rule enables the applicability of a transformation to be dependent upon other transformations being possible. That is, a transformation rule is of the form "if A transforms to B then C transforms to D". We shall generalise this rule so that the dependence upon other transformations may be context-specific. It is this rule form of transformation that provides the real *power* of our transformation system.

(*while logic*)	$\mathbf{while}(b,p); \mathbf{if}(\neg b, p_0, p_1) \Leftrightarrow_W \mathbf{while}(b,p); p_0$
(*while unfold*)	$\mathbf{while}(b,p) \Leftrightarrow_W \mathbf{if}(b, p; \mathbf{while}(b,p))$
(*while fold*)	$\dfrac{p \Rightarrow_W \mathbf{if}(b, p_1; p)}{p \Rightarrow_W \mathbf{while}(b, p_1)}$

Figure 3–1: transformations for **while**

We give many examples of the use of the system. These examples use an implementation which we discuss in a later chapter.

3.1 Transformations

In this section we define the relation $\Leftrightarrow_W$ over **WHILE**. The relation $\Leftrightarrow_W$ has as its base the relation $\Leftrightarrow_T$, with the universal quantification extended from **IF** to **WHILE**. The two additional axioms of the system, over the **IF** system, and a degenerate case of the additional rules are given in Fig.3–1. Again we include reflexivity, transitivity and monotonicity, but include symmetry only for $\Leftrightarrow_W$. The relation $\Rightarrow_W$, used in (*while fold*), is related to $\Leftrightarrow_W$ as follows: $a \Rightarrow_W b$ and $b \Rightarrow_W a$ *iff* $a \Leftrightarrow_W b$. Consequently the **WHILE** transformation system can be seen as being defined by the $\Rightarrow_W$ relation, using $\Leftrightarrow_W$ as an abbreviation. The (*while unfold*) axiom can be found in other systems (see for example **[Arsac 79]**, **[Pepper 79]**), but the (*while fold*) rule appears to be new as a transformation. The (*while fold*) rule is directly related to the least fixed-point characterization of the **while** statement (see **[deBakker 80]**). The soundness of (*while unfold*), with respect to strong equivalence, is obvious and requires no explanation, and (*while logic*) merely establishes the fact that the conditional of a **while** is false after the **while** has terminated. The (*while fold*) rule expresses in its premiss the ability to transform p into $\mathbf{if}(b, p_1;\ p)$. Using monotonicity this process can be repeated within the **if** statement. The conclusion is that we can transform p into

the **while** statement shown. The (*while fold*) rule does not preserve termination. Correspondingly, a transformation $p \Rightarrow_W p'$ does not preserve strong equivalence, but only weak equivalence $\sqsubseteq$. An example showing that this is necessary is given by setting $p_1 \triangleq$ **skip** and $b \triangleq$ *true* in (*while fold*). With the exception of the monotonicity, transitivity and symmetry rules, (*while fold*) is different to previous transformations in that it is a *rule* rather than an *axiom*. Although (*while fold*) only appears to introduce **while**s, it can in fact eliminate them, for example it can merge two loops into one. The system cannot always eliminate **while**s, namely it cannot eliminate a solitary **while**. We shall discuss this *restriction* later. The terms *unfold* and *fold* come by analogy with the UF transformation system (see [**Burstall/Darlington 77**]) detailed in Chapter 4.

Soundness proofs for these axioms and rules are more complicated than for the **IF** language as non-termination is involved. Details are given in Appendix B.

3.2 Examples

We now give some trivial examples to illustrate the use of (*while fold*) and (*while unfold*), and to show that generalisations of (*while fold*) are required to deal with *context-dependent* situations. These example *proofs* (or *derivations*), and those in section 3.4, have been done via an implemented system, detailed in Chapter 5. The format is a list of programs separated by the set of transformations used at each derivation step. The implementation uses $\Rightarrow_W$ throughout. Transformations for the booleans are used in addition to the transformations defining $\Rightarrow_W$. These boolean transformations are named via generic instances of the corresponding equivalences, e.g ($\neg b \wedge b \equiv$ *false*) is a transformation replacing an instance of $\neg b \wedge b$ by (in general, the corresponding instance of) *false*. The presentation of the examples in this thesis is a postprocessed LaTeX form of the implementation display. A slightly different syntax to the rest of the thesis is used in the example proofs, e.g **while** b **do** p **od** is used instead of **while**(b, p).

3.2.1 A Note on Linearization

Our implementation uses several pieces of notation in order to linearize the derivations, turning a tree into a linear sequence of steps separated by $\Rightarrow_W$'s; we need to explain these notations.

Proofs using just axioms of the form $A \Rightarrow_W B$ (where A and B are programs) and transitivity of $\Rightarrow_W$ are naturally written linearly, i.e

$$\dfrac{\dfrac{A \Rightarrow_W B \quad B \Rightarrow_W C}{A \Rightarrow_W C} \quad C \Rightarrow_W D}{A \Rightarrow_W D}$$

is now

$$\begin{array}{c} A \\ \Rightarrow \\ B \\ \Rightarrow \\ C \\ \Rightarrow \\ D \end{array}$$

The use of the monotonicity rules in proofs is linearized by using modules, written as "<<>>". If $\mathcal{C}$ is some context, then

$$\dfrac{A \Rightarrow_W \mathcal{C}[B] \quad \dfrac{B \Rightarrow_W D}{\mathcal{C}[B] \Rightarrow_W \mathcal{C}[D]}}{A \Rightarrow_W \mathcal{C}[D]}$$

is now

$$
\begin{array}{l}
A \\
\Rightarrow \\
\mathcal{C}[<< B >>] \\
<< * >> ---> \\
\qquad B \\
\qquad \Rightarrow \\
\qquad D \\
<--- << * >> \\
\mathcal{C}[D]
\end{array}
$$

The use of the (*while fold*) rule in proofs is linearized by taking advantage of it's special form, so that

$$\frac{A \Rightarrow_W B}{A \Rightarrow_W C}$$

is now

$$
\begin{array}{c}
A \\
\Rightarrow \\
B \\
\Rightarrow \\
C
\end{array}
$$

Because of this linearization, the use of the (*while fold*) rule in conjunction with transitivity hides A. This makes it difficult for someone reading a proof to deduce where (*while fold*) was used. We overcome this difficulty by using labels and comments referring to labels; this notation has been added to the postprocessed LaTeX form of the implementation display by hand.

Remark: Transformational proofs are naturally thought of in linear form, and so the usefulness of axioms in proofs is rather more obvious than the usefulness of rules. The (*while fold*) rule is not so natural in linear form, and consequently was not an obviously useful addition.

3.2.2 Examples of Transformations

Example 3.1

$\boxed{\textbf{if}(b, p_1, \textbf{while}(b, p_2)) \;\Leftrightarrow_W\; \textbf{if}(b, p_1, \textbf{skip})}$

Proof:

if b **then** p_1 **else while** b **do** p_2 **od fi**

$\Rightarrow$ (*while unfold*)

if b **then** p_1 **else if** b **then** p_2; **while** b **do** p_2 **od else skip fi fi**

$\Rightarrow$ (*if logic*$_2$ *intro*), $(\neg\, b \;\wedge\; b \;\equiv\; \mathit{false})$

if b **then** p_1 **else if** *false* **then** p_2; **while** b **do** p_2 **od else skip fi fi**

$\Rightarrow$ (*if elim*$_3$)

if b **then** p_1 **else skip fi**

□

Example 3.2 *(Combining Special Loops)*

$\boxed{\textbf{while}(b_1, p); \textbf{while}(b_2, p; \textbf{while}(b_1, p)) \;\Rightarrow_W\; \textbf{while}(b_1 \vee b_2, p)}$

Proof:

while b_1 **do** p **od** ; **while** b_2 **do** p; **while** b_1 **do** p **od od** (3.2.1)

$\Rightarrow$ (*while unfold*)

if b_1 **then** p; **while** b_1 **do** p **od else skip fi** ;
while b_2 **do** p; **while** b_1 **do** p **od od**

$\Rightarrow$ (; *elim*$_3$), (*if logic*$_1$ *intro*), (*skip elim*$_2$), (*while unfold*)

if $\neg\, b_1$
  then if b_2
    then p; **while** b_1 **do** p **od** ; **while** b_2 **do** p; **while** b_1 **do** p **od od**

else skip

fi

else p; **while** b_1 **do** p **od** ; **while** b_2 **do** p; **while** b_1 **do** p **od od**

fi

$\Rightarrow$ (*if logic*$_1$ *intro*)

if $\neg\, b_1$

then if $\neg\, b_2$

then skip

else p; **while** b_1 **do** p **od** ; **while** b_2 **do** p; **while** b_1 **do** p **od od**

fi

else p; **while** b_1 **do** p **od** ; **while** b_2 **do** p; **while** b_1 **do** p **od od**

fi

$\Rightarrow$ ($\wedge$ *intro*), (*if logic*$_1$ *intro*), (*if elim*$_1$), (; *assoc*)

if $\neg\,(\neg\, b_1 \;\wedge\; \neg\, b_2)$

then p; **while** b_1 **do** p **od** ; **while** b_2 **do** p; **while** b_1 **do** p **od od**

else skip

fi

$\Rightarrow$ $(\neg(b_1 \wedge b_2) \equiv \neg b_1 \vee \neg b_2), (\neg\neg b \equiv b), (\neg\neg b \equiv b)$

if $(b_1 \vee b_2)$

then p; **while** b_1 **do** p **od** ; **while** b_2 **do** p; **while** b_1 **do** p **od od**

else skip

fi

$\Rightarrow$ (*while fold*) applied to[1] (3.2.1)

while $(b_1 \vee b_2)$ **do** p **od**

□

---

[1]By applied to we mean taking (3.2.1) as p in (*while fold*)

Example 3.3

$\boxed{\mathbf{if}(b_1, \mathbf{while}(b_2, p), \mathbf{skip}) \Rightarrow_W \mathbf{while}(b_1, \mathbf{while}(b_2, p))}$

Proof:

if b_1 **then while** b_2 **do** p **od else skip fi** (3.3.1)

$\Rightarrow$ (*skip intro*$_3$), (*if intro*$_1$), (*if intro*$_1$), (*if intro*$_3$)

```
if b1
  then while b2 do p od ;
     if b1
       then if b2
              then skip
              else if false then p; while b2 do p od else skip fi
               fi
       else skip
     fi
  else skip
fi
```

$\Rightarrow$ ($false \equiv b \wedge \neg b$), ($b_0 \wedge b_1 \equiv b_1 \wedge b_0$)

```
if b1
  then while b2 do p od ;
     if b1
       then if b2
              then skip
              else if (¬ b2 ∧ b2)
                     then p; while b2 do p od
                     else skip
                     fi
          fi
       else skip
     fi
  else skip
fi
```

$\Rightarrow$ (*if logic*$_2$ *elim*), (*if logic*$_1$ *intro*), (*while unfold*)

if b_1
  then while b_2 **do** p **od** ;
    if b_1
      then if $\neg$ b_2 **then while** b_2 **do** p **od else skip fi**
      else skip
    fi
  else skip
fi

$\Rightarrow$ ($\wedge$ *intro*), (*if elim*$_1$)

if b_1
  then while b_2 **do** p **od** ;
    if (b_1 $\wedge$ $\neg$ b_2) **then while** b_2 **do** p **od else skip fi**
  else skip
fi

$\Rightarrow$ ($b_0 \wedge b_1 \equiv b_1 \wedge b_0$), ($\wedge$ *elim*), (*while logic elim*)

if b_1
  then while b_2 **do** p **od** ; **if** b_1 **then while** b_2 **do** p **od else skip fi**
  else skip
fi

$\Rightarrow$ (*while fold*) applied to (3.3.1)

while b_1 **do while** b_2 **do** p **od od**

□

Example 3.4

$\boxed{\text{while}(b, \text{while}(b, p)) \Rightarrow_W \text{while}(b, p)}$

Proof:

while b **do while** b **do** p **od od**

⇒ (*while unfold*), (*if logic*$_1$ *intro*), (*while unfold*), (*if logic*$_1$ *intro*)

```
if ¬ b
  then skip
  else if ¬ b then skip else p; while b do p od fi ;
     while b do while b do p od od
fi
```

⇒ (; *elim*$_3$)

```
if ¬ b
  then skip
  else if ¬ b
       then skip; while b do while b do p od od
       else p; while b do p od ; while b do while b do p od od
       fi
fi
```

⇒ (*if logic*$_2$ *intro*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$, $(\neg\neg b \equiv b)$

```
if ¬ b
  then skip
  else if (¬ b ∧ b)
       then skip; while b do while b do p od od
       else p; while b do p od ; while b do while b do p od od
       fi
fi
```

⇒ $(\neg b \wedge b \equiv$ *false*$)$

```
if ¬ b
  then skip
  else if false
       then skip; while b do while b do p od od
       else p; while b do p od ; while b do while b do p od od
       fi
fi
```

```
⇒ (if elim3), (; assoc)

if ¬ b
  then skip
  else p; << while b do p od ; while b do while b do p od od >>
fi

<< * >> --- >
    while b do p od ; while b do while b do p od od                    (3.4.1)
    ⇒ (while unfold)

    if b then p; while b do p od else skip fi ;
    while b do while b do p od od
    ⇒ (; elim3), (skip elim2)

    if b
      then p; while b do p od ; while b do while b do p od od
      else while b do while b do p od od
    fi
    ⇒ (while unfold)

    if b
      then p; while b do p od ; while b do while b do p od od
      else if b
             then while b do p od ; while b do while b do p od od
             else skip
           fi
      fi
    ⇒ (if logic2 intro), (¬ b ∧ b ≡ false)

    if b
      then p; while b do p od ; while b do while b do p od od
      else if false
             then while b do p od ; while b do while b do p od od
             else skip
```

fi

fi

⇒ (; *assoc*), (*if elim*$_3$)

if b

then p; **while** b **do** p **od** ; **while** b **do** **while** b **do** p **od** **od**

else skip

fi

⇒ (*while fold*) applied to (3.4.1)

while b **do** p **od**

< - - - << * >>

⇒ (*if logic*$_1$ *elim*)

if b **then** p; **while** b **do** p **od** **else skip fi**

⇒ (*while unfold*)

while b **do** p **od**

□

Examples 3.2, 3.3 and 3.4, which use (*while fold*) are of a special type since the resulting programs are a single **while**. Hence the **while** is formed regardless of context. Example 3.5, below, illustrates the need to form **while**s that are context-dependent. To enable us to do these types of examples we need to generalise the (*while fold*) rule above; we do this in the next section.

Example 3.5 *Consider*

$$\textbf{skip};\ s := s+1;\ \textbf{while}(x \geq 0, s := s+1;\ x := x-1)$$

This can be transformed into

$$\textbf{if}(x \geq 0, s := s+1;\ x := x-1;\ \textbf{skip});\ s := s+1;\ \textbf{while}(x \geq 0, s := s+1;\ x := x-1)$$

Now although this is equivalent to

$$\textbf{while}(x \geq 0, s := s+1;\ x := x-1);\ s := s+1;\ \textbf{while}(x \geq 0, s := s+1;\ x := x-1)$$

it cannot be transformed into such by simply using (while fold) since without the context $C[p] \triangleq p;\ s := s+1;\ \textbf{while}(x \geq 0, s := s+1;\ x := x-1)$, **skip** *could* *not* *be transformed into* $\textbf{if}(x \geq 0, s := s+1;\ x := x-1;\ \textbf{skip})$ *(since they are not equivalent).*

3.3 Context Dependent WHILE rules

In this section we consider generalising (*while fold*) to be context-specific, i.e we consider the formation of **while** in context situations. We shall denote a general context by $\mathcal{C}[]$. We aim to give rules of the form (for particular $\mathcal{C}$):

$$(form_1) \qquad \frac{\mathcal{C}[p] \Rightarrow_W \mathcal{C}[\textbf{if}(b_1, p_1;\ p)],\ P(\mathcal{C}, p, p_1, b_1)}{\mathcal{C}[p] \Rightarrow_W \mathcal{C}[\textbf{while}(b_1, p_1)]}$$

$P(\mathcal{C}, p, p_1, b_1)$ is another transformation premiss, possibly involving schemas $\mathcal{C}, p, p_1, b_1$. We must have such a premiss in certain instances to make the above sound. Consider the following program:

$$x := 0;\ \textbf{while}(y \geq x, x := x+2;\ y := y-1) \qquad (A)$$

Using sound axioms this can be transformed into:

$$x := 0;\ \textbf{if}(y \geq 0, x := x+2;\ y := y-1;\ \textbf{while}(y \geq x, x := x+2;\ y := y-1))$$

Hence using ($form_1$) without an extra premiss we could transform this into:

$$x := 0;\ \textbf{while}(y \geq 0, x := x+2;\ y := y-1) \qquad (B)$$

But (A) is not semantically equivalent to (B) (consider an initial value of $y = 2$, (A) gives $x = 2$ while (B) gives $x = 6$).

We shall consider instances of ($form_1$), and give premisses $P(-)$ such that the rules are sound. (*while fold*) is the simplest, which we have already seen,

(*while fold in context*$_1$)

$$\frac{\mathbf{if}(b,p,p_0) \Rightarrow_W \mathbf{if}(b,\mathbf{if}(b_1,p_1;\ p),p_0),\ \{b\}\mathbf{if}(b_1,p_1)\{b\}}{\mathbf{if}(b,p,p_0) \Rightarrow_W \mathbf{if}(b,\mathbf{while}(b_1,p_1),p_0)}$$

(*while fold in context*$_2$)

$$\frac{p;\ p_2 \Rightarrow_W \mathbf{if}(b_1,p_1;\ p);\ p_2}{p;\ p_2 \Rightarrow_W \mathbf{while}(b_1,p_1);\ p_2}$$

(*while fold in context*$_3$)

$$\frac{p_0;\ p \Rightarrow_W p_0;\mathbf{if}(b_1,p_1;\ p),\ p_0;\ p \Leftrightarrow_W \mathbf{if}(b_1,p_1;\ p_0;\ p,p_0)}{p_0;\ p \Rightarrow_W p_0;\ \mathbf{while}(b_1,p_1)}$$

Figure 3–2: while fold in context transformations

(*while fold in context*$_1$) (*while fold in context*$_2$) and (*while fold in context*$_3$) are given in Fig.3–2.

The first premiss of (*while fold in context*$_1$) gives the initial stage of formation of the **while** (when seen as a possibly infinite nesting of **if**s), in the context created by $\mathbf{if}(b,-,p_0)$. The second states that this context should be invariant with respect to the created **while**, i.e that we can continue the formation process given by the first premiss. The second premiss is written as a Hoare triple, as in this notation it is easier to state and more familiar, but we may consider it (equivalently, see Section 4.1) as the following transformation:

$$\forall S_1.\ \mathbf{if}(b,\mathbf{if}(b_1,p_1);\mathbf{if}(b,S_1)) \Leftrightarrow_W \mathbf{if}(b,\mathbf{if}(b_1,p_1);\ S_1)$$

Our second context rule, (*while fold in context*$_2$), is a great deal simpler. Here the use of a trailing context, p_2, imposes no extra restriction on the formation of a **while** over that of (*while fold*). That is, we require no $P(-)$ in (*form*$_1$) to make it sound for this particular instance. The final context rule, (*while fold in context*$_3$), is concerned with context as created by a preceding program statement p_0. The second premiss of (*while fold in context*$_3$) enables the first premiss to be used repeatedly (in a similar fashion to (*while fold in context*$_1$)). Unlike (*while fold in context*$_1$) however, the second premiss of (*while fold in context*$_3$) is required to *rebuild* the context it destroys. The previous example shows that the second premiss of this rule is needed.

These 'fold rules' taking account of context are not known to be complete in any sense. However, they are syntax-directed on the context and not just randomly chosen.

3.4 Examples using Full System

The (extended) **IF** axioms, (*while logic*), (*while unfold*), (*while fold*), (*while fold in context*$_1$), (*while fold in context*$_2$), and (*while fold in context*$_3$) constitute our full transformation system; a collective list of the full system is given in Appendix C. We include the degenerate rule (*while fold*) in this list, and in the implementation. We use the transformational form of (*while fold in context*$_1$), rather than the form using Hoare triples (we shall prove in the next section that either form can be used equivalently). With this transformation system we have a great deal of power, as we now show via some more substantial examples. These examples are longer than the previous examples of section 3.2. To make them more readable we use an unimplemented abbreviation mechanism; ellipses are used to abbreviate repeated parts of programs, as indicated by vertical lines in the right hand margin. In these examples we shall feel free to use higher level transformations (more specifically, derived **IF** axioms (*deriv*$_1$), (*deriv*$_2$), (*deriv*$_4$), (*deriv*$_5$), (*deriv*$_6$), (*deriv*$_7$), (*if logic*$_3$) and ($\vee$ *elim*); see Appendix C for details) and data axioms about the relations, functions and objects we use, e.g $(x + 1 > x) \equiv true$. The data axioms may not even form a *minimal* set, we just add what we need (within reason), when we need it. To linearize the rules (*while fold in context*$_1$) and (*while fold in context*$_3$) we incorporate a proof structure notation, "$*****$", to display what must be proven and the proof. So that

$$\frac{A \Rightarrow_W B \quad C \Leftrightarrow_W D}{A \Rightarrow_W E}$$

is now

A

$\Rightarrow$

B

$***\,*$

C

$\Rightarrow$

D

PROOF

$***\,*$

C

$\Rightarrow$

D

END OF (while fold in context) PROOF

$***\,*$

$\Rightarrow$

E

The *proof* premiss of both (*while fold in context*$_1$) and (*while fold in context*$_3$) requires $\Leftrightarrow_W$, rather than just $\Rightarrow_W$. Whether this is true or not is not checked by the system[2]. The notation "< *program* >", which only appears in example 3.10, is used as a schema variable for programs which is to be instantiated.

Example 3.6 *(Moving past Loops)*

	$s := s + 1;$ **while**$(x \geq 0, s := s + 1; x := x - 1)$
$\Rightarrow_W$	**while**$(x \geq 0, s := s + 1; x := x - 1); s := s + 1$

Proof:

[s := (s + 1)];
while ($x \geq 0$) **do** [s := (s + 1) & x := (x − 1)] **od**

---

[2]In general, the implementation lacks error checking routines and user friendly features.

$\Rightarrow$ (*skip intro*$_2$)

skip;
[s := (s + 1)];
while ($x \geq 0$) **do** [s := (s + 1) & x := (x − 1)] **od** (3.6.1)

$\Rightarrow$ (*skip elim*$_2$), (*while unfold*)

[s := (s + 1)];
if ($x \geq 0$)
  then [s := (s + 1) & x := (x − 1)];
    while ($x \geq 0$) **do** [s := (s + 1) & x := (x − 1)] **od**
  else skip
fi

$\Rightarrow$ (*if intro*$_3$)

[s := (s + 1)];
if ($x \geq 0$)
  then [s := (s + 1) & x := (x − 1)];
    while ($x \geq 0$) **do** [s := (s + 1) & x := (x − 1)] **od**
  else if *false*
      then [s := (s + 1) & x := (x − 1)];
        while ($x \geq 0$)
          do
            [s := (s + 1) & x := (x − 1)]
          od
      else skip
      fi
fi

$\Rightarrow$ (*if logic*$_3$ *intro*), ($b_0 \vee b_1 \equiv b_1 \vee b_0$)

[s := (s + 1)];
if ($x \geq 0$)
  then [s := (s + 1) & x := (x − 1)];
    while ($x \geq 0$) **do** [s := (s + 1) & x := (x − 1)] **od**
  else if (($x \geq 0$) $\vee$ *false*)... **fi**

fi

$\Rightarrow$ ($b \vee false \equiv b$), (; $elim_2$), (; *assoc*), (*while unfold*)

if $(x \geq 0)$
  then [$s := (s + 1)$]; [$s := (s + 1)$ & $x := (x - 1)$];
    while $(x \geq 0)$ **do** [$s := (s + 1)$ & $x := (x - 1)$] **od**
  else [$s := (s + 1)$];
    while $(x \geq 0)$ **do** [$s := (s + 1)$ & $x := (x - 1)$] **od**
fi

$\Rightarrow$ (; $elim_1$)

if $(x \geq 0)$
  then [$s := ((s + 1) + 1)$ & $x := (x - 1)$];
    while $(x \geq 0)$ **do** [$s := (s + 1)$ & $x := (x - 1)$] **od**
  else ...
fi

$\Rightarrow$ (; $intro_1$)

if $(x \geq 0)$
  then [$s := (s + 1)$ & $x := (x - 1)$]; [$s := (s + 1)$];
    while $(x \geq 0)$ **do** [$s := (s + 1)$ & $x := (x - 1)$] **od**
  else ...
fi

$\Rightarrow$ (; *assoc*), (*skip* $intro_2$), (; $intro_3$), (*skip* $intro_3$)

if $(x \geq 0)$
  then [$s := (s + 1)$ & $x := (x - 1)$]; **skip**
  else skip
fi ;
[$s := (s + 1)$];
while $(x \geq 0)$ **do** [$s := (s + 1)$ & $x := (x - 1)$] **od**

$\Rightarrow$ (*while fold in* $context_2$) applied to (3.6.1), (*while unfold*)

while $(x \geq 0)$ **do** [$s := (s + 1)$ & $x := (x - 1)$] **od** ;
[$s := (s + 1)$];

if $(x \geq 0)$
  then [$s := (s + 1)$ & $x := (x - 1)$];
    while $(x \geq 0)$ **do** [$s := (s + 1)$ & $x := (x - 1)$] **od**
  else skip
fi

$\Rightarrow$ (; *elim*$_2$), (*if logic*$_1$ *intro*)

while $(x \geq 0)$ **do** [$s := (s + 1)$ & $x := (x - 1)$] **od** ;
if $\neg (x \geq 0)$
  then [$s := (s + 1)$]; **skip**
  else [$s := (s + 1)$];
    [$s := (s + 1)$ & $x := (x - 1)$];
    while $(x \geq 0)$ **do** [$s := (s + 1)$ & $x := (x - 1)$] **od**
fi

$\Rightarrow$ (*while logic elim*), (*skip elim*$_3$)

while $(x \geq 0)$ **do** [$s := (s + 1)$ & $x := (x - 1)$] **od** ;
[$s := (s + 1)$]
□

The next example is the reverse of Example 3.2.

Example 3.7 *(Splitting Special Loops,* **[de Bakker 80]***)*

$\text{while}(b_1 \vee b_2, p) \Rightarrow_W \text{while}(b_1, p); \text{while}(b_2, p; \text{while}(b_1, p))$

Proof:

while $(b_1 \vee b_2)$ **do** p **od**

$\Rightarrow$ (*while unfold*), (*if intro*$_3$)

if $(b_1 \vee b_2)$
  then p; **while** $(b_1 \vee b_2)$ **do** p **od**
  else if *false* **then** p; **while** $(b_1 \vee b_2)$ **do** p **od** **else skip fi**
fi

$\Rightarrow$ $(false \equiv b \land \neg b), (b_0 \land b_1 \equiv b_1 \land b_0)$

```
if (b1 ∨ b2)
   then p; while (b1 ∨ b2) do p od
   else if (¬ (b1 ∨ b2) ∧ (b1 ∨ b2))
          then p; while (b1 ∨ b2) do p od
          else skip
          fi
fi
```

$\Rightarrow$ (*if logic*$_2$ *elim*), (*skip intro*$_2$), (*while unfold*)

```
if (b1 ∨ b2)
   then p; while (b1 ∨ b2) do p od
   else skip; while (b1 ∨ b2) do p od
fi
```

$\Rightarrow$ (; *intro*$_3$), ($\lor$ *elim*)

if b_1 **then** p **else if** b_2 **then** p **else skip fi fi** ; **while** $(b_1 \lor b_2)$ **do** p **od** (3.7.1)

$\Rightarrow$ (*if logic*$_3$ *intro*)

if b_1 **then** p **else if** $(b_2 \lor b_1)$ **then** p **else skip fi fi** ;
while $(b_1 \lor b_2)$ **do** p **od**

$\Rightarrow$ (; *elim*$_3$), (; *elim*$_3$), (*skip elim*$_2$), (*while unfold*)

```
if b1
   then p; while (b1 ∨ b2) do p od
   else if (b2 ∨ b1)
          then p; while (b1 ∨ b2) do p od
          else if (b1 ∨ b2)
                 then p; while (b1 ∨ b2) do p od
                 else skip
                 fi
          fi
fi
```

$\Rightarrow$ (*if logic*$_2$ *intro*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$, $(b_0 \vee b_1 \equiv b_1 \vee b_0)$, $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$

```
if b1
  then p; while (b1 ∨ b2) do p od
  else if (b2 ∨ b1)
         then p; while (b1 ∨ b2) do p od
         else if (¬ (b1 ∨ b2) ∧ (b1 ∨ b2))...fi
       fi
fi
```

$\Rightarrow$ $(\neg b \wedge b \equiv \mathit{false})$, (*if elim*$_3$), $(b_0 \vee b_1 \equiv b_1 \vee b_0)$

```
if b1
  then p; while (b1 ∨ b2) do p od
  else if (b1 ∨ b2) then p; while (b1 ∨ b2) do p od else skip fi
fi
```

$\Rightarrow$ (*while unfold*), (*if intro*$_3$), (*while unfold*)

```
if b1
  then p;
     if (b1 ∨ b2)
       then p; while (b1 ∨ b2) do p od
       else if false then p; while (b1 ∨ b2) do p od else skip fi
     fi
  else while (b1 ∨ b2) do p od
fi
```

$\Rightarrow$ $(\mathit{false} \equiv b \wedge \neg b)$, $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$

```
if b1
  then p;
     if (b1 ∨ b2)
       then p; while (b1 ∨ b2) do p od
       else if (¬ (b1 ∨ b2) ∧ (b1 ∨ b2))
              then p; while (b1 ∨ b2) do p od
              else skip
              fi
```

```
        fi
    else while (b1 ∨ b2) do p od
fi
```

⇒ (*if logic*$_2$ *elim*), (*skip intro*$_2$), (*while unfold*)

```
if b1
    then p;
        if (b1 ∨ b2)
            then p; while (b1 ∨ b2) do p od
            else skip; while (b1 ∨ b2) do p od
        fi
    else while (b1 ∨ b2) do p od
fi
```

⇒ (; *intro*$_3$), (; *assoc*)

```
if b1
    then p; if (b1 ∨ b2) then p else skip fi ; while (b1 ∨ b2) do p od
    else while (b1 ∨ b2) do p od
fi
```

⇒ (*skip intro*$_2$), (; *intro*$_3$)

```
if b1 then p; if (b1 ∨ b2) then p else skip fi else skip fi ;
while (b1 ∨ b2) do p od
```

⇒ (∨ *elim*)

```
if b1 then p; if b1 then p else if b2 then p else skip fi fi else skip fi ;
while (b1 ∨ b2) do p od
```

⇒ (*while fold in context*$_2$) applied to (3.7.1), (*while logic intro*)

```
while b1 do p od ;
<< if ¬ b1 then while (b1 ∨ b2) do p od else skip fi >>
```

```
<< * >> --->
    if ¬ b1 then while (b1 ∨ b2) do p od else skip fi        (3.7.3)
```

$\Rightarrow$ (*while unfold*), ($\vee$ *elim*), (*if logic*$_1$ *intro*)

```
if ¬ b1
  then if ¬ b1
         then if b2
                then p; while (b1 ∨ b2) do p od
                else skip
                fi
         else p; while (b1 ∨ b2) do p od
     fi
  else skip
fi
```

$\Rightarrow$ (*deriv*$_1$)

```
if ¬ b1
  then if b2 then p; << while (b1 ∨ b2) do p od >> else skip fi
  else skip
fi
```

```
<< * >> --->
    while (b1 ∨ b2) do p od
```

$\Rightarrow$ (*while unfold*), (*if intro*$_3$)

```
    if (b1 ∨ b2)
      then p; while (b1 ∨ b2) do p od
      else if false
             then p; while (b1 ∨ b2) do p od
             else skip
             fi
    fi
```

$\Rightarrow (false \equiv b \wedge \neg b), (b_0 \wedge b_1 \equiv b_1 \wedge b_0)$

```
    if (b1 ∨ b2)
      then p; while (b1 ∨ b2) do p od
      else if (¬ (b1 ∨ b2) ∧ (b1 ∨ b2))
```

then p; **while** $(b_1 \vee b_2)$ **do** p **od**

else skip

fi

fi

$\Rightarrow$ (*if logic*$_2$ *elim*),(*skip intro*$_2$),(*while unfold*)

if $(b_1 \vee b_2)$

then p; **while** $(b_1 \vee b_2)$ **do** p **od**

else skip; **while** $(b_1 \vee b_2)$ **do** p **od**

fi

$\Rightarrow$ (; *intro*$_3$),($\vee$ *elim*)

if b_1 **then** p **else if** b_2 **then** p **else skip fi fi** ; (3.7.2)

while $(b_1 \vee b_2)$ **do** p **od**

$\Rightarrow$ (*if logic*$_3$ *intro*)

if b_1 **then** p **else if** $(b_2 \vee b_1)$ **then** p **else skip fi fi** ;

while $(b_1 \vee b_2)$ **do** p **od**

$\Rightarrow$ (; *elim*$_3$),(; *elim*$_3$),(*skip elim*$_2$),(*while unfold*)

if b_1

then p; **while** $(b_1 \vee b_2)$ **do** p **od**

else if $(b_2 \vee b_1)$

then p; **while** $(b_1 \vee b_2)$ **do** p **od**

else if $(b_1 \vee b_2)$

then p; **while** $(b_1 \vee b_2)$ **do** p **od** |

else skip |

fi

fi

fi

$\Rightarrow$ (*if logic*$_2$ *intro*),$(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$

if b_1

then p; **while** $(b_1 \vee b_2)$ **do** p **od**

else if $(b_2 \vee b_1)$

```
            then p; while (b1 ∨ b2) do p od
            else if ((b1 ∨ b2) ∧ ¬ (b2 ∨ b1))...fi
          fi
fi
```

$\Rightarrow (b_0 \lor b_1 \equiv b_1 \lor b_0), (b_0 \land b_1 \equiv b_1 \land b_0), (\neg\, b \land b \equiv \mathit{false})$

```
if b1
  then p; while (b1 ∨ b2) do p od
  else if (b2 ∨ b1)
          then p; while (b1 ∨ b2) do p od
          else if false
                  then p; while (b1 ∨ b2) do p od
                  else skip
                fi
        fi
fi
```

$\Rightarrow (\mathit{if\ elim}_3), (b_0 \lor b_1 \equiv b_1 \lor b_0)$

```
if b1
  then p; while (b1 ∨ b2) do p od
  else if (b1 ∨ b2)
          then p; while (b1 ∨ b2) do p od
          else skip
        fi
fi
```

$\Rightarrow (\mathit{while\ unfold}), (\mathit{if\ intro}_3), (\mathit{while\ unfold})$

```
if b1
  then p;
     if (b1 ∨ b2)
        then p; while (b1 ∨ b2) do p od
        else if false
                then p; while (b1 ∨ b2) do p od
                else skip
              fi
```

```
      fi
    else while (b1 ∨ b2) do p od
fi
```

$\Rightarrow$ (*false* $\equiv b \wedge \neg b$), ($b_0 \wedge b_1 \equiv b_1 \wedge b_0$)

```
if b1
  then p;
      if (b1 ∨ b2)
         then p; while (b1 ∨ b2) do p od
         else if (¬ (b1 ∨ b2) ∧ (b1 ∨ b2))
                then p; while (b1 ∨ b2) do p od
                else skip
              fi
      fi
  else while (b1 ∨ b2) do p od
fi
```

$\Rightarrow$ (*if logic*$_2$ *elim*), (*skip intro*$_2$), (*while unfold*)

```
if b1
  then p;
      if (b1 ∨ b2)
         then p; while (b1 ∨ b2) do p od
         else skip; while (b1 ∨ b2) do p od
      fi
  else while (b1 ∨ b2) do p od
fi
```

$\Rightarrow$ (; *intro*$_3$), (; *assoc*)

```
if b1
  then p; if (b1 ∨ b2) then p else skip fi ;
      while (b1 ∨ b2) do p od
  else while (b1 ∨ b2) do p od
fi
```

$\Rightarrow$ (*skip intro*$_2$), (; *intro*$_3$)

if b_1 **then** p; **if** $(b_1 \lor b_2)$ **then** p **else skip fi** **else skip fi** ;

while $(b_1 \lor b_2)$ **do** p **od**

$\Rightarrow$ ($\lor$ *elim*)

if b_1

  then p; **if** b_1 **then** p **else** **if** b_2 **then** p **else skip fi** **fi**

  else skip

fi ; **while** $(b_1 \lor b_2)$ **do** p **od**

$\Rightarrow$ (*while fold in context*$_2$) applied to (3.7.2), (*while logic intro*)

while b_1 **do** p **od** ;

if $\neg\, b_1$ **then** **while** $(b_1 \lor b_2)$ **do** p **od** **else skip fi**

<'– – – << * >>

$\Rightarrow$ (*while logic elim*), (; *assoc*)

if $\neg\, b_1$

  then **if** b_2

    then p; **while** b_1 **do** p **od** ; **while** $(b_1 \lor b_2)$ **do** p **od**

    else skip

    fi

  else skip

fi

* * * **

if $\neg\, b_1$

  then **if** b_2 **then** p; **while** b_1 **do** p **od** **else skip fi** ;

    if $\neg\, b_1$ **then** S_1 **else skip fi**

  else skip

fi

$\Rightarrow$

if $\neg\ b_1$
  then **if** b_2 **then** p; **while** b_1 **do** p **od** **else skip fi** ; S_1
  else skip
fi

PROOF

*****

if $\neg\ b_1$
  then **if** b_2 **then** p; **while** b_1 **do** p **od** **else skip fi** ;
    if $\neg\ b_1$ **then** S_1 **else skip fi**
  else skip
fi

$\Rightarrow$ (; $elim_3$), (; *assoc*), (*while logic elim*), (*skip* $elim_2$)

if $\neg\ b_1$
  then **if** b_2
      then p; **while** b_1 **do** p **od** ; S_1
      else **if** $\neg\ b_1$ **then** S_1 **else skip fi**
      fi
  else skip
fi

$\Rightarrow$ (*if* $logic_1$ *intro*), ($deriv_6$)

if $\neg\ b_1$
  then **if** $\neg\ b_2$
      then **if** $(\neg\ b_1\ \wedge\ \neg\ b_2)$ **then** S_1 **else skip fi**
      else p; **while** b_1 **do** p **od** ; S_1
    fi
  else skip
fi

$\Rightarrow$ ($deriv_6$), $(b_0\ \wedge\ b_1\ \equiv\ b_1\ \wedge\ b_0)$

if $\neg\ b_1$
  then **if** $(\neg\ b_2\ \wedge\ \neg\ b_1)$
      then **if** $(\neg\ b_2\ \wedge\ \neg\ b_1)$ **then** S_1 **else skip fi**

else p; **while** b_1 **do** p **od** ; S_1

fi

else skip

fi

$\Rightarrow$ (*deriv*$_1$), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$, ($\wedge$ *elim*)

if $\neg b_1$

then **if** $\neg b_1$

then **if** $\neg b_2$ **then** S_1 **else** p; **while** b_1 **do** p **od** ; S_1 **fi**

else p; **while** b_1 **do** p **od** ; S_1

fi

else skip

fi

$\Rightarrow$ (*deriv*$_1$), (*if logic*$_1$ *elim*), (*skip intro*$_2$), (; *assoc*)

if $\neg b_1$

then **if** b_2 **then** p; **while** b_1 **do** p **od** ; S_1 **else skip**; S_1 **fi**

else skip

fi

$\Rightarrow$ (; *intro*$_3$)

if $\neg b_1$

then **if** b_2 **then** p; **while** b_1 **do** p **od** **else skip fi** ; S_1

else skip

fi

END OF (*while fold in context*$_1$) *PROOF*

*****

$\Rightarrow$ (*while fold in context*$_1$) applied to (3.7.3)

if $\neg b_1$ **then** **while** b_2 **do** p; **while** b_1 **do** p **od** **od** **else skip fi**
< - - - << * >>

$\Rightarrow$ (*while logic elim*)

while b_1 **do** p **od** ; **while** b_2 **do** p; **while** b_1 **do** p **od** **od**

□

Example 3.7 and example 3.2 express a semantic equivalence, proven using fixed point theory, taken from [**de Bakker 80**].

Example 3.8 *(Extracting from a Loop)*

while$(x > 0, y := 5; x := x - 1); y := n \Rightarrow_W$ while$(x > 0, x := x - 1); y := n$

Proof:

while $(x > 0)$ **do** $[\, y := 5\,]; [\, x := (x - 1)\,]$ **od** ; $[\, y := n\,]$ (3.8.1)

$\Rightarrow$ (*while unfold*), (*while unfold*)

if $(x > 0)$
  then $[\, y := 5\,]; [\, x := (x - 1)\,]$;
    if $(x > 0)$
      then $[\, y := 5\,]; [\, x := (x - 1)\,]$;
        while $(x > 0)$
          do
            $[\, y := 5\,]; [\, x := (x - 1)\,]$
          od
      else skip
    fi
  else skip
fi ; $[\, y := n\,]$

$\Rightarrow$ (; $elim_1$), (& *symm*)

if $(x > 0)$
  then $[\, y := 5 \,\&\, x := (x - 1)\,]$;
    if $(x > 0) \ldots$ **fi**
  else skip

```
fi ; [ y := n ]
 ⇒ (; intro1)

if (x > 0)
  then [ x := (x - 1) ]; [ y := 5 ];
     if (x > 0)...fi
  else skip
fi ; [ y := n ]
 ⇒ (; assoc), (; elim2), (; assoc), (; assoc)

if (x > 0)
  then [ x := (x - 1) ];
     if (x > 0)
       then [ y := 5 ]; [ y := 5 ];
          [ x := (x - 1) ];
          while (x > 0)
             do
               [ y := 5 ]; [ x := (x - 1) ]
             od
       else [ y := 5 ]; skip
     fi
  else skip
fi ; [ y := n ]
 ⇒ (; elim3), (; elim1)

if (x > 0)
  then [ x := (x - 1) ];
     if (x > 0)
       then [ y := 5 ];
          [ x := (x - 1) ];
          while (x > 0)...od
       else [ y := 5 ]; skip
     fi ; [ y := n ]
  else skip; [ y := n ]
fi
```

EX BIBL. UNIV. EDINBURGEN.

⇒ (; *assoc*), (; $elim_3$), (*skip* $elim_3$)

```
if (x > 0)
  then [ x := (x - 1) ];
      if (x > 0)
        then [ y := 5 ];
            [ x := (x - 1) ];
            while (x > 0)...od ; [ y := n ]
        else [ y := 5 ]; [ y := n ]
      fi
  else skip; [ y := n ]
fi
```

⇒ (*skip* $intro_2$), (; $elim_1$)

```
if (x > 0)
  then [ x := (x - 1) ];
      if (x > 0)
        then [ y := 5 ];
            [ x := (x - 1) ];
            while (x > 0)...od ; [ y := n ]
        else skip; [ y := n ]
      fi
  else skip; [ y := n ]
fi
```

⇒ (; $intro_3$), (; *assoc*)

```
if (x > 0)
  then [ x := (x - 1) ];
      if (x > 0)
        then [ y := 5 ];
            [ x := (x - 1) ];
            while (x > 0)...od
        else skip
      fi ; [ y := n ]
  else skip; [ y := n ]
```

fi

$\Rightarrow$ (; *intro*$_3$), (; *assoc*)

if $(x > 0)$
  then $[\, x := (x - 1) \,]$;
    if $(x > 0)$
      then $[\, y := 5 \,]; [\, x := (x - 1) \,]$;
        while $(x > 0) \ldots$ **od**
      else skip
    fi
  else skip
fi ; $[\, y := n \,]$

$\Rightarrow$ (*while unfold*)

if $(x > 0)$
  then $[\, x := (x - 1) \,]$;
    while $(x > 0)$ **do** $[\, y := 5 \,]; [\, x := (x - 1) \,]$ **od**
  else skip
fi ; $[\, y := n \,]$

$\Rightarrow$ (*while fold in context*$_2$) applied to (3.8.1)

while $(x > 0)$ **do** $[\, x := (x - 1) \,]$ **od** ; $[\, y := n \,]$

□

Example 3.9 *(Removing a Falsity,* [Scherlis 80]*)*

	if$(n \geq 0, i := 0;$ **while**$(n < i^2 \vee n \geq (i+1)^2, i := i+1))$
$\Rightarrow_W$	**if**$(n \geq 0, i := 0;$ **while**$(n \geq (i+1)^2, i := i+1))$.

Proof:

DATA RULES

$(r1)\ 0^2 \Rightarrow 0$
$(r2)\ 1^2 \Rightarrow 1$
$(r3)\ (0 + x) \Rightarrow x$

$(r4)\ (x < y) \Rightarrow \neg(x \geq y)$
$(r5)\ ((x \geq 0) \wedge (x \geq 1)) \Rightarrow (x \geq 1)$
$(r6)\ true \Rightarrow (0 = 0)$
$(r7)\ ((n \geq 1) \wedge (i = 0)) \Rightarrow (n \geq (i+1)^2)$
$(r8)\ ((n \geq (i+1)^2) \wedge (n \geq ((i+1)+1)^2)) \Rightarrow (n \geq ((i+1)+1)^2)$
$(r10)\ ((n \geq i^2) \wedge (n \geq (i+1)^2)) \Rightarrow (n \geq (i+1)^2)$

if $(n \geq 0)$
  then $[\, i := 0 \,]$;
    while $((n < i^2) \vee (n \geq (i+1)^2))$
      do
        $[\, i := (i+1) \,]$
      od
  else skip
fi

$\Rightarrow$ (*while unfold*), (*if logic*$_1$ *intro*), (; *elim*$_2$), $(r1)$, $(r3)$

if $\neg(n \geq 0)$
  then skip
  else if $((n < 0) \vee (n \geq 1^2))$
      then $[\, i := 0 \,]$;
        $[\, i := (i+1) \,]$;
        while $((n < i^2) \vee (n \geq (i+1)^2))$ ... **od**
      else $[\, i := 0 \,]$; **skip**
    fi
fi

$\Rightarrow$ (*if logic*$_2$ *intro*), $(\neg\neg\, b \equiv b)$, $(r2)$

if $\neg(n \geq 0)$
  then skip
  else if $((n \geq 0) \wedge ((n < 0) \vee (n \geq 1)))$
      then $[\, i := 0 \,]$; ...
      else $[\, i := 0 \,]$; **skip**
    fi
fi

$\Rightarrow$ (*if logic*$_1$ *elim*), $(b_0 \wedge (b_1 \vee b_2) \equiv (b_0 \wedge b_1) \vee (b_0 \wedge b_2))$

```
if (n ≥ 0)
  then if (((n ≥ 0) ∧ (n < 0)) ∨ ((n ≥ 0) ∧ (n ≥ 1)))
         then [ i := 0 ]; ...
         else [ i := 0 ]; skip
       fi
  else skip
fi
```

$\Rightarrow$ $(b_0 \wedge b_1 \equiv b_1 \wedge b_0), (r4), (b_0 \vee b_1 \equiv b_1 \vee b_0), (\neg\, b \wedge b \equiv \mathit{false}), (b \vee \mathit{false} \equiv b)$

```
if (n ≥ 0)
  then if ((n ≥ 0) ∧ (n ≥ 1))
         then [ i := 0 ]; ...
         else [ i := 0 ]; skip
       fi
  else skip
fi
```

$\Rightarrow$ (*if intro*$_2$), $(r6), (r5)$

```
if (n ≥ 0)
  then if (0 = 0)
         then if (n ≥ 1)
                then [ i := 0 ]; ...
                else [ i := 0 ]; skip
              fi
         else [ i := 0 ]; skip
       fi
  else skip
fi
```

$\Rightarrow$ (; *intro*$_2$)

```
if (n ≥ 0)
  then if (0 = 0)
         then [ i := 0 ];
```

```
            if (n ≥ 1)
              then ...
              else skip
            fi
          else [ i := 0 ]; skip
        fi
    else skip
fi
```

$\Rightarrow$ (; $intro_2$), ($\wedge$ $intro$), ($b_0 \wedge b_1 \equiv b_1 \wedge b_0$)

```
if (n ≥ 0)
    then [ i := 0 ];
        if ((n ≥ 1) ∧ (i = 0))
          then ...
          else if (i = 0) then skip else skip fi
        fi
    else skip
fi
```

$\Rightarrow$ (*if* $logic_1$ *intro*), ($r7$), (*if* $elim_1$), (*if* $intro_2$)

```
if (n ≥ 0)
    then [ i := 0 ];
        if ¬ (n ≥ (i + 1)²)
          then skip
          else if true
                then ...
                else [ i := (i + 1) ]
              fi
        fi
    else skip
fi
```

$\Rightarrow$ (*if* $logic_2$ *intro*), (*if* $logic_1$ *elim*), ($\neg\neg\, b \equiv b$)

```
if (n ≥ 0)
    then [ i := 0 ];
```

```
        if (n ≥ (i + 1)²)
          then if ((n ≥ (i + 1)²) ∧ true)
                 then ...
                 else [ i := (i + 1) ]
               fi
          else skip
        fi
   else skip
fi
```

$\Rightarrow (b_0 \wedge b_1 \equiv b_1 \wedge b_0), (true \wedge b \equiv b), (skip\ intro_3), (;\ intro_2)$

```
if (n ≥ 0)
   then [ i := 0 ];
        if (n ≥ (i + 1)²)
          then [ i := (i + 1) ];
               << if (n ≥ i²)
                  then while ((n < i²) ∨ (n ≥ (i + 1)²))
                         do
                           [ i := (i + 1) ]
                         od
                  else skip
               fi >>
          else skip
        fi
   else skip
fi
```

```
<< * >>  --->
          if (n ≥ i²)
            then while ((n < i²) ∨ (n ≥ (i + 1)²))
                   do
                     [ i := (i + 1) ]
                   od
            else skip
          fi
```

$\Rightarrow$ (*if logic*$_1$ *intro*),(*while unfold*)

if $\neg\,(n \geq i^2)$
  then skip
  else if $((n < i^2) \vee (n \geq (i+1)^2))$
    then [$i := (i+1)$];
      while $((n < i^2) \vee (n \geq (i+1)^2))$ ... **od**
    else skip
  fi
fi

$\Rightarrow$ (*if logic*$_2$ *intro*),$(\neg\neg\, b \equiv b)$

if $\neg\,(n \geq i^2)$
  then skip
  else if $((n \geq i^2) \wedge ((n < i^2) \vee (n \geq (i+1)^2)))$
    then ...
    else skip
  fi
fi

$\Rightarrow (b_0 \wedge (b_1 \vee b_2) \equiv (b_0 \wedge b_1) \vee (b_0 \wedge b_2)),(r4)$

if $\neg\,(n \geq i^2)$
  then skip
  else if $((n \geq i^2 \wedge \neg(n \geq i^2)) \vee (n \geq i^2 \wedge n \geq (i+1)^2))$
    then ...
    else skip
  fi
fi

$\Rightarrow (b_0 \wedge b_1 \equiv b_1 \wedge b_0),(b_0 \vee b_1 \equiv b_1 \vee b_0),(\neg b \wedge b \equiv \mathit{false}),(b \vee \mathit{false} \equiv b)$

if $\neg\,(n \geq i^2)$
  then skip
  else if $((n \geq i^2) \wedge (n \geq (i+1)^2))$
    then ...
    else skip

```
        fi
fi

 ⇒ (if logic₁ elim), (r10)

if (n ≥ i²)
   then if (n ≥ (i + 1)²)
           then ...
           else skip
        fi
   else skip
fi


*****
if (n ≥ i²)
   then if (n ≥ (i + 1)²)
           then [ i := (i + 1) ]
           else skip
        fi ; if (n ≥ i²) then S₁ else skip fi
   else skip
fi
⇒
if (n ≥ i²)
   then if (n ≥ (i + 1)²)
           then [ i := (i + 1) ]
           else skip
        fi ; S₁
   else skip
fi
PROOF
*****

if (n ≥ i²)
   then if (n ≥ (i + 1)²)
           then [ i := (i + 1) ]
           else skip
```

```
        fi ; if (n ≥ i²) then S₁ else skip fi
    else skip
fi
```

⇒ (*if logic*$_1$ *intro*)

```
if (n ≥ i²)
    then if ¬(n ≥ (i + 1)²)
            then skip
            else [ i := (i + 1) ]
        fi ; if (n ≥ i²) then S₁ else skip fi
    else skip
fi
```

⇒ (; *elim*$_3$), (*skip elim*$_2$)

```
if (n ≥ i²)
    then if ¬(n ≥ (i + 1)²)
            then if (n ≥ i²) then S₁ else skip fi
            else [ i := (i + 1) ];
                if (n ≥ i²) then S₁ else skip fi
        fi
    else skip
fi
```

⇒ (*if logic*$_1$ *elim*), (; *elim*$_2$), (*if logic*$_1$ *intro*)

```
if (n ≥ i²)
    then if (n ≥ (i + 1)²)
            then if ¬(n ≥ (i + 1)²)
                    then [ i := (i + 1) ]; skip
                    else [ i := (i + 1) ]; S₁
                fi
            else if (n ≥ i²) then S₁ else skip fi
        fi
    else skip
fi
```

$\Rightarrow$ (*deriv*$_6$), ($\neg\ b\ \wedge\ b\ \equiv$ *false*)

```
if (n ≥ i²)
  then if (n ≥ (i + 1)²)
         then if false
                then [ i := (i + 1) ]; skip
                else [ i := (i + 1) ]; S₁
              fi
         else if (n ≥ i²) then S₁ else skip fi
       fi
  else skip
fi
```

$\Rightarrow$ (*if logic*$_1$ *intro*), (*if logic*$_1$ *intro*), (*if elim*$_3$)

```
if (n ≥ i²)
  then if ¬ (n ≥ (i + 1)²)
         then if ¬ (n ≥ i²) then skip else S₁ fi
         else [ i := (i + 1) ]; S₁
       fi
  else skip
fi
```

$\Rightarrow$ (*deriv*$_6$), (*deriv*$_6$), ($b_0\ \wedge\ b_1\ \equiv\ b_1\ \wedge\ b_0$)

```
if (n ≥ i²)
  then if (¬ (n ≥ (i + 1)²) ∧ (n ≥ i²))
         then if (¬ (n ≥ i²) ∧ ((n ≥ i²) ∧ ¬ (n ≥ (i + 1)²)))
                then skip
                else S₁
              fi
         else [ i := (i + 1) ]; S₁
       fi
  else skip
fi
```

$\Rightarrow (a \wedge (b \wedge c) \equiv (a \wedge b) \wedge c), (\neg\, b \wedge b \equiv \mathit{false})$

if $(n \geq i^2)$
  then **if** $(\neg\,(n \geq (i + 1)^2) \wedge (n \geq i^2))$
    then **if** $(\mathit{false} \wedge \neg\,(n \geq (i + 1)^2))$
      then skip
      else S_1
    fi
    else $[\, i := (i + 1)\,]; S_1$
  fi
  else skip
fi

$\Rightarrow (\wedge\ \mathit{elim}), (b_0 \wedge b_1 \equiv b_1 \wedge b_0), (\mathit{if}\ \mathit{elim}_3)$

if $(n \geq i^2)$
  then **if** $((n \geq i^2) \wedge \neg\,(n \geq (i + 1)^2))$
    then S_1
    else $[\, i := (i + 1)\,]; S_1$
  fi
  else skip
fi

$\Rightarrow (\wedge\ \mathit{elim}), (\mathit{if}\ \mathit{logic}_1\ \mathit{elim}), (\mathit{skip}\ \mathit{intro}_2)$

if $(n \geq i^2)$
  then **if** $(n \geq i^2)$
    then **if** $(n \geq (i + 1)^2)$
      then $[\, i := (i + 1)\,]; S_1$
      else skip; S_1
    fi
    else $[\, i := (i + 1)\,]; S_1$
  fi
  else skip
fi

$\Rightarrow (\mathit{deriv}_1), (;\ \mathit{intro}_3)$

```
if (n ≥ i²)
   then if (n ≥ (i + 1)²)
          then [ i := (i + 1) ]
          else skip
        fi ; S₁
   else skip
fi
```

END OF (while fold in context$_1$*) PROOF*

* * * **

⇒ (*while fold in context*$_1$) applied to (3.9.1)

```
if (n ≥ i²)
   then while (n ≥ (i + 1)²)
            do
              [ i := (i + 1) ]
            od
   else skip
fi
```

```
<---  << * >>
```

⇒ (*if logic*$_1$ *intro*), (; *elim*$_2$)

```
if (n ≥ 0)
   then [ i := 0 ];
      if ¬ (n ≥ (i + 1)²)
         then skip
         else if (n ≥ (i + 1)²)
                then [ i := (i + 1) ];
                   while (n ≥ (i + 1)²)
                      do
                        [ i := (i + 1) ]
                      od
                else [ i := (i + 1) ]; skip
```

```
            fi
        fi
    else skip
fi
```

$\Rightarrow$ (*if logic*$_2$ *intro*), $(\neg\neg\, b \;\equiv\; b)$

```
if (n ≥ 0)
    then [ i := 0 ];
        if ¬ (n ≥ (i + 1)²)
            then skip
            else if ((n ≥ (i + 1)²) ∧ (n ≥ (i + 1)²))
                    then ...
                    else [ i := (i + 1) ]; skip
                fi
        fi
    else skip
fi
```

$\Rightarrow (b \;\equiv\; b \,\wedge\, true), (b \;\equiv\; \neg\neg\, b), (b \,\wedge\, b \;\equiv\; b)$

```
if (n ≥ 0)
    then [ i := 0 ];
        if ¬ (n ≥ (i + 1)²)
            then skip
            else if (¬ ¬ (n ≥ (i + 1)²) ∧ true)
                    then ...
                    else [ i := (i + 1) ]; skip
                fi
        fi
    else skip
fi
```

$\Rightarrow$ (*if logic*$_2$ *elim*), (*if logic*$_1$ *elim*), (*if elim*$_2$)

```
if (n ≥ 0)
    then [ i := 0 ];
        if (n ≥ (i + 1)²)
```

```
        then ...
        else skip
      fi
  else skip
fi
```

⇒ (*while unfold*)

```
if (n ≥ 0)
  then [ i := 0 ];
      while (n ≥ (i + 1)²) do [ i := (i + 1) ] od
  else skip
fi
```

□

Example 3.9 is taken from **[Scherlis 80]**, where it is expressed in a functional language, and transformed using EP and assertion propagation rules.

In the next example we prove $a \Leftrightarrow_W b$ by proving $a \Rightarrow_W b$ and $b \Rightarrow_W a$ separately. We do not include any details of the $b \Rightarrow_W a$ part, as it is similar to, but much longer than, the $a \Rightarrow_W b$ part.

Example 3.10 *(***[van Diepen/de Roever 86]***)*

$$\mathbf{while}(b, \mathbf{while}(b_1 \wedge b, p_1); \mathbf{while}(\neg b_1 \wedge b, p_2)) \Leftrightarrow_W \mathbf{while}(b, \mathbf{if}(b_1, p_1, p_2))$$

Proof:

$$\mathbf{while}\ b\ \mathbf{do}\ \ \mathbf{while}\ (b_1 \wedge b)\ \mathbf{do}\ p_1\ \mathbf{od}\ ;\ \mathbf{while}\ (\neg\, b_1 \wedge b)\ \mathbf{do}\ p_2\ \mathbf{od}\ \ \mathbf{od} \quad (3.10)$$

⇒ (*while unfold*),(*while unfold*)

```
if b
  then if (b1 ∧ b) then p1; while (b1 ∧ b) do p1 od else skip fi ;
      while (¬ b1 ∧ b) do p2 od ;
      while b
        do
```

while $(b_1 \wedge b)$ **do** p_1 **od** ; **while** $(\neg b_1 \wedge b)$ **do** p_2 **od**
od
else skip
fi

$\Rightarrow$ (; *assoc*), (; *elim*$_3$), (*skip elim*$_2$)

if b
then if $(b_1 \wedge b)$
then p_1; **while** $(b_1 \wedge b)$ **do** p_1 **od** ;
while $(\neg b_1 \wedge b)$ **do** p_2 **od** ;
while b
do
while $(b_1 \wedge b)$ **do** p_1 **od** ;
while $(\neg b_1 \wedge b)$ **do** p_2 **od**
od
else while $(\neg b_1 \wedge b)$ **do** p_2 **od** ;
while b
do
while $(b_1 \wedge b)$ **do** p_1 **od** ;
while $(\neg b_1 \wedge b)$ **do** p_2 **od**
od
fi
else skip
fi

$\Rightarrow$ (*if logic*$_1$ *intro*), (*while unfold*)

if $\neg b$
then skip
else if $(b_1 \wedge b)$
then ...
else if $(\neg b_1 \wedge b)$
then p_2; **while** $(\neg b_1 \wedge b)$ **do** p_2 **od**
else skip
fi ;
while b

```
            do
               while (b1 ∧ b) do p1 od ;
               while (¬ b1 ∧ b) do p2 od
            od
    fi
fi
```

$\Rightarrow (b_0 \wedge b_1 \equiv b_1 \wedge b_0), (b \equiv \neg\neg\, b), (\textit{if logic}_2\ \textit{elim}), (;\ \textit{elim}_3)$

```
if ¬ b
   then skip
   else if b1
          then ...
          else if (¬ b1 ∧ b)
                 then p2; while (¬ b1 ∧ b) do p2 od ;
                     while b
                        do
                           while (b1 ∧ b) do p1 od ;
                           while (¬ b1 ∧ b) do p2 od
                        od
                 else skip;
                     while b
                        do
                           while (b1 ∧ b) do p1 od ;
                           while (¬ b1 ∧ b) do p2 od
                        od
               fi
    fi
fi
```

$\Rightarrow (\textit{if logic}_2\ \textit{elim}), (\textit{if logic}_1\ \textit{intro}), (\textit{if logic}_2\ \textit{intro}), (b_0 \wedge b_1 \equiv b_1 \wedge b_0), (\neg\neg b \equiv b)$

```
if ¬ b
   then skip
   else if (¬ b1 ∧ b)
          then if b
                 then p2; while (¬ b1 ∧ b) do p2 od ;
```

```
                    while b
                        do
                            while (b1 ∧ b) do p1 od ;
                            while (¬ b1 ∧ b) do p2 od
                        od
                else skip;
                    while b
                        do
                            while (b1 ∧ b) do p1 od ;
                            while (¬ b1 ∧ b) do p2 od
                        od
            fi
        else ...
    fi
fi
```

$\Rightarrow (\wedge\ elim), (if\ logic_1\ intro), (b\ \equiv\ b\ \wedge\ true), (b\ \equiv\ \neg\neg\ b)$

```
if ¬ b
   then skip
   else if ¬ b1
          then if ¬ b
                 then p1; while (b1 ∧ b) do p1 od ;
                      while (¬ b1 ∧ b) do p2 od ;
                      while b
                          do
                              while (b1 ∧ b) do p1 od ;
                              while (¬ b1 ∧ b) do p2 od
                          od
                 else if (¬ ¬ b ∧ true)
                         then p2; while (¬ b1 ∧ b) do p2 od ;
                              while b
                                  do
                                      while (b1 ∧ b) do p1 od ;
                                      while (¬ b1 ∧ b) do p2 od
                                  od
```

```
                    else skip;
                        while b
                            do
                              while (b1 ∧ b) do p1 od ;
                              while (¬ b1 ∧ b) do p2 od
                            od
                fi
        fi
      else ...
    fi
fi
```

$\Rightarrow$ (*if logic*$_2$ *elim*), (*if logic*$_1$ *elim*), ($\wedge$ *intro*), ($b_0 \wedge b_1 \equiv b_1 \wedge b_0$), ($b \equiv \neg\neg b$), (*if logic*$_2$ *elim*), (*if logic*$_1$ *elim*), (*if logic*$_1$ *elim*), (*if elim*$_2$), (*if elim*$_1$), (; *assoc*), (; *assoc*)

```
if b
  then if b1
        then ...
        else p2; while (¬ b1 ∧ b) do p2 od ;
            while b
                do
                  while (b1 ∧ b) do p1 od ;
                  while (¬ b1 ∧ b) do p2 od
                od
      fi
  else skip
fi
```

$\Rightarrow$ (; *assoc*), (*if intro*$_1$), (*while unfold*)

```
if b
  then if b1
        then p1;
            if b
              then while (b1 ∧ b) do p1 od ;
                  while (¬ b1 ∧ b) do p2 od ;
```

```
                while b
                    do
                        while (b1 ∧ b) do p1 od ;
                        while (¬ b1 ∧ b) do p2 od
                    od
            else if (b1 ∧ b)
                then p1; while (b1 ∧ b) do p1 od
                else skip
                fi ; while (¬ b1 ∧ b) do p2 od ;
                while b
                    do
                        while (b1 ∧ b) do p1 od ;
                        while (¬ b1 ∧ b) do p2 od
                    od
            fi
        else p2; while (¬ b1 ∧ b) do p2 od ;
            while b
                do
                    while (b1 ∧ b) do p1 od ;
                    while (¬ b1 ∧ b) do p2 od
                od
        fi
    else skip
fi
```

$\Rightarrow$ (; *assoc*), (; *elim*$_3$), (*if logic*$_2$ *intro*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$

```
if b
    then if b1
        then p1;
            if b
                then while (b1 ∧ b) do p1 od ;
                    while (¬ b1 ∧ b) do p2 od ;
                    while b
                        do
                            while (b1 ∧ b) do p1 od ;
```

```
                        while (¬ b1 ∧ b) do p2 od
                      od
               else if (¬ b ∧ (b ∧ b1))
                    then p1; while (b1 ∧ b) do p1 od ;
                       while (¬ b1 ∧ b) do p2 od ;
                       while b
                          do
                            while (b1 ∧ b) do p1 od ;
                            while (¬ b1 ∧ b) do p2 od
                          od
                    else skip;
                       while (¬ b1 ∧ b) do p2 od ;
                       while b
                          do
                            while (b1 ∧ b) do p1 od ;
                            while (¬ b1 ∧ b) do p2 od
                          od
                 fi
            fi
         else ...
       fi
  else skip
fi
```

$\Rightarrow (b_1 \wedge (b_2 \wedge b_3) \equiv (b_1 \wedge b_2) \wedge b_3), (\neg\, b \wedge b \equiv \mathit{false}), (\mathit{false} \wedge b \equiv \mathit{false})$

```
if b
  then if b1
        then p1;
           if b
             then while (b1 ∧ b) do p1 od ;
                while (¬ b1 ∧ b) do p2 od ;
                while b
                   do
                     while (b1 ∧ b) do p1 od ;
                     while (¬ b1 ∧ b) do p2 od
```

```
                        od
                 else if false
                        then p1; while (b1 ∧ b) do p1 od ;
                             while (¬ b1 ∧ b) do p2 od ;
                             while b
                                 do
                                    while (b1 ∧ b) do p1 od ;
                                    while (¬ b1 ∧ b) do p2 od
                                 od
                        else skip;
                             while (¬ b1 ∧ b) do p2 od ;
                             while b
                                 do
                                    while (b1 ∧ b) do p1 od ;
                                    while (¬ b1 ∧ b) do p2 od
                                 od
                   fi
             fi
         else ...
      fi
   else skip
fi
```

$\Rightarrow$ (*if elim*$_3$), (*skip elim*$_2$), (*while unfold*), (; *elim*$_3$), (*if logic*$_2$ *intro*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$

```
if b
   then if b1
          then p1;
               if b
                  then while (b1 ∧ b) do p1 od ;
                       while (¬ b1 ∧ b) do p2 od ;
                       while b
                           do
                              while (b1 ∧ b) do p1 od ;
                              while (¬ b1 ∧ b) do p2 od
                           od
```

```
                else  if (¬ b ∧ (b ∧ ¬ b1))
                      then p2;  while (¬ b1 ∧ b) do p2 od ;
                         while b
                            do
                               while (b1 ∧ b) do p1 od ;
                               while (¬ b1 ∧ b) do p2 od
                            od
                      else skip;
                         while b
                            do
                               while (b1 ∧ b) do p1 od ;
                               while (¬ b1 ∧ b) do p2 od
                            od
             fi
         fi
      else ...
    fi
  else skip
fi
```

$\Rightarrow (b_1 \wedge (b_2 \wedge b_3) \equiv (b_1 \wedge b_2) \wedge b_3), (\neg b \wedge b \equiv \mathit{false}), (\mathit{false} \wedge b \equiv \mathit{false}),$
$(\mathit{if}\ \mathit{elim}_3), (\mathit{skip}\ \mathit{elim}_2), (\mathit{while}\ \mathit{unfold})$

```
if b
  then  if b1
      then p1;
         if b
           then  while (b1 ∧ b) do p1 od ;
              while (¬ b1 ∧ b) do p2 od ;
              while b
                 do
                    while (b1 ∧ b) do p1 od ;
                    while (¬ b1 ∧ b) do p2 od
                 od
           else  if b
                then  while (b1 ∧ b) do p1 od ;
```

```
                        while (¬ b1 ∧ b) do p2 od ;
                        while b
                           do
                              while (b1 ∧ b) do p1 od ;
                              while (¬ b1 ∧ b) do p2 od
                           od
                     else skip
                  fi
            fi
         else ...
      fi
   else skip
fi
```

$\Rightarrow$ (*if logic*$_2$ *intro*), $(\neg\ b\ \wedge\ b\ \equiv\ false)$, (*if elim*$_3$), (*while unfold*), (; *assoc*)

```
if b
   then if b1
          then p1;
             while b
                do
                   while (b1 ∧ b) do p1 od ;
                   while (¬ b1 ∧ b) do p2 od
                od
          else p2;
             <<  while (¬ b1 ∧ b) do p2 od ;
             while b
                do
                   while (b1 ∧ b) do p1 od ;
                   while (¬ b1 ∧ b) do p2 od
                od  >>
      fi
   else skip
fi
```

```
<< * >>  --->
```

```
while (¬ b1 ∧ b) do p2 od ;
while b
    do
       while (b1 ∧ b) do p1 od ;
       while (¬ b1 ∧ b) do p2 od
    od
 ⇒ (if intro1), (while unfold), (; elim3)

if (¬ b1 ∧ b)
   then while (¬ b1 ∧ b) do p2 od ; ...
   else if (¬ b1 ∧ b)
           then p2; while (¬ b1 ∧ b) do p2 od ; ...
           else skip; ...
        fi
fi
 ⇒ (if logic2 intro), (¬ b ∧ b ≡ false), (if elim3), (skip elim2)

if (¬ b1 ∧ b)
   then while (¬ b1 ∧ b) do p2 od ; ...
   else ...
fi
 ⇒ (b0 ∧ b1 ≡ b1 ∧ b0), (skip intro2), (if intro2)

if (b ∧ ¬ b1)
   then if true then skip else < program > fi ;
        while (¬ b1 ∧ b) do p2 od ; ...
   else ...
fi
 ⇒ (∧ elim), (; assoc), (; elim3)

if b
   then if ¬ b1
           then if true
                   then skip;
                        while (¬ b1 ∧ b) do p2 od ; ...
```

```
                else < program >;
                    while (¬ b_1 ∧ b) do p_2 od ; ...
            fi
        else ...
    fi
  else ...
fi
```

$\Rightarrow (deriv_6), (;\ intro_3), (true \wedge b \equiv b)$

```
if b
  then if ¬ b_1
        then if ¬ b_1 then skip else < program > fi ;
            while (¬ b_1 ∧ b) do p_2 od ; ...
        else ...
    fi
  else ...
fi
```

$\Rightarrow (\wedge\ intro), (b_0 \wedge b_1 \equiv b_1 \wedge b_0), (if\ logic_1\ elim), (if\ elim_1)$

```
if (¬ b_1 ∧ b)
  then if b_1 then < program > else skip fi ;
      while (¬ b_1 ∧ b) do p_2 od ; ...
  else ...
fi
```

$\Rightarrow (\wedge\ elim), (;\ elim_3)$

```
if ¬ b_1
  then if b
        then if b_1
                then < program >;
                    while (¬ b_1 ∧ b) do p_2 od ; ...
                else skip;
                    while (¬ b_1 ∧ b) do p_2 od ; ...
            fi
        else ...
```

```
        fi
    else ...
fi
```

$\Rightarrow$ $(deriv_6), (;\ intro_3)$

```
if ¬ b1
    then if b
            then if (b1 ∧ b)
                    then < program >
                    else skip
                fi ;
                while (¬ b1 ∧ b) do p2 od ; ...
            else ...
        fi
    else ...
fi
```

$\Rightarrow$ $(\wedge\ intro), (;\ assoc), (if\ intro_2), (while\ unfold), (if\ elim_1)$

```
if (¬ b1 ∧ b)
    then if true
            then while (b1 ∧ b) do p1 od ;
                while (¬ b1 ∧ b) do p2 od ; ...
            else skip
        fi
    else ...
fi
```

$\Rightarrow$ $(\wedge\ elim), (deriv_6), (true\ \wedge\ b\ \equiv\ b)$

```
if ¬ b1
    then if b
            then if b
                    then while (b1 ∧ b) do p1 od ;
                        while (¬ b1 ∧ b) do p2 od ; ...
                    else skip
                fi
```

```
                else ...
            fi
        else ...
      fi
```

$\Rightarrow$ ($\wedge$ *intro*), (*while unfold*), (*if elim*$_1$)

```
      if (¬ b1 ∧ b)
        then ...
        else ...
      fi
```

$\Rightarrow$ (*if elim*$_1$)

```
      while b
         do
           while (b1 ∧ b) do p1 od ;
           while (¬ b1 ∧ b) do p2 od
         od
```

<--- << * >>

$\Rightarrow$ (; *intro*$_3$)

```
if b
  then if b1 then p1 else p2 fi ;
       while b
          do
             while (b1 ∧ b) do p1 od ; while (¬ b1 ∧ b) do p2 od
          od
  else skip
fi
```

$\Rightarrow$ (*while fold*) applied to (3.10)

```
while b do if b1 then p1 else p2 fi od
```

□

Example 3.10 is a rule of the system detailed in **[van Diepen/de Roever 86]**. We conjecture that our system can derive all the rules of this system; the only other non-trivial transformations are "moving a statement through a loop", an example of which we have done using W (see example 3.6), and the addition of a cycle to a loop, (*while unfold*) in reverse.

In the previous examples we have mostly only managed to show $a \Rightarrow_W b$ even though $b \Rightarrow_W a$ may also be true. As an extreme case of this we cannot eliminate a solitary **while** using our transformation system, although we can introduce one. With respect to our system, having done $a \Rightarrow_W b$, trying to do $b \Rightarrow_W a$ is tantamount to a termination proof for b. This can be seen as follows: consider $a \Rightarrow_W b$; if a is non-terminating then so is b by soundness of the transformation system; if a is terminating then b may be non-terminating, **but** if we can transform $b \Rightarrow_W a$ then since a is terminating, b must be terminating. This *restriction* of not being able to eliminate a solitary **while** and in general of not being able to do termination proofs via transformation, becomes more significant when we consider the use of higher level transformations. For example, suppose we have transformed $a \Rightarrow_W b$; in order to use this as a higher level transformation replacing b by a we need to do a termination proof, otherwise the system becomes unsound.

A **Redefinition** rule was added to UF (see **[Burstall/Darlington 77]**) to enable the UF system to eliminate solitary loops, e.g

$$G(n) \Leftarrow \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } G(n-1)$$

can be transformed into $G(n) \Leftarrow 0$ using **Redefinition. Redefinition** cannot be used transitively.

Chapter 4

Relative Completeness

In this chapter we shall investigate the theoretical power of our transformation system W. Unfortunately we cannot give an absolute completeness result, as we did for the **IF** system, since such a (finite) transformation system would imply decidability of the halting problem (see [**Kibler 78**]). Second best are relative completeness results. Given some other *good* transformation system, we wish to prove that our system is complete relative to it. That is, our system can do whatever the good system can do. We shall give two relative completeness results in this chapter.

Through the construction of our transformation system we have shown that transformations, assertions and verification are closely related. More specifically we have shown that Hoare triples naturally arise in (*while fold in* $context_1$). Although we have used this transformation rule in our examples, we have not used Hoare's logic. We have used a form of (*while fold in* $context_1$) with a transformation premiss. Our first relative completeness result will prove that our transformation system is *as powerful* as Hoare's logic (with the usual simple consequence rule). This will prove that the two forms of (*while fold in* $context_1$), using a Hoare triple premiss and using a transformation premiss, are equivalent.

There are few well established imperative systems. We therefore choose to compare our W system with the Unfold/Fold (UF) system of [**Burstall/Darlington 77**]. This is a very powerful and well known system, which we shall describe later.

Our second relative completeness result proves that UF is at least *as powerful* as a subset of W. We conjecture that the UF system is lacking the full power of W.

Since we use translation with transformation, in these relative completeness results, we discuss the problems that arise through the interaction of these related concepts. We argue that for any relative completeness result to be fair we need to restrict the translations used. We thus define a set of *relatively reasonable* translations that may be used in order to compare the *power* of transformation systems. Note, however, that our results comparing W and UF use a translation procedure that we have not proven to be relatively reasonable.

4.1 Mimicking Hoare's Logic

As mentioned earlier there is a correspondence between Hoare triples and transformations. In this section we shall detail this correspondence and use it show that our transformation system has all the power of Hoare's logic (with the usual simple consequence rule). This will then prove that the Hoare triple premiss of (*while fold in context*$_1$) can be written using a transformation without any loss of power to the system. In practice the notation of assertions, for example Hoare triples, is less cumbersome than the notation of transformations. The purpose of this section therefore is not to promote transformations as an alternative to assertions but to show that transformations provide a uniform framework that is adequate to capture the power of assertions.

The first step is to express a Hoare triple as a semantic equivalence, so that the transformation system which works over this semantic equivalence can be compared to Hoare's logic which works over the Hoare triples. This is informally expressed as follows:

$$\{p\}S\{q\} \textit{ is } \textbf{true} \textit{ is equivalent to } \forall S_1.\mathbf{if}(p, S; \mathbf{if}(q, S_1)) \equiv \mathbf{if}(p, S; S_1)$$

The formula '$\{p\}S\{q\}$ is **true**' has the following informal meaning: whenever p holds before the execution of S (and S terminates) then q holds after the execution

of S. One can see that this semantic equivalence matches the meaning of the Hoare triple, i.e if p is false or S is undefined then the right hand side and left hand side of the semantic equivalence are the same, if p is true then, after S, $\mathbf{if}(q, S_1)$ is equivalent to S_1, that is q is true.

We cannot always perform the above *translation* as p and q in the Hoare triple may contain quantifiers in general and our boolean expressions do not. We could introduce bounded quantification into our boolean expressions in order to reason over data structures such as arrays.

The following Lemma provides a formal statement (and thus enables us to prove it) of the above translation; the notation used is that of the semantics, defined in Appendix A.

Lemma 4.1 $\forall p, q, S$
$\forall \Phi, \forall \sigma, \sigma' \in STORE_\Phi.$
if $(\langle p, \sigma\rangle \xrightarrow{*} \langle true, \sigma\rangle \wedge \langle S, \sigma\rangle \xrightarrow{*} \sigma')$
then $\langle q, \sigma'\rangle \xrightarrow{*} \langle true, \sigma'\rangle$
iff
$\forall S_1.\ \mathbf{if}(p, S; \mathbf{if}(q, S_1)) \equiv \mathbf{if}(p, S; S_1)$

Proof:
(outline)
We start with the definition of the semantic equivalence of $\mathbf{if}(p, S;\ \mathbf{if}(q, S_1)) \equiv \mathbf{if}(p, S;\ S_1)$, i.e $\forall \Phi, \forall \sigma \in STORE_\Phi.exec(\mathbf{if}(p, S;\ \mathbf{if}(q, S_1)), \sigma) \approx exec(\mathbf{if}(p, S;\ S_1), \sigma)$, and expand the definition of *exec* in Appendix A as follows:

$$\forall \Phi, \forall \sigma \in STORE_\Phi.exec(\mathbf{if}(p, S;\ \mathbf{if}(q, S_1)), \sigma) \approx exec(\mathbf{if}(p, S;\ S_1), \sigma)$$

Consider the case $\langle p, \sigma\rangle \xrightarrow{*} \langle true, \sigma\rangle$. In this case we have,

$$(\langle S;\ S_1, \sigma\rangle \xrightarrow{*} \sigma'' \Leftrightarrow \langle S;\ \mathbf{if}(q, S_1), \sigma\rangle \xrightarrow{*} \sigma'')$$

Expanding semantic definitions this gives,

$$(\langle S, \sigma\rangle \xrightarrow{*} \sigma' \wedge (\langle S_1, \sigma'\rangle \xrightarrow{*} \sigma'' \Leftrightarrow \langle \mathbf{if}(q, S_1), \sigma'\rangle \xrightarrow{*} \sigma'')) \vee (\neg \exists \sigma' \in STORE_\Phi.\langle S, \sigma\rangle \xrightarrow{*} \sigma' \wedge true)$$

Consider $(\langle S_1, \sigma' \rangle \xrightarrow{*} \sigma'' \Leftrightarrow \langle \mathbf{if}(q, S_1), \sigma' \rangle \xrightarrow{*} \sigma'')$, either $(\langle q, \sigma' \rangle \xrightarrow{*} \langle true, \sigma' \rangle \wedge true)$ or $(\langle q, \sigma' \rangle \xrightarrow{*} \langle false, \sigma' \rangle \wedge false)$.[1] Hence the result follows. □

Lemma 4.1 gives a formal correspondence between correctness proofs (programs with assertions) and program transformations. In other words, we can use Lemma 4.1 to replace the Hoare triple in the (*while fold in context*$_1$) rule by the corresponding transformation. The natural question to ask now is whether the resulting *pure* transformation system is as powerful as the transformation system with (*while fold in context*$_1$) written using the Hoare triple, and the system having Hoare's logic axioms and rules in addition? This is the same as asking whether the transformation system is as *powerful* as Hoare's logic, using Lemma 4.1 to translate between the two. We shall prove that it is, i.e

Theorem 4.1 *if* $\vdash_{HL} \{p\}S\{q\}$
then $\forall S_1.\ \mathbf{if}(p, S; \mathbf{if}(q, S_1)) \Leftrightarrow_W \mathbf{if}(p, S; S_1)$

This states that if $\{p\}S\{q\}$ follows by Hoare's logic (written $\vdash_{HL}$) then the corresponding semantic equivalence, as given by Lemma 4.1, can be shown to hold by our transformation system. Both $\vdash_{HL}$ and $\Leftrightarrow_W$ are relative to an underlying system for the first order language on top of which the **WHILE** language is built.

The proof of Theorem 4.1 is by showing that each axiom and rule of Hoare's logic is derivable via transformations. The single axiom of Hoare's logic forms the base case, and the rules form the step. By the induction hypothesis the corresponding (by Lemma 4.1) semantic equivalences of the premises of these rules can be assumed to follow by our transformation system. The following two lemmas will be used in the derivation of the Hoare's logic **while** rule, and are given here to ease presentation.

[1] For this to follow we must make a technical assumption about our language such that $\exists S_1 \in \mathbf{WHILE}\ .(\langle S_1, \sigma' \rangle \xrightarrow{*} \sigma'' \wedge \sigma' \neq \sigma'')$

Lemma 4.2

if $\forall S_1.\mathbf{if}(p \wedge e, S; \mathbf{if}(p, S_1)) \Leftrightarrow_W \mathbf{if}(p \wedge e, S; S_1)$

then $\forall S_1.\mathbf{if}(p, \mathbf{if}(e \vee \neg p, S); \mathbf{if}(p, S_1)) \Leftrightarrow_W \mathbf{if}(p, \mathbf{if}(e \vee \neg p, S); S_1)$

Proof:

Consider $\mathbf{if}(p, \mathbf{if}(e \vee \neg p, S); \mathbf{if}(p, S_1))$

$$
\begin{aligned}
&\overset{(;\ elim_3)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(e \vee \neg p, S; \mathbf{if}(p, S_1), \mathbf{skip}; \mathbf{if}(p, S_1))) \\
&\overset{(skip\ elim_2),(deriv_6)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}((e \vee \neg p) \wedge p, S; \mathbf{if}(p, S_1), \mathbf{if}(p, S_1)))
\end{aligned}
$$

now since $(e \vee \neg p) \wedge p \;\equiv\; (e \wedge p) \vee (\neg p \wedge p) \;\equiv\; e \wedge p$

$$
\begin{aligned}
&\overset{(bool_1)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(e \wedge p, S; \mathbf{if}(p, S_1), \mathbf{if}(p, S_1))) \\
&\overset{(if\ logic_1\ intro}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(\neg(e \wedge p), \mathbf{if}(p, S_1), S; \mathbf{if}(p, S_1))) \\
&\overset{(deriv_6)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(\neg(e \wedge p) \wedge p, \mathbf{if}(p, S_1), S; \mathbf{if}(p, S_1))) \\
&\overset{(deriv_6)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(\neg(e \wedge p) \wedge p, \mathbf{if}(\neg(e \wedge p) \wedge p \wedge p, S_1), S; \mathbf{if}(p, S_1)))
\end{aligned}
$$

now since $p \wedge p \equiv p \equiv p \wedge true$

$$
\begin{aligned}
&\overset{(bool_1)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(\neg(e \wedge p) \wedge p, \mathbf{if}(\neg(e \wedge p) \wedge p \wedge true, S_1), S; \mathbf{if}(p, S_1))) \\
&\overset{(deriv_6)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(\neg(e \wedge p) \wedge p, \mathbf{if}(true, S_1), S; \mathbf{if}(p, S_1))) \\
&\overset{(deriv_6)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(\neg(e \wedge p), \mathbf{if}(true, S_1), S; \mathbf{if}(p, S_1))) \\
&\overset{(if\ logic_1\ elim)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(e \wedge p, S; \mathbf{if}(p, S_1), \mathbf{if}(true, S_1))) \\
&\overset{(if\ elim_2)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(e \wedge p, S; \mathbf{if}(p, S_1), S_1)) \\
&\overset{(deriv_1)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(e \wedge p, \mathbf{if}(e \wedge p, S; \mathbf{if}(p, S_1)), S_1)) \\
&\overset{(assumption)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(e \wedge p, \mathbf{if}(e \wedge p, S; S_1), S_1)) \\
&\overset{(deriv_1)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(e \wedge p, S; S_1, S_1)) \\
&\overset{(;\ elim_3)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(e \wedge p, S); S_1)
\end{aligned}
$$

□

Lemma 4.3

if $\forall S_1.\ \mathbf{if}(p \wedge e, S; \mathbf{if}(p, S_1)) \Leftrightarrow_W \mathbf{if}(p \wedge e, S; S_1)$

then $\forall S_1.\ \mathbf{if}(p, \mathbf{if}(e, S); \mathbf{if}(p, S_1)) \Leftrightarrow_W \mathbf{if}(p, \mathbf{if}(e, S); S_1)$

Proof:

Consider $\mathbf{if}(p, \mathbf{if}(e, S); \mathbf{if}(p, S_1))$

$$
\begin{array}{cl}
\overset{(;\ elim_3)}{\Leftrightarrow_W} & \mathbf{if}(p, \mathbf{if}(e, S; \mathbf{if}(p, S_1), \mathbf{skip}; \mathbf{if}(p, S_1))) \\
\overset{(if\ logic_1)}{\Leftrightarrow_W} & \mathbf{if}(\neg p, \mathbf{skip}, \mathbf{if}(e, S; \mathbf{if}(p, S_1), \mathbf{skip}; \mathbf{if}(p, S_1))) \\
\overset{(if\ logic_2)}{\Leftrightarrow_W} & \mathbf{if}(\neg p, \mathbf{skip}, \mathbf{if}(p \wedge e, S; \mathbf{if}(p, S_1), \mathbf{skip}; \mathbf{if}(p, S_1))) \\
\overset{(deriv_1)}{\Leftrightarrow_W} & \mathbf{if}(\neg p, \mathbf{skip}, \mathbf{if}(p \wedge e, \mathbf{if}(p \wedge e, S; \mathbf{if}(p, S_1)), \mathbf{skip}; \mathbf{if}(p, S_1))) \\
\overset{(assumption),(deriv_1)}{\Leftrightarrow_W} & \mathbf{if}(\neg p, \mathbf{skip}, \mathbf{if}(p \wedge e, S; S_1, \mathbf{skip}; \mathbf{if}(p, S_1))) \\
\overset{(if\ logic_1),(skip\ elim_2)}{\Leftrightarrow_W} & \mathbf{if}(p, \mathbf{if}(p \wedge e, S; S_1, \mathbf{if}(p, S_1))) \\
\overset{(deriv_6),(if\ logic_2\ intro)}{\Leftrightarrow_W} & \mathbf{if}(p, \mathbf{if}(e, S; S_1, \mathbf{if}(p \wedge \neg e, S_1))) \\
\overset{(deriv_6),(deriv_2)}{\Leftrightarrow_W} & \mathbf{if}(p, \mathbf{if}(e, S; S_1, S_1)) \\
\overset{(skip\ intro_2),(;\ intro_3)}{\Leftrightarrow_W} & \mathbf{if}(p, \mathbf{if}(e, S); S_1)
\end{array}
$$

□

Proof of Theorem 4.1:

base:

assignment axiom	$\{p[t/x]\}x := t\{p\}$

Consider $\mathbf{if}(p[t/x], x := t; \mathbf{if}(p, S_1))$

$$
\begin{array}{cl}
\overset{(;\ elim_2)}{\Leftrightarrow_W} & \mathbf{if}(p[t/x], \mathbf{if}(p[t/x], x := t; S_1, x := t; \mathbf{skip})) \\
\overset{(deriv_1)}{\Leftrightarrow_W} & \mathbf{if}(p[t/x], x := t; S_1)
\end{array}
$$

Hence by soundness of rules,

$$\mathbf{if}(p[t/x], x := t; \mathbf{if}(p, S_1)) \equiv \mathbf{if}(p[t/x], x := t; S_1)$$

and so by Lemma 4.1 $\{p[t/x]\}x := t\{p\}$

step: we proceed by cases.

composition rule	$\dfrac{\{p\}S_1\{r\}, \{r\}S_2\{q\}}{\{p\}S_1; S_2\{q\}}$

We may assume

$$\forall S_3.\ \mathbf{if}(p, S_1;\ \mathbf{if}(r, S_3)) \ \Leftrightarrow_W \ \mathbf{if}(p, S_1;\ S_3) \tag{4.1}$$

$$\forall S_3.\ \mathbf{if}(r, S_2;\ \mathbf{if}(q, S_3)) \ \Leftrightarrow_W \ \mathbf{if}(r, S_2;\ S_3) \tag{4.2}$$

Consider $\mathbf{if}(p, S_1;\ \mathbf{if}(r, S_2;\ \mathbf{if}(q, S_3)))$

$$\begin{aligned} &\overset{(4.1)}{\Leftrightarrow_W} && \mathbf{if}(p, S_1;\ S_2;\ \mathbf{if}(q, S_3)) \\ &\overset{(assoc;)}{\Leftrightarrow_W} && \mathbf{if}(p, (S_1;\ S_2);\ \mathbf{if}(q, S_3)) \end{aligned}$$

Also $\mathbf{if}(p, S_1;\ \mathbf{if}(r, S_2;\ \mathbf{if}(q, S_3)))$

$$\begin{aligned} &\overset{(4.2)}{\Leftrightarrow_W} && \mathbf{if}(p, S_1;\ \mathbf{if}(r, S_2;\ S_3)) \\ &\overset{(4.1)}{\Leftrightarrow_W} && \mathbf{if}(p, S_1;\ S_2;\ S_3) \\ &\overset{(assoc;)}{\Leftrightarrow_W} && \mathbf{if}(p, (S_1;\ S_2);\ S_3) \end{aligned}$$

Hence by soundness of rules

$$\mathbf{if}(p, (S_1;\ S_2);\ \mathbf{if}(q, S_3)) \equiv \mathbf{if}(p, (S_1;\ S_2);\ S_3)$$

and so by Lemma 4.1 $\{p\}S_1;\ S_2\{q\}$

if then else rule	$\dfrac{\{p \wedge e\}S_1\{q\},\ \{p \wedge \neg e\}S_2\{q\}}{\{p\}\mathbf{if}(e, S_1, S_2)\{q\}}$

We may assume

$$\forall S_3.\ \mathbf{if}(p \wedge e, S_1;\ \mathbf{if}(q, S_3)) \ \Leftrightarrow_W \ \mathbf{if}(p \wedge e, S_1;\ S_3) \tag{4.3}$$

$$\forall S_3.\ \mathbf{if}(p \wedge \neg e, S_2;\ \mathbf{if}(q, S_3)) \ \Leftrightarrow_W \ \mathbf{if}(p \wedge \neg e, S_2;\ S_3) \tag{4.4}$$

Consider $\mathbf{if}(p, \mathbf{if}(e, S_1, S_2);\ \mathbf{if}(q, S_3))$

$$\begin{aligned} &\overset{(;\ elim_3)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(e, S_1;\ \mathbf{if}(q, S_3), S_2;\ \mathbf{if}(q, S_3))) \\ &\overset{(deriv_2)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(p \wedge e, S_1;\ \mathbf{if}(q, S_3), \mathbf{if}(p \wedge \neg e, S_2;\ \mathbf{if}(q, S_3,)))) \\ &\overset{(4.4)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(p \wedge e, S_1;\ \mathbf{if}(q, S_3), \mathbf{if}(p \wedge \neg e, S_2;\ S_3))) \\ &\overset{(4.3)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(p \wedge e, S_1;\ S_3, \mathbf{if}(p \wedge \neg e, S_2;\ S_3))) \\ &\overset{(deriv_2)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(e, S_1;\ S_3, S_2;\ S_3)) \\ &\overset{(;\ elim_3)}{\Leftrightarrow_W} && \mathbf{if}(p, \mathbf{if}(e, S_1, S_2);\ S_3) \end{aligned}$$

Hence by soundness of rules

$$\mathbf{if}(p, \mathbf{if}(e, S_1, S_2);\ \mathbf{if}(q, S_3)) \equiv \mathbf{if}(p, \mathbf{if}(e, S_1, S_2);\ S_3)$$

and so by Lemma 4.1 $\{p\}\mathbf{if}(e, S_1, S_2)\{q\}$.

while rule	$\dfrac{\{p \wedge e\}S\{p\}}{\{p\}\mathbf{while}(e, S)\{p \wedge \neg e\}}$

We may assume

$$\forall S_1.\ \mathbf{if}(p \wedge e, S;\ \mathbf{if}(p, S_1)) \ \Leftrightarrow_W\ \mathbf{if}(p \wedge e, S;\ S_1) \tag{4.5}$$

Consider $\mathbf{if}(p, \mathbf{while}(e, S);\ \mathbf{if}(p \wedge \neg e, S_1))$

$$\overset{(deriv_4)}{\Rightarrow_W}\ \mathbf{if}(p, \mathbf{if}(p, \mathbf{while}(e, S));\ \mathbf{if}(p \wedge \neg e, S_1)) \tag{4.6}$$

Now consider the following subprogram of (4.6), $\mathbf{if}(p, \mathbf{while}(e, S))$

$$\begin{array}{cl} \overset{(while\ unfold)}{\Rightarrow_W} & \mathbf{if}(p, \mathbf{if}(e, S;\ \mathbf{while}(e, S))) \\ \overset{(deriv_5)}{\Rightarrow_W} & \mathbf{if}(p, \mathbf{if}(e \vee \neg p, S;\ \mathbf{while}(e, S))) \\ \overset{(while\ fold\ in\ context_1)}{\Rightarrow_W} & \mathbf{if}(p, \mathbf{while}(e \vee \neg p, S)) \end{array}$$

To justify the last step we must prove $\{p\}\mathbf{if}(e \vee \neg p, S)\{p\}$; this follows, in transformation terms, via Lemma 4.2 ((4.5) is the assumption).

Substituting this back into (4.6), i.e using monotonicity rules.

$$\begin{array}{cl} \Rightarrow_W & \mathbf{if}(p, \mathbf{if}(p, \mathbf{while}(e \vee \neg p, S));\ \mathbf{if}(p \wedge \neg e, S_1)) \\ \overset{(deriv_4)}{\Rightarrow_W} & \mathbf{if}(p, \mathbf{while}(e \vee \neg p, S);\ \mathbf{if}(p \wedge \neg e, S_1)) \\ & \text{now since } \neg(p \wedge \neg e)\ \equiv\ \neg p \vee \neg(\neg e)\ \equiv\ \neg p \vee e \\ \overset{(while\ logic)}{\Rightarrow_W} & \mathbf{if}(p, \mathbf{while}(e \vee \neg p, S);\ S_1) \end{array}$$

$$\overset{(deriv_4)}{\Rightarrow_W}\ \mathbf{if}(p, \mathbf{if}(p, \mathbf{while}(e \vee \neg p, S));\ S_1) \tag{4.7}$$

Consider the following subprogram of (4.7), $\mathbf{if}(p, \mathbf{while}(e \vee \neg p, S))$

$$\begin{array}{cl} \overset{(while\ unfold)}{\Rightarrow_W} & \mathbf{if}(p, \mathbf{if}(e \vee \neg p, S;\ \mathbf{while}(e \vee \neg p, S))) \\ \overset{(deriv_5)}{\Rightarrow_W} & \mathbf{if}(p, \mathbf{if}(e, S;\ \mathbf{while}(e \vee \neg p, S))) \\ \overset{(while\ fold\ in\ context_1)}{\Rightarrow_W} & \mathbf{if}(p, \mathbf{while}(e, S)) \end{array}$$

To justify this last step we must prove $\{p\}\mathbf{if}(e,S)\{p\}$; this follows, in transformation terms via Lemma 4.3 ((4.5) is the assumption).

Substituting this back into (4.7), i.e using monotonicity rule.

$$\begin{array}{ll} \Rightarrow_W & \mathbf{if}(p,\mathbf{if}(p,\mathbf{while}(e,S));\ S_1) \\ \overset{(deriv_4)}{\Rightarrow_W} & \mathbf{if}(p,\mathbf{while}(e,S);\ S_1) \end{array}$$

Hence by soundness, since we have shown above that

$$\mathbf{if}(p,\mathbf{while}(e,S);\ \mathbf{if}(p \wedge \neg e, S_1)) \Rightarrow_W \mathbf{if}(p,\mathbf{while}(e,S);\ S_1)$$

then

$$\mathbf{if}(p,\mathbf{while}(e,S);\ \mathbf{if}(p \wedge \neg e, S_1)) \ \sqsubseteq \ \mathbf{if}(p,\mathbf{while}(e,S);\ S_1) \tag{4.8}$$

We now prove the reverse to get strong equivalence, thus consider $\mathbf{if}(p,\mathbf{while}(e,S);\ S_1)$

$$\overset{(deriv_4)}{\Leftrightarrow_W} \ \mathbf{if}(p,\mathbf{if}(p,\mathbf{while}(e,S));\ S_1) \tag{4.9}$$

Consider the following subprogram of (4.9), $\mathbf{if}(p,\mathbf{while}(e,S))$

$$\begin{array}{rl} \overset{(while\ unfold)}{\Rightarrow_W} & \mathbf{if}(p,\mathbf{if}(e,S;\ \mathbf{while}(e,S))) \\ \overset{(deriv_5)}{\Rightarrow_W} & \mathbf{if}(p,\mathbf{if}(e \vee \neg p,S;\ \mathbf{while}(e,S))) \\ \overset{(while\ fold\ in\ context_1)}{\Rightarrow_W} & \mathbf{if}(p,\mathbf{while}(e \vee \neg p,S)) \end{array}$$

To justify this last step we need to prove $\{p\}\mathbf{if}(e \vee \neg p,S)\{p\}$; this we have already proven in Lemma 4.2. Substituting this back into (4.9), i.e using monotonicity rule, we have

$$\begin{array}{rl} \Rightarrow_W & \mathbf{if}(p,\mathbf{if}(p,\mathbf{while}(e \vee \neg p,S));\ S_1) \\ \overset{(deriv_4)}{\Rightarrow_W} & \mathbf{if}(p,\mathbf{while}(e \vee \neg p,S);\ S_1) \\ \overset{(while\ logic)}{\Rightarrow_W} & \mathbf{if}(p,\mathbf{while}(e \vee \neg p,S);\ \mathbf{if}(p \wedge \neg e,S_1)) \end{array}$$

$$\overset{(deriv_4)}{\Rightarrow_W} \ \mathbf{if}(p,\mathbf{if}(p,\mathbf{while}(e \vee \neg p,S));\ \mathbf{if}(p \wedge \neg e,S_1)) \tag{4.10}$$

Consider the following subprogram of (4.10), $\mathbf{if}(p,\mathbf{while}(e \vee \neg p,S))$. From above, where we considered such a subprogram before, we have that

$$\Rightarrow_W \ \mathbf{if}(p,\mathbf{while}(e,S))$$

Substituting this back into (4.10)

$$\begin{array}{ll} \Rightarrow_W & \mathbf{if}(p, \mathbf{if}(p, \mathbf{while}(e, S)); \mathbf{if}(p \wedge \neg e, S_1)) \\ \overset{(deriv_4)}{\Rightarrow_W} & \mathbf{if}(p, \mathbf{while}(e, S); \mathbf{if}(p \wedge \neg e, S_1)) \end{array}$$

Hence by soundness, since we have shown above that

$$\mathbf{if}(p, \mathbf{while}(e, S); S_1) \Rightarrow_W \mathbf{if}(p, \mathbf{while}(e, S); \mathbf{if}(p \wedge \neg e, S_1))$$

then

$$\mathbf{if}(p, \mathbf{while}(e, S); S_1) \sqsubseteq \mathbf{if}(p, \mathbf{while}(e, S); \mathbf{if}(p \wedge \neg e, S_1))$$

Hence by (4.8),

$$\mathbf{if}(p, \mathbf{while}(e, S); S_1) \equiv \mathbf{if}(p, \mathbf{while}(e, S); \mathbf{if}(p \wedge \neg e, S_1))$$

consequence rule	$\dfrac{p \Rightarrow p_1 \ \{p_1\}S\{q_1\} \ q_1 \Rightarrow q}{\{p\}S\{q\}}$

We may thus assume

$$\forall S_1.\ \mathbf{if}(p_1, S; \mathbf{if}(q_1, S_1)) \Leftrightarrow_W \mathbf{if}(p_1, S; S_1) \tag{4.11}$$

$$p \Rightarrow p_1\ (i.e\ \neg p \vee p_1) \equiv true \tag{4.12}$$

$$q_1 \Rightarrow q\ (i.e\ \neg q_1 \vee q) \equiv true \tag{4.13}$$

Consider $\mathbf{if}(p_1, S; \mathbf{if}(q_1, S_1))$, using (4.13) and $q_1 \equiv q_1 \wedge (\neg q_1 \vee q) \equiv q_1 \wedge q$

$$\begin{array}{ll} \overset{(bool_1)}{\Leftrightarrow_W} & \mathbf{if}(p_1, S; \mathbf{if}(q_1 \wedge q, S_1)) \\ \overset{(\wedge\ elim)}{\Leftrightarrow_W} & \mathbf{if}(p_1, S; \mathbf{if}(q_1, \mathbf{if}(q, S_1, S_2), S_2)) \\ \overset{(4.11)}{\Leftrightarrow_W} & \mathbf{if}(p_1, S; \mathbf{if}(q, S_1)) \end{array}$$

Hence by soundness of the rules (and symmetry of $\equiv$) and using (4.11)

$$\mathbf{if}(p_1, S; \mathbf{if}(q, S_1)) \equiv \mathbf{if}(p_1, S; S_1) \tag{4.14}$$

Now consider $\mathbf{if}(p, S; \mathbf{if}(q, S_1))$, using (4.12) and $p \equiv p \wedge (\neg p \vee p_1) \equiv p \wedge p_1$

$$\begin{array}{ll} \overset{(bool_1)}{\Leftrightarrow_W} & \mathbf{if}(p \wedge p_1, S; \mathbf{if}(q, S_1)) \\ \overset{(\wedge\ elim)}{\Leftrightarrow_W} & \mathbf{if}(p, \mathbf{if}(p_1, S; \mathbf{if}(q, S_1))) \\ \overset{(4.14)}{\Leftrightarrow_W} & \mathbf{if}(p, \mathbf{if}(p_1, S; S_1)) \\ \overset{(\wedge\ intro)}{\Leftrightarrow_W} & \mathbf{if}(p \wedge p_1, S; S_1) \\ \overset{(bool_1)}{\Leftrightarrow_W} & \mathbf{if}(p, S; S_1) \end{array}$$

Hence by soundness

$$\mathbf{if}(p, S; \mathbf{if}(q, S_1)) \equiv \mathbf{if}(p, S; S_1)$$

□

There seems to be a circularity at first glance. We use (*while fold in context*$_1$) in our derivation of the Hoare axioms and rules, and the (*while fold in context*$_1$) rule uses Hoare triples! But if we merely rewrite this part of the rule as the equivalent transformation, then there is clearly no such confusion. The derivations then go to show that by making this replacement, instead of keeping Hoare triples and boosting the system with Hoare axioms and rules, we lose nothing.

4.2 UF Transformation System

In this section we shall briefly describe the UF transformation system of Burstall and Darlington (see **[Burstall/Darlington 77]**). This system has two major assets: it uses a functional language of first order recursion equations, allowing succinct programs and a simple transformation system; although a simple transformation system it is powerful due to the fact that it uses rules as well as axioms (the explicit use of rules is hidden by the use of multiple definitions). We shall use a slight variant of the original language and system. We define the syntax as follows:

$$\mathbf{UF\text{-}lang} \;::=\; t_0 : \left\{ \begin{array}{ccc} G_1(\bar{x}) & \Leftarrow & t_1 \\ & \& & \\ & \vdots & \\ & \& & \\ G_m(\bar{y}) & \Leftarrow & t_m \end{array} \right\}$$

$$t \;::=\; v \mid x \mid f_i^n(t_1, \ldots, t_n) \mid \mathbf{if}(b, t_1, t_2) \mid G_i(t) \;\; i = 1, \ldots, m \mid (t_1, \ldots, t_k)$$

A program in **UF-lang** is a *term* and a set of recursive equations. The term t_0 can be seen as the main program and the recursive equations as user-defined functions. Terms of **UF-lang** are made up values v, variables x, basic functions of other terms

(*Defn. Intro.*)	$t :\{E\} \Rightarrow_{UF} t :\{E \& G(\bar{x}) \Leftarrow t_1\} \quad G \notin E$
(*Unfolding*)	$\dfrac{G_0(\bar{x}) \Leftarrow t_0, G_1(\bar{y}) \Leftarrow t_1(G_0(\bar{t})) \in E}{t :\{E\} \Rightarrow_{UF} t :\{E \& G_1(\bar{y}) \Leftarrow t_1[t_0[\bar{t}/\bar{x}]/G_0(\bar{t})]\}}$
(*Folding*)	$\dfrac{G_0(\bar{x}) \Leftarrow t_0, G_1(\bar{y}) \Leftarrow t_1(t_0[\bar{t}/\bar{x}]) \in E}{t :\{E\} \Rightarrow_{UF} t :\{E \& G_1(\bar{y}) \Leftarrow t_1[G_0(\bar{t})/t_0[\bar{t}/\bar{x}]]\}}$
(*Proc. Unfold*)	$G(\bar{t}) :\{E \& G(\bar{x}) \Leftarrow t_0\} \Rightarrow_{UF} t_0[\bar{t}/\bar{x}] :\{E \& G(\bar{x}) \Leftarrow t_0\}$
(*Proc. Fold*)	$t_0[\bar{t}/\bar{x}] :\{E \& G(\bar{x}) \Leftarrow t_0\} \Rightarrow_{UF} G(\bar{t}) :\{E \& G(\bar{x}) \Leftarrow t_0\}$
(*Laws*)	$\dfrac{t = t_1}{t :\{E\} \Rightarrow_{UF} t_1 :\{E\}}$
(*Laws*)	$\dfrac{t = t_1}{t_0 :\{E \& G(\bar{x}) \Leftarrow t\} \Rightarrow_{UF} t_0 :\{E \& G(\bar{x}) \Leftarrow t_1\}}$

Figure 4–1: UF transformations

f_i^n (for example '+'), conditional terms, non-basic functions G_i, and tuples of terms (we abbreviate $(t_1, \ldots, t_k)$ by $\bar{t}$). We shall not give the conditions for a **UF-lang** program to be well-formed, such as for each equation $G(\bar{x}) \Leftarrow t$, $X_k \supseteq FV(t)$, or give a semantics for the **UF-lang**. The reader is referred to **[Manna 74]** or **[Pettorossi 84]** for such details.

The UF transformations (over **UF-lang**) are defined in Fig. 4–1, by the relation $\Rightarrow_{UF}$. We use E to denote a set of equations, and $G \notin E$ to mean that there does not exist an equation $G(\bar{x}) \Leftarrow t$ in E, i.e the non-basic function name G is new. We use $t'(t)$ to mean that t occurs in t'. The (*Folding*) and (*Unfolding*) axioms may be used with $G_0(\bar{x})$ and $G_1(\bar{y})$ as the same equation. The (*Proc. Unfold*) axiom (and its reverse (*Proc. Fold*)) are only needed because of the syntax change. The (*Laws*) axioms use '=' to denote equivalence ot terms, so that any term may be replaced by any equivalent term. Note that via the UF transformations we can

introduce multiple, but consistent, definitions for G_i. We therefore also include a (*Defn. Elim.*) transformation, to eliminate such multiple definitions; this was implicit in the original system. The transformations only preserve weak equivalence, i.e they may introduce non-termination (see **[Kott 85]** for a simple example showing how non-termination can be introduced). We assume that the transformations are used in such a way that all the resulting programs are well-formed.

We give two examples illustrating the use of the UF system on **UF-lang** programs.

Example 4.1

$$G_2(G_3(x+1)) : \left\{ \begin{array}{l} G_2(x) \Leftarrow \mathbf{if}(b_2, G_2(G_3(x+1)), x) \\ G_3(x) \Leftarrow \mathbf{if}(b_1, G_3(x+1), x) \end{array} \right\}$$

$\Rightarrow_{UF}$

$$G_1(x+1) : \left\{ \begin{array}{l} G_1(x) \Leftarrow \mathbf{if}(b_1, G_1(x+1), G_2(x)) \\ G_2(x) \Leftarrow \mathbf{if}(b_2, G_2(G_3(x+1)), x) \\ G_3(x) \Leftarrow \mathbf{if}(b_1, G_3(x+1), x) \end{array} \right\}$$

Proof:

Let us denote the functional program on the left hand side of $\Rightarrow_{UF}$ by $t : \{E\}$, then $t : \{E\}$ can be transformed as follows:

$$\overset{(Defn.\ Intro.)}{\Rightarrow_{UF}} \quad t : \{E \& G_1(x) \Leftarrow G_2(G_3(x))\}$$

$$\overset{(Proc.\ Fold)}{\Rightarrow_{UF}} \quad G_1(x+1) : \{E \& G_1(x) \Leftarrow G_2(G_3(x))\}$$

$$\overset{(Unfolding)}{\Rightarrow_{UF}} \quad G_1(x+1) : \left\{ \begin{array}{l} E \& \\ G_1(x) \Leftarrow G_2(G_3(x)) \& \\ G_1(x) \Leftarrow G_2(\mathbf{if}(b_1, G_3(x+1), x)) \end{array} \right\}$$

$$\overset{(Laws)}{\Rightarrow_{UF}} \quad G_1(x+1) : \left\{ \begin{array}{l} E \& \\ G_1(x) \Leftarrow G_2(G_3(x)) \& \\ G_1(x) \Leftarrow \mathbf{if}(b_1, G_2(G_3(x+1)), G_2(x)) \end{array} \right\}$$

$$\overset{(Folding)}{\Rightarrow_{UF}} \quad G_1(x+1) : \left\{ \begin{array}{l} E \& \\ G_1(x) \Leftarrow G_2(G_3(x)) \& \\ G_1(x) \Leftarrow \mathbf{if}(b_1, G_1(x+1), G_2(x)) \end{array} \right\}$$

$$\overset{(Defn.\ Elim.)}{\Rightarrow_{UF}} \quad G_1(x+1) : \left\{ \begin{array}{l} E \& \\ G_1(x) \Leftarrow \mathbf{if}(b_1, G_1(x+1), G_2(x)) \end{array} \right\}$$

□

We use this example in the following example, which is the functional equivalent of Example 3.2.

Example 4.2

$$G_1(x) : \left\{ \begin{array}{l} G_1(x) \Leftarrow \mathbf{if}(b_1, G_1(x+1), G_2(x)) \\ G_2(x) \Leftarrow \mathbf{if}(b_2, G_2(G_3(x+1)), x) \\ G_3(x) \Leftarrow \mathbf{if}(b_1, G_3(x+1), x) \end{array} \right\}$$

$\Rightarrow_{UF}$

$$G_4(x) : \{G_4(x) \Leftarrow \mathbf{if}(b_1 \vee b_2, G_4(x+1), x)\}$$

Proof:

Let us denote the functional program on the left hand side of $\Rightarrow_{UF}$ by $G_1(x) : \{E\}$,

then $G_1(x) :\{E\}$ can be transformed as follows:

$$\overset{(Defn.\ Intro.)}{\Rightarrow_{UF}} \quad G_1(x) :\{E \& G_4(x) \Leftarrow G_1(x)\}$$

$$\overset{(Proc.\ Fold)}{\Rightarrow_{UF}} \quad G_4(x) :\{E \& G_4(x) \Leftarrow G_1(x)\}$$

$$\overset{(Unfolding)}{\Rightarrow_{UF}} \quad G_4(x) : \left\{ \begin{array}{l} E\& \\ G_4(x) \Leftarrow G_1(x)\& \\ G_4(x) \Leftarrow \mathbf{if}(b_1, G_1(x+1), \mathbf{if}(b_2, G_2(G_3(x+1)), x)) \end{array} \right\}$$

$$\overset{(Laws)}{\Rightarrow_{UF}} \quad G_4(x) : \left\{ \begin{array}{l} E\& \\ G_4(x) \Leftarrow G_1(x)\& \\ G_4(x) \Leftarrow \mathbf{if}(\neg b_1, \mathbf{if}(\neg b_2, x, G_2(G_3(x+1))), G_1(x+1)) \end{array} \right\}$$

$$\overset{Example\ 4.1}{\Rightarrow_{UF}} \quad G_4(x) : \left\{ \begin{array}{l} E\& \\ G_4(x) \Leftarrow G_1(x)\& \\ G_4(x) \Leftarrow \mathbf{if}(\neg b_1, \mathbf{if}(\neg b_2, x, G_1(x+1)), G_1(x+1)) \end{array} \right\}$$

$$\overset{(Laws)}{\Rightarrow_{UF}} \quad G_4(x) : \left\{ \begin{array}{l} E\& \\ G_4(x) \Leftarrow G_1(x)\& \\ G_4(x) \Leftarrow \mathbf{if}(b_1 \vee b_2, G_1(x+1), x) \end{array} \right\}$$

$$\overset{(Folding)}{\Rightarrow_{UF}} \quad G_4(x) : \left\{ \begin{array}{l} E\& \\ G_4(x) \Leftarrow G_1(x)\& \\ G_4(x) \Leftarrow \mathbf{if}(b_1 \vee b_2, G_1(x+1), x) \\ G_4(x) \Leftarrow \mathbf{if}(b_1 \vee b_2, G_4(x+1), x) \end{array} \right\}$$

$$\overset{(Defn.\ Elim.)}{\Rightarrow_{UF}} \quad G_4(x) : \{G_4(x) \Leftarrow \mathbf{if}(b_1 \vee b_2, G_4(x+1), x)\}$$

□

4.3 Derivation of W via UF

In this section we give a set of transformations enabling us to translate **WHILE** programs into equivalent **UF-lang** programs. A similar set of transformations is given in **[Henderson 80]**. We prove that some of the (functional) axioms and rules induced by translation from W can be derived via UF. In trying to derive W via UF we will show some possible weaknesses of UF.

(*in elim*$_1$)	$[\textbf{skip}]\ \textbf{in}\ t \ \rightarrow\ t$
(*in elim*$_2$)	$[\bar{x} := \bar{c}]\ \textbf{in}\ t \ \rightarrow\ t[\bar{c}/\bar{x}]$
(*in elim*$_3$)	$[w_1;\ w_2]\ \textbf{in}\ t \ \rightarrow\ [w_1]\ \textbf{in}\ ([w_2]\ \textbf{in}\ t)$
(*in elim*$_4$)	$[\textbf{if}(b, w_1, w_2)]\ \textbf{in}\ t \ \rightarrow\ \textbf{if}(b, [w_1]\ \textbf{in}\ t, [w_2]\ \textbf{in}\ t)$
(*in elim*$_5$)	$[\textbf{while}(b, w)]\ \textbf{in}\ t \ \rightarrow\ G(\bar{y})$ and add the following equation: $G(\bar{y}) \Leftarrow \textbf{if}(b, G([w]\ \textbf{in}\ \bar{y}), t)$ where $Y_k \triangleq VAR(b) \cup VAR(w) \cup FV(t)$

Figure 4–2: WHILE → UF-lang translation procedure

We (temporarily) extend **UF-lang** by adding the following clause to terms:

$$[\textbf{WHILE}\ ]\ \textbf{in}\ t$$

The value of $[w]$ **in** t is the value of t in the context produced by executing w. In order to translate $w \in$ **WHILE** into a (functional) program of **UF-lang** we must choose a tuple of variables whose value we are interested in after w has executed. Let us suppose we choose $\bar{x}$. We can construct $[w]$ **in** $\bar{x}$ as a program in the temporarily extended **UF-lang,** and use the transformations in Fig. 4–2 to drive out the **in** construct. The resulting program is a program in **UF-lang** and, provided the transformations in Fig. 4–2 are sound, is equivalent to $[w]$ **in** $\bar{x}$. The set of transformations over the extended **UF-lang**, as given in Fig. 4–2, have a *translation property.* The transformations are such that if $A \rightarrow B$, then A can be replaced by B, where A contains a subexpression of the form $[w]$ **in** t and B either contains no such expressions or contains only shorter expressions of this kind. Hence repeated application of these transformations will drive **in** expressions out of the program.

To illustrate how the set of transformations in Fig. 4–2 may be used as a translation procedure, and to show how a transformation in W may be mimicked in UF, we consider the following example:

$$x := 0 \Rightarrow_W x := 0;\ \mathbf{while}(x > 0, x := x - 1;\ y := y + 1) \quad (A)$$

This transformation is derivable via W using (*skip intro*$_3$), (*if intro*$_3$), (; *elim*$_2$), (*bool*$_1$), (; *intro*$_2$) and (*while unfold*). In order to use the transformations of Fig. 4–2, and so translate imperative transformation (A) into a functional transformation, we construct $[x := 0]$ **in** (x, y) and $[x := 0;\ \mathbf{while}(x > 0, x := x - 1;\ y := y + 1)]$ **in** (x, y). Using (*in elim*$_2$), $[x := 0]$ **in** (x, y) becomes the term (0,y). Now let us consider how $[x := 0;\ \mathbf{while}(x > 0, x := x - 1;\ y := y + 1)]$ **in** (x, y) translates:

$$\begin{array}{ll}
\overset{(in\ elim_3)}{\rightarrow} & [x := 0]\ \mathbf{in}\ [\mathbf{while}(x > 0, x := x - 1;\ y := y + 1)]\ \mathbf{in}\ (x, y) \\
\overset{(in\ elim_5)}{\rightarrow} & [x := 0]\ \mathbf{in}\ (G(x, y)) : \\
 & \{G(x, y) \Leftarrow \mathbf{if}(x > 0, G([x := x - 1;\ y := y + 1]\ \mathbf{in}\ (x, y)), (x, y))\} \\
\overset{(in\ elim_3),(in\ elim_2)}{\rightarrow} & G(0, y) : \{G(x, y) \Leftarrow \mathbf{if}(x > 0, G(x - 1, y + 1), (x, y))\}
\end{array}$$

Therefore, the imperative transformation (A) translates into the following functional transformation:

$$(0, y) \Rightarrow_F G(0, y) : \{G(x, y) \Leftarrow \mathbf{if}(x > 0, G(x - 1, y + 1), (x, y))\} \quad (B)$$

We use $\Rightarrow_F$ to denote the functional transformation system induced by translation (using the transformations in Fig. 4–2) from $\Rightarrow_W$. We now show how (B) may be derived using the UF transformation system.

$$\begin{array}{ll}
 & (0, y) \\
\overset{(Defn.\ Intro.)}{\Rightarrow_{UF}} & (0, y) : \{G(x, y) \Leftarrow \mathbf{if}(x > 0, G(x - 1, y + 1), (x, y))\} \\
\overset{(Laws)}{\Rightarrow_{UF}} & \mathbf{if}(0 > 0, G(0 - 1, y + 1), (0, y)) : \\
 & \{G(x, y) \Leftarrow \mathbf{if}(x > 0, G(x - 1, y + 1), (x, y))\} \\
\overset{(Proc.\ Fold)}{\Rightarrow_{UF}} & G(0, y) : \{G(x, y) \Leftarrow \mathbf{if}(x > 0, G(x - 1, y + 1), (x, y))\}
\end{array}$$

Note that in this derivation (*if intro*$_3$) and (*bool*$_1$) have been mimicked in UF via (*Laws*). What we attempt to prove in the remainder of this section is that

$\Rightarrow_F \subseteq \Rightarrow_{UF}$, i.e the functional transformations induced from W are derivable via UF.

We proceed by translating each side of an imperative axiom to give a functional axiom, which we then attempt to show derivable via the UF system (similarly for the imperative rules).

Lemma 4.4 *The induced functional transformation of (; elim$_3$) is reflexivity.*

Proof:

Consider how the left hand side of (; $elim_3$) is translated, i.e

$$\begin{array}{rl} & [\mathbf{if}(b, p_0, p_1);\ p_2]\ \mathbf{in}\ \bar{x} \\ \overset{(in\ elim_3)}{\rightarrow} & [\mathbf{if}(b, p_0, p_1)]\ \mathbf{in}\ ([p_2]\ \mathbf{in}\ \bar{x}) \\ \overset{(in\ elim_4)}{\rightarrow} & \mathbf{if}(b, t_0, t_1) :\{E_0 \& E_1 \& E_2\} \end{array}$$

where we assume

$$\begin{array}{rcl} [p_2]\ \mathbf{in}\ \bar{x} & \overset{*}{\rightarrow} & t_2 :\{E_2\} \\ [p_1]\ \mathbf{in}\ t_2 & \overset{*}{\rightarrow} & t_1 :\{E_1\} \\ [p_0]\ \mathbf{in}\ t_2 & \overset{*}{\rightarrow} & t_0 :\{E_0\} \end{array}$$

Now consider how the right hand side is translated,

$$\begin{array}{rl} & [\mathbf{if}(b, p_0;\ p_2, p_1;\ p_2)]\ \mathbf{in}\ \bar{x} \\ \overset{(in\ elim_4)}{\rightarrow} & \mathbf{if}(b, [p_0;\ p_2]\ \mathbf{in}\ \bar{x}, [p_1;\ p_2]\ \mathbf{in}\ \bar{x}) \\ \overset{(in\ elim_3)}{\rightarrow} & \mathbf{if}(b, [p_0]\ \mathbf{in}\ ([p_2]\ \mathbf{in}\ \bar{x}), [p_1]\ \mathbf{in}\ ([p_2]\ \mathbf{in}\ \bar{x})) \end{array}$$

Using the same assumptions as above, we get exactly the translation of the left hand side. □

Other **if** axioms will correspond to other simple properties (or (*Laws*)) of **if** in UF. The more interesting cases are those axioms and rules added to the **IF** system to give the powerful **WHILE** system.

Lemma 4.5 *The induced functional transformation of (while logic) is derivable via (Laws) in UF.*

Proof:

Consider the left hand side of (*while logic*):

$$\begin{array}{ll} & [\mathbf{while}(b,p);\,\mathbf{if}(\neg b,p_0,p_1)]\ \mathbf{in}\ \bar{x} \\ \stackrel{(in\ elim_3),(in\ elim_4)}{\longrightarrow} & [\mathbf{while}(b,p)]\ \mathbf{in}\ \mathbf{if}(\neg b,t_0,t_1)\ :\{E_0\&E_1\} \\ \stackrel{(in\ elim_5)}{\longrightarrow} & G(\bar{y})\ :\{E_0\&E_1\&G(\bar{y}) \Leftarrow \mathbf{if}(b,G(t),\mathbf{if}(\neg b,t_0,t_1))\&E\} \end{array}$$

where we assume

$$\begin{array}{lll} [p_0]\ \mathbf{in}\ \bar{x} & \stackrel{*}{\rightarrow} & t_0\ :\{E_0\} \\ [p_1]\ \mathbf{in}\ \bar{x} & \stackrel{*}{\rightarrow} & t_1\ :\{E_1\} \\ [p]\ \mathbf{in}\ \bar{y} & \stackrel{*}{\rightarrow} & t\ :\{E\} \end{array}$$

Now consider the right hand side:

$$\begin{array}{ll} & [\mathbf{while}(b,p);\,p_0]\ \mathbf{in}\ \bar{x} \\ \stackrel{(in\ elim_3)}{\longrightarrow} & [\mathbf{while}(b,p)]\ \mathbf{in}\ ([p_0]\ \mathbf{in}\ \bar{x}) \\ \stackrel{(in\ elim_5)}{\longrightarrow} & G(\bar{y})\ :\{E_0\&G(\bar{y}) \Leftarrow \mathbf{if}(b,G(t),t_0)\&E\} \end{array}$$

The induced functional transformation is derivable in UF by (*Laws*) and (*Defn. Intro.*). The induced functional transformation of the reverse of (*while logic*) is derivable by (*Laws*) and (*Defn. Elim.*). □

Lemma 4.6 *The induced functional transformation of (while unfold) is derivable via (Proc Unfold) in UF (and a property of the construct* **in***).*

Proof:

Consider how the left hand side of (*while unfold*) is translated, i.e

$$\begin{array}{ll} & [\mathbf{while}(b,p)]\ \mathbf{in}\ \bar{x} \\ \stackrel{(in\ elim_5)}{\longrightarrow} & G(\bar{y})\ :\{G(\bar{y}) \Leftarrow \mathbf{if}(b,G(t),\bar{x})\&E\} \end{array}$$

where we assume $[p]\ \mathbf{in}\ \bar{y} \stackrel{*}{\rightarrow} t\ :\{E\}$. Now consider the right hand side,

$$\begin{array}{ll} & [\mathbf{if}(b,p;\,\mathbf{while}(b,p))]\ \mathbf{in}\ \bar{x} \\ \stackrel{(in\ elim_4),(in\ elim_1)}{\longrightarrow} & \mathbf{if}(b,[p;\,\mathbf{while}(b,p)]\ \mathbf{in}\ \bar{x},\bar{x}) \\ \stackrel{(in\ elim_3)}{\longrightarrow} & \mathbf{if}(b,[p]\ \mathbf{in}\ ([\mathbf{while}(b,p)]\ \mathbf{in}\ \bar{x}),\bar{x}) \\ \stackrel{(in\ elim_5)}{\longrightarrow} & (\mathbf{if}(b,[p]\ \mathbf{in}\ G(\bar{y}),\bar{x})\ :\{G(\bar{y}) \Leftarrow \mathbf{if}(b,G(t),\bar{x})\&E\} \end{array}$$

So the induced functional transformation is derivable in UF via (*Proc. Unfold*) and the following property of the **in** construct:

$$[p] \textbf{ in } G(\bar{y}) \equiv G([p] \textbf{ in } \bar{y})$$

The reverse can be derived in UF by (*Proc. Fold*) and this property of **in**. When used in conjunction with monotonicity, (*whileunfold*) is derived in UF by (*Unfolding*). □

Before we consider the remaining while fold in context rules, we consider the simpler instance of (*while fold*).

Lemma 4.7 *The induced functional transformation of (while fold) is derivable in UF via (Defn. Intro.), (Proc.Fold) and (Folding).*

Proof:
From the premiss of the (*while fold*) rule we can assume that the following transformation follows via UF:

$$[p] \textbf{ in } \bar{x} \Rightarrow_{UF} [\textbf{if}(b, p_1; p)] \textbf{ in } \bar{x}$$

Let us assume that $[p]$ **in** $\bar{x} \xrightarrow{*} t :\{E\}$, then by (*in elim*$_4$)

$$t :\{E\} \Rightarrow_{UF} \textbf{if}(b, [p_1] \textbf{ in } t, \bar{x}) :\{E\}$$

Now by (*in elim*$_5$), [**while**(b, p_1)] **in** $\bar{x}$ becomes

$$G(\bar{y}) :\{G(\bar{y}) \Leftarrow \textbf{if}(b, G([p_1] \textbf{ in } \bar{y}), \bar{x})\}$$

where $Y_k \triangleq VAR(b) \cup VAR(p_1) \cup FV(t)$. Now let $Z_k \triangleq DV(p_1)$, then $[p_1]$ **in** $t \equiv t[[p_1] \textbf{ in } \bar{z}/\bar{z}]$ and since $Z_k \subseteq Y_k$ then

$$[p_1] \textbf{ in } t \equiv t[[p_1] \textbf{ in } \bar{y}/\bar{y}]$$

So assuming $[p_1]$ **in** $\bar{y} \xrightarrow{*} t_1 :\{E_1\}$ we have by assumption that the following transformation is derivable in UF:

$$t :\{E\} \Rightarrow_{UF} \textbf{if}(b, t[t_1/\bar{y}], \bar{x}) :\{E \& E_1\}$$

The induced functional transformation of the conclusion of (*while fold*) is by our assumptions:

$$t : \{E\} \Rightarrow_F G(\bar{y}) : \{E_1 \& G(\bar{y}) \Leftarrow \mathbf{if}(b, G(t_1), \bar{x})\}$$

We now show how this is derivable in UF:

$$\begin{array}{rl} & t : \{E\} \\ \overset{(Defn.\ Intro.)}{\Rightarrow_{UF}} & t : \{E \& G(\bar{y}) \Leftarrow t\} \\ \overset{(Proc.\ Fold)}{\Rightarrow_{UF}} & G(\bar{y}) : \{E \& G(\bar{y}) \Leftarrow t\} \\ \overset{(assumption)}{\Rightarrow_{UF}} & G(\bar{y}) : \{E \& E_1 \& G(\bar{y}) \Leftarrow \mathbf{if}(b, t[t_1/\bar{y}], \bar{x})\} \\ \overset{(Folding)}{\Rightarrow_{UF}} & G(\bar{y}) : \{E \& E_1 \& G(\bar{y}) \Leftarrow \mathbf{if}(b, G(t_1), \bar{x})\} \end{array}$$

Note that in this derivation the (*Defn. Intro.*) transformation was such that the new equation was well formed, since $FV(t) \subseteq Y_k$ □

To show how the translation of (*while fold in context*$_2$) can be derived in UF, we consider the following example which is the translation of Example 3.6:

Example 4.3

$$G_1(x, s+1) : \{G_1(x,s) \Leftarrow \mathbf{if}(x \geq 0, G_1(x-1, s+1), (x,s))\}$$

$\Rightarrow_F$

$$\mathbf{if}(x \geq 0, G_1(x-1, s+2), G_1(x, s+1)) : \{G_1(x,s) \Leftarrow \mathbf{if}(x \geq 0, G_1(x-1, s+1), (x,s))\}$$

$\Rightarrow_F$

$$G_2(x,s) : \left\{ \begin{array}{l} G_2(x,s) \Leftarrow \mathbf{if}(x \geq 0, G_2(x-1, s+1), G_1(x, s+1)) \\ G_1(x,s) \Leftarrow \mathbf{if}(x \geq 0, G_1(x-1, s+1), (x,s)) \end{array} \right\}$$

Proof:

We may assume that

$$G_1(x, s+1) : \{G_1(x,s) \Leftarrow \mathbf{if}(x \geq 0, G_1(x-1, s+1), (x,s))\}$$

$\Rightarrow_{UF}$

$$\mathbf{if}(x \geq 0, G_1(x-1, s+2), G_1(x, s+1)) : \{G_1(x,s) \Leftarrow \mathbf{if}(x \geq 0, G_1(x-1, s+1), (x,s))\}$$

since in Example 3.6 this follows by imperative transformations not including while fold in context rules. Consider then

$$\mathbf{if}(x \geq 0, G_1(x-1, s+2), G_1(x, s+1)) : \{G_1(x,s) \Leftarrow \mathbf{if}(x \geq 0, G_1(x-1, s+1), (x, s))\}$$

$\Rightarrow_F$

$$G_2(x,s) : \left\{ \begin{array}{l} G_2(x,s) \Leftarrow \mathbf{if}(x \geq 0, G_2(x-1, s+1), G_1(x, s+1)) \\ G_1(x,s) \Leftarrow \mathbf{if}(x \geq 0, G_1(x-1, s+1), (x, s)) \end{array} \right\}$$

We can derive this in UF using (*Defn. Intro*), the assumption and (*Folding*). □

We conjecture that the remaining while fold in context rules are not derivable in UF.

Conjecture 4.1 *The induced functional transformations of (while fold in context*$_1$*) and (while fold in context*$_3$*) are* <u>*not*</u> *derivable via UF.*

Scherlis (see **[Scherlis 80]**) adds an assertion propagation system, and uses induction, to take account of context. We conjecture that the induced functional transformations of (*while fold in context*$_1$) and (*while fold in context*$_3$) can be derived in UF with such an addition. As some justification we note that Example 3.9 uses (*while fold in context*$_1$), and that the functional equivalent of Example 3.9 is derived in **[Scherlis 80]** using EP (and consequently UF, since the EP system can be simulated by the UF system) with the addition of this assertion propagation system.

4.4 A Note on Translation

Since translation and transformation are related, transformation can be seen as translation within a language, not any old translation can be used in the proof of results concerning the relative power of transformation systems. In fact, we conjecture that by choosing the right translation we can prove anything about the comparison of two transformation systems.

At the very least we need to understand translations better before any results in this chapter can really mean anything. We shall attempt to describe and analyse the problems that arise due to translation in detail, using a more general setting. We shall work towards a definition of reasonableness by gradually, and hopefully intuitively, building up the conditions of our final definition. We start by giving a rather informal definition of translation.

Definition 4.1 *A* **translation** *from language L_1 to language L_2 is a semantic preserving function.*

Obviously there does not always exist such a translation, and it is not generally unique if there does exist one. Consider two languages L_1 and L_2, and two (transformation system) relations R_1 and R_2 over programs in these languages respectively (programs are given by context-free languages). Now suppose we wish to compare R_1 and R_2; we cannot compare the sets unless $L_1 = L_2$, and comparisons of $|R_1|$ and $|R_2|$ (cardinality) do not mean much. What we need to compare is $T(R_1)$ and R_2 where $T : L_1 \rightarrow L_2$ is a translation, or R_1 and $T'(R_2)$ where $T' : L_2 \rightarrow L_1$ is a translation. We use $T(R)$ to mean $\{(T(a_0), ..., T(a_n))|(a_0, ..., a_n) \in R\}$.

We have claimed that a comparison based on translations, as defined above, is not independent of the translation, and so we cannot prove the following proposition:

Proposition 4.1 *if* $\exists\, T.\, T(R_1) \subseteq R_2$ *then* $\forall\, T'.\, T'(R_1) \subseteq R_2$

Here T and T' denote translations, and $\subseteq$ is a concrete example of a comparison

(stating that transformation system R_2 is no less powerful than R_1). We are thus lead to considering the following:

Proposition 4.2 *if* $\exists\ T \subseteq REAS$ *(set of reasonable translations).* $T(R_1) \subseteq R_2$ *then* $\forall\ T' \subseteq REAS\ T'(R_1) \subseteq R_2$

This proposition states that if we can prove $T(R_1) \subseteq R_2$ for some reasonable translation T, then we can prove it for all reansonable translations. This would enable us to give results independent of the translation, at least reasonable ones. Our problem can thus be seen as trying to find a natural set, REAS, such that we can prove Proposition 4.2.

Ignoring termination, in semantic terms transformation does not change the denotational meaning of a program, but may change its operational meaning (that is, how the underlying abstract machine evaluates the program). Translation, on the other hand, should change neither the denotational nor operational meaning. To be more exact, a translation should provide a *simulation* between operational semantics.

Consider a machine M_1 with configurations C_1 and operational semantics given by the relation $\rightarrow_1$. Similarly, let M_2 be a machine with configurations C_2 and operational semantics given by the relation $\rightarrow_2$. We define a simulation as follows:

Definition 4.2 *T is a simulation between $\rightarrow_2$ and $\rightarrow_1$ (i.e machine M_2 simulates machine M_1, via T) provided T is an onto function between configurations C_2 and C_1 s.t.*

$$\overset{*}{\rightarrow}_1 oT \subseteq (To \overset{*}{\rightarrow}_2) \downarrow (dom\ T)$$

i.e $\forall b \in dom\ T$ *if* $T(b) \overset{*}{\rightarrow}_1 a$ *then* $\exists b'.\ b \overset{*}{\rightarrow}_2 b' \wedge T(b') = a$

Here *dom* T denotes the domain of T, o is function composition, and $\downarrow$ restriction.

We thus define the set of reasonable translations, REAS, as the set of those translations that are simulations.

We must make some assumptions about our transformation system (relations). We must assume that the transformation systems are transitive and are more than just simulations (within the same language). This latter condition can be expressed as follows:

$$\forall \text{ simulations } S \subseteq L \times L, S \subset R$$

where L is the language and R the transformation system relation. Clearly, if the transformation systems themselves have less than the power of simulations then a comparison via simulations would not be independent of the simulation (a form of our original problem).

In order to prove proposition 4.2 we need to show that the 'difference' between reasonable translations can be expressed as simulations, and is thus insignificant in comparison to the transformation systems.

The definitions of reasonable translation and simulation we have provided are general and hence probably too weak to enable us to prove proposition 4.2. It is a difficult task to add further conditions to our definitions, and still be practical. In order to do this we need to examine properties of 'translations'. For example, if the definition of a simulation, and thus reasonable translation, is changed so that the function must be 1-1 then the proof of proposition 4.2 is easy. However such a restriction does not seem practical. Alternatively we could define our set of reasonable translations as pairs of simulations, providing some sort of bisimulation. The practicality of this, weaker, restriction is also questionable.

THIS PAGE IS LEFT INTENTIONALLY BLANK.

Chapter 5

Implementation

In this chapter we shall discuss the implementation used to produce the examples in Chapter 3. We briefly motivate the need for some implemented system and discuss how we were able to use the synthesizer generator (see **[Reps/Teitelbaum 85]**). We give an example dialogue with the system and, in the final section of this chapter, reveal some of the implementation details.

5.1 Motivation

It is obvious that with such low level axioms and rules, transformation is a very laborious task; we need an electronic friend! The implementation we had in mind was not a fully automatic system, or even a semi-automatic system. Rather it was the least ambitious, an electronic piece of paper. This has many obvious advantages over ordinary paper:

- as long as our system is correct, we cannot make any erroneous steps;
- we do not have to keep writing out the slightly changed programs;
- we undoubtedly have a neater and better structured output;
- we create an environment within which we can build tricks to help us transform programs, i.e a semi-automatic system.

Our specific system also has the following advantages:

- structured editing;
- automatic selection of applicable transformations, via pattern matching.

5.2 Using the Synthesizer Generator

We were very fortunate to be able to base our system on the **synthesizer generator** (SGEN) (see **[Reps/Teitelbaum 85]**). SGEN is a tool for implementing language-based editing environments. The generator creates a full-screen editor for manipulating objects according to a specification, written in a language based on the concepts of term algebra and attribute grammars. An editor specification consists of a list of declarations, defining a language's abstract syntax, context-sensitive relationships, display format etc. The great advantage of using SGEN is that a kernel of features, such as basic editing commands and window commands, are provided, making SGEN ideal for prototype development.

Being able to create a structured editor for our **WHILE** language was very useful, but how did SGEN help us build a transformation system? The answer lies in the fact that SGEN contains the ability to specify editor commands for restructuring objects (*transformation declarations*), when the selected component matches the specified structural pattern. This enables us to create a transformation system on top of the editor specification. We must be careful to only invoke such transformations when the editing is actually finished; we can do this with SGEN by using two forms of syntax corresponding to an editable version of the language and an uneditable version.

However, producing a single **WHILE** language program and transforming it is not enough. We need some history of the transformations applied in order to use the (*while fold*) and while fold in context rules. We implemented such information by constructing an editor for a language for representing a *derivation list* of **WHILE** programs.

In SGEN each object to be edited/transformed is represented as a consistently attributed derivation tree. In response to each modification incremental analysis is performed, updating values throughout the tree. The collection of attributes can be seen as a set of derived facts that can be displayed, or used to control the editing process. We use attributes to display what transformations have been applied with respect to the previous element in the derivation list. This information is derived from invisible structures, that is structures that are not displayed, left by transformations.

5.3 An Example Dialogue

We shall now examine in detail the dialogue with the system, as we transform a small example:

Example 5.1 *(Extracting from a Loop)*

$$\textbf{while}(i \leq 9, i := i + 1; p := p + i); i := n$$

$$\Rightarrow_W \quad i := i + 1; \textbf{while}(i \leq 10, p := p + i; i := i + 1); i := n$$

Unless the user has loaded a previous example, the starting point of any transformation will be the following:

```
datarules

<datarule>



initial program

<program>
```

From this *template* an initial program is created using the structured editing facilities of SGEN. This involves making a *selection*, that is clicking the mouse on an

editable structure (visually the selected structure is highlighted). Each program template can then be "transformed" into any of the **WHILE** program constructs, by selecting the appropriate menu item with the mouse. Once this editing is deemed complete the entire program can be similarly transformed, by using a menu selection **start**, into an uneditable version. The display in the case of our small example would then be as follows:

```
datarules

<datarule>



WHILE lte(i,nine) DO [i:=plus(i,one)];[p:=plus(p,i)] OD;i:=n

==>

WHILE lte(i,nine) DO [i:=plus(i,one)];[p:=plus(p,i)] OD;i:=n
```

Note that we have `lte(i,nine)` rather than $i \leq 9$. In general the display we use for examples, as in Chapter 3, is a postprocessed LaTeX form of the SGEN display.

Transformation of the lower program can be made in a similar fashion to the previous editing stage. Each time a selection is made within a program of the derivation list, a list of applicable transformations is displayed. In addition to the transformations of the W system, the implementation also contains **step** and **delete** transformations. The **step** transformation allows the user to expand the number of intermediate steps in the derivation list, by copying the last program in the current list. The user then transforms this program. The **delete** transformation allows the user to *backup*, by deleting a tail section of the derivation list.

We give below the entire derivation list, interspersed with comments explaining how certain things came about.

The datarules can be added at any time, in the same way the original program was constructed, i.e via structured editing, and we thus assume the data rules we require for this example have already been input.

DATA RULES

$(r1)\ (x \leq 9) \Rightarrow ((x + 1) \leq 10)$

while $(i \leq 9)$ **do** $[\, i := (i + 1) \,]; [\, p := (p + i) \,]$ **od** ; $[\, i := n \,]$

$\Rightarrow$ (*while unfold*)

if $(i \leq 9)$
  then $[\, i := (i + 1) \,]; [\, p := (p + i) \,];$
    while $(i \leq 9)$ **do** $[\, i := (i + 1) \,]; [\, p := (p + i) \,]$ **od**
  else skip
fi ; $[\, i := n \,]$

A record of the transformations applied is kept in between the appropriate programs. So for example, the difference between the two programs above is the application of the (*while unfold*) transformation, while the difference between the lower of the two programs above and the program directly below is the application of the transformations $(;\ elim_3)$, $(;\ intro_1)$ and $(;\ assoc)$. The latter difference also shows that more than one transformation can be applied before using **step**. For technical reasons only a limited amount of transformation is possible before a **step** transformation must be invoked.

$\Rightarrow (;\ elim_3), (;\ intro_1), (;\ assoc)$

if $(i \leq 9)$
  then $[\, i := (i + 1) \,]; [\, p := (p + i) \,];$
    while $(i \leq 9)$ **do** $[\, i := (i + 1) \,]; [\, p := (p + i) \,]$ **od** ;
   $[\, i := n \,]$
  else skip; $[\, i := (i + 1) \,]; [\, i := n \,]$
fi

The application of the transformation (; $intro_1$) requires the user to postulate a sequence of *ma*s considered equivalent to the existing *ma*. A **replace** transformation then makes the replacement, if valid.

$\Rightarrow$ (; $intro_3$), (*skip* $intro_3$), (*skip* $elim_2$), (; *assoc*)

```
if (i ≤ 9)
   then [ i := (i + 1) ];
      [ p := (p + i) ];
       while (i ≤ 9) do [ i := (i + 1) ]; [ p := (p + i) ] od
   else [ i := (i + 1) ]; skip
fi ; [ i := n ]
```

$\Rightarrow$ ($r1$)

```
if ((i + 1) ≤ 10)
   then [ i := (i + 1) ];
      [ p := (p + i) ];
       while (i ≤ 9) do [ i := (i + 1) ]; [ p := (p + i) ] od
   else [ i := (i + 1) ]; skip
fi ; [ i := n ]
```

A **datarule** transformation allows the user to specify the name of a datarule to be used at the specific point selected; in the above case ($r1$). A **use datarule** transformation then makes the replacement, if valid.

$\Rightarrow$ (; $intro_2$), (; *assoc*)

```
[ i := (i + 1) ];
<<  if (i ≤ 10)
   then [ p := (p + i) ];
       while (i ≤ 9) do [ i := (i + 1) ]; [ p := (p + i) ] od
   else skip
fi ; [ i := n ] >>
```

```
<< * >>  --->
    if (i ≤ 10)
      then [ p := (p + i) ];
          while (i ≤ 9)
              do
                [ i := (i + 1) ]; [ p := (p + i) ]
              od
      else skip
    fi ; [ i := n ]
```

Similarly to the (; $intro_1$) transformation, (; $intro_2$) requires the user to postulate a sequence, consisting of a *ma* and a conditional, equivalent to the existing conditional. A **replace** transformation then makes the replacement, if valid.

Modules, indicated by << >>, are used for folding in contexts not used, and for separately transforming small parts of large programs. A **start module** transformation makes the current selection into a module, copying the selection into a slightly indented derivation list. This module can then be transformed in the same way as a *top-level* program.

$\Rightarrow$ (; $elim_3$), (*while unfold*)

```
if (i ≤ 10)
  then [ p := (p + i) ];
      if (i ≤ 9)
        then [ i := (i + 1) ]; [ p := (p + i) ];
            while (i ≤ 9)
                do
                  [ i := (i + 1) ]; [ p := (p + i) ]
                od
        else skip
      fi ; [ i := n ]
  else skip; [ i := n ]
fi
```

$\Rightarrow$ (; *assoc*), (; $elim_3$), (*skip* $elim_2$), (; $intro_1$)

```
if (i ≤ 10)
  then [ p := (p + i) ];
     if (i ≤ 9)
       then [ i := (i + 1) ]; [ p := (p + i) ];
          while (i ≤ 9)
             do
               [ i := (i + 1) ]; [ p := (p + i) ]
             od ; [ i := n ]
       else [ i := (i + 1) ]; [ i := n ]
     fi
  else skip; [ i := n ]
fi
```

$\Rightarrow$ (; $intro_3$), (; *assoc*)

```
if (i ≤ 10)
  then [ p := (p + i) ];
     if (i ≤ 9)
       then [ i := (i + 1) ]; [ p := (p + i) ];
          while (i ≤ 9)
             do
               [ i := (i + 1) ]; [ p := (p + i) ]
             od
       else [ i := (i + 1) ]
     fi ; [ i := n ]
  else skip; [ i := n ]
fi
```

$\Rightarrow$ (; $intro_3$), ($r1$)

```
if (i ≤ 10)
  then [ p := (p + i) ];
     if ((i + 1) ≤ 10)
       then [ i := (i + 1) ]; [ p := (p + i) ];
          while (i ≤ 9)
```

```
                do
                  [ i := (i + 1) ]; [ p := (p + i) ]
                od
          else [ i := (i + 1) ]
        fi
      else skip
    fi ; [ i := n ]
```

⇒ (*skip intro*$_3$), (; *assoc*), (; *intro*$_2$), (; *assoc*)

```
    if (i ≤ 10)
      then [ p := (p + i) ]; [ i := (i + 1) ];
        if (i ≤ 10)
          then [ p := (p + i) ];
            while (i ≤ 9)
                do
                  [ i := (i + 1) ]; [ p := (p + i) ]
                od
          else skip
        fi
      else skip
    fi ; [ i := n ]
```

⇒ (*while fold in context*$_2$)

```
    while (i ≤ 10) do [ p := (p + i) ]; [ i := (i + 1) ] od ;
    [ i := n ]
```

< - - -   << * >>

⇒

```
[ i := (i + 1) ];
while (i ≤ 10) do [ p := (p + i) ]; [ i := (i + 1) ] od ; [ i := n ]
```

Similarly to (*while fold*), (*while fold in context*$_2$) only requires the user to select a tail section of the derivation list (within the same module, if any). The system then attempts to fold the current program with the head of this tail section. The

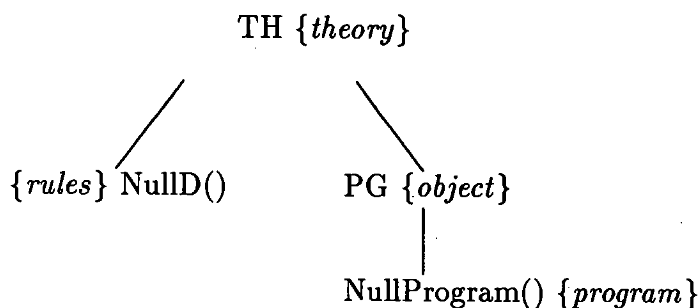


Figure 5–1:

transformations (*while fold in* $context_1$) and (*while fold in* $context_3$) also require the user to do a transformational proof, corresponding to the second premiss in these transformations. The proposition to be proven is automatically generated. The user terminates the proof with a **finish proof** transformation, and the system checks that the conclusion of the proof is the required one. Examples 3.7 and 3.9 contain proof structures.

A **finish module** transformation puts the transformed version of the original module back into the context of the containing program.

5.4 Implementation Details

In this section we detail the abstract syntax of the language defined in the implementation. This language is used to represent a derivation list of **WHILE** programs, rather than just single **WHILE** programs. We give the abstract syntax trees at various stages in the transformational development of a simple program, and explain how these trees evolve with respect to user interaction. We annotate the trees with types enclosed in curly brackets, and abbreviate repeated parts from previous trees with vertical ellipses.

Figure 5–1 is the abstract syntax tree of the starting point (as given on page 108). NullProgram() is the *completing term* for the type *program* and has a display declaration of " < *program* >". Similarly NullD() is the completing term

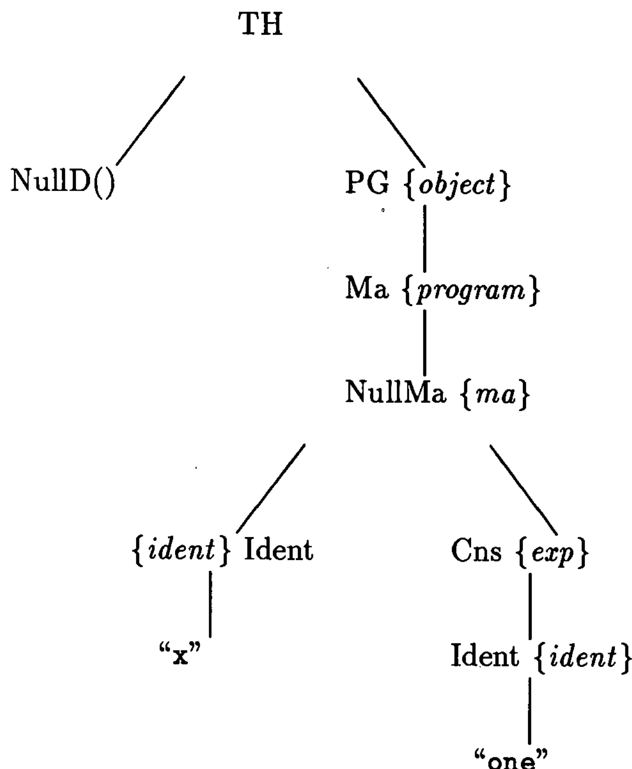


Figure 5–2:

for type *rules*. These completing terms act as templates for inserting syntactic constructs of the corresponding type.

In order to focus on the abstract syntax we choose a trivial program: $x := 1$ (or as it would appear in SGEN, `x:=one`). Figure 5–2 shows the abstract syntax tree at the point in development at which this program has been typed in. The **start** transformation performs the following tree transformation on Figure 5–2: PG(p)⇒DV(*MakePair*(*Mapp*(p))). The resulting tree is given in Figure 5–3. The function *Mapp* translates the constructs of the language into uneditable versions; for example Ma (the multiple assignment constructor) becomes MA (of type *prg*, as oppose to *program*). Both the editable and uneditable versions of the core language, **WHILE**, have the same display declarations. The *Makepair* function converts an object of type *prg* (an uneditable program) into a PAIR of programs (a derivation list).

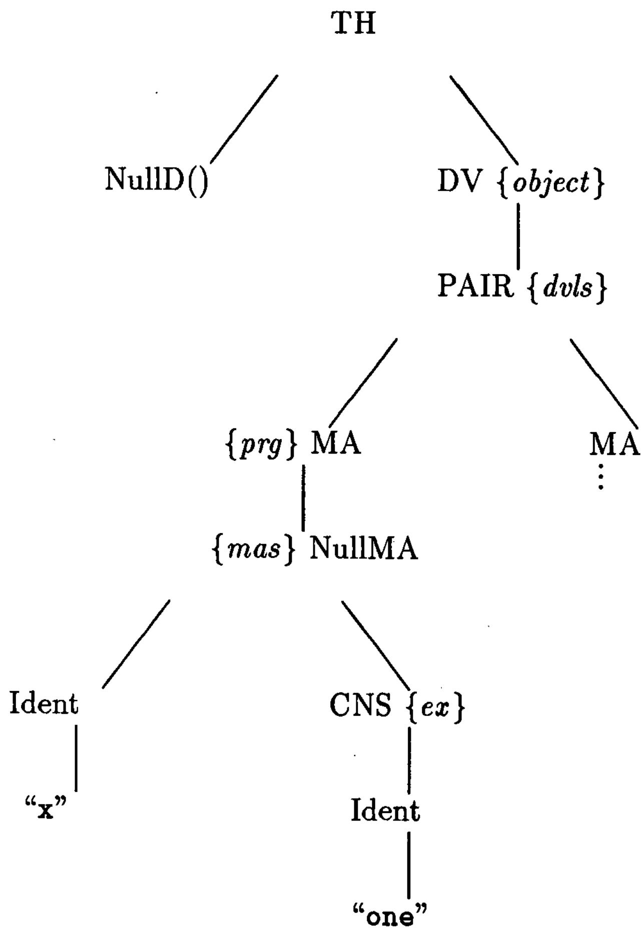


Figure 5–3:

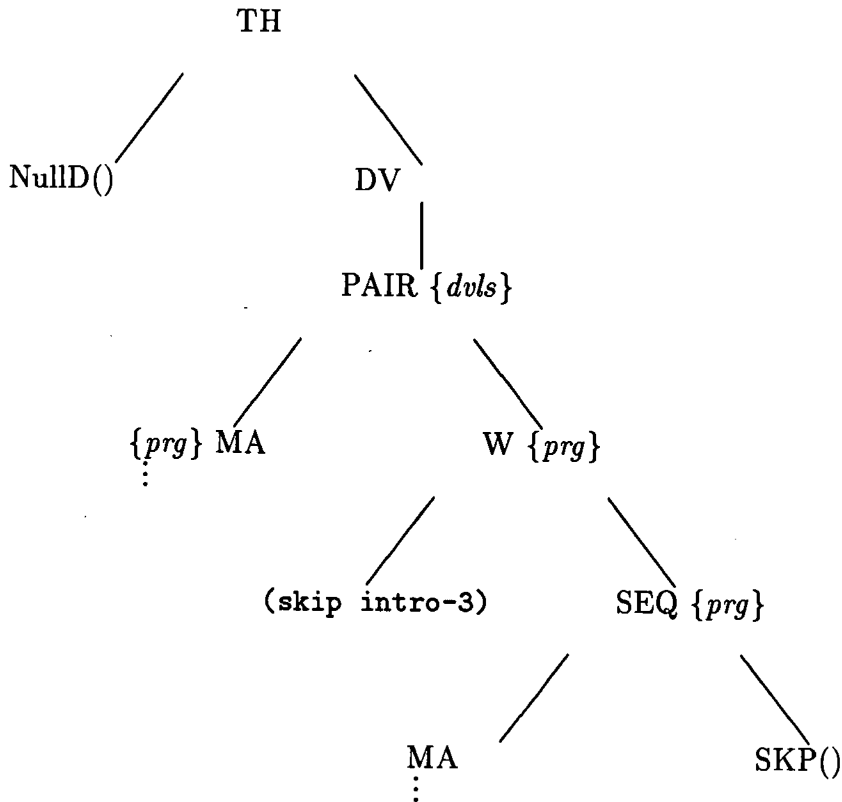


Figure 5–4:

Now suppose we transform **x:=one** into **x:=one;skip**, using (*skip intro*$_3$). The tree then becomes as in Figure 5–4. The (invisible) structure W is used to hold information about which transformation has been applied and where the transformation has been applied. Because the transformation declarations corresponding to our transformation system are invoked through pattern matching objects of type *prg*, the W structure disables transformations which should be applicable. For this reason only a limited amount of transformation can be done at each step, and a **step** transformation must be invoked to clear away the W structures.

Using the **step** transformation on an object of type *dvls* performs the following tree transformation: PAIR(p,p1)⇒STEP(p,PAIR(p1,*strip*(p1))). The function

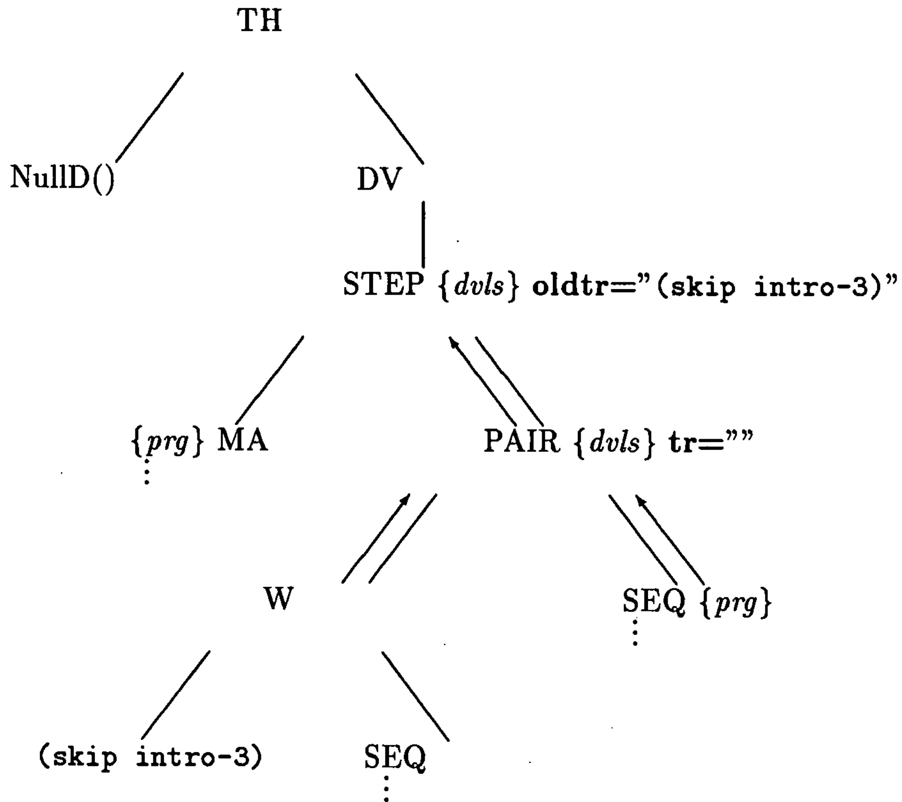


Figure 5–5:

strip clears away the W structures. Using **step** on the tree in Figure 5–4 we get the tree in Figure 5–5. We also include on this tree arrows which indicate how the attributes move up the tree to be displayed at the points indicated. The attribute **oldtr** displays at each STEP the transformations that have been applied at an element in the derivation list to get the next element (except when the next element is the last element, or current program). The attribute **tr** displays the transformations that have been applied to the current program.

Chapter 6

Summary and Further Work

6.1 Summary

In this thesis we have developed a transformation system for an imperative **WHILE** language. This system uses transformation rules in addition to axioms, extending the usual notion of transformation. That is, if A, B, C, D are programs, in addition to the axiom form of transformation, "A transforms to B", we have also the rule form of transformation, "if A transforms to B then C transforms to D". We have shown that the rule form of transformation needs to be extended to take account of context, and we have provided three such rules.

We have given many examples. In particular we have given solutions to examples taken from [**de Bakker 80**], [**Scherlis 80**] and [**van Diepen/de Roever 86**]. The examples are very long-winded and our prototype implementation has only begun to make such derivations slightly less tedious.

We have shown that by stating the partial correctness of a program as a semantic equivalence, we can prove programs correct by transformation, and further such a method using our transformation system is as powerful as Hoare's logic. As a consequence of this result, we have a *pure* transformation system, although we may use assertions as a notational convenience.

As another *relative completeness* result, we have proven that a subset of our system, W, can be derived by the unfold/fold system for functional languages (see [**Burstall/Darlington 77**]). However, we argue that to be a fair comparison more work must be done on the proof. This further work amounts to proving that

the translations, used to pass between the imperative and functional languages, are no more powerful than the transformation systems that are under comparison, i.e the translations are *relatively reasonable.*

6.2 Further Work

6.2.1 Extending the Theory

There are several places in this thesis where we have raised questions and not fully answered them. We reiterate these questions below as areas of possible further research.

- The while fold in context rules are not known to be complete in any sense. Therefore how do we know that we do not need any more such rules? Are there any more (non-derivable) while fold in context rules? How would we prove that we have all the generalisations of (*while fold*)?

- It is a difficult problem to find a natural example of an equivalence that our transformation system does not capture. The fact that we can derive Hoare's logic, itself complete in a restricted sense, means we could consider some restricted forms of completeness. As a start in this direction, the proof that every **WHILE** program has a semantically equivalent counterpart which only has single nested **while**s is by transformation (see **[Harel 80]** for discussion), and so it may be possible to derive a version of this proof using our system.

- Although we do not gain any *power* by using assertions in our system, they maybe useful as syntactic sugaring since they have some useful properties. "Extra" transformation steps are needed to eliminate the invariant context created in order to use (*while fold in* $context_1$), whereas with "ordinary" assertions we can simply drop them since they are not part of the program.

However we have done the transformations to allow us to drop the *context* that we have created, via transformations. e.g

$$\begin{aligned} p_0; p_1; p_2 &\Leftrightarrow_W p_0; \mathbf{if}(b, p_1, \mathbf{skip}); p_2 \\ &\Leftrightarrow_W p_0; \mathbf{if}(b, p_1', \mathbf{skip}); p_2 \\ &\Leftrightarrow_W p_0; p_1'; p_2 \end{aligned}$$

The last line is a single step since it follows that b is true after p_0 from the first line.

- We could consider the converse of Theorem 4.1, i.e does there exist p, q and S for which correctness can be proven by transformation but for which correctness does not follow by Hoare Logic. In fact a mismatch occurs since the Hoare derivations only use the *(while fold in context$_1$)* rule. So do the other rules give extra power to Hoare logic type problems or are they simply just not required for the special form of transformation?

- Using Lemma 4.1 we can *translate* Hoare Logic results into a transformational setting, e.g results such as the characterization problem (there exist constructs for which it is impossible to give a complete Hoare system (see **[Clarke 84]**)), total correctness and Cook completeness.

- The relative completeness work and the notion of *relatively reasonable* translation need to be investigated further. How do we prove a translation to be relatively reasonable? Are the translations we have used in Chapter 4 relatively reasonable, and hence the results fair?

- We have conjectured that folding in context has no equivalent in UF, and so by adding such generalisations of **Folding** to UF we could obtain a more powerful functional system than UF. It would be interesting to prove our conjecture and find the generalisations of **Folding**. It would also be interesting to see how the mimicking of Hoare's logic result for our imperative transformation system translates into UF. How do these relate to Scherlis' assertion system (see **[Scherlis 80]**, **[Scherlis 81]**)?

6.2.2 Extending the Language

As the **WHILE** language we use is adequate to compute each partial recursive function, any new control construct can be translated into our **WHILE** language. Consider the problem of introducing a new construct into our language, and giving transformations for it. When dealing with programs containing higher level constructs, we could translate (compile) them into the **WHILE** language, develop some transformation strategy and translate the resulting program back into a form using the higher level constructs. This creates a higher-level transformation in the high level language that is correct by construction. If the resulting program cannot be translated back into a form using the high-level constructs, then the transformation simply cannot be done at this higher level.

We need to add arrays to our **WHILE** language in order for it to be adequate to compute each partial recursive function over an abstract structure. We can give transformations for arrays.

6.2.3 Extending the Implementation

We conclude this chapter with a few ideas on how to begin to make our implemented system more useful.

We have stated previously that the use of higher level transformations, enabling users to build up a library or theory of transformations, is needed in order for a generative system to be useful. A major drawback in the implementation at present, is that of not being able to use higher level transformations (or *tactics*). We have built several such transformations into the system (see Appendix C) mainly to cut down on the length of presentation of examples in this thesis, but it is impossible in SGEN to enable the user to define tactics (unless one goes into a completely different representation which makes no use of the SGEN transformation facilities). One solution to this problem of defining tactics is to use a goal-directed strategy in transforming. The user develops the transformation leaving certain steps (lemmas) as *obvious*. The system can then (concurrently in background) try to match these lemma against a catalogue of previously "proven"

transformations. Any outstanding ones are presented to the user at the end of the derivation, to be proved as normal. This solution calls for an efficient structuring of the catalogue of higher level transformations. A possible approach here is to use abstraction to structure the transformations. Consider a tree whose nodes are schematic programs. Two nodes have a common parent if both nodes are instances of that parent. Transformations are associated with nodes matching the left-hand sides of the transformations. To search for a transformation to apply to a lemma, the system starts at the node matching the lemma and moves upwards through it's parents, if necessary.

Appendix A

Semantics

The semantics of the first-order language L is given by a L-**structure** Φ, consisting of the following:

- a non-empty set $|\Phi| = \mathcal{A}$
- for each symbol $R_j^n \in L$ of arity n, a relation $\mathcal{R}_j^n$ in $\mathcal{A}^n$
- for each symbol $f_j^n \in L$ of arity n, a function $\mathcal{F}_j^n$ in $\mathcal{A}^n \rightarrow \mathcal{A}$

For a given L-structure, let $STORE_\Phi = \{s | s : X \rightarrow \mathcal{A}\}$. To give the semantics of **WHILE** programs we use a very simple abstract machine (as in **[Plotkin 81]**). We use E_L and B_L[1] to denote the sets of expressions and boolean expressions built on top of L, respectively. We introduce configurations $\Gamma \triangleq \{\langle e, \sigma \rangle | e \in E, \sigma \in STORE_\Phi\}$ and a relation $\gg$; $\langle e, \sigma \rangle \gg \langle e', \sigma' \rangle$ *meaning* the evaluation of e (with store σ) results in the expression e' (with store σ'). Terminal configurations for E are defined as $\{\langle m, \sigma \rangle | m \in \mathcal{A}, \sigma \in STORE_\Phi\} \triangleq T_E$. The transition rules are as follows:

(e_1) $\qquad \langle x, \sigma \rangle \gg \langle \sigma(x), \sigma \rangle$

[1]We shall drop the L subscript, as it is constant throughout.

$$(e_2) \qquad \frac{\langle e_i, \sigma\rangle \gg \langle m_i, \sigma\rangle \; m_i \in \mathcal{A} \; i = 1, \ldots, n}{\langle f_j^n(e_1, \ldots, e_n), \sigma\rangle \gg \langle m, \sigma\rangle} \; (m \triangleq \mathcal{F}_j^n(m_1, \ldots, m_n) \in \mathcal{A})$$

Equivalence for $e, e' \in E$ is then defined as follows:

$$e \equiv e' \; \mathit{iff} \; \forall\Phi, \forall\sigma \in STORE_\Phi \; eval(e, \sigma) = eval(e', \sigma)$$

where $eval(e, \sigma) = m$ *iff* $\langle e, \sigma\rangle \gg \langle m, \sigma\rangle \in T_E$.

We now take $\Gamma \triangleq \{\langle b, \sigma\rangle | b \in B, \sigma \in STORE_\Phi\}$, and terminal configurations for B as $\{\langle true, \sigma\rangle | \sigma \in STORE_\Phi\} \cup \{\langle false, \sigma\rangle | \sigma \in STORE_\Phi\} \triangleq T_B$. The rules are as follows[2]:

$$(b_1) \qquad \frac{\langle e_i, \sigma\rangle \gg \langle m_i, \sigma\rangle \; m_i \in \mathcal{A} \; i = 1, \ldots, n}{\langle R_j^n(e_1, \ldots, e_n), \sigma\rangle \gg \langle true, \sigma\rangle} \; \text{if } (m_1, \ldots, m_n) \in \mathcal{R}_j^n$$

$$(b_1) \qquad \frac{\langle e_i, \sigma\rangle \gg \langle m_i, \sigma\rangle \; m_i \in \mathcal{A} \; i = 1, \ldots, n}{\langle R_j^n(e_1, \ldots, e_n), \sigma\rangle \gg \langle false, \sigma\rangle} \; \text{if } (m_1, \ldots, m_n) \notin \mathcal{R}_j^n$$

$$(b_2) \qquad \frac{\langle b, \sigma\rangle \gg \langle true, \sigma\rangle}{\langle \neg b, \sigma\rangle \gg \langle false, \sigma\rangle}$$

$$(b_3) \qquad \frac{\langle b, \sigma\rangle \gg \langle false, \sigma\rangle}{\langle \neg b, \sigma\rangle \gg \langle true, \sigma\rangle}$$

$$(b_4) \qquad \frac{\langle b, \sigma\rangle \gg \langle t, \sigma\rangle \; \langle b', \sigma\rangle \gg \langle t', \sigma\rangle \; t, t' \in \{true, false\}}{\langle b \vee b', \sigma\rangle \gg \langle t'', \sigma\rangle} \; (t'' \triangleq b \; or \; b')$$

$$(b_5) \qquad \frac{\langle b, \sigma\rangle \gg \langle t, \sigma\rangle \; \langle b', \sigma\rangle \gg \langle t', \sigma\rangle \; t, t' \in \{true, false\}}{\langle b \wedge b', \sigma\rangle \gg \langle t'', \sigma\rangle} \; (t'' \triangleq b \; and \; b')$$

[2]We use the same symbol, $\gg$, as a relation between E and B.

Equivalence for $b, b' \in B$ is defined as follows:

$$b \equiv b' \textit{ iff } \forall \Phi,\ \forall \sigma \in STORE_{\Phi}\ eval(b,\sigma) = eval(b',\sigma)$$

where $eval(b,\sigma) = t$ *iff* $\langle b,\sigma\rangle \gg \langle t,\sigma\rangle \in T_B$

We now take $\Gamma \triangleq \{\langle w,\sigma\rangle | w \in \textbf{WHILE},\ \sigma \in STORE_{\Phi}\} \cup STORE_{\Phi}$ and terminal configurations for **WHILE** as $T_{\textbf{WHILE}} \triangleq STORE_{\Phi}$. The rules are as follows:

(w_0) $$\langle \textbf{skip}, \sigma\rangle \rightarrow \sigma$$

(w_1) $$\frac{\langle c_i,\sigma\rangle \gg \langle m_i,\sigma\rangle\ m_i \in \mathcal{A}\ i = 1,\ldots,n}{\langle (x_1 := c_1 \ \&\ldots\&\ x_n := c_n),\sigma\rangle \rightarrow \sigma[x_1/m_1,\ldots,x_n/m_n]}$$

where $\sigma[-]$ (function extension)[3] is defined as:

$$\sigma[x_1/m_1,\ldots,x_n/m_n] = \begin{cases} \sigma(y) & y \notin \{x_1,\ldots,x_n\} \\ m_i & y = x_i\ (i = 1,\ldots,n) \end{cases}$$

(w_2) $$\frac{\langle w,\sigma\rangle \rightarrow \sigma'}{\langle w;\ w',\sigma\rangle \rightarrow \langle w',\sigma'\rangle}$$

(w_2) $$\frac{\langle w,\sigma\rangle \rightarrow \langle w_0,\sigma'\rangle}{\langle w;\ w',\sigma\rangle \rightarrow \langle w_0;\ w',\sigma'\rangle}$$

(w_3) $$\frac{\langle b,\sigma\rangle \gg \langle true,\sigma\rangle}{\langle \textbf{if}(b,w,w'),\sigma\rangle \rightarrow \langle w,\sigma\rangle}$$

---

[3]We use [-] to mean both *substitution* and *functional extension*, but the usage should be clear from context.

$$(w_4) \qquad \frac{\langle b, \sigma\rangle \gg \langle \mathit{false}, \sigma\rangle}{\langle \mathbf{if}(b, w, w'), \sigma\rangle \rightarrow \langle w', \sigma\rangle}$$

$$(w_5) \qquad \frac{\langle b, \sigma\rangle \gg \langle \mathit{true}, \sigma\rangle}{\langle \mathbf{while}(b, w), \sigma\rangle \rightarrow \langle w; \mathbf{while}(b, w), \sigma\rangle}$$

$$(w_6) \qquad \frac{\langle b, \sigma\rangle \gg \langle \mathit{false}, \sigma\rangle}{\langle \mathbf{while}(b, w), \sigma\rangle \rightarrow \sigma}$$

Equivalence of **WHILE** programs is defined as follows:

$$w \equiv w' \textit{ iff } \forall \Phi,\ \forall \sigma \in STORE_\Phi \ exec(w, \sigma) \approx exec(w', \sigma)$$

where

$$exec(w, \sigma) = \begin{cases} \sigma' & \text{if } \langle w, \sigma\rangle \xrightarrow{*} \sigma' \in T_{\mathbf{WHILE}} \\ \bot(\text{undefined}) & \text{otherwise} \end{cases}$$

The function *exec* is a partial function and so $\approx$, and consequently $\equiv$, denote **strong equivalence**, i.e both are undefined or both are defined and equal. We shall also use **weak equivalence**, defined as follows:

$$w \sqsubseteq w' \textit{ iff } w \equiv w' \ \underline{or} \ exec(w', \sigma) \textit{ is undefined}$$

(written as $exec(w, \sigma) \simeq exec(w', \sigma)$).

We shall require the following lemmas in the soundness proofs. These lemmas state simple properties of the transition relation. We denote the transitive closure of $\rightarrow$ by $\xrightarrow{*}$.

Lemma A.1 *if* $\langle i_0, \sigma\rangle \xrightarrow{*} \sigma'$ *then* $\langle i_0; i_1, \sigma\rangle \xrightarrow{*} \langle i_1, \sigma'\rangle$

Proof:

Proof follows by induction on the length of the transition sequence $\langle i_0, \sigma\rangle \xrightarrow{*} \sigma'$.
□

Lemma A.2 $\forall\Phi\ \forall\sigma \in STORE_\Phi$
$(\forall e \in E\ \exists\sigma' \in T_E.\ \langle e,\sigma\rangle \gg \sigma')$ *and*
$(\forall b \in B\ \exists\sigma'' \in T_B.\ \langle b,\sigma\rangle \gg \sigma'')$

Proof:

Proof follows by structural induction □

Now for all **IF** programs i_0, exec(i_0,σ) is defined, i.e reaches a terminal configuration.

Lemma A.3 $\forall\Phi\ \forall\sigma \in STORE_\Phi$
$\forall i_0 \in \mathbf{IF}\ \ \exists\sigma' \in T_{\mathbf{IF}}\ .\ \ \langle i_0,\sigma\rangle \stackrel{*}{\rightarrow} \sigma'$

Proof:

Proof follows by structural induction □

The next two lemmas state that each of the transition relations, corresponding to the syntactic categories, is deterministic. (see also [**Plotkin 81**])

Lemma A.4 $\forall\Phi\ \forall\sigma,\sigma',\sigma'' \in STORE_\Phi$
if $(\forall e \in E\ \langle e,\sigma\rangle \gg \sigma'\ \wedge\ \langle e,\sigma\rangle \gg \sigma''$ *where* $\sigma',\sigma'' \in T_E)$ *then* $(\sigma' = \sigma'')$
and
if $(\forall b \in B\ \langle b,\sigma\rangle \stackrel{*}{\rightarrow} \sigma'\ \wedge\ \langle b,\sigma\rangle \stackrel{*}{\rightarrow} \sigma''$ *where* $\sigma',\sigma'' \in T_B)$ *then* $(\sigma' = \sigma'')$

Proof:

Proof follows by structural induction □

Lemma A.5 $\forall\Phi\ \forall\sigma,\sigma',\sigma'' \in STORE_\Phi$
if $(\forall i_0 \in \mathbf{IF}\ \ \langle i_0,\sigma\rangle \stackrel{*}{\rightarrow} \sigma'\ \wedge\ \langle i_0,\sigma\rangle \stackrel{*}{\rightarrow} \sigma''$ *where* $\sigma',\sigma'' \in T_{\mathbf{IF}}\ )$ *then* $(\sigma' = \sigma'')$

Proof:

Proof follows by structural induction □

Lemma A.6 $\forall i_0, i_1 \in \mathbf{IF}$.
if $\langle i_0;\ i_1, \sigma\rangle \xrightarrow{*} \sigma'$, *then* $\exists \sigma''$. $\langle i_0, \sigma\rangle \xrightarrow{*} \sigma''$ *and* $\langle i_1, \sigma''\rangle \xrightarrow{*} \sigma'$

Proof:

Assume $\langle i_0;\ i_1, \sigma\rangle \xrightarrow{*} \sigma'$, $i_0, i_1 \in \mathbf{IF}$

Consider $\langle i_0, \sigma\rangle$,

since $i_0 \in \mathbf{IF}$, then by Lemma A.3 $\exists \sigma''.\langle i_0, \sigma\rangle \xrightarrow{*} \sigma''$

Hence using Lemma A.1, $\langle i_0;\ i_1, \sigma\rangle \xrightarrow{*} \langle i_1, \sigma''\rangle$

Now since $i_1 \in \mathbf{IF}$, then by Lemma A.3 $\exists \sigma'''.\langle i_1, \sigma''\rangle \xrightarrow{*} \sigma'''$

Hence $\langle i_0;\ i_1, \sigma\rangle \xrightarrow{*} \sigma'''$

Now by Lemma A.5 $\sigma''' = \sigma'$,

Hence $\langle i_0, \sigma\rangle \xrightarrow{*} \sigma''$, $\langle i_1, \sigma''\rangle \xrightarrow{*} \sigma'$ □

Lemma A.7

if $\langle x_1 := c_1\ \&\ldots\&\ x_n := c_n, \sigma\rangle \xrightarrow{*} \sigma''$
then $(i{=}1,\ldots,n)$ $\exists m_i \in \mathcal{A}$. $\langle c_i, \sigma\rangle \xrightarrow{*} \langle m_i, \sigma\rangle$ *and* $\sigma'' = \sigma[m_1/x_1, \ldots, m_n/x_n]$

Proof:

Proof follows by Lemma A.2, (w_1) and Lemma (A.4) □

Lemma A.8

if $\langle f_j^n(c_1, \ldots, c_n), \sigma\rangle \gg \langle m, \sigma\rangle$
then $(i{=}1,\ldots,n)$ $\exists m_i \in \mathcal{A}$. $\langle c_i, \sigma\rangle \gg \langle m_i, \sigma\rangle$ *and* $m = \mathcal{F}_j^n(m_1, \ldots, m_n)$

Proof:

Proof follows by Lemma A.2 and (e_2) □

Lemma A.9

if $\langle c_i, \sigma\rangle \gg \langle m_i, \sigma\rangle$ $(i{=}1,\ldots,n)$
then $eval(e[x_1/c_1 \ldots x_n/c_n], \sigma) = eval(e, \sigma[x_1/m_1 \ldots x_n/m_n])$
and $eval(b[x_1/c_1 \ldots x_n/c_n], \sigma) = eval(b, \sigma[x_1/m_1 \ldots x_n/m_n])$

Proof:

Proof follows by simultaneous structural induction, and uses Lemma A.2 and Lemma A.8 □

Lemma A.10

if $\langle c_i, \sigma \rangle \gg \langle m_i, \sigma \rangle$ *(i=1,...,n)*

then $exec(p,\sigma[x_1/m_1, \ldots, x_n/m_n]) = exec((x_1 := c_1 \; \& \ldots \& \; x_n := c_n);p,\sigma)$

Proof:

Proof follows using Lemma A.6, Lemma A.5 and (w_1) □

We shall also require the following, some of which are extensions of the previous Lemmas from **IF** to **WHILE**.

Lemma A.11 $\forall \Phi \; \forall \sigma, \sigma', \sigma'' \in STORE_\Phi$

$\forall w_0 \in$ **WHILE** *if* $(\langle w_0, \sigma \rangle \xrightarrow{*} \sigma' \; \wedge \; \langle w_0, \sigma \rangle \xrightarrow{*} \sigma''$ *where* $\sigma', \sigma'' \in T_{\mathbf{WHILE}})$

then $(\sigma' = \sigma'')$

Proof:

Proof follows by structural induction □

Lemma A.12 $\forall p, p' \in$ **WHILE**

if $\langle p, \sigma \rangle \xrightarrow{*} \sigma''$ *and* $\langle p, \sigma \rangle \xrightarrow{*} \langle p', \sigma' \rangle$ *then* $\langle p', \sigma' \rangle \xrightarrow{*} \sigma''$

Proof:

Assume $\langle p, \sigma \rangle \xrightarrow{*} \sigma''$ and $\langle p, \sigma \rangle \xrightarrow{*} \langle p', \sigma' \rangle$

Now assume $\neg \exists \sigma'''. \langle p', \sigma' \rangle \xrightarrow{*} \sigma'''$ then

$\neg \exists \sigma'''. \langle p, \sigma \rangle \xrightarrow{*} \sigma'''$ (contradiction)

So $\exists \sigma'''. \langle p', \sigma' \rangle \xrightarrow{*} \sigma'''$ and hence

$\langle p, \sigma \rangle \xrightarrow{*} \sigma'''$ and $\sigma''' = \sigma''$ by Lemma A.11

Thus $\langle p', \sigma' \rangle \xrightarrow{*} \sigma''$ □

Lemma A.13 $\forall p_0, p_1 \in$ **WHILE**
if $\langle p_0;\ p_1, \sigma\rangle \xrightarrow{*} \sigma'$, *then* $\exists\sigma''.\ \langle p_0, \sigma\rangle \xrightarrow{*} \sigma''$ *and* $\langle p_1, \sigma''\rangle \xrightarrow{*} \sigma'$

Proof:

Assume $\langle p_0;\ p_1, \sigma\rangle \xrightarrow{*} \sigma'$
Now assume $\neg\exists\sigma''.\langle p_0, \sigma\rangle \xrightarrow{*} \sigma''$ then
$\neg\exists\sigma'.\langle p_0;\ p_1, \sigma\rangle \xrightarrow{*} \sigma'$ (contradiction)
So $\exists\sigma''.\langle p_0, \sigma\rangle \xrightarrow{*} \sigma''$ and hence
$\langle p_0;\ p_1, \sigma\rangle \xrightarrow{*} \langle p_1, \sigma''\rangle$ by Lemma A.1,
and so as $\langle p_0;\ p_1, \sigma\rangle \xrightarrow{*} \sigma'$ then by lemma A.12 $\langle p_1, \sigma''\rangle \xrightarrow{*} \sigma'$ $\square$

Appendix B

Soundness and Completeness Proofs

In this appendix we shall present sketchy outlines of some of the soundness proofs of the axioms and rules in the W system, and of proofs of the completeness theorems quoted in the main body of the thesis. We begin with the soundness proofs.

B.1 Soundness of $\Leftrightarrow_I$

Theorem B.1 *(Soundness of $\Leftrightarrow_I$)*
$\forall i_0, i_1 \in \mathbf{IF}$ *if* $i_0 \Leftrightarrow_I i_1$ *then* $i_0 \equiv i_1$

We proceed by proving each axiom and rule of $\Leftrightarrow_I$ to be sound. These proofs are simple, although lengthy. We only include a few such proofs to give a taste of the style. The strategy in proving $i_0 \Leftrightarrow_I i_1$ is, using Lemma A.3, to assume $\exists \sigma'.\langle i_0, \sigma\rangle \xrightarrow{*} \sigma'$, and then show that $\langle i_1, \sigma\rangle \xrightarrow{*} \sigma'$.

(; *elim*$_2$) $\qquad \bar{x} := \bar{c}; \mathbf{if}(b, i_0, i_1) \Leftrightarrow_I \mathbf{if}(b[c_1/x_1 \ldots c_n/x_n], \bar{x} := \bar{c}; i_0, \bar{x} := \bar{c}; i_1)$

Proof:
Assume $\langle \bar{x} := \bar{c}; \mathbf{if}(b, i_0, i_1), \sigma\rangle \xrightarrow{*} \sigma'$,
then by Lemma A.6 $\exists \sigma''.\langle \bar{x} := \bar{c}, \sigma\rangle \xrightarrow{*} \sigma''$ and $\langle \mathbf{if}(b, i_0, i_1), \sigma''\rangle \xrightarrow{*} \sigma'$
Now by Lemma A.7

$\exists m_i (i = 1, \ldots, n) \in \mathcal{A}. \langle c_i, \sigma \rangle \gg \langle m_i, \sigma \rangle \ \wedge \sigma'' = \sigma[x_1/m_1 \ldots x_n/m_n]$,
hence $\langle \mathbf{if}(b, p_0, p_1), \sigma[x_1/m_1 \ldots x_n/m_n] \rangle \xrightarrow{*} \sigma'$
We now consider cases of $\langle b, \sigma[x_1/m_1 \ldots x_n/m_n] \rangle$

(a) $\langle b, \sigma[x_1/m_1 \ldots x_n/m_n] \rangle \gg \langle true, \sigma[x_1/m_1 \ldots x_n/m_n] \rangle$
then by (w_3) (and Lemma A.5)
$\langle p_0, \sigma[x_1/m_1 \ldots x_n/m_n] \rangle \xrightarrow{*} \sigma'$

(b) $\langle b, \sigma[x_1/m_1 \ldots x_n/m_n] \rangle \gg \langle false, \sigma[x_1/m_1 \ldots x_n/m_n] \rangle$
then by (w_4) (and Lemma A.5)
$\langle p_1, \sigma[x_1/m_1 \ldots x_n/m_n] \rangle \xrightarrow{*} \sigma'$

Now consider the right hand side of the axiom:
$\langle \mathbf{if}(b[c_1/x_1 \ldots c_n/x_n], ma; p_0, ma; p_1), \sigma \rangle$
where $ma \triangleq (x_1 := c_1 \ \& \ldots \& \ x_n := c_n)$
From the above we have $\langle c_i, \sigma \rangle \gg \langle m_i, \sigma \rangle \ (i = 1, \ldots, n)$
Thus considering the cases above

(a) $\langle b, \sigma[x_1/m_1 \ldots x_n/m_n] \rangle \gg \langle true, \sigma[x_1/m_1 \ldots x_n/m_n] \rangle$
then $\langle b[c_1/x_1 \ldots c_n/x_n], \sigma \rangle \gg \langle true, \sigma \rangle$ by Lemma A.9, and
$\langle \mathbf{if}(b[c_1/x_1 \ldots c_n/x_n], ma; p_0, ma; p_1), \sigma \rangle \xrightarrow{*} \langle ma; p_0, \sigma \rangle$ by (w_3)
Also by Lemma A.10, since $\langle p_0, \sigma[x_1/m_1 \ldots x_n/m_n] \rangle \xrightarrow{*} \sigma'$,
$\langle ma; p_0, \sigma \rangle \xrightarrow{*} \sigma'$
which gives the required result by composing $\xrightarrow{*}$s.

(b) This case follows similarly.

□

(*skip elim*$_1$) $\quad$ **skip** $\Leftrightarrow_I x := x$

Proof:

Assume $\langle x := x, \sigma \rangle \xrightarrow{*} \sigma'$

then by Lemma A.7 $\exists m_1 \in \mathcal{A}.\langle x, \sigma\rangle \gg \langle m_1, \sigma\rangle \wedge \sigma' = \sigma[x/m_1]$

Now by (e_1), $\langle x, \sigma\rangle \gg \langle \sigma(x), \sigma\rangle$

So by Lemma A.4 $m_1 = \sigma(x)$, and $\sigma[x/\sigma(x)] = \sigma$

Hence $\langle x := x, \sigma\rangle \xrightarrow{*} \sigma$

By (w_0) $\langle \mathbf{skip}, \sigma\rangle \rightarrow \sigma$ □

$$(mono_2) \qquad \frac{i_0 \Leftrightarrow_I i_0',\ i_1 \Leftrightarrow_I i_1'}{\mathbf{if}(b, i_0, i_1) \Leftrightarrow_I \mathbf{if}(b, i_0', i_1')}$$

Proof:

Assume $i_0 \Leftrightarrow_I i_0'$ and $i_1 \Leftrightarrow_I i_1'$ are sound, i.e $i_0 \equiv i_0'$ and $i_1 \equiv i_1'$

Consider cases as follows

(a) $\langle b, \sigma\rangle \gg \langle true, \sigma\rangle$

Assuming $\langle \mathbf{if}(b, i_0, i_1), \sigma\rangle \xrightarrow{*} \sigma'$

then by (w_3), $\langle \mathbf{if}(b, i_0, i_1), \sigma\rangle \rightarrow \langle i_0, \sigma\rangle$

hence by Lemma A.3 and Lemma A.5 $\langle i_0, \sigma\rangle \xrightarrow{*} \sigma'$

Since $i_0 \equiv i_0'$, then by definition of $\equiv$,

$\langle i_0', \sigma\rangle \xrightarrow{*} \sigma'$, hence $\langle \mathbf{if}(b, i_0', i_1'), \sigma\rangle \rightarrow \langle i_0', \sigma\rangle \xrightarrow{*} \sigma'$

(b) $\langle b, \sigma\rangle \gg \langle false, \sigma\rangle$

This case follows similarly to the previous.

□

Similar proofs follow for the remaining axioms of $\Leftrightarrow_I$, and for the axioms of $\Leftrightarrow_N$.

B.2 Reduction to Normal Form

Lemma B.1 $\forall i \in \mathbf{IF} \quad \exists n \in NF.i \Leftrightarrow_I n$

Proof:

(by structural induction on i)

base: There are two cases to consider here,

$i \triangleq \mathbf{skip}$: This case follows using (*skip intro*$_1$) and (*if intro*$_1$).

$i \triangleq ma$: This case follows using (*if intro*$_1$).

step: There are two cases to consider in this case also,

$i \triangleq \mathbf{if}(b, i_0, i_1)$: By induction hypothesis $\exists i'_0, i'_1 \in NF.i_0 \Leftrightarrow_I i'_0$ and $i_1 \Leftrightarrow_I i'_1$. Hence using monotonicity $\mathbf{if}(b, i_0, i_1) \Leftrightarrow_I \mathbf{if}(b, i'_0, i'_1)$. We now prove that $\mathbf{if}(b, i'_0, i'_1) \Leftrightarrow_I i' \in NF$ by induction on the number of nested ifs, $lnf()$. Formally we define $lnf()$ as follows:

$lnf(ma) = lnf(\mathbf{skip}) = 0$

$lnf(\mathbf{if}(b, i, i')) = 1 + lnf(i) + lnf(i')$

$lnf(i;\, i') = lnf(i) + lnf(i')$

base :- $lnf(\mathbf{if}(b, i'_0, i'_1)) = 3$, since i'_0, i'_1 must at least be of the form

$i'_0 \triangleq \mathbf{if}(b_0, ma_0, ma'_0)$

$i'_1 \triangleq \mathbf{if}(b_1, ma_1, ma'_1)$

Hence using ($\wedge$ *intro*) we have

$\mathbf{if}(b, i'_0, i'_1)$

$\Leftrightarrow_I$

$\mathbf{if}(b \wedge b_0, ma_0, \mathbf{if}(b, ma'_0, \mathbf{if}(b_1, ma_1, ma'_1))) \in NF$.

step :- Now assume the hypothesis true for i. $lnf(i) \leq k$ $(k \geq 3)$, and consider $\mathbf{if}(b, \mathbf{if}(b_0, ma_0, i''_0), \mathbf{if}(b_1, ma_1, i''_1))$, where

$lnf(\mathbf{if}(b, \mathbf{if}(b_0, ma_0, i''_0), \mathbf{if}(b_1, ma_1, i''_1))) = k + 1$,

i.e $lnf(i''_0) + lnf(i''_1) = k - 2$. Now using ($\wedge$ *intro*) we have,

$\mathbf{if}(b \wedge b_0, ma_0, \mathbf{if}(b, i_0'', \mathbf{if}(b_1, ma_1, i_1'')))$, and

$\mathit{lnf}(\mathbf{if}(b, i_0'', \mathbf{if}(b_1, ma_1, i_1''))) = 1 + \mathit{lnf}(i_0'') + \mathit{lnf}(i_1'') = k$.

Hence by induction hypothesis,

$\mathbf{if}(b, i_0'', \mathbf{if}(b_1, ma_1, i_1'')) \Leftrightarrow_I i_2 \in NF$, and thus

$\mathbf{if}(b, \mathbf{if}(b_0, ma_0, i_0''), \mathbf{if}(b_1, ma_1, i_1''))$

$\Leftrightarrow_I \mathbf{if}(b \wedge b_0, ma_0, i_2) \in NF$.

$i \triangleq i_0; i_1$: This case is similar to $i \triangleq \mathbf{if}(b, i_0, i_1)$ case.

□

B.3 Completeness of $\Leftrightarrow_N$ for IF

We shall use the following lemma in the proof of completeness; this lemma enables us to consider completeness for a further restricted form of **IF** programs.

Lemma B.2 $\forall n_0, n_1 \in NF,\ \exists n_2, n_3 \in NF. n_0 \Leftrightarrow_N n_2 \wedge n_1 \Leftrightarrow_N n_3$ *and* n_2 *is of the form*

$$\mathbf{if}(b_1, ma_1^2, \mathbf{if}(b_2, ma_2^2, \ldots, \mathbf{if}(b_k, ma_k^2, ma_{k+1}^2)\ldots))$$

and n_3 *is of the form*

$$\mathbf{if}(b_1, ma_1^3, \mathbf{if}(b_2, ma_2^3, \ldots, \mathbf{if}(b_k, ma_k^3, ma_{k+1}^3)\ldots))$$

where $\forall i, j (i \neq j)\ b_i \wedge b_j = \mathit{false}$, *and*

$$ma_i^2 \triangleq x_0^i := e_0^i \& \ldots \& x_n^i := e_n^i \& y_0^i := e_{n+1}^i \& \ldots \& y_m^i := e_{n+m+1}^i$$

$$ma_i^3 \triangleq x_0^i := d_0^i \& \ldots \& x_n^i := d_n^i \& z_0^i := d_{n+1}^i \& \ldots \& z_m^i := d_{n+k+1}^i$$

Proof:

Let

$$n_0 \triangleq \mathbf{if}(a_1^0, ma_1^0, \mathbf{if}(a_2^0, ma_2^0, \ldots, \mathbf{if}(a_n^0, ma_n^0, ma_{n+1}^0)\ldots))$$

and

$$n_1 \triangleq \mathbf{if}(a_1^1, ma_1^1, \mathbf{if}(a_2^1, ma_2^1, \ldots, \mathbf{if}(a_m^1, ma_m^1, ma_{m+1}^1)\ldots))$$

Using (*if logic*$_2$) we can transform these into forms such that the new conditionals, say $a_1^2, \ldots, a_n^2$ and $a_1^3, \ldots, a_m^3$, satisfy

$$\forall i,j\ (i \neq j)\ (a_i^2 \wedge a_j^2 \equiv \mathit{false}) \wedge (a_i^3 \wedge a_j^3 \equiv \mathit{false})$$

Now consider $\mathbf{if}(a_i^2, ma_i^0, p),\ i = 1, \ldots, n$. Since

$$(a_1^3 \vee \ldots \vee a_m^3 \vee (\neg a_1^3 \wedge \ldots \wedge \neg a_m^3)) \equiv \mathit{true}$$

and $a_i^2 \wedge \mathit{true} \equiv a_i^2$, then

$$a_i^2 \equiv (a_i^2 \wedge a_1^3) \vee \ldots \vee (a_i^2 \wedge a_m^3) \vee (a_i^2 \wedge (\neg a_1^3 \wedge \ldots \wedge \neg a_m^3))$$

Hence using the derivable transformation ($\vee$ *elim*) (see Appendix C) and (*if intro*$_1$), n_0 becomes

$$\mathbf{if}(a_1^2 \wedge a_1^3, ma_1^0, \ldots \mathbf{if}(a_1^2 \wedge a_m^3, ma_1^0, \mathbf{if}(a_1^2 \wedge (\neg a_1^3 \wedge \ldots \wedge \neg a_m^3), ma_n^0, \ldots$$

$$\mathbf{if}(a_n^2 \wedge a_1^3, ma_n^0, \ldots \mathbf{if}(a_n^2 \wedge a_m^3, ma_n^0, \mathbf{if}(a_n^2 \wedge (\neg a_1^3 \wedge \ldots \wedge \neg a_m^3), ma_n^0,$$

$$\mathbf{if}(\neg a_1^2 \wedge \ldots \wedge \neg a_n^2 \wedge a_1^3, ma_{n+1}^0, \ldots \mathbf{if}(\neg a_1^2 \wedge \ldots \wedge \neg a_n^2 \wedge a_m^3, ma_{n+1}^0, ma_{n+1}^0)\ldots)\ldots))\ldots)\ldots))\ldots)$$

Repeating the above for n_1 (and using ($b_0 \wedge b_1 \equiv b_1 \wedge b_0$)) we obtain the same form, with ma_i^0 replaced by ma_i^1. Using (& *assoc*), (& *ident*) and (& *symm*) we can put the *ma*s in the special form required. Note that the conditionals have the required property because of the conditions that the individual a_i^2 and a_i^3 satisfy. □

Theorem B.2 *(Completeness of* $\Leftrightarrow_N$*)*
$\forall n, n' \in NF$ *if* $n \equiv n'$ *then* $n \Leftrightarrow_N n'$

Proof:
By Lemma B.2 we need only consider $n \equiv n'$, where n, n' are of the form:

$$n \triangleq \mathbf{if}(b_1, ma_1^2, \mathbf{if}(b_2, ma_2^2, \ldots, \mathbf{if}(b_k, ma_k^2, ma_{k+1}^2)\ldots))$$

, and

$$n' \triangleq \textbf{if}(b_1, ma_1^3, \textbf{if}(b_2, ma_2^3, \ldots, \textbf{if}(b_k, ma_k^3, ma_{k+1}^3)\ldots))$$

where $\forall i, j (i \neq j)\ b_i \wedge b_j = \mathit{false}$, and

$$ma_i^2 \triangleq x_0^i := e_0^i \& \ldots \& x_n^i := e_n^i \& y_0^i := e_{n+1}^i \& \ldots \& y_m^i := e_{n+m+1}^i$$

$$ma_i^3 \triangleq x_0^i := d_0^i \& \ldots \& x_n^i := d_n^i \& z_0^i := d_{n+1}^i \& \ldots \& z_m^i := d_{n+k+1}^i$$

Result follows by induction on k, using semantics to give possible cases and using $\Leftrightarrow_N$ transformations to show how all these cases are derivable via transformation. □

B.4 Soundness of $\Rightarrow_W$

In this section we shall prove the soundness of the W system. There are two subsections to consider; the first corresponding to the extension of the **IF** system, within which we shall only prove a single example, and the second subsection corresponding to the new **while** axioms and rules (as given in Fig. 3–1 and Fig. 3–2).

Our strategy for proving soundness is slightly more complicated than in the **IF** case as we must consider non-termination. To prove $p \Rightarrow_W p'$ sound, we must prove that $\forall \Phi, \forall \sigma \in STORE_\Phi . exec(p, \sigma) \simeq exec(p', \sigma)$, i.e

$$(p \Leftrightarrow_W p') \vee \neg \exists \sigma' \in T_{\mathbf{WHILE}} . \langle p', \sigma \rangle \xrightarrow{*} \sigma'$$

To prove $p \Leftrightarrow_W p'$ sound, we need only prove

$$\langle p, \sigma \rangle \xrightarrow{*} \sigma' \ \Rightarrow \ \langle p', \sigma \rangle \xrightarrow{*} \sigma', \sigma' \in T_{\mathbf{WHILE}}$$

and

$$\neg \exists \sigma' \in T_{\mathbf{WHILE}} . \langle p, \sigma \rangle \xrightarrow{*} \sigma' \ \Rightarrow \ \neg \exists \sigma' \in T_{\mathbf{WHILE}} . \langle p', \sigma \rangle \xrightarrow{*} \sigma'$$

The *only if* directions follow from these.

B.4.1 The IF System Extended

(; $elim_3$) $\quad$ $\mathbf{if}(b, p_0, p_1);\ p_2 \Leftrightarrow_W \mathbf{if}(b, p_0;\ p_2, p_1;\ p_2)$

Proof:

We have two cases corresponding to whether the program on the left hand side of the axiom reaches a terminal configuration or not, i.e corresponding to whether it terminates or not.

(1) In this case we assume $\langle \mathbf{if}(b, p_0, p_1);\ p_2, \sigma\rangle \xrightarrow{*} \sigma'$, and consider $\langle \mathbf{if}(b, p_0;\ p_2, p_1;\ p_2), \sigma\rangle$. We have two further cases to consider corresponding to the outcome of $\langle b, \sigma\rangle$:

(a) $\langle b, \sigma\rangle \gg \langle true, \sigma\rangle$

In this case, by (w_3) and (w_2), we have

$\langle \mathbf{if}(b, p_0, p_1);\ p_2, \sigma\rangle \rightarrow \langle p_0;\ p_2, \sigma\rangle$

and so by Lemma A.12 $\langle p_0;\ p_2, \sigma\rangle \xrightarrow{*} \sigma'$

Now, also by (w_3), we have

$\langle \mathbf{if}(b, p_0;\ p_2, p_1;\ p_2), \sigma\rangle \rightarrow \langle p_0;\ p_2, \sigma\rangle$

and therefore $\langle \mathbf{if}(b, p_0;\ p_2, p_1;\ p_2), \sigma\rangle \rightarrow \sigma'$

(b) $\langle b, \sigma\rangle \gg \langle false, \sigma\rangle$

This case is similar to the above.

(2) In this case we have $exec(\mathbf{if}(b, p_0, p_1);\ p_2, \sigma) = \perp$ (that is $\neg\exists\sigma'.\langle \mathbf{if}(b, p_0, p_1);\ p_2, \sigma\rangle \xrightarrow{*} \sigma'$), and consider $\langle \mathbf{if}(b, p_0;\ p_2, p_1;\ p_2), \sigma\rangle$. Again we have two further subcases to consider, corresponding to the outcome of $\langle b, \sigma\rangle$:

(a) $\langle b, \sigma\rangle \gg \langle true, \sigma\rangle$

In this case, by (w_3) and (w_2), we have

$\langle \mathbf{if}(b, p_0, p_1);\ p_2, \sigma\rangle \rightarrow \langle p_0;\ p_2, \sigma\rangle$

and by assumption $\neg\exists\sigma''.\langle p_0;\ p_2, \sigma\rangle \xrightarrow{*} \sigma''$,

otherwise $\langle \mathbf{if}(b, p_0, p_1);\ p_2, \sigma\rangle \xrightarrow{*} \sigma''$

Now, also by (w_3), we have

$\langle \mathbf{if}(b, p_0; p_2, p_1; p_2), \sigma\rangle \rightarrow \langle p_0; p_2, \sigma\rangle$

and therefore $\neg\exists\sigma''.\langle \mathbf{if}(b, p_0; p_2, p_1; p_2), \sigma\rangle \xrightarrow{*} \sigma''$

(b) $\langle b, \sigma\rangle \gg \langle \mathit{false}, \sigma\rangle$

This case is similar to the above.

□

B.4.2 The while Axioms and Rules

In the following soundness proofs we again have two cases corresponding to whether a terminal configuration is reached or not. For all of the soundness proofs, except (*while unfold*), we need to enhance our definition of $exec(p, \sigma)$ to be more specific, i.e

$$exec(\mathbf{while}(b,p),\sigma) = \begin{cases} \sigma' & \text{if } \exists n \geq 0, \sigma_0, \ldots, \sigma_n . \sigma = \sigma_n \wedge \\ & \langle b, \sigma_i\rangle \gg \langle \mathit{true}, \sigma_i\rangle \wedge \langle p, \sigma_i\rangle \xrightarrow{*} \sigma_{i+1}\ (i = 0, \ldots, n-1) \wedge \\ & \langle b, \sigma_n\rangle \gg \langle \mathit{false}, \sigma_n\rangle \\ \perp & \text{otherwise} \end{cases}$$

(*while logic*) $\quad \mathbf{while}(b,p); \mathbf{if}(\neg b, p_0, p_1) \Leftrightarrow_W \mathbf{while}(b,p); p_0$

Proof:

We have two cases corresponding to the outcome of $\langle \mathbf{while}(b,p), \sigma\rangle$:

(1) Assume $\exists n \geq 0, \sigma_0, \ldots, \sigma_n . \sigma = \sigma_n \wedge \langle b, \sigma_i\rangle \gg \langle \mathit{true}, \sigma_i\rangle \wedge \langle p, \sigma_i\rangle \xrightarrow{*} \sigma_{i+1}$ $(i = 0, \ldots, n-1)$ $\wedge \langle b, \sigma_n\rangle \gg \langle \mathit{false}, \sigma_n\rangle$ then,

$\langle \mathbf{while}(b,p); \mathbf{if}(\neg b, p_0, p_1), \sigma\rangle \xrightarrow{*} \langle \mathbf{if}(\neg b, p_0, p_1), \sigma_n\rangle \rightarrow \langle p_0, \sigma_n\rangle$

Similarly $\langle \mathbf{while}(b,p); p_0, \sigma\rangle \xrightarrow{*} \langle p_0, \sigma_n\rangle$

(2) $exec(\mathbf{while}(b,p),\sigma) = \perp$

In this case both, $exec(\mathbf{while}(b,p); \mathbf{if}(\neg b, p_0, p_1), \sigma) = \perp$ and $exec(\mathbf{while}(b,p); p_0, \sigma)$ $\perp$

□

(*while unfold*) $\textbf{while}(b,p) \Leftrightarrow_W \textbf{if}(b,p;\textbf{while}(b,p))$

Proof:

We have two cases corresponding to whether the program on the left hand side of the axiom reaches a terminal configuration or not.

(1) Assume $\langle \textbf{while}(b,p),\sigma\rangle \xrightarrow{*} \sigma'$, and consider $\langle \textbf{if}(b,p;\ \textbf{while}(b,p)),\sigma\rangle$. We have two further subcases:

(a) $\langle b,\sigma\rangle \gg \langle true,\sigma\rangle$
By (w_5), $\langle \textbf{while}(b,p),\sigma\rangle \rightarrow \langle p;\textbf{while}(b,p),\sigma\rangle$,
and therefore by Lemma A.12
$\langle p;\textbf{while}(b,p),\sigma\rangle \xrightarrow{*} \sigma'$
Now by (w_3)
$\langle \textbf{if}(b,p;\textbf{while}(b,p)),\sigma\rangle \rightarrow \langle p;\textbf{while}(b,p),\sigma\rangle$
and therefore $\langle \textbf{if}(b,p;\textbf{while}(b,p)),\sigma\rangle \xrightarrow{*} \sigma'$

(b) $\langle b,\sigma\rangle \gg \langle false,\sigma\rangle$
By (w_6)
$\langle \textbf{while}(b,p),\sigma\rangle \xrightarrow{*} \sigma$, and by Lemma A.11 $\sigma' = \sigma$
By (w_4) and (w_0) $\langle \textbf{if}(b,p;\textbf{while}(b,p)),\sigma\rangle \rightarrow \sigma$

(2) In this case we assume $exec(\textbf{while}(b,p),\sigma) = \perp$, and consider $\langle \textbf{if}(b,p;\ \textbf{while}(b,p)),\sigma\rangle$. We have two further subcases:

(a) $\langle b,\sigma\rangle \gg \langle true,\sigma\rangle$
By (w_5) $\langle \textbf{while}(b,p),\sigma\rangle \rightarrow \langle p;\textbf{while}(b,p),\sigma\rangle$,
and so by assumption $\neg\exists\sigma''.\langle p;\textbf{while}(b,p),\sigma\rangle \xrightarrow{*} \sigma''$,
otherwise $\langle \textbf{while}(b,p),\sigma\rangle \xrightarrow{*} \sigma''$
By (w_3)

$\langle \textbf{if}(b, p; \textbf{while}(b, p)), \sigma \rangle \rightarrow \langle p; \textbf{while}(b, p), \sigma \rangle$

and therefore $\neg \exists \sigma''. \langle \textbf{if}(b, p; \textbf{while}(b, p)), \sigma \rangle \xrightarrow{*} \sigma''$

(b) $\langle b, \sigma \rangle \gg \langle \mathit{false}, \sigma \rangle$

This leads to a contradiction of the assumption, and so cannot hold.

□

$$(\mathit{while\ fold}) \qquad \frac{p \;\Rightarrow_W\; \textbf{if}(b, p_1; p)}{p \;\Rightarrow_W\; \textbf{while}(b, p_1)}$$

Proof:

From the premiss of this rule we may assume $\forall \Phi, \forall \sigma \in \mathit{STORE}_\Phi, \mathit{exec}(p, \sigma) \simeq \mathit{exec}(\textbf{if}(b, p_1; p), \sigma)$. We have two cases to consider corresponding to the outcome of $\mathit{exec}(\textbf{while}(b, p_1), \sigma)$.

(1) Assume $\exists n \geq 0, \sigma_0, \ldots, \sigma_n. \sigma = \sigma_n \wedge \langle b, \sigma_i \rangle \gg \langle \mathit{true}, \sigma_i \rangle \wedge \langle p_1, \sigma_i \rangle \xrightarrow{*} \sigma_{i+1}\ (i = 0, \ldots, n-1)\ \wedge \langle b, \sigma_n \rangle \gg \langle \mathit{false}, \sigma_n \rangle$

We have two further subcases to consider:

(a) Consider $\langle p, \sigma \rangle \xrightarrow{*} \sigma'$

By assumption either $\langle \textbf{if}(b, p_1; p), \sigma \rangle \xrightarrow{*} \sigma'$ or $\mathit{exec}(\textbf{if}(b, p_1; p), \sigma) = \bot$.

Consider $\langle \textbf{if}(b, p_1; p), \sigma \rangle \xrightarrow{*} \sigma'$,

since $\langle b, \sigma_0 \rangle \gg \langle \mathit{true}, \sigma_0 \rangle$ (and $\sigma_0 = \sigma$),

$\langle \textbf{if}(b, p_1; p), \sigma \rangle \xrightarrow{*} \langle p_1; p, \sigma \rangle$ by (w_3),

and we also have that $\langle p_1, \sigma \rangle \xrightarrow{*} \sigma_1$, and so by (w_2)

$\langle p_1; p, \sigma \rangle \rightarrow \langle p, \sigma_1 \rangle$.

Continuing this process we get,

$\langle \textbf{if}(b, p_1; p), \sigma \rangle \xrightarrow{*} \sigma_n$,

and so by Lemma A.11, $\sigma' = \sigma_n$, and so

$\langle \textbf{while}(b, p_1), \sigma \rangle \xrightarrow{*} \sigma'$.

Hence in this case $\mathit{exec}(p, \sigma) \approx \mathit{exec}(\textbf{while}(b, p_1), \sigma)$.

Now consider $\mathit{exec}(\textbf{if}(b, p_1; p), \sigma) = \bot$.

By assumption $\langle b, \sigma\rangle \gg \langle true, \sigma\rangle \wedge \langle p_1, \sigma\rangle \xrightarrow{*} \sigma_1$, hence
$exec(p, \sigma_1) = \bot$

By premiss assumption $exec(\mathbf{if}(b, p_1;\ p), \sigma_1) = \bot$.

Continuing this process we get $exec(\mathbf{if}(b, p_1;\ p), \sigma_n) = \bot$,

and since by assumption $\langle b, \sigma_n\rangle \gg \langle false, \sigma_n\rangle$,

this implies that $exec(\mathbf{skip}, \sigma_n) = \bot$, which is false.

Hence a contradiction, so this case cannot hold.

(b) $exec(p, \sigma) = \bot$

By assumption $exec(\mathbf{if}(b, p_1;\ p), \sigma) = \bot$,

and, as in the previous case,

since $\langle b, \sigma_0\rangle \gg \langle true, \sigma_0\rangle$ (and $\sigma_0 = \sigma$),

$\langle \mathbf{if}(b, p_1;\ p), \sigma\rangle \xrightarrow{*} \langle p_1;\ p, \sigma\rangle$ by (w_3),

and we also have that $\langle p_1, \sigma\rangle \xrightarrow{*} \sigma_1$, and so by (w_2)

$\langle p_1;\ p, \sigma\rangle \rightarrow \langle p, \sigma_1\rangle$.

Hence $\langle \mathbf{if}(b, p_1;\ p), \sigma\rangle \xrightarrow{*} \langle \mathbf{if}(b, p_1;\ p), \sigma_1\rangle$,

and so continuing this process we get,

$\langle \mathbf{if}(b, p_1;\ p), \sigma\rangle \xrightarrow{*} \sigma_n$, a contradiction.

(2) $exec(\mathbf{while}(b, p_1), \sigma) = \bot$.

In this case we cannot prove anything, and the premiss assumption is not contradicted.

Overall we therefore have only weak equivalence; $exec(p, \sigma) \approx exec(\mathbf{while}(b, p_1), \sigma)$ or $exec(\mathbf{while}(b, p_1), \sigma) = \bot$. □

The soundness proofs for the while fold in context rules follow the same strategy as for (*while fold*). We outline one case in the soundness proof of (*while fold in context*$_3$).

Consider $\langle p_0;\ \mathbf{while}(b_1, p_1), \sigma\rangle$ and assume $\exists n \geq 0, \sigma_0, \ldots, \sigma_n . \langle b_1, \sigma_i\rangle \gg \langle true, \sigma_i\rangle \wedge \langle p_1, \sigma_i\rangle \xrightarrow{*} \sigma_{i+1}$ $(i = 0, \ldots, n-1)$ $\wedge \langle b_1, \sigma_n\rangle \gg \langle false, \sigma_n\rangle$, where $\langle p_0, \sigma\rangle \xrightarrow{*} \sigma_0$.
Now consider $\langle p_0;\ p, \sigma\rangle \xrightarrow{*} \sigma'$. From the premisses of (*while fold in context*$_3$) we have:

$$exec(p_0;\ p, \sigma) \simeq exec(\mathbf{if}(b_1, p_1;\ p_0;\ p, p_0), \sigma)$$

and

$$exec(p_0;\ p, \sigma) \approx exec(p_0;\ \mathbf{if}(b_1, p_1;\ p, \mathbf{skip}), \sigma)$$

Hence $\langle \mathbf{if}(b_1, p_1;\ p_0;\ p, p_0), \sigma\rangle \xrightarrow{*} \sigma'$ (or $exec(\mathbf{if}(b_1, p_1;\ p_0;\ p, p_0), \sigma) = \bot$)
Repeating this replacement n times, and using the premises of (*while fold in context*$_3$) (and their consequences) n times we have:

$$\langle p_0;\ \mathbf{if}(b_1, p_1, \ldots \mathbf{if}(b_1, p_1;\ p)\ldots), \sigma\rangle \xrightarrow{*} \sigma'$$

Thus by assumptions for this case, $\langle \mathbf{if}(b_1, p_1;\ p), \sigma_n\rangle \xrightarrow{*} \sigma'$,
and so $\langle \mathbf{skip}, \sigma_n\rangle \xrightarrow{*} \sigma'$. Hence $\sigma_n = \sigma'$, and so $\langle p_0;\ \mathbf{while}(b_1, p_1), \sigma\rangle \xrightarrow{*} \sigma'$.

Appendix C

List of Transformations

Tables C-1 and C-2 form a collective list of our W system transformations.

$(;\ elim_1)$	$\bar{x} := \bar{c};\ \bar{y} := \bar{d} \Leftrightarrow_W \mathop{\&}_{x_j \in X_n - Y_m} x_j := c_j$ $\mathop{\&}_{y_j \in Y_m} y_j := d_j[c_1/x_1 \ldots c_n/x_n]$
$(;\ elim_2)$	$\bar{x} := \bar{c};\ \mathbf{if}(b, p_0, p_1) \Leftrightarrow_W \mathbf{if}(b[c_1/x_1 \ldots c_n/x_n], \bar{x} := \bar{c};\ p_0, \bar{x} := \bar{c};\ p_1)$
$(;\ elim_3)$	$\mathbf{if}(b, p_0, p_1);\ p_2 \Leftrightarrow_W \mathbf{if}(b, p_0;\ p_2, p_1;\ p_2)$
$(\wedge\ intro)$	$\mathbf{if}(b_0, \mathbf{if}(b_1, p_0, p_1), p_2) \Leftrightarrow_W \mathbf{if}(b_0 \wedge b_1, p_0, \mathbf{if}(b_0, p_1, p_2))$
$(skip\ elim_1)$	$\mathbf{skip} \Leftrightarrow_W x := x$
$(skip\ elim_2)$	$\mathbf{skip};\ p \Leftrightarrow_W p$
$(skip\ elim_3)$	$p;\ \mathbf{skip} \Leftrightarrow_W p$
$(if\ elim_1)$	$\mathbf{if}(b, p, p) \Leftrightarrow_W p$
$(if\ elim_2)$	$\mathbf{if}(true, p_0, p_1) \Leftrightarrow_W p_0$
$(if\ elim_3)$	$\mathbf{if}(false, p_0, p_1) \Leftrightarrow_W p_1$
$(\&\ ident)$	$ma\ \&\ y := y \Leftrightarrow_W ma$
$(\&\ symm)$	$ma \& ma' \Leftrightarrow_W ma' \& ma$
$(if\ logic_1)$	$\mathbf{if}(b, p_0, p_1) \Leftrightarrow_W \mathbf{if}(\neg b, p_1, p_0)$
$(if\ logic_2)$	$\mathbf{if}(b_0, p_0, \mathbf{if}(b_1, p_1, p_2)) \Leftrightarrow_W \mathbf{if}(b_0, p_0, \mathbf{if}(\neg b_0 \wedge b_1, p_1, p_2))$

Figure C–1: transformation system W

(; *assoc*) $\quad i_0; (i_1; i_2) \Leftrightarrow_W (i_0; i_1); i_2$

(& *assoc*) $\quad ma\&(ma'\&ma'') \Leftrightarrow_W (ma\&ma')\&ma''$

(*exp/bool*) $$\frac{b \Rightarrow (c_i \equiv d_i \quad i = 1, \ldots, n)}{\mathbf{if}(b, \bar{x} := \bar{c}, n) \Leftrightarrow_W \mathbf{if}(b, \bar{x} := \bar{d}, n)}$$

($bool_1$) $$\frac{b_0 \equiv b_1}{\mathbf{if}(b_0, p_0, p_1) \Leftrightarrow_W \mathbf{if}(b_1, p_0, p_1)}$$

($bool_2$) $$\frac{b_0 \equiv b_1}{\mathbf{while}(b_0, p) \Leftrightarrow_W \mathbf{while}(b_1, p)}$$

(*while logic*) $\quad \mathbf{while}(b, p); \mathbf{if}(\neg b, p_0, p_1) \Leftrightarrow_W \mathbf{while}(b, p); p_0$

(*while unfold*) $\quad \mathbf{while}(b, p) \Leftrightarrow_W \mathbf{if}(b, p; \mathbf{while}(b, p))$

(*while fold*) $$\frac{p \Rightarrow_W \mathbf{if}(b, p_1; p)}{p \Rightarrow_W \mathbf{while}(b, p_1)}$$

(*while fold in context*$_1$)
$$\frac{\mathbf{if}(b, p, p_0) \Rightarrow_W \mathbf{if}(b, \mathbf{if}(b_1, p_1; p), p_0), \ \{b\}\mathbf{if}(b_1, p_1)\{b\}}{\mathbf{if}(b, p, p_0) \Rightarrow_W \mathbf{if}(b, \mathbf{while}(b_1, p_1), p_0)}$$

(*while fold in context*$_2$) $$\frac{p; p_2 \Rightarrow_W \mathbf{if}(b_1, p_1; p); p_2}{p; p_2 \Rightarrow_W \mathbf{while}(b_1, p_1); p_2}$$

(*while fold in context*$_3$)
$$\frac{p_0; p \Rightarrow_W p_0; \mathbf{if}(b_1, p_1; p), \ p_0; p \Leftrightarrow_W \mathbf{if}(b_1, p_1; p_0; p, p_0)}{p_0; p \Rightarrow_W p_0; \mathbf{while}(b_1, p_1)}$$

Figure C–2: transformation system W (contd.)

C.1 Derived Transformations

The following derived transformations are used in the body of this thesis, and are built into the implementation.

$(deriv_1)$	$\mathbf{if}(b, \mathbf{if}(b, p_1, p_2), p_3) \Leftrightarrow_W \mathbf{if}(b, p_1, p_3)$

Proof:

if b **then if** b **then** p_1 **else** p_2 **fi else** p_3 **fi**

$\Rightarrow$ (*if logic*$_1$ *intro*), (*if logic*$_1$ *intro*)

if $\neg\, b$ **then** p_3 **else if** $\neg\, b$ **then** p_2 **else** p_1 **fi fi**

$\Rightarrow$ (*if logic*$_2$ *intro*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$, $(\neg\neg\, b \equiv b)$

if $\neg\, b$ **then** p_3 **else if** $(\neg\, b \wedge b)$ **then** p_2 **else** p_1 **fi fi**

$\Rightarrow (\neg\, b \wedge b \equiv \mathit{false})$

if $\neg\, b$ **then** p_3 **else if** *false* **then** p_2 **else** p_1 **fi fi**

$\Rightarrow$ (*if logic*$_1$ *elim*), (*if elim*$_3$)

if b **then** p_1 **else** p_3 **fi**

□

$(deriv_2)$	$\mathbf{if}(b_1, \mathbf{if}(b_2, p_1, p_2), p_3) \Leftrightarrow_W \mathbf{if}(b_1, \mathbf{if}(b_2 \wedge b_1, p_1, \mathbf{if}(b_1 \wedge \neg b_2, p_2)), p_3)$

Proof:

if b_1 **then if** b_2 **then** p_1 **else** p_2 **fi else** p_3 **fi**

$\Rightarrow$ (*if intro*$_2$)

if b_1

then if b_2 **then** p_1 **else if** *true* **then** p_2 **else** $<$ *program* $>$ **fi fi**

else p_3

fi

$\Rightarrow$ (*if logic*$_1$ *intro*), (*if logic*$_2$ *intro*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$

if $\neg\ b_1$

then p_3

else if b_2

then p_1

else if $(true \wedge \neg\ b_2)$ **then** p_2 **else** $< program >$ **fi**

fi

fi

$\Rightarrow$ (*if logic*$_2$ *intro*), (*if logic*$_1$ *elim*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$, $(\neg\neg b \equiv b)$, $(true \wedge b \equiv b)$

if b_1

then if $(b_2 \wedge b_1)$

then p_1

else if $\neg\ b_2$ **then** p_2 **else** $< program >$ **fi**

fi

else p_3

fi

$\Rightarrow$ (*if logic*$_1$ *intro*), (*if logic*$_1$ *intro*)

if $\neg\ b_1$

then p_3

else if $\neg\ (b_2 \wedge b_1)$

then if $\neg\ b_2$ **then** p_2 **else** $< program >$ **fi**

else p_1

fi

fi

$\Rightarrow$ (*if logic*$_2$ *intro*), (*if logic*$_1$ *elim*), (*if logic*$_1$ *intro*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$, $(\neg\neg b \equiv b)$, (*if intro*$_2$)

if b_1

then if $\neg\ (\neg\ (b_2 \wedge b_1) \wedge b_1)$

then p_1

else if *true*

then if $\neg\, b_2$ **then** p_2 **else** $<$ *program* $>$ **fi**

else $<$ *program* $>$

fi

fi

else p_3

fi

$\Rightarrow$ (*if logic*$_2$ *intro*), (*if logic*$_1$ *elim*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0), (\neg\neg\, b \equiv b)$

if b_1

then if $(\neg\,(b_2 \wedge b_1) \wedge b_1)$

then if $(true \wedge (\neg\,(b_2 \wedge b_1) \wedge b_1))$

then if $\neg\, b_2$ **then** p_2 **else** $<$ *program* $>$ **fi**

else $<$ *program* $>$

fi

else p_1

fi

else p_3

fi

$\Rightarrow (b_0 \wedge b_1 \equiv b_1 \wedge b_0), (true \wedge b \equiv b)$

if b_1

then if $(b_1 \wedge \neg\,(b_2 \wedge b_1))$

then if $(\neg\,(b_2 \wedge b_1) \wedge b_1)$

then if $\neg\, b_2$ **then** p_2 **else** $<$ *program* $>$ **fi**

else $<$ *program* $>$

fi

else p_1

fi

else p_3

fi

$\Rightarrow$ (*if logic*$_1$ *intro*), ($\wedge$ *elim*), (*if logic*$_1$ *intro*)

if $\neg\, b_1$

then p_3

```
    else if ¬ b1
             then p1
             else if ¬ (b2 ∧ b1)
                      then if (¬ (b2 ∧ b1) ∧ b1)
                               then if ¬ b2 then p2 else < program > fi
                               else < program >
                           fi
                      else p1
                  fi
         fi
fi
```

$\Rightarrow$ (*if logic*$_2$ *intro*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$, $(\neg\neg b \equiv b)$, $(\neg b \wedge b \equiv false)$

```
if ¬ b1
    then p3
    else if false
             then p1
             else if ¬ (b2 ∧ b1)
                      then if (¬ (b2 ∧ b1) ∧ b1)
                               then if ¬ b2 then p2 else < program > fi
                               else < program >
                           fi
                      else p1
                  fi
         fi
fi
```

$\Rightarrow$ (*if logic*$_1$ *elim*), (*if elim*$_3$), (*if logic*$_1$ *intro*), ($\wedge$ *elim*), (*if logic*$_1$ *intro*), $(\neg\neg b \equiv b)$

```
if b1
    then if ¬ ¬ (b2 ∧ b1)
             then p1
             else if (b2 ∧ b1)
                      then < program >
                      else if b1
                               then if ¬ b2 then p2 else < program > fi
```

```
                    else < program >
                fi
            fi
        fi
  else p3
fi
```

$\Rightarrow$ (*if logic*$_2$ *intro*), $(\neg\neg\, b \equiv b)$, (*if logic*$_1$ *elim*), $(\neg\, b \wedge b \equiv \mathit{false})$

```
if b1
  then if ¬ (b2 ∧ b1)
        then if false
              then < program >
              else if b1
                    then if ¬ b2 then p2 else < program > fi
                    else < program >
                  fi
            fi
        else p1
      fi
  else p3
fi
```

$\Rightarrow$ (*if elim*$_3$), ($\wedge$ *intro*), (*if elim*$_1$), (*if logic*$_1$ *elim*)

```
if b1
  then if (b2 ∧ b1)
        then p1
        else if (b1 ∧ ¬ b2) then p2 else skip fi
        fi
  else p3
fi
```

□

$(deriv_4)$	$\mathbf{if}(b, p_1; p_2, p_3) \Leftrightarrow_W \mathbf{if}(b, \mathbf{if}(b, p_1, \mathbf{skip}); p_2, p_3)$

Proof:

if b **then** $p_1; p_2$ **else** p_3 **fi**

$\Rightarrow$ (*if logic*$_1$ *intro*), (*if intro*$_2$)

if $\neg\, b$ **then** p_3 **else** **if** $true$ **then** $p_1; p_2$ **else** p_2 **fi** **fi**

$\Rightarrow$ (*if logic*$_2$ *intro*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$, $(\neg\neg\, b \equiv b)$, (*skip intro*$_2$)

if $\neg\, b$ **then** p_3 **else** **if** $(true \wedge b)$ **then** $p_1; p_2$ **else** **skip**; p_2 **fi** **fi**

$\Rightarrow$ (*if logic*$_1$ *elim*), (*; intro*$_3$), $(true \wedge b \equiv b)$

if b **then** **if** b **then** p_1 **else** **skip** **fi** ; p_2 **else** p_3 **fi**

□

$(deriv_5)$	$\mathbf{if}(b_1, \mathbf{if}(b_2, p, \mathbf{skip}), \mathbf{skip}) \Leftrightarrow_W \mathbf{if}(b_1, \mathbf{if}(b_2 \vee \neg b_1, p, \mathbf{skip}), \mathbf{skip})$

Proof:

if b_1 **then** **if** b_2 **then** p **else** **skip** **fi** **else** **skip** **fi**

$\Rightarrow$ (*if logic*$_1$ *intro*), (*if logic*$_1$ *intro*)

if $\neg\, b_1$ **then** **skip** **else** **if** $\neg\, b_2$ **then** **skip** **else** p **fi** **fi**

$\Rightarrow$ (*if logic*$_2$ *intro*), (*if logic*$_1$ *elim*), (*if logic*$_1$ *intro*), $(\neg(b_0 \wedge b_1) \equiv \neg b_0 \vee \neg b_1)$,
$(b_0 \vee b_1 \equiv b_1 \vee b_0)$, $(\neg\neg b \equiv b)$, $(\neg\neg b \equiv b)$

if b_1 **then** **if** $(b_2 \vee \neg\, b_1)$ **then** p **else** **skip** **fi** **else** **skip** **fi**

□

$(deriv_6)$	$\mathbf{if}(b_0, \mathbf{if}(b_1, p_1, p_2), p_3) \Leftrightarrow_W \mathbf{if}(b_0, \mathbf{if}(b_1 \wedge b_0, p_1, p_2), p_3)$

Proof:

if b_0 **then** **if** b_1 **then** p_1 **else** p_2 **fi** **else** p_3 **fi**

$\Rightarrow$ (*if logic*$_1$ *intro*)

if $\neg\ b_0$ **then** p_3 **else if** b_1 **then** p_1 **else** p_2 **fi fi**

$\Rightarrow$ (*if logic*$_2$ *intro*), (*if logic*$_1$ *elim*), $(b_0 \wedge b_1 \equiv b_1 \wedge b_0)$, $(\neg\neg\ b \equiv b)$

if b_0 **then if** $(b_1 \wedge b_0)$ **then** p_1 **else** p_2 **fi else** p_3 **fi**

□

(*deriv*$_7$)	$\mathbf{if}(b, p_1, \mathbf{if}(b, p_2, p_3)) \Leftrightarrow_W \mathbf{if}(b, p_1, p_3)$

Proof:

if b **then** p_1 **else if** b **then** p_2 **else** p_3 **fi fi**

$\Rightarrow$ (*if logic*$_2$ *intro*), $(\neg\ b \wedge b \equiv \mathit{false})$

if b **then** p_1 **else if** *false* **then** p_2 **else** p_3 **fi fi**

$\Rightarrow$ (*if elim*$_3$)

if b **then** p_1 **else** p_3 **fi**

□

(*if logic*$_3$)	$\mathbf{if}(b_1, p_1, \mathbf{if}(b_2, p_2, p_3)) \Leftrightarrow_W \mathbf{if}(b_1, p_1, \mathbf{if}(b_2 \vee b_1, p_2, p_3))$

Proof:

if b_1

  then p_1

  else if b_2 **then** p_2 **else** p_3

fi

$\Rightarrow$ (*if logic*$_2$ *intro*)

if b_1

  then p_1

  else if $b_2 \wedge \neg b_1$ **then** p_2 **else** p_3 **fi**

$\Rightarrow (b \equiv b \vee false), (false \equiv b \wedge \neg b), ((b \wedge a) \vee (c \wedge a) \equiv (b \vee c) \wedge a)$

if b_1
  then p_1
  else **if** $(b_2 \vee b_1) \wedge \neg b_1$ **then** p_2 **else** p_3
fi

$\Rightarrow$ (*if logic*$_2$ *elim*)

if b_1
  then p_1
  else **if** $b_2 \vee b_1$ **then** p_2 **else** p_3
fi

□

(∨ *elim*)	$\mathbf{if}(b_1 \vee b_2, p_1, p_2) \Leftrightarrow_W \mathbf{if}(b_1, p_1, \mathbf{if}(b_2, p_1, p_2))$

Proof:

if $b_1 \vee b_2$
  then p_1
  else p_2
fi

$\Rightarrow (b_1 \vee b_2 \equiv \neg(\neg b_1 \wedge \neg b_2))$, (*if logic*$_1$ *elim*)

if $\neg b_1 \wedge \neg b_2$
  then p_2
  else p_1
fi

$\Rightarrow$ (*if intro*$_1$)

if $\neg b_1$
  then **if** $\neg b_2$ **then** p_2 **else** p_1 **fi**
  else p_1
fi

$\Rightarrow$ (*if logic*$_1$ *elim*), (*if logic*$_1$ *elim*)

if b_1

  then p_1

  else if b_2 **then** p_1 **else** p_2 **fi**

fi

□

Bibliography

[Aho/Ullman 72] A.V. Aho, J.D. Ullman, *Optimization of Straight line Programs*, pp1-19 in SIAM J. Comput. 1, 1972.

[Arsac 79] Jacques Arsac, *Syntactic Source-to-Source Transforms and Program Manipulation*, pp43-54 in CACM 22(1), 1979.

[Arsac 85] J. Arsac, *Foundations of Programming*, Academic Press, 1985.

[Back 87] R.J.R. Back, *A Calculus of Refinements for Program Derivations*, Abo Akademi Technical Report, 1987.

[Backus 78] John Backus, *Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs*, pp613-641 in CACM 21(8), 1978.

[Backus 81a] J. Backus, *The Algebra of Functional Programs: Function Level Reasoning, Linear Equations, and Extended Definitions*, pp1-43 in *Formalization of Programming Concepts*, G. Goos, J. Hartmanis (Editors), Springer-Verlag, 1981.

[Backus 81b] J. Backus, *Is Computer Science Based on the Wrong Fundamental Concept of 'Program'? An Extended*

Concept, pp133-165 in *Algorithmic Languages*, J. de Bakker, van Vliet (Editors), North-Holland, 1981.

[de Bakker 80] Jaco de Bakker, *Mathematical Theory of Program Correctness*, Prentice-Hall, 1980.

[Balzer 81] Robert Balzer, *Transformational Implementation: An Example*, pp3-14 in IEEE Trans. on Soft. Eng. SE-7(1), 1981.

[Barstow 85] D. Barstow, *On Convergence toward a database of program transformations*, pp1-9 in TOPLAS 7(1), 1985.

[Bloom/Tindell 83] S.L. Bloom, R.Tindell, *Varieties of IF-THEN-ELSE*, pp677-707 in SIAM J. Comput. 12(4), 1983.

[Burstall/Darlington 77] R.M. Burstall, J. Darlington, *A Transformation System for Developing Recursive Programs*, pp44-67 in JACM 24(1), 1977.

[Cheatham et al 81] Thomas Cheatham, Glenn Holloway, Judy Townley, *Program Refinement by Transformation*, pp430-437 in IEEE 5th Int. Conf. on Soft. Eng., 1981.

[Chusho 80] T. Chusho, *A Good Program = A Structured Program + Optimization Commands*, pp269-274 in IFIP '80, 1980.

[Cousineau 77] G. Cousineau, *Transformation de Programmes Iteratifs*, pp33-74 in *Programmation*, B. Robinet (Editor), Dunod, 1977.

[Darlington 81] J. Darlington, *The Stuctured Description of Algorithm Derivations*, pp221-250 in *Algorithmic Lan-*

guages, J. de Bakker, van Vliet (Editors), North-Holland, 1981.

[Darlington 82] John Darlington, *Program Transformation*, pp193-215 in *Functional Programming: an advanced course*, J. Darlington, P. Henderson and D.A. Turner (Editors), Cambridge, 1982.

[Darlington/Burstall 76] J. Darlington, R.M. Burstall, *A System which Automatically Improves Programs*, pp41-60 in Acta Informatica 6, 1976.

[Dershowitz 81] Nachum Dershowitz, *The Evolution of Programs: Program Abstraction and Instantiation*, pp79-89 in 5th Int. Conf. on Soft. Eng., 1981.

[van Diepen/de Roever 86] N.W.P. van Diepen, W.P. de Roever, *Program Derivation through Transformations: The Evolution of List Copying Algorithms*, pp213-272 in Science of Computer Programming 6, 1986.

[Ershov 82] A.P.Ershov, *Mixed Computation: Potential Applications and Problems for Study*, pp41-67 in TCS 18, 1982.

[Feather 82] Martin Feather, *A System for Assisting Program Transformation*, pp1-20 in TOPLAS 4(1), 1982.

[Gerhart 75] S.L. Gerhart, *Correctness Preserving Program Transformations*, pp54-66 in 2nd ACM Symp. on Principles of Programming Languages, 1975.

[Goldberg 86] A. Goldberg, *Knowledge-based Programming: a survey of program design and construction techniques*,

pp752-769 in IEEE Trans. on Soft. Eng. SE-12(7), 1986.

[Guessarian 85] I. Guessarian, *Survey on Classes of Interpretations and some of their Applications*, pp383-409 in *Algebraic Methods in Semantics*, M. Nivat, J.C. Reynolds (Editors), Cambridge, 1985.

[Harel 80] David Harel, *On Folk Theorems*, pp379-388 in CACM 23(7), 1980.

[Harrison/Khoshnevisan 86] P.G. Harrison, H. Khoshnevisan, *The Transformation of Linear recursive Functions into Iterative Form*, Imperial College Technical Report, 1986.

[Hoare et al 85] C.A.R. Hoare, J. He, I.J. Hayes, C.C. Morgan, J.W. Sanders, I.H. Sorensen, J.M. Spivey, B.A. Sufrin, A.W. Roscoe, *Laws of Programming: A Tutorial Paper*, Oxford Univeristy PRG Technical Monograph 45, 1985.

[Huet/Lang 78] G. Huet, B. Lang, *Proving and Applying Program Transformations Expressed with second-order patterns*, pp31-55 in Acta Informatica 11, 1978.

[Kibler 78] D.F. Kibler, *Power, Efficiency and Correctness of Transformation Systems*, University of California Ph.D Thesis, 1978.

[Koga 85] A. Koga, *On Program transformation with Tupling Technique*, Kyoto University Technical Report, 1985.

[Kott 78] L. Kott, *About Transformation System: A Theoretical Study*, pp232-247 in *Program Transformations*, B. Robinet (Editor), Dunod, 1978.

[Kott 85] L. Kott, *Unfold/Fold Program Transformations*, pp412-433 in *Algebraic Methods in Semantics*, M. Nivat, J.C. Reynolds (Editors), Cambridge, 1985.

[Lovemann 77] D.B. Lovemann, *Program Improvement by Source-to-Source Transformation*, pp121-145 in JACM 24(1), 1977.

[Manna 74] Zohar Manna, *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[Marcotty/Ledgard 87] Micheal Marcotty, Henry Ledgard, *The World of Programming Languages*, Springer-Verlag, 1987.

[Maher/Sleeman 83] B. Maher, D.H. Sleeman, *Automatic Program Improvement: Variable Usage Transformations*, pp236-264 in TOPLAS 5(2), 1983.

[Mason 86] Ian A. Mason, *The Semantics of Destructive Lisp*, CSLI Lecture Notes 5, 1986.

[McCarthy 63] J. McCarthy, *A Basis for a Mathematical Theory of Computation*, pp33-70 in *Computer Programming and Formal Systems*, P. Braffort, D. Hirchberg (Editors), North-Holland, 1963.

[Meertens 83] Lambert Meertens, *Algorithmics*, pp289-334 in Proc. of the CWI Symp., 1983.

[Mycroft 81] Alan Mycroft, *Abstract Interpretation and Optimising Transformations for Applicative Programs*, University of Edinburgh Computer Science Dept. Ph.D Thesis, 1981.

[Neilson 81] F. Neilson, *Program Transformations in a Denotational Setting*, pp359-379 in TOPLAS 7(3), 1985.

[Partsch/Steinbruggen 83] H. Partsch, R. Steinbruggen, *Program Transformation Systems*, pp199-236 in Computing Surveys, 15(3), 1983.

[Pepper 79] P. Pepper, *A Study on Transformational Semantics*, pp322-405 in *Program Construction*, F.L. Bauer, M. Broy, LNCS 69, Springer-Verlag, 1979.

[Pettorossi 84] Alberto Pettorossi, *Methodologies for Transformations and Memoing in Applicative Languages*, University of Edinburgh Computer Science Dept. Ph.D Thesis, 1984.

[Pettorossi/Proietti 88] Alberto Pettorossi, Maurizio Proietti, *Deriving Programs which avoids both Call-by-Need and Multiple Traversals of Structures*, working paper distributed at IFIP WG2.1 (Rome), 1988.

[Plotkin 81] G.D. Plotkin, *A Structural Approach to Operational Semantics*, Aarhus University Technical Report DAIMI FN-19, 1981.

[Reps/Teitelbaum 85] Thomas Reps, Tim Teitelbaum, *The Synthesizer Generator Reference Manual*, Cornell University Computer Science University, 1985.

[Sabelfield 78] V.K. Sabelfield, *Aquivalele Transformationen fur Flussdiagramme*, pp127-155 in Acta Informatica 10(2), 1978.

[Scherlis 80] W.L. Scherlis, *Expression Procedures and Program Derivation*, Stanford University Computer Science Dept. Ph.D Thesis, 1980.

[Scherlis 81] W.L. Scherlis, *Program Improvement by Internal Specialization*, pp41-49 in 8th *ACM Symposium on Principles of Programming Languages*, 1981.

[Scherlis/Scott 83] W.L. Scherlis, D. Scott, *First Steps Towards Inferential Programming*, pp199-211 in *Information Processing '83*, R.E.A. Mason (Editor), North-Holland, 1983.

[Smith et al 85] Douglas Smith, Gordon Kotik, Stephen Westfold, *Research on Knowledge-Based Software Environments at Kestrel Institute*, pp1278-1295 in IEEE Trans. on Soft. Eng. SE-11(11), 1985.

[Standish et al 76] T.A. Standish, D.C. Harriman, D.F. Kibler, J.M. Neighbours, *The Irvine Program Transformation Catalogue*, University of California Technical Report, 1976.

[Steinbruggen 82] Ralf Steinbruggen, *Program Development Using Transformational Expressions*, Technische Universitat Munchen TUM-I8206, 1982.

[Williams 82] J.H. Williams, *On the Development of the Algebra of Functional Programs*, pp733-757 in TOPLAS 4(4), 1982.