January 1993

The Relatedness and Comparative Utility of Various Approaches to Operational Semantics

Raymond McDowell
University of Pennsylvania

The Relatedness and Comparative Utility of Various Approaches to Operational Semantics

Abstract

We examine three approaches to operational semantics: transition semantics, natural semantics, and reduction semantics. First we compare the style and expressive power of the three forms of semantics by using them to construct semantics for various language features. Program abortion, interleaving, and block structure particularly distinguish the three. Natural semantics was very good at specifying "large granularity" features such as blocks, but is apparently unable to capture interleaving because of its "small granularity". On the other hand, transition semantics and reduction semantics easily express "small granularity" features but have difficulty with "large granularity" features. Reduction semantics provide especially concise specifications of non-sequential control constructs such as abortion and interleaving. We also analyze the utility of the different approaches for two application areas: implementation correctness and type soundness. For these two applications, natural semantics seems to allow simpler proofs, although we do not generalize this conclusion to other areas.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-16.

The Relatedness and Comparative Utility of Various Approaches to Operational Semantics

MS-CIS-93-16
LINC LAB 246

Raymond C. McDowell

University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

February 1993

The Relatedness and Comparative Utility of Various Approaches to Operational Semantics [1]

Raymond C. McDowell

January 28, 1993

Abstract

We examine three approaches to operational semantics: transition semantics, natural semantics, and reduction semantics. First we compare the style and expressive power of the three forms of semantics by using them to construct semantics for various language features. Program abortion, interleaving, and block structure particularly distinguish the three. Natural semantics was very good at specifying "large granularity" features such as blocks, but is apparently unable to capture interleaving because of its "small granularity". On the other hand, transition semantics and reduction semantics easily express "small granularity" features but have difficulty with "large granularity" features. Reduction semantics provide especially concise specifications of non-sequential control constructs such as abortion and interleaving. We also analyze the utility of the different approaches for two application areas: implementation correctness and type soundness. For these two applications, natural semantics seems to allow simpler proofs, although we do not generalize this conclusion to other areas.

[1]This report is based on the Ph.D. Area Examination of the author.

Contents

List of Tables

Acknowledgements

I would like to thank Carl Gunter, the chair of my area examination committee, for suggesting the topic for my exam and for encouraging me to revise the resulting document for publication as a technical report. He and the other members of the committee, Dale Miller and Val Tannen, also provided a helpful critique of the original document. A special thanks also goes to Jon Riecke, who acted as a sounding board for ideas early on in the development of the paper, and to Myra VanInwegen, who provided a detailed review of the draft version of the technical report.

1 Introduction to Operational Semantics

The purpose of formal semantics is to rigorously specify the meaning of a program. In operational semantics, the meaning of a program is specified in terms of an execution model, that is, the semantics provide an abstract representation of how the program is executed on a machine. For applicative languages, the value resulting from a program's execution is often given as the meaning of the program. For imperative languages, the meaning is the changes made by the program to the machine state.

In this paper we will examine three approaches to operational semantics: transition semantics, natural semantics, and reduction semantics. This section will introduce the three approaches by presenting the semantics for a simple language using each approach. In the second section we extend the language (and the semantics) presented in this section to include additional features. This will allow us to compare the style and expressive power of the three forms of semantics. Then we will discuss applications of operational semantics in the third section, analyzing the utility of the different approaches for each task. Finally, we conclude with a general summary of our findings.

1.1 A Simple Imperative Language

The language we will use to illustrate the three operational semantics techniques is a simple imperative language called **While** [NN92]. The abstract syntax of the language is given by

$$s \ ::= \ x := e \mid \mathbf{skip} \mid s_1; s_2 \mid \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2 \mid \mathbf{while}\, e\, \mathbf{do}\, s$$

where x ranges over variables, e ranges over expressions, and s, s_1, and s_2 range over statements. To simplify matters, we will not concern ourselves with the syntax and semantics of expressions. We will assume integer and boolean expressions and a semantic function

$$\mathcal{E} : \mathbf{Exp} \to (\mathbf{State} \to \mathbf{Value})$$

that defines the meaning of an expression. We use **Exp**, **State**, and **Value** to represent the sets of all **While** expressions, states, and values, respectively. A state σ is a mapping of variables to values, so if

$$\sigma = \{x \mapsto 3, y \mapsto 2\}$$

then

$$\mathcal{E}[\![x + y]\!]\sigma = 5$$

We will also not concern ourselves with typing of expressions; we will assume that all expressions are well-typed.

1.2 Transition Semantics

The first form of operational semantics we will look at is transition semantics. It is also called structural operational semantics or Plotkin-style semantics because of Gordon Plotkin's seminal report about it [Plo81]. Transition semantics define the meaning of a program via a transition relation $\longrightarrow$ between machine configurations. A machine configuration is either

- a pair $\langle s, \sigma \rangle$, consisting of a statement s and a state σ, or
- a state σ.

The transition relation describes the individual steps in the evaluation of a program. If $\langle s, \sigma \rangle \longrightarrow \langle s', \sigma' \rangle$ this indicates that one step of the execution of statement s from the state σ results in the new state σ'. The statement s' represents the computation remaining to be done for the complete execution of s. A transition of the form $\langle s, \sigma \rangle \longrightarrow \sigma'$ represents the final step of execution. The transition relation is defined by a set of rules and axioms. A transition semantics for **While** is given in Table 1; this is a slightly modified version of the semantics given in [NN92].

Only one rule will apply to a given configuration, so the derivation sequence is uniquely determined by the initial statement and state. Thus we can express the meaning of a **While** program as a semantic function

$$\mathcal{S}_{\mathrm{ts}} : \mathbf{Stmt} \to (\mathbf{State} \to \mathbf{State})$$

where **Stmt** is the set of all **While** statements. We define

$$\mathcal{S}_{\mathrm{ts}}[\![s]\!]\sigma = \sigma' \text{ if } \langle s, \sigma \rangle \longrightarrow^{*} \sigma'$$

where $\longrightarrow^{*}$ is the reflexive-transitive closure of $\longrightarrow$. Note that the meaning of a program is a *partial* function from **State** to **State**. This is because a given program executing from an initial state may never reach a final state; it may terminate abnormally or never terminate.[2]

[2]These possibilities will be further discussed as we investigate extensions of the language **While**.

$\langle x := e, \sigma\rangle \longrightarrow \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]$		$[asg_{ts}]$
$\langle \mathbf{skip}, \sigma\rangle \longrightarrow \sigma$		$[skip_{ts}]$
$\dfrac{\langle s_1, \sigma\rangle \longrightarrow \langle s_1', \sigma'\rangle}{\langle s_1; s_2, \sigma\rangle \longrightarrow \langle s_1'; s_2, \sigma'\rangle}$		$[comp^1_{ts}]$
$\dfrac{\langle s_1, \sigma\rangle \longrightarrow \sigma'}{\langle s_1; s_2, \sigma\rangle \longrightarrow \langle s_2, \sigma'\rangle}$		$[comp^2_{ts}]$
$\langle \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \sigma\rangle \longrightarrow \langle s_1, \sigma\rangle$	if $\mathcal{E}[\![e]\!]\sigma = \text{true}$	$[if^{true}_{ts}]$
$\langle \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \sigma\rangle \longrightarrow \langle s_2, \sigma\rangle$	if $\mathcal{E}[\![e]\!]\sigma = \text{false}$	$[if^{false}_{ts}]$
$\langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \longrightarrow \langle s; \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle$	if $\mathcal{E}[\![e]\!]\sigma = \text{true}$	$[while^{true}_{ts}]$
$\langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \longrightarrow \sigma$	if $\mathcal{E}[\![e]\!]\sigma = \text{false}$	$[while^{false}_{ts}]$

Table 1: Transition Semantics for **While**

Note that the transition semantics gives us a "two-dimensional" structure from which the meaning of a program is derived [Plo81]. The "horizontal" dimension is the derivation sequence $\langle s, \sigma \rangle \longrightarrow^* \sigma'$; then for each step in the sequence there is a "vertical" proof of the validity of the transition. In a corresponding manner, proofs of program properties using transition semantics can also be "two-dimensional". The main structure of the proof will be done by induction on the length of the derivation sequence; each step in this induction may in turn involve a proof by induction on the height of the derivation tree for the corresponding transition. We shall see examples of this sort of proof shortly.

1.3 Natural Semantics

Like transition semantics, natural semantics[3] [Kah87] define the semantics of a program via a relation between machine configurations. However, natural semantics abstract away from the details of the computation, defining a relation between a configuration $\langle s, \sigma \rangle$ and the state σ' resulting from the *complete* execution of s from σ. Table 2 gives the natural semantics of **While** from [NN92].

Again we can define a semantic function

$$\mathcal{S}_{\text{ns}}[\![s]\!]\sigma = \sigma' \text{ if } \langle s, \sigma \rangle \Downarrow \sigma'$$

We can see from this that a natural semantics defines meaning in terms of a single derivation rather than a derivation sequence. Thus it does not have the two-dimensional structure that a transition semantics does; it has only the "vertical" dimension of the proof of the relation $\langle s, \sigma \rangle \Downarrow \sigma'$. Correspondingly, proofs of program properties using natural semantics are usually done by induction on the height of the derivation tree.

1.4 Reduction Semantics

Reduction semantics [FH89], like transition semantics, define a relation that describes incremental steps in the evaluation of a program. However, reduction semantics have a more complex system of specifying the relation than transition semantics. This form of semantics is based on term rewriting; the

[3] Just to make things really confusing, natural semantics are also sometimes referred to as structural operational semantics and Plotkin-style semantics, e.g. in [Tof90] and [WF91].

$\langle x := e, \sigma\rangle \Downarrow \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]$		$[asg_{ns}]$
$\langle \mathbf{skip}, \sigma\rangle \Downarrow \sigma$		$[skip_{ns}]$
$\dfrac{\langle s_1, \sigma\rangle \Downarrow \sigma_1 \quad \langle s_2, \sigma_1\rangle \Downarrow \sigma_2}{\langle s_1; s_2, \sigma\rangle \Downarrow \sigma_2}$		$[comp_{ns}]$
$\dfrac{\langle s_1, \sigma\rangle \Downarrow \sigma'}{\langle \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \sigma\rangle \Downarrow \sigma'}$	if $\mathcal{E}[\![e]\!]\sigma = \text{true}$	$[if_{ns}^{true}]$
$\dfrac{\langle s_2, \sigma\rangle \Downarrow \sigma'}{\langle \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \sigma\rangle \Downarrow \sigma'}$	if $\mathcal{E}[\![e]\!]\sigma = \text{false}$	$[if_{ns}^{false}]$
$\dfrac{\langle s, \sigma\rangle \Downarrow \sigma' \quad \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma'\rangle \Downarrow \sigma''}{\langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \Downarrow \sigma''}$	if $\mathcal{E}[\![e]\!]\sigma = \text{true}$	$[while_{ns}^{true}]$
$\langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \Downarrow \sigma$	if $\mathcal{E}[\![e]\!]\sigma = \text{false}$	$[while_{ns}^{false}]$

Table 2: Natural Semantics for **While**

core of a reduction semantics is a set of axioms defining a rewriting relation $\leadsto$ between configurations.

A set of evaluation contexts $E[\,]$ are then defined to describe the syntactic contexts in which the $\leadsto$ reductions can take place. In term rewriting systems, a term can be reduced if any subterm of the term matches the left-hand side of a rewrite rule. The evaluation contexts in a reduction semantics serve to restrict the subterms which are considered for matching against the rewrite rules.

Finally, another set of axioms is used to define a relation $\longmapsto$ in terms of the $\leadsto$ relation and the contexts $E[\,]$. It is this relation which defines the meaning of a program in reduction semantics.

A reduction semantics for **While** is given in Table 3.[4] Here the $\leadsto$ relation rewrites configurations to equivalent configurations. The evaluation contexts restrict this rewriting to the left-most statement; this specifies the left-to-right evaluation order we want for our semantics of **While**.

The semantic function for the reduction semantics is defined by

$$\mathcal{S}_{\mathrm{rs}}[\![s]\!]\sigma = \sigma' \text{ if } \langle s, \sigma \rangle \longmapsto^{*} \sigma'$$

where $\longmapsto^{*}$ is the reflexive-transitive closure of $\longmapsto$. It is not difficult to see that this is well-defined:

- If the program consists of a single **skip** statement, only Rule $[skip_{rs}]$ will apply. (The only $\leadsto$ axiom that applies to **skip** statements is Rule $[comp_{rs}]$, which requires that the skip be followed by another statement.)
- Otherwise, Rule $[context_{rs}]$ will apply. The $\leadsto$ axiom and evaluation context will then be uniquely determined by the left-most statement of the program.

The complex description of the $\longmapsto$ relation can give a complex structure to proofs of program properties using reduction semantics. However, each piece of the description is simple, so each piece of the proof can involve fewer cases. The outer level of the proof, as with transition semantics, will be by induction on the length of the derivation sequence. The induction steps of this proof, however, will use induction on the structure of the evaluation context $E[\,]$ used in the reduction step.

[4]It may seem from this example that the additional complexity of reduction semantics does not result in any gain over transition semantics - however, we shall see differently as we extend **While** with more advanced features.

$$\langle x := e, \sigma\rangle \rightsquigarrow \langle \mathbf{skip}, \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\rangle \qquad [asg_{rs}]$$

$$\langle \mathbf{skip}; s, \sigma\rangle \rightsquigarrow \langle s, \sigma\rangle \qquad [comp_{rs}]$$

$$\langle \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \sigma\rangle \rightsquigarrow \langle s_1, \sigma\rangle \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{true} \qquad [if_{rs}^{true}]$$

$$\langle \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \sigma\rangle \rightsquigarrow \langle s_2, \sigma\rangle \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{false} \qquad [if_{rs}^{false}]$$

$$\langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \rightsquigarrow \langle s; \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{true} \qquad [while_{rs}^{true}]$$

$$\langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \rightsquigarrow \langle \mathbf{skip}, \sigma\rangle \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{false} \qquad [while_{rs}^{false}]$$

$$E \ ::= \ [\,] \mid E; s$$

$$\langle E[s], \sigma\rangle \longmapsto \langle E[s'], \sigma'\rangle \quad \text{if } \langle s, \sigma\rangle \rightsquigarrow \langle s', \sigma'\rangle \qquad [context_{rs}]$$

$$\langle \mathbf{skip}, \sigma\rangle \longmapsto \sigma \qquad [skip_{rs}]$$

Table 3: Reduction Semantics for **While**

1.5 Equivalence Theorems

In this section we illustrate the proof techniques mentioned earlier by proving the three semantics of **While** to be equivalent. We do this by showing that both the natural semantics and the reduction semantics are equivalent to the transition semantics. But first we need to prove a useful lemma about the transition semantics for **While**:

Lemma 1 If $\langle s_1; s_2, \sigma\rangle \longrightarrow^k \sigma'$ then there exist a state σ'' and natural numbers k_1 and k_2 such that $\langle s_1, \sigma\rangle \longrightarrow^{k_1} \sigma''$ and $\langle s_2, \sigma''\rangle \longrightarrow^{k_2} \sigma'$ where $k_1 + k_2 = k$.

Proof: We prove this by induction on the length k of the derivation sequence $\langle s_1; s_2, \sigma\rangle \longrightarrow^k \sigma'$.

basis: If $k = 0$ then the result holds vacuously.

induction: We prove the result holds for $k > 0$, assuming it holds for all natural numbers less than k. If we consider the first transition step of the derivation sequence, the proof of the transition must end with either Rule $[comp^1_{ts}]$ or Rule $[comp^2_{ts}]$. We look at each case in turn:

Rule $[comp^1_{ts}]$:] We can write the derivation sequence as

$$\langle s_1; s_2, \sigma\rangle \longrightarrow \langle s_1'; s_2, \sigma_1\rangle \longrightarrow^{k-1} \sigma'$$

Applying the induction hypothesis to the latter part of the derivation sequence yields state σ'' and numbers k_1 and k_2 such that $\langle s_1', \sigma_1\rangle \longrightarrow^{k_1} \sigma''$ and $\langle s_2, \sigma''\rangle \longrightarrow^{k_2} \sigma'$ where $k_1 + k_2 = k - 1$. Putting this together with the premise of Rule $[comp^1_{ts}]$ we have our result:

$$\langle s_1, \sigma\rangle \longrightarrow \langle s_1', \sigma_1\rangle \longrightarrow^{k_1} \sigma''$$

and

$$\langle s_2, \sigma''\rangle \longrightarrow^{k_2} \sigma'$$

where $(k_1 + 1) + k_2 = k$.

Rule $[comp^2_{ts}]$: In this case the derivation sequence is of the form

$$\langle s_1; s_2, \sigma\rangle \longrightarrow \langle s_2, \sigma''\rangle \longrightarrow^{k-1} \sigma'$$

Thus our result is immediate; from the premise of the rule we have $\langle s_1, \sigma\rangle \longrightarrow \sigma''$ and from the latter part of the derivation sequence we have $\langle s_2, \sigma''\rangle \longrightarrow^{k-1} \sigma'$, where clearly $1 + (k - 1) = k$. □

Now we show the equivalence of the transition and natural semantics of **While**.

Theorem 2 For every statement s of **While**, $\mathcal{S}_{ts}[\![s]\!] = \mathcal{S}_{ns}[\![s]\!]$.

Proof: This is a modified presentation of the proof in [NN92]. We need to show that for every statement s and states σ and σ',

$$\langle s, \sigma \rangle \longrightarrow^* \sigma' \quad \Leftrightarrow \quad \langle s, \sigma \rangle \Downarrow \sigma'$$

($\Rightarrow$) We prove this direction by induction on the length k of the derivation sequence $\langle s, \sigma \rangle \longrightarrow^* \sigma'$.

basis: If $k = 0$ then the result holds vacuously.

induction: We proceed by induction on the height h of the derivation tree for the first transition in the derivation sequence.

basis: If $h = 0$ then the transition is derived from of one of the axioms $[asg_{ts}]$, $[skip_{ts}]$, $[if_{ts}^{true}]$, $[if_{ts}^{false}]$, $[while_{ts}^{true}]$, or $[while_{ts}^{false}]$. The proof for the axioms $[asg_{ts}]$, $[skip_{ts}]$, and $[while_{ts}^{false}]$ are immediate. We show the proof for Axiom $[if_{ts}^{true}]$ only.

Axiom $[if_{ts}^{true}]$**:** Assume that $\mathcal{E}[\![e]\!]\sigma = \text{true}$ and that

$$\langle \textbf{if}\, e\, \textbf{then}\, s_1\, \textbf{else}\, s_2, \sigma \rangle \longrightarrow \langle s_1, \sigma \rangle \longrightarrow^* \sigma'$$

The outer induction hypothesis applied to the derivation sequence $\langle s_1, \sigma \rangle \longrightarrow^* \sigma'$ yields $\langle s_1, \sigma \rangle \Downarrow \sigma'$. Thus

$$\langle \textbf{if}\, e\, \textbf{then}\, s_1\, \textbf{else}\, s_2, \sigma \rangle \Downarrow \sigma'$$

holds by Axiom $[if_{ns}^{true}]$.

induction: We now assume that $h > 0$ and consider the last rule in the derivation tree. We will show the proof for Rule $[comp_{ts}^2]$ only.

Rule $[comp_{ts}^2]$**:** If we apply Lemma 1 to

$$\langle s_1; s_2, \sigma \rangle \longrightarrow^k \sigma'$$

we get a state σ'' and derivation sequences $\langle s_1, \sigma \rangle \longrightarrow^{k_1} \sigma''$ and $\langle s_2, \sigma'' \rangle \longrightarrow^{k_2} \sigma'$ where $k_1 + k_2 = k$. The outer induction hypothesis applied to each of these smaller derivation sequences yields $\langle s_1, \sigma \rangle \Downarrow \sigma''$ and $\langle s_2, \sigma'' \rangle \Downarrow \sigma'$. Rule $[comp_{ns}]$ then gives us

$$\langle s_1; s_2, \sigma \rangle \Downarrow \sigma'.$$

($\Leftarrow$) We prove this direction by induction on the height h of the derivation tree for $\langle s, \sigma\rangle \Downarrow \sigma'$.

basis: If $h = 0$ the result is immediate.

induction: For $h > 0$ we consider the last rule in the derivation tree. We will show only the cases for the rules $[comp_{ns}]$ and $[if_{ns}^{true}]$.

Rule $[comp_{ns}]$**:** Assume that $\langle s_1; s_2, \sigma''\rangle \Downarrow \sigma'$. By applying the induction hypothesis to the premises $\langle s_1, \sigma\rangle \Downarrow \sigma''$ and $\langle s_2, \sigma''\rangle \Downarrow \sigma'$ we get $\langle s_1, \sigma\rangle \longrightarrow^* \sigma''$ and $\langle s_2, \sigma''\rangle \longrightarrow^* \sigma'$. Thus

$$\langle s_1; s_2, \sigma\rangle \longrightarrow^* \langle s_2, \sigma''\rangle \longrightarrow^* \sigma'$$

Rule $[if_{ns}^{true}]$**:** Assume that $\mathcal{E}[\![e]\!]\sigma = \text{true}$ and that

$$\langle \textbf{if}\, e\, \textbf{then}\, s_1\, \textbf{else}\, s_2, \sigma\rangle \Downarrow \sigma'$$

The induction hypothesis applied to the rule premise $\langle s_1, \sigma\rangle \Downarrow \sigma'$ yields $\langle s_1, \sigma\rangle \longrightarrow^* \sigma'$. By Rule $[if_{ts}^{true}]$ we have

$$\langle \textbf{if}\, e\, \textbf{then}\, s_1\, \textbf{else}\, s_2, \sigma\rangle \longrightarrow \langle s_1, \sigma\rangle$$

Thus

$$\langle \textbf{if}\, e\, \textbf{then}\, s_1\, \textbf{else}\, s_2, \sigma\rangle \longrightarrow \langle s_1, \sigma\rangle \longrightarrow^* \sigma'$$

□

Finally, we prove the equivalence of the transition and reduction semantics.

Theorem 3 For every statement s of **While**, $\mathcal{S}_{ts}[\![s]\!] = \mathcal{S}_{rs}[\![s]\!]$.

Proof: We need to show that for every statement s and states σ and σ',

$$\langle s, \sigma\rangle \longrightarrow^* \sigma' \quad \Leftrightarrow \quad \langle s, \sigma\rangle \longmapsto^* \sigma'$$

($\Rightarrow$) We prove this direction by induction on the length k of the derivation sequence $\langle s, \sigma\rangle \longrightarrow^* \sigma'$.

basis: If $k = 0$ then the result holds vacuously.

induction: If $k > 0$ then the derivation sequence contains at least one transition, i.e. it is of the form

$$\langle s, \sigma\rangle \longrightarrow \gamma \longrightarrow^* \sigma'$$

where γ is a configuration. We claim that the transition $\langle s, \sigma\rangle \longrightarrow \gamma$ has a corresponding derivation sequence $\langle s, \sigma\rangle \longmapsto^* \gamma$. This is all we need to prove, since then by the induction hypothesis on $\gamma \longrightarrow^* \sigma'$ we have $\gamma \longmapsto^* \sigma'$, so

$$\langle s, \sigma\rangle \longmapsto^* \gamma \longmapsto^* \sigma'$$

We prove the claim by induction on the height h of the derivation tree for the transition $\langle s, \sigma\rangle \longrightarrow \gamma$.

basis: If $h = 0$ then the transition is derived from of one of the axioms $[asg_{ts}]$, $[skip_{ts}]$, $[if_{ts}^{true}]$, $[if_{ts}^{false}]$, $[while_{ts}^{true}]$, or $[while_{ts}^{false}]$. The proof for the axioms $[skip_{ts}]$, $[if_{ts}^{true}]$, $[if_{ts}^{false}]$ and $[while_{ts}^{true}]$ are simple. We show the proof for Axiom $[asg_{ts}]$ only.

Axiom $[asg_{ts}]$**:** The transition is of the form

$$\langle x := e, \sigma\rangle \longrightarrow \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]$$

Thus Axiom $[context_{rs}]$ applies via Axiom $[asg_{rs}]$ with the empty context $[\,]$, giving

$$\langle x := e, \sigma\rangle \longmapsto \langle \mathbf{skip}, \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\rangle$$

Finally, using Axiom $[skip_{rs}]$ we obtain

$$\langle x := e, \sigma\rangle \longmapsto \langle \mathbf{skip}, \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\rangle \longmapsto \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]$$

induction: If $h > 0$ then the last rule of the proof is either Rule $[comp_{ts}^1]$ or Rule $[comp_{ts}^2]$. We show only the proof for Rule $[comp_{ts}^2]$.

Rule $[comp_{ts}^2]$**:** Assume the transition is of the form

$$\langle s_1; s_2, \sigma\rangle \longrightarrow \langle s_2, \sigma_1\rangle$$

Applying the inner induction hypothesis to the premise of the rule we obtain $\langle s_1, \sigma\rangle \longmapsto^* \sigma_1$. The last reduction in this sequence must be by Axiom $[skip_{rs}]$, so

$$\langle s_1, \sigma\rangle \longmapsto^* \langle \mathbf{skip}, \sigma_1\rangle$$

Every reduction in this sequence must be by Axiom $[context_{rs}]$; by expanding the context $E[\,]$ in each reduction to $E[\,]; s_2$ we obtain

$$\langle s_1; s_2, \sigma\rangle \longmapsto^* \langle \mathbf{skip}; s_2, \sigma_1\rangle$$

Since

$$\langle \mathbf{skip}; s_2, \sigma_1\rangle \longmapsto \langle s_2, \sigma_1\rangle$$

by Axiom $[context_{rs}]$ via Axiom $[comp_{rs}]$ and the empty context $[\,]$, we have

$$\langle s_1; s_2, \sigma\rangle \longmapsto^* \langle \mathbf{skip}; s_2, \sigma_1\rangle \longmapsto \langle s_2, \sigma_1\rangle$$

($\Leftarrow$) We prove this direction by induction on the length k of the derivation sequence $\langle s, \sigma\rangle \longmapsto^* \sigma'$.

basis: If $k = 0$ then the result holds vacuously.

induction: We proceed by case analysis of the rule used to obtain the first reduction in the derivation sequence.

Axiom $[skip_{rs}]$**:** Immediate.

Axiom $[context_{rs}]$**:** We claim that if $\langle s, \sigma\rangle \longmapsto \langle s_2, \sigma_2\rangle$ then either[5]

- $\langle s, \sigma\rangle \longrightarrow \langle s_2, \sigma_2\rangle$; or
- $\langle s_2, \sigma_2\rangle \longmapsto \langle s_3, \sigma_3\rangle$ by Rule $[context_{rs}]$ via Rule $[comp_{rs}]$ and $\langle s, \sigma\rangle \longrightarrow \langle s_3, \sigma_3\rangle$; or
- $s_2 = \mathbf{skip}$ and $\langle s, \sigma\rangle \longrightarrow \sigma_2$.

This is all we need to prove, since

- if $\langle s, \sigma\rangle \longrightarrow \langle s_2, \sigma_2\rangle$, then applying the induction hypothesis to $\langle s_2, \sigma_2\rangle \longmapsto^* \sigma'$ we get $\langle s_2, \sigma_2\rangle \longrightarrow^* \sigma'$;
- if $\langle s_2, \sigma_2\rangle \longmapsto \langle s_3, \sigma_3\rangle$ and $\langle s, \sigma\rangle \longrightarrow \langle s_3, \sigma_3\rangle$, then applying the induction hypothesis to $\langle s_3, \sigma_3\rangle \longmapsto^* \sigma'$ we get

  $$\langle s_3, \sigma_3\rangle \longrightarrow^* \sigma'$$

- if $s_2 = \mathbf{skip}$ and $\langle s, \sigma\rangle \longrightarrow \sigma_2$, then $\langle \mathbf{skip}, \sigma_2\rangle \longmapsto \sigma_2$ by Axiom $[skip_{rs}]$ so $\sigma_2 = \sigma'$ and we have $\langle s, \sigma\rangle \longrightarrow \sigma'$.

We prove our claim by induction on the structure of the context $E[\,]$ used in the reduction $\langle s, \sigma\rangle \longmapsto \langle s_2, \sigma_2\rangle$.

$E = [\,]$: Straightforward case analysis of the $\leadsto$ axioms.

$E = E'; s_1$: We know by Axiom $[context_{rs}]$ that there exist statements s' and s_2' such that $s = E[s']$, $s_2 = E[s_2']$, and

$$\langle s', \sigma\rangle \leadsto \langle s_2', \sigma_2\rangle$$

[5]Note that in essence we are creating a bisimulation between transition semantics configurations and reduction semantics configurations. This idea is discussed further in Section 3.1.

Thus $\langle E'[s'],\sigma\rangle \longmapsto \langle E'[s_2'],\sigma_2\rangle$, and by the induction hypothesis either

- $\langle E'[s'],\sigma\rangle \longrightarrow \langle E'[s_2'],\sigma_2\rangle$, in which case
  $$\langle E[s'],\sigma\rangle \longrightarrow \langle E[s_2'],\sigma_2\rangle$$
  by Rule $[comp^1_{ts}]$;
- $\langle E'[s_2'],\sigma_2\rangle \longmapsto \langle s_3',\sigma_3\rangle$ and $\langle E'[s'],\sigma\rangle \longrightarrow \langle s_3',\sigma_3\rangle$, in which case
  $$\langle E[s_2'],\sigma_2\rangle \longmapsto \langle s_3';s_1,\sigma_3\rangle$$
  and
  $$\langle E[s'],\sigma\rangle \longrightarrow \langle s_3';s_1,\sigma_3\rangle$$
- $E'[s_2'] =$ **skip** and $\langle E'[s'],\sigma\rangle \longrightarrow \sigma_2$, in which case
  $$\langle E[s'],\sigma\rangle \longmapsto \langle s_1,\sigma_2\rangle$$
  and
  $$\langle E[s_2'],\sigma\rangle \longrightarrow \langle s_1,\sigma_2\rangle$$

□

2 Expressiveness of Operational Semantics

In this section we explore the expressive power of the three forms of operational semantics. We do this by extending the language **While** introduced in the last section, and comparing the ease with which the different approaches to semantics capture the meaning of the new language constructs.

2.1 Abortion

First we add an abortion command to the language. This command stops the execution of the complete program. We will explore several alternatives of what this means, but first let us give the syntax for **While** with the new command.

$$\begin{array}{lll} s & ::= & x := e \mid \mathbf{skip} \mid s_1 ; s_2 \mid \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2 \mid \\ & & \mathbf{while}\, e\, \mathbf{do}\, s \mid \mathbf{abort} \end{array}$$

2.1.1 Sticky Semantics

The initial semantics we will consider for our extended language is given by the same rules as the semantics for **While** without the **abort** command. Thus in the transition and reduction semantics, **abort** results in a "stuck" configuration, since no rule applies. Similarly, in natural semantics, **abort** results in no relation holding. Hanne and Flemming Nielson conclude that **abort** is semantically equivalent to **while true do skip** in natural semantics, since both statements fail to converge to a value [NN92]. But for transition semantics they conclude that the two are distinguished: one results in a stuck configuration and the other in an infinite derivation. This distinction is accurate, but relies on their choice of definition for semantic equivalence.[6] They define two programs to be semantically equivalent with respect to a transition semantics if, when started in the same state, both terminate in the same state or both have infinite derivation sequences. For natural semantics, they define two programs to be semantically equivalent only if they both converge to the same final state. Note, however, that since we have not changed the semantics, the equivalence results from Section 1.5 still hold. So if we instead define two programs to be semantically equivalent with respect to a semantics if the semantic function returns the same value for both programs, none of the semantics distinguish looping from abortion.

[6]. . . and on their semantics, as we'll see shortly.

$$\langle \textbf{abort}, \sigma \rangle \longrightarrow \langle \sigma \rangle \qquad [abort_{ts}]$$

$$\frac{\langle s_1, \sigma \rangle \longrightarrow \langle \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \longrightarrow \langle \sigma' \rangle} \qquad [comp^3_{ts}]$$

Table 4: Extension to Transition Semantics for Abnormal Termination

$$\langle \textbf{abort}, \sigma \rangle \Downarrow \langle \sigma \rangle \qquad [abort_{ns}]$$

$$\frac{\langle s_1, \sigma \rangle \Downarrow \langle \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \Downarrow \langle \sigma' \rangle} \qquad [comp^2_{ns}]$$

Table 5: Extension to Natural Semantics for Abnormal Termination

The definition of semantic equivalence we choose to use will largely depend upon our focus. If we are primarily concerned with the specification of the meanings of programs,[7] the definition derived from the semantic function is appropriate. If we are interested in proving properties of a program based on the derivation of its meaning, then a definition based on the relation $\longrightarrow$, $\Downarrow$, or $\longmapsto$ is appropriate.

2.1.2 Abnormal Termination Semantics

With the semantics given in the last section, abortion and looping were indistinguishable at the level of the semantic function; the function only gives a result for normally terminating programs. However, it is possible for the semantic function to give values for all terminating programs by having the function return both a final state and a termination status.[8] To do this, of course, we must change our semantics.

Tables 4, 5, and 6 provide additional rules for the transition, natural, and

[7]This may seem like wanting a formal semantics just for its own right. But in fact we may want it to avoid ambiguity or lack of precision in a language specification, even if we do not intend to formally manipulate the semantics.

[8]To be fair to the Nielsons, they do qualify their conclusion with this possibility.

$$\langle E[\textbf{abort}], \sigma\rangle \longmapsto \langle\sigma\rangle \qquad [abort_{rs}]$$

Table 6: Extension to Reduction Semantics for Abnormal Termination

reduction semantics of **While** with abortion for this purpose. Basically, the **abort** command results in its state being tagged; commands occurring after an abortion have no effect. The semantic functions must now be changed to recognize the tagged state. The new semantic functions are defined by

$$\mathcal{S}_{\text{ts}}[\![s]\!]\sigma = \begin{cases} \langle\textbf{normal}, \sigma'\rangle & \text{if } \langle s, \sigma\rangle \longrightarrow^* \sigma' \\ \langle\textbf{abnormal}, \sigma'\rangle & \text{if } \langle s, \sigma\rangle \longrightarrow^* \langle\sigma'\rangle \end{cases}$$

$$\mathcal{S}_{\text{ns}}[\![s]\!]\sigma = \begin{cases} \langle\textbf{normal}, \sigma'\rangle & \text{if } \langle s, \sigma\rangle \Downarrow \sigma' \\ \langle\textbf{abnormal}, \sigma'\rangle & \text{if } \langle s, \sigma\rangle \Downarrow \langle\sigma'\rangle \end{cases}$$

$$\mathcal{S}_{\text{rs}}[\![s]\!]\sigma = \begin{cases} \langle\textbf{normal}, \sigma'\rangle & \text{if } \langle s, \sigma\rangle \longmapsto^* \sigma' \\ \langle\textbf{abnormal}, \sigma'\rangle & \text{if } \langle s, \sigma\rangle \longmapsto^* \langle\sigma'\rangle \end{cases}$$

2.1.3 Exit Semantics

Another way to view the **abort** command is as a way to specify normal termination from an arbitrary point in the program.[9] The rules specifying the transition semantics and natural semantics are the same for exit and abnormal termination semantics. The reduction semantics change slightly as shown in Table 7. The new semantic functions are defined by

$$\mathcal{S}_{\text{ts}}[\![s]\!]\sigma = \sigma' \;\text{ if } \langle s, \sigma\rangle \longrightarrow^* \sigma' \text{or } \langle s, \sigma\rangle \longrightarrow^* \langle\sigma'\rangle$$

$$\mathcal{S}_{\text{ns}}[\![s]\!]\sigma = \sigma' \;\text{ if } \langle s, \sigma\rangle \Downarrow \sigma' \text{or } \langle s, \sigma\rangle \Downarrow \langle\sigma'\rangle$$

$$\mathcal{S}_{\text{rs}}[\![s]\!]\sigma = \sigma' \;\text{ if } \langle s, \sigma\rangle \longmapsto^* \sigma'$$

Note that the transition and natural semantics must still distinguish between normal and "abnormal" termination in the rules in order to avoid the statements following the **abort** from having an effect on the state. These semantics continue to consider each statement after the **abort**, until the end

[9] **exit** would be a more usual name than **abort** for the command with this meaning.

$$\langle E[\mathbf{abort}], \sigma\rangle \longmapsto \sigma \qquad [exit_{rs}]$$

Table 7: Extension to Reduction Semantics for Exiting Termination

of the program is reached. In the reduction semantics this is not necessary because of its use of evaluation contexts rather than proof rules; in particular, a conclusion of an outer rule cannot be used as a premise. Thus at the outer level we can specify that the "execution" of the program should halt when an **abort** command is encountered; the semantic function is now the same as it was for the original transition semantics for **While**.

2.1.4 Conclusions

Since transition semantics and reduction semantics specify the "small steps" of execution with their relations, they allow the distinction between looping and abnormal termination to be reflected at the level of that relation. By abstracting away from the details of computation, natural semantics also abstracts away from this distinction, leaving it in the derivation tree. With any of these forms of semantics, abnormal termination may be encoded into the state to raise its visibility to the relation or semantic function level.

In this section we have also seen that the bilevel rules and evaluation contexts of reduction semantics seem to allow easier specification of aberrations from the normal sequential flow of control.

2.2 Non-Determinism

We now turn to another language construct: non-deterministic choice. The syntax for our language with this feature is given by

$$\begin{array}{lll} s & ::= & x := e \mid \mathbf{skip} \mid s_1; s_2 \mid \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2 \mid \\ & & \mathbf{while}\, e\, \mathbf{do}\, s \mid s_1\, \mathbf{or}\, s_2 \end{array}$$

The command s_1 **or** s_2 can result in the execution of s_1 or in the execution of s_2; either one is okay. The extensions to our semantics for this construct are given in tables 8, 9, and 10.[10] Note that non-determinism in the language

[10]The extensions to the transition and natural semantics for non-determinism are taken from [NN92].

$$\langle s_1\,\mathbf{or}\,s_2, \sigma\rangle \longrightarrow \langle s_1, \sigma\rangle \qquad [or^1_{ts}]$$

$$\langle s_1\,\mathbf{or}\,s_2, \sigma\rangle \longrightarrow \langle s_2, \sigma\rangle \qquad [or^2_{ts}]$$

Table 8: Extension to Transition Semantics for Non-Determinism

$$\frac{\langle s_1, \sigma\rangle \Downarrow \sigma'}{\langle s_1\,\mathbf{or}\,s_2, \sigma\rangle \Downarrow \sigma'} \qquad [or^1_{ns}]$$

$$\frac{\langle s_2, \sigma\rangle \Downarrow \sigma'}{\langle s_1\,\mathbf{or}\,s_2, \sigma\rangle \Downarrow \sigma'} \qquad [or^2_{ns}]$$

Table 9: Extension to Natural Semantics for Non-Determinism

results in non-determinism in the semantics. Whereas before the form of the program uniquely determined the derivation tree or sequence, now more than one rule may be applicable at any given time. Also note that it is now possible for a program to have different meanings depending on the choices made. Thus to keep our semantic function well-defined we must allow it to return a function mapping a state to a set of states:

$$\mathcal{S} : \mathbf{Stmt} \to (\mathbf{State} \to \mathcal{P}(\mathbf{State}))$$

where $\mathcal{P}(X)$ is the power-set of X. The new definitions for the semantic functions are

$$\mathcal{S}_{\mathrm{ts}}[\![s]\!]\sigma = \{\sigma' \mid \langle s, \sigma\rangle \longrightarrow^* \sigma'\}$$

$$\mathcal{S}_{\mathrm{ns}}[\![s]\!]\sigma = \{\sigma' \mid \langle s, \sigma\rangle \Downarrow \sigma'\}$$

$$\mathcal{S}_{\mathrm{rs}}[\![s]\!]\sigma = \{\sigma' \mid \langle s, \sigma\rangle \longmapsto^* \sigma'\}$$

Non-determinism also introduces the possibility of a choice between a terminating and non-terminating branch, e.g.

$$((x := 2; x := x + 2)\,\mathbf{or}\,(\mathbf{while}\,\mathbf{true}\,\mathbf{do}\,\mathbf{skip}))$$

$$\langle s_1\,\mathbf{or}\,s_2, \sigma\rangle \rightsquigarrow \langle s_1, \sigma\rangle \qquad [or^1_{rs}]$$

$$\langle s_1\,\mathbf{or}\,s_2, \sigma\rangle \rightsquigarrow \langle s_2, \sigma\rangle \qquad [or^2_{rs}]$$

Table 10: Extension to Reduction Semantics for Non-Determinism

The Nielsons observe that the natural semantics give only

$$\langle((x := 2; x := x + 2)\,\mathbf{or}\,(\mathbf{while\,true\,do\,skip})), \sigma\rangle \Downarrow \sigma[x \mapsto 4]$$

whereas the transition semantics (and reduction semantics) give two derivation sequences: one finite and the other infinite [NN92]. Thus they conclude that with natural semantics non-determinism suppresses looping, if possible, while transition semantics does not suppress looping.

This also is a superficial analysis[11]: in natural semantics looping is expressed as an infinite derivation tree – such a derivation tree is still possible for the given statement. In transition semantics (and reduction semantics) looping is expressed as an infinite derivation sequence, which we've already mentioned can occur with the above statement. The difference is the level at which the looping appears. For natural semantics, it is in the proofs of the $\Downarrow$ relation, so the relation doesn't hold for infinite loops.[12] Since looping appears in the derivation sequence for transition and reduction semantics, the $\longrightarrow$ or $\longmapsto$ relation emphatically holds in the case of an infinite loop – an infinite chain of derivations is formed. At the level of the semantic function, all three semantics are still equivalent; that is, given a statement and state, all three functions return the same (perhaps empty) set of states. Only choices resulting in termination are reflected in the result of the semantic function. In that sense, all three semantics suppress looping, and only statements that *always* loop return an empty set of states. In this case, whether our desire is to simply specify in a formal manner the meaning of non-deterministic programs, or we wish also to use the semantics to prove properties of a program in the language, all three semantic styles are equally useful in capturing non-determinism.

[11]Also at issue is whether the suppression of looping is a desirable characteristic. We may wish our semantics to reflect the possibility of looping.

[12]Note that this makes the $\Downarrow$ relation only semi-decidable, while the $\longrightarrow$ and $\longmapsto$ relations are decidable. Of course, $\longrightarrow^*$ and $\longmapsto^*$ are only semi-decidable ...

2.3 Parallelism

Our third extension to the language **While** is to introduce a command that allows the execution of (compound) statements to be interleaved. Our syntax is now

$$\begin{array}{lll} s & ::= & x := e \mid \mathbf{skip} \mid s_1; s_2 \mid \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2 \mid \\ & & \mathbf{while}\, e\, \mathbf{do}\, s \mid s_1\, \mathbf{par}\, s_2 \end{array}$$

Tables 11 and 12 extend the basic transition and reduction semantics for **While** to include this feature.[13] Note that the use of contexts allow a very concise specification of interleaving in the reduction semantics. The Nielsons give the following strawman extension for the natural semantics of **While**:

$$\frac{\langle s_1, \sigma \rangle \Downarrow \sigma_1 \quad \langle s_2, \sigma_1 \rangle \Downarrow \sigma_2}{\langle s_1\, \mathbf{par}\, s_2, \sigma \rangle \Downarrow \sigma_2}$$

$$\frac{\langle s_2, \sigma \rangle \Downarrow \sigma_2 \quad \langle s_1, \sigma_2 \rangle \Downarrow \sigma_1}{\langle s_1\, \mathbf{par}\, s_2, \sigma \rangle \Downarrow \sigma_1}$$

They then conclude:

> However, it is easy to see that this will not do because the rules only express that either s_1 is executed before s_2 or vice versa. This means that we have lost the ability to *interleave* the execution of two statements. Furthermore, it seems impossible to be able to express this in the natural semantics because we consider the execution of a statement as an atomic entity that cannot be split into smaller pieces.

Thus we have encountered a feature which one form of semantics is not simply less suited to express – rather the form of semantics is not even capable of expressing this feature. We explore this issue further in Appendix A through a sequence of additional attempts to capture parallelism with natural semantics.

2.4 Block Structure

The final construct we shall consider for our language is block structure. This feature allows variables with limited scope. The syntax for **While**

[13]The extension to the transition semantics for parallelism is taken from [NN92].

$$\frac{\langle s_1, \sigma\rangle \longrightarrow \langle s_1', \sigma'\rangle}{\langle s_1 \,\mathbf{par}\, s_2, \sigma\rangle \longrightarrow \langle s_1' \,\mathbf{par}\, s_2, \sigma'\rangle} \qquad [par_{ts}^1]$$

$$\frac{\langle s_1, \sigma\rangle \longrightarrow \sigma'}{\langle s_1 \,\mathbf{par}\, s_2, \sigma\rangle \longrightarrow \langle s_2, \sigma'\rangle} \qquad [par_{ts}^2]$$

$$\frac{\langle s_2, \sigma\rangle \longrightarrow \langle s_2', \sigma'\rangle}{\langle s_1 \,\mathbf{par}\, s_2, \sigma\rangle \longrightarrow \langle s_1 \,\mathbf{par}\, s_2', \sigma'\rangle} \qquad [par_{ts}^3]$$

$$\frac{\langle s_2, \sigma\rangle \longrightarrow \sigma'}{\langle s_1 \,\mathbf{par}\, s_2, \sigma\rangle \longrightarrow \langle s_1, \sigma'\rangle} \qquad [par_{ts}^4]$$

Table 11: Extension to Transition Semantics for Parallelism

$$\langle \mathbf{skip\,par\,skip}, \sigma\rangle \rightsquigarrow \langle \mathbf{skip}, \sigma\rangle \qquad [par_{rs}]$$

$$E \quad ::= \quad [\,] \mid E; s \mid E \,\mathbf{par}\, s \mid s \,\mathbf{par}\, E$$

Table 12: Extension to Reduction Semantics for Parallelism

with block structure is given by

$$\begin{array}{rcl} s & ::= & x := e \mid \textbf{skip} \mid s_1; s_2 \mid \textbf{if}\, e\, \textbf{then}\, s_1\, \textbf{else}\, s_2 \mid \\ & & \textbf{while}\, e\, \textbf{do}\, s \mid \textbf{begin}\, d\, s\, \textbf{end} \\ \\ d & ::= & \textbf{var}\, x := e \mid d_1; d_2 \end{array}$$

Each block begins with a sequence of variable declarations d. The scope of these variables is limited to the block itself; such a variable may only be referenced in the succeeding declarations and in the code s for the block. The body of the block may itself contain other blocks, allowing nested scopes.

An extension to the natural semantics to encorporate block structure is shown in Table 13.[14] A new relation $\Downarrow_d$ is introduced for the declarations; the relation $\langle d, \sigma \rangle \Downarrow_d \sigma'$ holds when σ' is the extension of σ to include the variables declared in d. Note the manipulation of the state in Rule $[block_{ns}]$ to limit the scope of the local variables.

The Nielsons state

> It is somewhat harder to specify a [transition] semantics for the extended language. One approach is to replace states with a structure that is similar to the run-time stacks used in the implementation of block structured languages. Another is to extend the statements with fragments of the state.

Indeed, blocks are more difficult to express in transition semantics and reduction semantics than in natural semantics. This is because natural semantics allow us to deal with the block and its effects as a whole. Since transition and reduction semantics focus on the incremental steps of execution, block entry and block exit are separate in the derivation sequence. Thus we need to somehow pass along the block's set of variables and state at block entry time so that when the end of the block is reached its local variables can be removed and any global variables shadowed by them can be restored.

Table 14 gives an extension to the transition semantics using states σ that map from locations to values and environments ρ that map from variables to locations.[15] This separates the issue of linking names to variables (since more than one variable can have the same name with block structure) from

[14]The extension to the natural semantics for block structure is a slightly modified version of that in [NN92].

[15]The extension to the transition semantics for block structure is patterned after the semantics in [Plo81].

$$\frac{\langle d, \sigma\rangle \Downarrow_d \sigma' \qquad \langle s, \sigma'\rangle \Downarrow \sigma''}{\langle \mathbf{begin}\, d\, s\, \mathbf{end}, \sigma\rangle \Downarrow \sigma''[\mathrm{DV}(d) \mapsto \sigma]} \qquad [block_{ns}]$$

$$\langle \mathbf{var}\, x := e, \sigma\rangle \Downarrow_d \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma] \qquad [decl^1_{ns}]$$

$$\frac{\langle d_1, \sigma\rangle \Downarrow_d \sigma_1 \quad \langle d_2, \sigma_1\rangle \Downarrow_d \sigma_2}{\langle d_1; d_2, \sigma\rangle \Downarrow_d \sigma_2} \qquad [decl^2_{ns}]$$

$$\mathrm{DV}(\mathbf{var}\, x := e) = \{x\}$$

$$\mathrm{DV}(d_1; d_2) = \mathrm{DV}(d_1) \cup \mathrm{DV}(d_2)$$

$$\sigma_1[X \mapsto \sigma_2](x) = \begin{cases} \sigma_2(x) & \text{if } x \in X \\ \sigma_1(x) & \text{if } x \notin X \end{cases}$$

Table 13: Extension to Natural Semantics for Block Structure

the issue of linking variables to values. Since each variable has its own location, an assignment to a local variable doesn't affect any global variables with the same name. Thus only the environment, which is concerned with the issue of linking names to variables, needs to be restored at block exit time. This is done by keeping the block's extension to the environment separate from the enclosing environment. The rules of Table 1 would also have to be modified to encorporate the environment; for example, Rule $[asg_{ts}]$ would become

$$\rho \vdash \langle x := e, \sigma\rangle \longrightarrow \sigma[\rho(x) \mapsto \mathcal{E}[\![e]\!](\sigma \circ \rho)]$$

Note that the rules use a modified syntax to treat environments as declarations. This extension to the declaration syntax is only for the use of the semantics; the user of the language doesn't see this change.

For the reduction semantics in Table 15 we followed the Nielson's suggestion of replacing states σ with stacks of states Σ. We use the notation $\sigma; \Sigma$ to represent the result of pushing state σ onto stack Σ. At block entry a new local state is pushed onto the stack for each local variable; at block exit these states are removed.[16]

It is clear that the natural semantics have an advantage over the transition and reduction semantics for capturing block structure. Its level of abstraction makes it easier to specify things at the block level. It should be noted, however, that techniques such as those used in the extension to the transition semantics here become necessary for natural semantics as well to model features such as procedures and static scoping.[17] Natural semantics still retains an expressive advantage, but the advantage becomes less than that shown here.

2.5 Expressivity Conclusions

Natural semantics distinguishes itself from transition and reduction semantics in its abstraction away from the detailed steps of computation. As we have seen, this is an advantage for specifying "large granularity" features such as blocks, but a disadvantage for "small granularity" features such as interleaving. Conversely, transition semantics and reduction semantics more easily express "small granularity" features.

[16]It is possible to collapse these states so that each block has a single state on the stack; this complicates the reduction semantics slightly.

[17]The interested reader is referred to [NN92] for a natural semantics of an extension of **While** with these features.

$$d \quad ::= \quad \mathbf{var}\, x := e \mid \rho \mid d_1; d_2$$

$$\frac{\rho \vdash \langle d, \sigma \rangle \longrightarrow_d \langle d', \sigma' \rangle}{\rho \vdash \langle \mathbf{begin}\, d\, s\, \mathbf{end}, \sigma \rangle \longrightarrow \langle \mathbf{begin}\, d'\, s\, \mathbf{end}, \sigma' \rangle} \qquad [block^1_{ts}]$$

$$\frac{\rho[\rho'] \vdash \langle s, \sigma \rangle \longrightarrow \langle s', \sigma' \rangle}{\rho \vdash \langle \mathbf{begin}\, \rho'\, s\, \mathbf{end}, \sigma \rangle \longrightarrow \langle \mathbf{begin}\, \rho'\, s'\, \mathbf{end}, \sigma' \rangle} \qquad [block^2_{ts}]$$

$$\frac{\rho[\rho'] \vdash \langle s, \sigma \rangle \longrightarrow \sigma'}{\rho \vdash \langle \mathbf{begin}\, \rho'\, s\, \mathbf{end}, \sigma \rangle \longrightarrow \sigma'} \qquad [block^3_{ts}]$$

$$\rho \vdash \langle \mathbf{var}\, x := e, \sigma \rangle \longrightarrow_d \langle \{x \mapsto \ell\}, \sigma[\ell \mapsto \mathcal{E}[\![e]\!](\sigma \circ \rho)] \rangle \quad \text{where } \ell \text{ is a new location} \quad [decl^1_{ts}]$$

$$\rho \vdash \langle \rho'; \rho'', \sigma \rangle \longrightarrow_d \langle \rho'[\rho''], \sigma \rangle \qquad [decl^2_{ts}]$$

$$\frac{\rho \vdash \langle d_1, \sigma \rangle \longrightarrow_d \langle d_1', \sigma' \rangle}{\rho \vdash \langle d_1; d_2, \sigma \rangle \longrightarrow_d \langle d_1'; d_2, \sigma' \rangle} \qquad [decl^3_{ts}]$$

$$\frac{\rho[\rho'] \vdash \langle d, \sigma \rangle \longrightarrow_d \langle d', \sigma' \rangle}{\rho \vdash \langle \rho'; d, \sigma \rangle \longrightarrow_d \langle \rho'; d', \sigma' \rangle} \qquad [decl^4_{ts}]$$

$$\rho[\rho'](x) = \begin{cases} \rho'(x) & \text{if } x \in \mathrm{dom}(\rho') \\ \rho(x) & \text{otherwise} \end{cases}$$

Table 14: Extension to Transition Semantics for Block Structure

$$d \quad ::= \quad \mathbf{var}\, x := e \mid \mathbf{novar} \mid d_1; d_2$$

$$\langle \mathbf{begin}\, d_1; d_2\, s\, \mathbf{end}, \Sigma\rangle \rightsquigarrow \langle \mathbf{begin}\, d_1\, \mathbf{begin}\, d_2\, s\, \mathbf{end}\, \mathbf{end}, \Sigma\rangle \qquad [block^1_{rs}]$$

$$\langle \mathbf{begin\, var}\, x := e\, s\, \mathbf{end}, \Sigma\rangle \rightsquigarrow \langle \mathbf{begin\, novar}\, s\, \mathbf{end}, \{x \mapsto \mathcal{E}[\![e]\!]\Sigma\}; \Sigma\rangle \quad [block^2_{rs}]$$

$$\langle \mathbf{begin\, novar\, skip\, end}, \sigma; \Sigma\rangle \rightsquigarrow \langle \mathbf{skip}, \Sigma\rangle \qquad [block^3_{rs}]$$

$$E \quad ::= \quad [\,] \mid \mathbf{begin\, novar}\, E\, \mathbf{end} \mid E; s$$

$$\Sigma \quad ::= \quad \sigma \mid \sigma; \Sigma$$

$$(\sigma;\Sigma)[x \mapsto v] = \begin{cases} \sigma[x \mapsto v]; \Sigma & \text{if } x \in \mathrm{dom}(\sigma) \\ \sigma; (\Sigma[x \mapsto v]) & \text{otherwise} \end{cases}$$

$$(\sigma;\Sigma)(x) = \begin{cases} \sigma(x) & \text{if } x \in \mathrm{dom}(\sigma) \\ \Sigma(x) & \text{otherwise} \end{cases}$$

Table 15: Extension to Reduction Semantics for Block Structure

Reduction semantics distinguishes itself from transition semantics (and natural semantics) in that its use of bilevel rules and evaluation contexts allow it to be more flexible and powerful. This especially shows in the specification of non-sequential control constructs such as abortion or interleaving. Reduction semantics have also been used for concise semantics of exceptions and continuations in ML [WF91]. On the other hand, the evaluation contexts and bilevel rules also give reduction semantics a subtlety that makes its semantic specifications more difficult to write and understand.

3 Applications of Operational Semantics

In this section we will explore two applications of operational semantics. Our intent is to compare the utility of the three approaches to operational semantics for these applications.

3.1 Implementation Correctness

The first application we will look at is the correctness verification of a language implementation. We will provide an implementation of **While** via a translation into the assembly language of an abstract machine. Then we shall proceed to prove that for any **While** program, the semantics of the assembly language translation is the same as that of the original program.

The abstract machine that we will use has configurations of the form $\langle C, E, S\rangle$, where C is code (a sequence of instructions), E is an evaluation stack (a list of values), and S is the machine's storage (a mapping from variables to values).[18] The machine's instructions are given by

$$\begin{array}{rcl} I & ::= & \mathbf{FETCH}{-}x \mid \mathbf{STORE}{-}x \mid \mathbf{NOOP} \mid \\ & & \mathbf{BRANCH}(C, C) \mid \mathbf{LOOP}(C, C) \end{array}$$

Additional instructions are necessary to compute expression values, but we shall not be concerned with that here. Sequences of instructions are built using the : operator:

$$C \quad ::= \quad \epsilon \mid I : C$$

and the :: operator is used to append two sequences:

$$\begin{array}{rcl} \epsilon :: C & = & C \\ (I : C_1) :: C_2 & = & I : (C_1 :: C_2) \end{array}$$

In order to accomplish our task of proving an implementation correct, we will need a formal semantics for this assembly language. Thus we define in Table 16 the $\rhd$ relation between abstract machine configurations. If $\gamma \rhd \gamma'$, then one step of execution transforms configuration γ into configuration γ'. Note that the machine halts with a configuration of the form $\langle \epsilon, E, S\rangle$. The semantic function for the machine is defined by

$$\mathcal{C}_{am} : \mathbf{Code} \rightarrow (\mathbf{State} \rightarrow \mathbf{State})$$

[18]The abstract machine we are using comes from [NN92].

$$\langle \mathbf{FETCH}{-}x : C,\ E,\ S\rangle \rhd \langle C,\ S(x) : E,\ S\rangle \qquad [fetch_{am}]$$

$$\langle \mathbf{STORE}{-}x : C,\ z : E,\ S\rangle \rhd \langle C,\ E,\ S[x \mapsto z]\rangle \qquad [store_{am}]$$

$$\langle \mathbf{NOOP} : C,\ E,\ S\rangle \rhd \langle C,\ E,\ S\rangle \qquad [noop_{am}]$$

$$\langle \mathbf{BRANCH}(C_1, C_2) : C,\ t : E,\ S\rangle \rhd \langle C_1 :: C,\ E,\ S\rangle \quad \text{if } t = \text{true} \qquad [branch_{am}^{true}]$$

$$\langle \mathbf{BRANCH}(C_1, C_2) : C,\ t : E,\ S\rangle \rhd \langle C_2 :: C,\ E,\ S\rangle \quad \text{if } t = \text{false} \qquad [branch_{am}^{false}]$$

$$\langle \mathbf{LOOP}(C_1, C_2) : C,\ E,\ S\rangle \rhd \langle C_1 :: (\mathbf{BRANCH}(C_2 :: (\mathbf{LOOP}(C_1, C_2) : \epsilon),\ \mathbf{NOOP}) : C),\ E,\ S\rangle \qquad [loop_{am}]$$

Table 16: Abstract Machine Semantics

$$\mathcal{C}_{am}[\![C]\!]S = S' \ \text{ if } \langle C, \epsilon, S\rangle \rhd^* \langle \epsilon, E, S'\rangle$$

where **Code** is the set of all sequences of machine instructions.

We now specify the translation from **While** to our machine language. We do this via a function

$$\mathcal{T}_S : \mathbf{Stmt} \to \mathbf{Code}$$

We shall assume the existence of a translation function for expressions:[19]

$$\mathcal{T}_E : \mathbf{Exp} \to \mathbf{Code}$$

The translation function for statements is given in Table 17.

We will now prove the correctness of this translation; that is, we will prove that for any **While** program, the semantics of the program is preserved by the translation. For convenience, we define the following semantic function

$$\mathcal{S}_{tr}[\![s]\!] = (\mathcal{C}_{am} \circ \mathcal{T}_S)[\![s]\!]$$

Since we have not specified the semantics of expressions in **While** nor their translation to the machine language, we shall assume the following lemma.[20]

[19]The curious reader is referred to [NN92] for a definition of this function.

[20]See [NN92] for a proof of this lemma.

$$\mathcal{T}_S[\![x := e]\!] = \mathcal{T}_E[\![e]\!] :: (\mathbf{STORE}\text{-}x : \epsilon) \tag{1}$$

$$\mathcal{T}_S[\![\mathbf{skip}]\!] = \mathbf{NOOP} : \epsilon \tag{2}$$

$$\mathcal{T}_S[\![s_1; s_2]\!] = \mathcal{T}_S[\![s_1]\!] :: \mathcal{T}_S[\![s_2]\!] \tag{3}$$

$$\begin{aligned}&\mathcal{T}_S[\![\mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2]\!]\\ &\qquad = \mathcal{T}_E[\![e]\!] :: (\mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon)\end{aligned} \tag{4}$$

$$\mathcal{T}_S[\![\mathbf{while}\, e\, \mathbf{do}\, s]\!] = \mathbf{LOOP}(\mathcal{T}_E[\![e]\!], \mathcal{T}_S[\![s]\!]) : \epsilon \tag{5}$$

Table 17: Translation of **While** Statements

Lemma 4 For every expression e in **While** and storage mapping S,

$$\langle \mathcal{T}_E[\![e]\!], \epsilon, S\rangle \rhd^* \langle \epsilon, \mathcal{E}[\![e]\!]S : \epsilon, S\rangle$$

□

We first prove the correctness of the implementation with respect to the natural semantics for **While.**

Theorem 5 For every statement s of **While**, $\mathcal{S}_{ns}[\![s]\!] = \mathcal{S}_{tr}[\![s]\!]$.

Proof: This is a modified presentation of the proof in [NN92]. We have to show that for all states σ and σ'

$$E = \epsilon \text{ and } \langle s, \sigma\rangle \Downarrow \sigma' \quad \Leftrightarrow \quad \langle \mathcal{T}_S[\![s]\!], \epsilon, \sigma\rangle \rhd^* \langle \epsilon, E, \sigma'\rangle$$

($\Rightarrow$) We prove this direction by induction on the height h of the derivation tree for $\langle s, \sigma\rangle \Downarrow \sigma'$.

basis: If $h = 0$ then the tree consists of one of the axioms $[asg_{ns}]$, $[skip_{ns}]$, or $[while_{ns}^{false}]$. We shall only show the proof for Axiom $[asg_{ns}]$

Axiom $[asg_{ns}]$**:** We are given $\langle x := e, \sigma\rangle \Downarrow \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]$. From (1) we have

$$\mathcal{T}_S[\![x := e]\!] = \mathcal{T}_E[\![e]\!] :: (\mathbf{STORE}\text{-}x : \epsilon)$$

In general, given $\langle C_1, E, S\rangle \rhd^* \langle C_1', E', S'\rangle$, we can conclude $\langle C_1 :: C_2, E, S\rangle \rhd^* \langle C_1' :: C_2, E', S'\rangle$ for any C_2. Thus, applying Lemma 4, we get

$$\begin{aligned}&\langle \mathcal{T}_E[\![e]\!] :: (\mathbf{STORE}{-}x : \epsilon),\ \epsilon,\ \sigma\rangle\\ &\qquad \rhd^* \langle \mathbf{STORE}{-}x : \epsilon,\ \mathcal{E}[\![e]\!]\sigma : \epsilon,\ \sigma\rangle\end{aligned}$$

We can then apply Axiom [$store_{am}$] to get

$$\begin{aligned}&\langle \mathcal{T}_E[\![e]\!] :: (\mathbf{STORE}{-}x : \epsilon),\ \epsilon,\ \sigma\rangle\\ &\qquad \rhd^* \langle \mathbf{STORE}{-}x : \epsilon,\ \mathcal{E}[\![e]\!]\sigma : \epsilon,\ \sigma\rangle\\ &\qquad \rhd \langle \epsilon,\ \epsilon,\ \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\rangle\end{aligned}$$

induction: For $h > 0$ we proceed by case analysis of the last rule in the derivation tree. We only show the proofs for the rules [$comp_{ns}$] and [if_{ns}^{true}].

Rule [$comp_{ns}$]**:** We are given

$$\langle s_1; s_2, \sigma\rangle \Downarrow \sigma'$$

From (3) we have $\mathcal{T}_S[\![s_1; s_2]\!] = \mathcal{T}_S[\![s_1]\!] :: \mathcal{T}_S[\![s_2]\!]$. The application of the induction hypothesis to the premises $\langle s_1, \sigma\rangle \Downarrow \sigma''$ and $\langle s_2, \sigma''\rangle \Downarrow \sigma'$ yields

$$\langle \mathcal{T}_S[\![s_1]\!], \epsilon, \sigma\rangle \ \rhd^* \langle \epsilon, \epsilon, \sigma''\rangle$$

and

$$\langle \mathcal{T}_S[\![s_2]\!], \epsilon, \sigma''\rangle \ \rhd^* \langle \epsilon, \epsilon, \sigma'\rangle$$

Thus

$$\begin{aligned}\langle \mathcal{T}_S[\![s_1]\!] :: \mathcal{T}_S[\![s_2]\!],\ \epsilon,\ \sigma\rangle &\rhd^* \langle \mathcal{T}_S[\![s_2]\!],\ \epsilon,\ \sigma''\rangle\\ &\rhd^* \langle \epsilon,\ \epsilon,\ \sigma'\rangle\end{aligned}$$

Rule [if_{ns}^{true}]**:** We are given

$$\langle \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \sigma\rangle \Downarrow \sigma'$$

and $\mathcal{E}[\![e]\!]\sigma = \text{true}$. Equation (4) gives us

$$\begin{aligned}&\mathcal{T}_S[\![\mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2]\!]\\ &\qquad = \mathcal{T}_E[\![e]\!] :: (\mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon)\end{aligned}$$

Applying Lemma 4 yields

$$\begin{aligned}&\langle \mathcal{T}_E[\![e]\!] :: (\mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon),\ \epsilon,\ \sigma\rangle \rhd^*\\ &\qquad \langle \mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon,\ \mathcal{E}[\![e]\!]\sigma : \epsilon,\ \sigma\rangle\end{aligned}$$

Since $\mathcal{E}[\![e]\!]\sigma = \text{true}$, Axiom $[branch_{am}^{true}]$ gives

$$\langle \mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon, \mathcal{E}[\![e]\!]\sigma : \epsilon, \sigma\rangle \rhd \langle \mathcal{T}_S[\![s_1]\!], \epsilon, \sigma\rangle$$

Now by applying the induction hypothesis to the premise $\langle s_1, \sigma\rangle \Downarrow \sigma'$ of Rule $[if_{ns}^{true}]$ we get

$$\langle \mathcal{T}_S[\![s_1]\!], \epsilon, \sigma\rangle \rhd^* \langle \epsilon, \epsilon, \sigma'\rangle$$

Thus putting it all together,

$$\begin{aligned}\langle \mathcal{T}_E[\![e]\!] :: (\mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon),\ \epsilon,\ \sigma\rangle \\ \rhd^* \langle \mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon,\ \mathcal{E}[\![e]\!]\sigma : \epsilon,\ \sigma\rangle \\ \rhd \langle \mathcal{T}_S[\![s_1]\!],\ \epsilon,\ \sigma\rangle \\ \rhd^* \langle \epsilon,\ \epsilon,\ \sigma'\rangle\end{aligned}$$

($\Leftarrow$) We prove this direction by induction on the length k of the computation sequence

$$\langle \mathcal{T}_S[\![s]\!], \epsilon, \sigma\rangle \rhd^* \langle \epsilon, E, \sigma'\rangle$$

basis: If $k = 0$ then the result holds vacuously.

induction: For $k > 0$ we proceed by case analysis of the statement s. We show the proof only for assignment, composition, and if statements.

Statement $x := e$**:** By (1) we have

$$\mathcal{T}_S[\![x := e]\!] = \mathcal{T}_E[\![e]\!] :: (\mathbf{STORE}\text{-}x : \epsilon)$$

We are given

$$\langle \mathcal{T}_E[\![e]\!] :: (\mathbf{STORE}\text{-}x : \epsilon),\ \epsilon,\ \sigma\rangle \rhd^k \langle \epsilon,\ E,\ \sigma'\rangle$$

We claim without proof that a result analogous to Lemma 1 holds for the abstract machine semantics. Thus there exists a machine configuration of the form $\langle \epsilon, E', \sigma''\rangle$ such that

$$\langle \mathcal{T}_E[\![e]\!], \epsilon, \sigma\rangle \rhd^{k_1} \langle \epsilon, E', \sigma''\rangle$$

and

$$\langle \mathbf{STORE}\text{-}x : \epsilon, E', \sigma''\rangle \rhd^{k_2} \langle \epsilon, E, \sigma'\rangle$$

where $k_1 + k_2 = k$. From Lemma 4 we get that $E' = \mathcal{E}[\![e]\!]\sigma : \epsilon$ and $\sigma'' = \sigma$. Then by Axiom $[store_{am}]$, $E = \epsilon$, and $\sigma' = \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]$. It now follows from Axiom $[asg_{ns}]$ that

$$\langle x := e, \sigma\rangle \Downarrow \sigma'$$

Statement $s_1; s_2$: We are given that

$$\langle \mathcal{T}_S[\![s_1]\!] :: \mathcal{T}_S[\![s_2]\!],\ \epsilon,\ \sigma\rangle \rhd^k \langle\epsilon,\ E,\ \sigma'\rangle$$

There must be a configuration $\langle\epsilon, E', \sigma''\rangle$ such that

$$\langle \mathcal{T}_S[\![s_1]\!], \epsilon, \sigma\rangle \rhd^{k_1} \langle\epsilon, E', \sigma''\rangle$$

and

$$\langle \mathcal{T}_S[\![s_2]\!], E', \sigma''\rangle \rhd^{k_2} \langle\epsilon, E, \sigma'\rangle$$

where $k_1 + k_2 = k$. The induction hypothesis applied to these computation sequences yields $E' = \epsilon$, $\langle s_1, \sigma\rangle \Downarrow \sigma''$, $E = \epsilon$, and $\langle s_2, \sigma''\rangle \Downarrow \sigma'$. Thus

$$\langle s_1; s_2, \sigma\rangle \Downarrow \sigma'$$

by Rule [$comp_{ns}$].

Statement $\textbf{if}\, e\, \textbf{then}\, s_1\, \textbf{else}\, s_2$: By (4)

$$\begin{aligned}&\mathcal{T}_S[\![\textbf{if}\, e\, \textbf{then}\, s_1\, \textbf{else}\, s_2]\!]\\ &\qquad = \mathcal{T}_E[\![e]\!] :: (\textbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon)\end{aligned}$$

We are given that

$$\langle \mathcal{T}_E[\![e]\!] :: (\textbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon),\ \epsilon,\ \sigma\rangle \rhd^k \langle\epsilon,\ E,\ \sigma'\rangle$$

Thus there must be a configuration of the form $\langle\epsilon, E', \sigma''\rangle$ such that

$$\langle \mathcal{T}_E[\![e]\!], \epsilon, \sigma\rangle \rhd^{k_1} \langle\epsilon, E', \sigma''\rangle$$

and

$$\langle \textbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon, E', \sigma''\rangle \rhd^{k_2} \langle\epsilon, E, \sigma'\rangle$$

where $k_1 + k_2 = k$. By Lemma 4, $E' = \mathcal{E}[\![e]\!]\sigma : \epsilon$ and $\sigma'' = \sigma$. If $\mathcal{E}[\![e]\!]\sigma = \text{true}$ then we have

$$\begin{aligned}&\langle \textbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon, \mathcal{E}[\![e]\!]\sigma : \epsilon, \sigma\rangle\\ &\qquad \rhd \langle \mathcal{T}_S[\![s_1]\!], \epsilon, \sigma\rangle\\ &\qquad \rhd^{k_2-1} \langle\epsilon, E, \sigma'\rangle\end{aligned}$$

The application of the induction hypothesis to the latter part of this computation sequence yields $E = \epsilon$ and $\langle s_1, \sigma\rangle \Downarrow \sigma'$. Thus Rule [$if^{true}_{ns}$] gives

$$\langle \textbf{if}\, e\, \textbf{then}\, s_1\, \textbf{else}\, s_2, \sigma\rangle \Downarrow \sigma'$$

The case for $\mathcal{E}[\![e]\!]\sigma = \text{false}$ is similar. □

Since we proved the three semantics equivalent in Section 1.5, we can now conclude that the implementation of **While** is correct with respect to all three semantics. However, since our purpose is to compare the utility of each approach to the task, we will also directly prove the correctness with respect to transition and reduction semantics.

Theorem 6 For every statement s of **While**, $\mathcal{S}_{ts}[\![s]\!] = \mathcal{S}_{tr}[\![s]\!]$.

Proof (sketch): We have to show that for all states σ and σ'

$$E = \epsilon \text{ and } \langle s, \sigma\rangle \longrightarrow^* \sigma' \quad \Leftrightarrow \quad \langle \mathcal{T}_S[\![s]\!], \epsilon, \sigma\rangle \rhd^* \langle \epsilon, E, \sigma'\rangle$$

We first define the following bisimulation relation between configurations of the transition semantics and those of the abstract machine semantics:

$$\langle s, \sigma\rangle \approx_{ts} \langle \mathcal{T}_S[\![s]\!], \epsilon, \sigma\rangle$$

$$\sigma \approx_{ts} \langle \epsilon, \epsilon, \sigma\rangle$$

($\Rightarrow$) We claim that if $\gamma_{ts} \approx_{ts} \gamma_{am}$ and $\gamma_{ts} \longrightarrow \gamma'_{ts}$ then there exists a configuration γ'_{am} such that $\gamma_{am} \rhd^+ \gamma'_{am}$ and $\gamma'_{ts} \approx_{ts} \gamma'_{am}$. By an induction on the length of the derivation sequence $\langle s, \sigma\rangle \longrightarrow^* \sigma'$, this claim extends to sequences of transition steps, completing the proof in this direction. We prove the claim by induction on the height h of the derivation tree of $\gamma_{ts} \longrightarrow \gamma'_{ts}$.

basis: If $h = 0$ then the tree consists of one of the axioms $[asg_{ts}]$, $[skip_{ts}]$, $[if_{ts}^{true}]$, $[if_{ts}^{false}]$, $[while_{ts}^{true}]$, or $[while_{ts}^{false}]$. We shall only show the proof for $[asg_{ts}]$ and $[if_{ts}^{true}]$.

Axiom $[asg_{ts}]$**:** We are given that

$$\langle x := e, \sigma\rangle \longrightarrow \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]$$

By (1) $\mathcal{T}_S[\![x := e]\!] = \mathcal{T}_E[\![e]\!] :: (\mathbf{STORE}{-}x : \epsilon)$. Lemma 4 gives us

$$\begin{aligned}&\langle \mathcal{T}_E[\![e]\!] :: (\mathbf{STORE}{-}x : \epsilon),\ \epsilon,\ \sigma\rangle \\ &\qquad \rhd^* \langle \mathbf{STORE}{-}x : \epsilon,\ \mathcal{E}[\![e]\!]\sigma : \epsilon,\ \sigma\rangle\end{aligned}$$

Finally

$$\langle \mathbf{STORE}{-}x : \epsilon, \mathcal{E}[\![e]\!]\sigma : \epsilon, \sigma\rangle \rhd \langle \epsilon, \epsilon, \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\rangle$$

by Axiom $[store_{am}]$.

Axiom $[if_{ts}^{true}]$**:** We are given that

$$\langle \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \sigma\rangle \longrightarrow \langle s_1, \sigma\rangle$$

and $\mathcal{E}[\![e]\!]\sigma = \text{true}$. By (4)

$$\mathcal{T}_S[\![\mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2]\!] \\ = \mathcal{T}_E[\![e]\!] :: (\mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon)$$

Lemma 4 gives us

$$\langle \mathcal{T}_E[\![e]\!] :: (\mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon),\ \epsilon,\ \sigma\rangle \\ \rhd^* \langle \mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon,\ \mathcal{E}[\![e]\!]\sigma : \epsilon,\ \sigma\rangle$$

Finally, since $\mathcal{E}[\![e]\!]\sigma = \text{true}$, Axiom $[branch_{am}^{true}]$ gives us

$$\langle \mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon, \mathcal{E}[\![e]\!]\sigma : \epsilon, \sigma\rangle \rhd \langle \mathcal{T}_S[\![s_1]\!], \epsilon, \sigma\rangle$$

induction: For $h > 0$ we proceed by case analysis of the last rule in the derivation tree. We only show the proof for Rule $[comp_{ts}^2]$.

Axiom $[comp_{ts}^2]$**:** We are given that

$$\langle s_1; s_2, \sigma\rangle \longrightarrow \langle s_2, \sigma'\rangle$$

By (3) $\mathcal{T}_S[\![s_1; s_2]\!] = \mathcal{T}_S[\![s_1]\!] :: \mathcal{T}_S[\![s_2]\!]$. Applying the induction hypothesis to the premise $\langle s_1, \sigma\rangle \longrightarrow \sigma'$ yields

$$\langle \mathcal{T}_S[\![s_1]\!] :: \mathcal{T}_S[\![s_2]\!],\ \epsilon,\ \sigma\rangle \rhd^* \langle \mathcal{T}_S[\![s_2]\!], \epsilon, \sigma'\rangle$$

($\Leftarrow$) Assume that $\gamma_{ts} \approx_{ts} \gamma_{am}^1$ and

$$\gamma_{am}^1 \rhd \gamma_{am}^2 \rhd \ldots \rhd \gamma_{am}^k$$

where $k > 1$ and only γ_{am}^1 and γ_{am}^k have empty evaluation stacks. We claim that there exists a configuration γ_{ts}' such that $\gamma_{ts} \longrightarrow \gamma_{ts}'$ and $\gamma_{ts}' \approx_{ts} \gamma_{am}^k$. Again this claim extends by induction to complete the proof in this direction. The claim is proven by case analysis of the statement in γ_{ts}. The proofs of each case are similar to those in the ($\Rightarrow$) direction, except we must also demonstrate that no intermediate γ_{as}^i has an empty stack. As long as expression evaluation doesn't leave an empty stack at intermediate stages, the proof is straightforward. □

The proof of correctness with respect to reduction semantics is similar in approach to that for transition semantics, but is much more complex. To establish a bisimulation relation between configurations of transition semantics and configurations of abstract machine semantics, we would need

$\langle x := e, \sigma\rangle \rightsquigarrow \langle \mathbf{noop}, \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\rangle$		$[asg_{rs}]$
$\langle \mathbf{skip}, \sigma\rangle \rightsquigarrow \langle \mathbf{noop}, \sigma\rangle$		$[skip_{rs}]$
$\langle \mathbf{noop}; s, \sigma\rangle \rightsquigarrow \langle s, \sigma\rangle$		$[comp_{rs}]$
$\langle \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \sigma\rangle \rightsquigarrow \langle s_1, \sigma\rangle$	if $\mathrm{E}[\![e]\!]\sigma = \mathrm{true}$	$[if_{rs}^{true}]$
$\langle \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \sigma\rangle \rightsquigarrow \langle s_2, \sigma\rangle$	if $\mathrm{E}[\![e]\!]\sigma = \mathrm{false}$	$[if_{rs}^{false}]$
$\langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \rightsquigarrow \langle s; \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle$	if $\mathrm{E}[\![e]\!]\sigma = \mathrm{true}$	$[while_{rs}^{true}]$
$\langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \rightsquigarrow \langle \mathbf{noop}, \sigma\rangle$	if $\mathrm{E}[\![e]\!]\sigma = \mathrm{false}$	$[while_{rs}^{false}]$

$$E \quad ::= \quad [\,] \mid E; s$$

$\langle E[s], \sigma\rangle \longmapsto \langle E[s'], \sigma'\rangle$	if $\langle s, \sigma\rangle \rightsquigarrow \langle s', \sigma'\rangle$	$[context_{rs}]$
$\langle \mathbf{noop}, \sigma\rangle \longmapsto \sigma$		$[noop_{rs}]$

Table 18: Modified Reduction Semantics for **While**

to distinguish between **skip** commands in the original **while** program and those introduced by the $\rightsquigarrow$ relation. To do this we could change all occurrences of **skip** in the $\rightsquigarrow$ and $\longmapsto$ axioms to a new statement (say **noop**) and add the axiom

$$\langle \mathbf{skip}, \sigma \rangle \rightsquigarrow \langle \mathbf{noop}, \sigma \rangle$$

The resulting semantics is shown in Table 18; the equivalence of this semantics with that of Table 3 is easily shown. We can now establish the bisimulation relation

$$\langle s, \sigma \rangle \approx_{rs} \langle \mathcal{T}_S[\![s]\!], \epsilon, \sigma \rangle \quad \text{if } s \neq E[\mathbf{noop}] \text{ for any context } E. \tag{6}$$

$$\sigma \approx_{rs} \langle \epsilon, \epsilon, \sigma \rangle \tag{7}$$

Note that, in contrast with the bisimulation for transition semantics, there are configurations on both sides that do not occur in the relation.

Theorem 7 For every statement s of **While**, $\mathcal{S}_{rs}[\![s]\!] = \mathcal{S}_{tr}[\![s]\!]$.

Proof (sketch): We want to show that for all states σ and σ'

$$\langle s, \sigma \rangle \longmapsto^* \sigma' \quad \Leftrightarrow \quad \langle \mathcal{T}_S[\![s]\!], \epsilon, \sigma \rangle \rhd^* \langle \epsilon, \epsilon, \sigma' \rangle$$

($\Rightarrow$) Assume that $\gamma_{rs}^1 \approx_{rs} \gamma_{am}$ and

$$\gamma_{rs}^1 \longmapsto \gamma_{rs}^2 \longmapsto \ldots \longmapsto \gamma_{rs}^k$$

where $k > 1$ and only γ_{rs}^1 and γ_{rs}^k contain statements that are **noop**-free. We claim that there exists a configuration γ'_{am} such that $\gamma_{am} \rhd^+ \gamma'_{am}$ and $\gamma_{rs}^k \approx_{rs} \gamma'_{am}$. This claim extends by induction on the length of the derivation sequence to complete the proof in this direction. Since γ_{rs}^1 is **noop**-free, the reduction $\gamma_{rs}^1 \longmapsto \gamma_{rs}^2$ cannot be by Axiom [$noop_{rs}$]. Thus we prove the claim by structural induction on the context E used in Axiom [$context_{rs}$] to derive this reduction.

basis: We are given that $E = [\,]$ and proceed by case analysis of the statement in γ_{rs}^1. We shall only show the proof for $x := e$ and **if** e **then** s_1 **else** s_2.

Statement $x := e$: By Axiom $[asg_{rs}]$

$$\langle x := e, \sigma\rangle \longmapsto \langle \mathbf{noop}, \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\rangle$$

Now Axiom $[noop_{rs}]$ applies to give

$$\langle \mathbf{noop}, \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\rangle \longmapsto \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]$$

Since $\sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]$ does not contain any occurrences of **noop**, it is γ_{rs}^k. By (1), $\mathcal{T}_S[\![x := e]\!] = \mathcal{T}_E[\![e]\!] :: (\mathbf{STORE}\!-\!x : \epsilon)$. Applying Lemma 4, we get

$$\begin{aligned}&\langle \mathcal{T}_E[\![e]\!] :: (\mathbf{STORE}\!-\!x : \epsilon),\ \epsilon,\ \sigma\rangle\\ &\qquad \rhd^* \langle \mathbf{STORE}\!-\!x : \epsilon,\ \mathcal{E}[\![e]\!]\sigma : \epsilon,\ \sigma\rangle\end{aligned}$$

We can then apply Axiom $[store_{am}]$ to get

$$\langle \mathbf{STORE}\!-\!x : \epsilon, \mathcal{E}[\![e]\!]\sigma : \epsilon, \sigma\rangle \rhd \langle \epsilon, \epsilon, \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\rangle$$

Since

$$\sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma] \approx_{rs} \langle \epsilon, \epsilon, \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\rangle$$

we are done.

Statement $\mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2$: We will assume $\mathcal{E}[\![e]\!]\sigma$ = true; the case for $\mathcal{E}[\![e]\!]\sigma$ = false is similar. By Axiom $[if_{rs}^{true}]$

$$\langle \mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2, \sigma\rangle \longmapsto \langle s_1, \sigma\rangle$$

Since $\mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2$ is **noop**-free, s_1 must be, so $\gamma_{rs}^k = \langle s_1, \sigma\rangle$. By (4),

$$\begin{aligned}&\mathcal{T}_S[\![\mathbf{if}\, e\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2]\!]\\ &\qquad = \mathcal{T}_E[\![e]\!] :: (\mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon)\end{aligned}$$

Applying Lemma 4 and then Axiom $[branch_{am}^{true}]$ we get

$$\begin{aligned}&\langle \mathcal{T}_E[\![e]\!] :: (\mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon),\ \epsilon,\ \sigma\rangle\\ &\qquad \rhd^* \langle \mathbf{BRANCH}(\mathcal{T}_S[\![s_1]\!], \mathcal{T}_S[\![s_2]\!]) : \epsilon,\ \mathcal{E}[\![e]\!]\sigma : \epsilon,\ \sigma\rangle\\ &\qquad \rhd \langle \mathcal{T}_S[\![s_1]\!],\ \epsilon,\ \sigma\rangle\end{aligned}$$

induction: The proof for context $E = E'; s$ follows directly from the induction hypothesis.

($\Leftarrow$) Assume that $\gamma_{rs} \approx_{rs} \gamma_{am}$ and

$$\gamma_{am}^1 \rhd \gamma_{am}^2 \rhd \cdots \rhd \gamma_{am}^k$$

where $k > 1$ and only γ_{am}^1 and γ_{am}^k have empty evaluation stacks. We claim that there exists a configuration γ'_{rs} such that $\gamma_{rs} \longmapsto^+ \gamma'_{rs}$ and $\gamma'_{rs} \approx_{rs} \gamma_{am}^k$. Again this claim extends by induction to complete the proof in this direction. The proof of this claim is similar to the analogous one for transition semantics. $\square$

The Nielsons comment

> ...[the bisimulation approach] relies on the two semantics proceeding in *lock-step*: that one is able to find configurations in the two derivation sequences that correspond to one another (as specified in the bisimulation relation). Often this is not possible and then one has to raise the level of abstraction for one of the semantics. This is exactly what happens when the [transition] semantics is replaced by the natural semantics: we do not care about the individual steps of the execution but only on the result.

The proof for reduction semantics showed that the lock-step can be somewhat flexible; in the worst-case the two semantics would only correlate at the beginning and end of the program. Of course, the proof complexity increases with the length of the derivation sequences between corresponding configurations, so the natural semantics does have an edge here.

Note that what we have really done is introduced yet another kind of operational semantics - abstract machine semantics - and proven an equivalence result for it, like those in Section 1.5. Often it is desirable to have several different semantics for a language, because each is better for certain purposes. For example, the abstract machine semantics would be useful in guiding or checking a language implementation on an actual machine. There are, of course, "non-operational" semantics as well, such as denotational semantics and axiomatic semantics. One would like to be certain that all the semantics being used for a given language are equivalent. While it is difficult to generalize based on the examples here, it appears that the natural semantics lead to simpler equivalence proofs than the other operational semantics.

3.2 Type Soundness

Type soundness is a relationship between a type system and a semantics for a given language. If the type system and semantics are sound and the type system associates a type τ with an expression, then the meaning of the expression according to the semantics will have type τ.

The type system of **While** is straightforward. Thus for this section we introduce an expression-based language **Exp**. The abstract syntax for the language is given by

$$e \;::=\; c \mid x \mid \mathbf{fn}\, x \Rightarrow e \mid e_1 e_2 \mid \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2$$

where c ranges over constants, x ranges over variables, and e, e_1, and e_2 range over expressions. Included among the constants are the operators :=, **ref**, and !. The reference operator **ref** applied to a value evaluates to a new location containing that value. The dereference operator ! applied to a location evaluates to the value in that location. The assignment operator := applied to a location and a value changes the location to contain the value.

The types of **Exp** are defined by

$$\tau \;::=\; \pi \mid \alpha \mid \tau_1 \to \tau_2 \mid \tau\,\mathrm{ref}$$

where π ranges over type constants (such as int, bool, etc.), α ranges over type variables, and τ, τ_1, and τ_2 range over types.

Type schemes capture the notion of polymorphic types, and are defined by

$$\psi \;::=\; \tau \mid \forall\alpha.\psi$$

We use FTV(ψ) to specify the free type variables of type scheme ψ. A substitution S is a map from type variables to types; we shall also apply substitutions to types, meaning by this that the substitution is applied to all type variables in the type. We say that a type scheme $\psi = \forall\alpha_1 \ldots \alpha_n.\tau$ generalizes a type τ' (written $\psi \succ \tau'$) if there exists a substitution S with domain $\{\alpha_1, \ldots, \alpha_n\}$ such that $S(\tau) = \tau'$.

Finally, we introduce type environments that map from variables to type schemes:

$$\Gamma : \mathbf{Var} \to \mathbf{TypeScheme}$$

to give type bindings to free variables. The type inference system in Table 19 gives type judgements of the form $\Gamma \rhd e : \tau$, indicating that expression e has type τ in type environment Γ.[21]

We must be careful how we infer types in a language such as **Exp** with both references and polymorphism. The problem is illustrated by the following example:[22]

$$\mathbf{let}\, x = \mathbf{ref}(\mathbf{fn}\, y \Rightarrow y)\,\mathbf{in}\,(\mathbf{fn}\, z \Rightarrow (!x)\mathbf{true})(:= x\,(\mathbf{fn}\, n \Rightarrow (+\,1\,n)))$$

[21]This type inference system is from [Tof90].

[22]This example is from [WF91].

$\Gamma \rhd x : \tau$	if $\Gamma(x) \succ \tau$	$[var_{type}]$
$\Gamma \rhd c : \tau$	if TypeOf$(c) \succ \tau$	$[const_{type}]$
$\dfrac{\Gamma[x \mapsto \tau_1] \rhd e : \tau_2}{\Gamma \rhd \mathbf{fn}\, x \mathrel{=>} e : \tau_1 \to \tau_2}$		$[fn_{type}]$
$\dfrac{\Gamma \rhd e_1 : \tau_1 \to \tau_2 \quad \Gamma \rhd e_2 : \tau_1}{\Gamma \rhd e_1 e_2 : \tau_2}$		$[appl_{type}]$
$\dfrac{\Gamma \rhd e_1 : \tau_1 \quad \Gamma[x \mapsto \text{Close}_\Gamma(\tau_1)] \rhd e_2 : \tau_2}{\Gamma \rhd \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \tau_2}$	if e_1 is non-expansive	$[let^{non}_{type}]$
$\dfrac{\Gamma \rhd e_1 : \tau_1 \quad \Gamma[x \mapsto \text{AppClose}_\Gamma(\tau_1)] \rhd e_2 : \tau_2}{\Gamma \rhd \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \tau_2}$	if e_1 is expansive	$[let^{exp}_{type}]$

$$\text{Close}_\Gamma(\tau) = \forall\alpha_1 \ldots \alpha_n.\tau \quad \text{where } \{\alpha_1, \ldots, \alpha_n\} = \text{FTV}(\tau) \setminus \text{FTV}(\Gamma)$$

$$\text{AppClose}_\Gamma(\tau) = \forall\alpha_1 \ldots \alpha_n.\tau \qquad \text{where } \{\alpha_1, \ldots, \alpha_n\} = (\text{FTV}(\tau) \setminus \text{FTV}(\Gamma)) \cap \mathbf{AppTypeVar}$$

Table 19: Type Inference System for **Exp**

If the type of x is generalized to $\forall\alpha.(\alpha \to \alpha)\,\mathrm{ref}$, then x will be treated as having type $(\mathbf{int} \to \mathbf{int})\,\mathrm{ref}$ during the assignment and type $(\mathbf{bool} \to \mathbf{bool})\,\mathrm{ref}$ during the dereference. The result is that the above expression is well-typed, even though it evaluates to $(+\,1\,\mathbf{true})$.

To avoid this we ensure that stored values are not used polymorphically.[23] Thus we divide our type variables into two disjoint sets: applicative type variables $\{t, t_1, \ldots\}$ and imperative type variables $\{u, u_1, \ldots\}$. Imperative type variables will range over types of values that are (perhaps) stored in one of the locations; applicative type variables will range only over types of values that we are certain are not stored in any location. We will use the term imperative type to describe a type containing no applicative type variables. Thus for a value to be stored, it must have an imperative type. We also require substitutions to map imperative type variables to imperative types.

Similarly, we distinguish between expressions whose evaluation might introduce new locations, which we call expansive, and expressions which definitely do not, which we call non-expansive. For our purposes it suffices to consider function applications and let expressions to be expansive and all other expressions (viz., all values) to be non-expansive. The type inference rules for **let** treat non-expansive and expansive expressions differently. The types of non-expansive expressions are fully generalized (Rule $[let^{non}_{type}]$), while the types of expansive expressions are only generalized over applicative type variables (Rule $[let^{exp}_{type}]$). This latter restriction prevents the types of any new locations introduced in the expansive expression from being generalized.

The types of the language constants are give by a function

$$\mathrm{TypeOf} : \mathbf{Const} \to \mathbf{TypeScheme}$$

where

$$\mathrm{TypeOf}(\mathbf{ref}) = \forall u.u \to u\,\mathrm{ref} \tag{8}$$

$$\mathrm{TypeOf}(!) = \forall t.t\,\mathrm{ref} \to t \tag{9}$$

$$\mathrm{TypeOf}(:=) = \forall t.t\,\mathrm{ref} \to t \to t \tag{10}$$

Note that (8) ensures that every value stored has an imperative type.

We also extend the type inference system to apply to configurations with the following definition:

[23]See [Tof90] and [WF91] for further information and references on this problem and its solution.

Definition 8 $\Gamma \rhd \langle e, \sigma \rangle : \tau$ if the type inference system extended with the axioms $\Gamma \rhd \ell_1 : u_1 \,\mathrm{ref}, \ldots, \Gamma \rhd \ell_n : u_n \,\mathrm{ref}$, where $\{\ell_1, \ldots, \ell_n\}$ is the domain of σ and $u_1, \ldots, u_n$ are new imperative type variables, proves $\Gamma \rhd e : \tau$ and $\Gamma \rhd \sigma(\ell_i) : u_i$, for $1 \leq i \leq n$.

In expression based languages, the meaning of a program is defined in terms of its evaluation as well as the changes it makes to the machine state. Thus our semantic functions will have type

$$\mathcal{E} : \mathbf{Exp} \rightarrow (\mathbf{State} \rightarrow \mathbf{Value} \times \mathbf{State})$$

Mads Tofte gave a natural semantics for **Exp** using environments such as those introduced in Section 2.4 and proved the soundness of the type inference system of Table 19 with respect to those semantics [Tof90]. His proof is quite complex, however, and required the invention of a new proof technique.

Andrew Wright and Matthias Felleisen later presented a much simpler type soundness proof for a reduction semantics which uses value substitution rather than environments [WF91]. Following this lead, Tables 20, 21, and 22, contain transition, natural, and reduction semantics for **Exp** using value substitution. The corresponding semantic functions are

$$\mathcal{E}_{\mathbf{ts}}[\![e]\!]\sigma = \langle v, \sigma' \rangle \ \text{ if } \langle e, \sigma \rangle \longrightarrow^* \langle v, \sigma' \rangle$$

$$\mathcal{E}_{\mathbf{ns}}[\![e]\!]\sigma = \langle v, \sigma' \rangle \ \text{ if } \langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$$

$$\mathcal{E}_{\mathbf{rs}}[\![e]\!]\sigma = \langle v, \sigma' \rangle \ \text{ if } \langle e, \sigma \rangle \longmapsto^* \langle v, \sigma' \rangle$$

We now proceed to prove type soundness with respect to each of these semantics. We will need the following lemma about value substitution and types.[24]

Lemma 9 If $\Gamma[x \mapsto \forall \alpha_1 \ldots \alpha_n.\tau] \rhd e : \tau'$, $\{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\Gamma) = \emptyset$, and $\Gamma \rhd v : \tau$, then $\Gamma \rhd e[v/x] : \tau'$. □

We first prove type soundness for the transition semantics:

Theorem 10 If $\rhd \langle e, \sigma \rangle : \tau$ and $\langle e, \sigma \rangle \longrightarrow^* \langle v, \sigma' \rangle$, then $\rhd \langle v, \sigma' \rangle : \tau$.

[24]Since this lemma is used in all three of the soundness proofs, its proof is not relevant to our comparison. The interested reader is referred to [WF91] for a proof.

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e_1', \sigma' \rangle}{\langle e_1 e_2, \sigma \rangle \longrightarrow \langle e_1' e_2, \sigma' \rangle} \qquad [app^1_{ts}]$$

$$\frac{\langle e_2, \sigma \rangle \longrightarrow \langle e_2', \sigma' \rangle}{\langle v_1 e_2, \sigma \rangle \longrightarrow \langle v_1 e_2, \sigma' \rangle} \qquad [app^2_{ts}]$$

$$\langle (\mathbf{fn}\, x \Rightarrow e) v, \sigma \rangle \longrightarrow \langle e[v/x], \sigma \rangle \qquad [fn_{ts}]$$

$$\langle \mathbf{ref}\, v, \sigma \rangle \longrightarrow \langle \ell, \sigma[\ell \mapsto v] \rangle \qquad \text{where } \ell \text{ is a new location} \quad [ref_{ts}]$$

$$\langle !\ell, \sigma \rangle \longrightarrow \langle \sigma(\ell), \sigma \rangle \qquad [deref_{ts}]$$

$$\langle (:= \ell_1) v_2, \sigma \rangle \longrightarrow \langle v_2, \sigma[\ell_1 \mapsto v_2] \rangle \qquad [asg_{ts}]$$

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e_1', \sigma' \rangle}{\langle \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2, \sigma \rangle \longrightarrow \langle \mathbf{let}\, x = e_1'\, \mathbf{in}\, e_2, \sigma' \rangle} \qquad [let^1_{ts}]$$

$$\langle \mathbf{let}\, x = v_1\, \mathbf{in}\, e_2, \sigma \rangle \longrightarrow \langle e_2[v_1/x], \sigma \rangle \qquad [let^2_{ts}]$$

$$v \;::=\; c \mid \ell \mid \mathbf{fn}\, x \Rightarrow e \mid\; := \ell$$

Table 20: Transition Semantics for **Exp**

$$\langle v, \sigma \rangle \Downarrow \langle v, \sigma \rangle \qquad [val_{ns}]$$

$$\frac{\begin{array}{c}\langle e_1, \sigma \rangle \Downarrow \langle \mathbf{fn}\, x' \texttt{=>}\, e', \sigma_1 \rangle \quad \langle e_2, \sigma_1 \rangle \Downarrow \langle v_2, \sigma_2 \rangle \\ \langle e'[v_2/x'], \sigma_2 \rangle \Downarrow \langle v', \sigma' \rangle\end{array}}{\langle e_1 e_2, \sigma \rangle \Downarrow \langle v', \sigma' \rangle} \qquad [fn_{ns}]$$

$$\frac{\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \mathbf{ref}\, e, \sigma \rangle \Downarrow \langle \ell, \sigma'[\ell \mapsto v] \rangle} \quad \text{where } \ell \text{ is a new location} \qquad [ref_{ns}]$$

$$\frac{\langle e, \sigma \rangle \Downarrow \langle \ell, \sigma' \rangle}{\langle !e, \sigma \rangle \Downarrow \langle \sigma'(\ell), \sigma' \rangle} \qquad [deref_{ns}]$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow \langle \ell_1, \sigma_1 \rangle \quad \langle e_2, \sigma_1 \rangle \Downarrow \langle v_2, \sigma_2 \rangle}{\langle (:= e_1) e_2, \sigma \rangle \Downarrow \langle v_2, \sigma_2[\ell_1 \mapsto v_2] \rangle} \qquad [asg_{ns}]$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow \langle v_1, \sigma_1 \rangle \quad \langle e_2[v_1/x], \sigma_1 \rangle \Downarrow \langle v_2, \sigma_2 \rangle}{\langle \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2, \sigma \rangle \Downarrow \langle v_2, \sigma_2 \rangle} \qquad [let_{ns}]$$

$$v \ ::= \ c \mid \ell \mid \mathbf{fn}\, x \texttt{=>}\, e \mid \ := \ell$$

Table 21: Natural Semantics for **Exp**

$$\langle (\mathbf{fn}\, x \texttt{=>} e)v, \sigma\rangle \rightsquigarrow \langle e[v/x], \sigma\rangle \qquad [fn_{rs}]$$

$$\langle \mathbf{ref}\, v, \sigma\rangle \rightsquigarrow \langle \ell, \sigma[\ell \mapsto v]\rangle \quad \text{where } \ell \text{ is a new location} \qquad [ref_{rs}]$$

$$\langle !\ell, \sigma\rangle \rightsquigarrow \langle \sigma(\ell), \sigma\rangle \qquad [deref_{rs}]$$

$$\langle (:= \ell)v, \sigma\rangle \rightsquigarrow \langle v, \sigma[\ell \mapsto v]\rangle \qquad [asg_{rs}]$$

$$\langle \mathbf{let}\, x = v\, \mathbf{in}\, e, \sigma\rangle \rightsquigarrow \langle e[v/x], \sigma\rangle \qquad [let_{rs}]$$

$$E \ ::= \ [\,] \mid Ee \mid vE \mid \mathbf{let}\, x = E\, \mathbf{in}\, e$$

$$\langle E[e], \sigma\rangle \longmapsto \langle E[e'], \sigma'\rangle \quad \text{if } \langle e, \sigma\rangle \rightsquigarrow \langle e', \sigma'\rangle \qquad [context_{rs}]$$

$$v \ ::= \ c \mid \ell \mid \mathbf{fn}\, x \texttt{=>} e \mid \ := \ell$$

Table 22: Reduction Semantics for **Exp**

Proof: This theorem follows from Lemma 11 by a simple induction on the length of the derivation sequence $\langle e, \sigma \rangle \longrightarrow^* \langle v, \sigma' \rangle$. □

Lemma 11 If $\Gamma \rhd \langle e_a, \sigma \rangle : \tau_a$ and $\langle e_a, \sigma \rangle \longrightarrow \langle e'_a, \sigma' \rangle$, then $\Gamma \rhd \langle e'_a, \sigma' \rangle : \tau_a$. If, furthermore, $\Gamma \rhd \langle e_b, \sigma \rangle : \tau_b$, then $\Gamma \rhd \langle e_b, \sigma' \rangle : \tau_b$.

Proof: By Definition 8, the type inference system extended by

$$\Gamma \rhd \ell_i : u_i \,\text{ref} \tag{11}$$

for all $\ell_i \in \text{dom}(\sigma)$ proves $\Gamma \rhd e_a : \tau_a$ and

$$\Gamma \rhd \sigma(\ell_i) : u_i \tag{12}$$

for all $\ell_i \in \text{dom}(\sigma)$. We claim that if we further extend the type inference system with an axiom of the form of (11) for each $\ell_i \in \text{dom}(\sigma') \setminus \text{dom}(\sigma)$, then $\Gamma \rhd e'_a : \tau_a$ and $\Gamma \rhd \sigma'(\ell_i) : u_i$ are provable in this system, for all $\ell_i \in \text{dom}(\sigma')$. Once we have proved this claim, it only remains to show that if $\Gamma \rhd e_b : \tau_b$ is provable in the type inference system as extended for σ, then it is also provable using the system as further extended for σ'. But since the type inference system as extended for σ' includes all rules in the system as extended for σ, this is immediate.

We prove the above claim by induction on the height of the proof of $\langle e_a, \sigma \rangle \longrightarrow \langle e'_a, \sigma' \rangle$. We perform a case analysis of the last rule in the proof. We show only some of the cases; the remainder are similar and no more difficult.

Rule $[asg_{ts}]$: We are given $\Gamma \rhd (:= \ell_1)v_2 : \tau_a$, so by the structure of the typing rules

$$\Gamma \rhd \ell_1 : \tau_a \,\text{ref} \tag{13}$$

and $\Gamma \rhd v_2 : \tau_a$. Since e'_a is v_2, it only remains to show

$$\Gamma \rhd \sigma[\ell_1 \mapsto v_2](\ell_i) : u_i$$

for all $\ell_i \in \text{dom}(\sigma)$. Since $\ell_1 \in \text{dom}(\sigma)$, this follows from (12) as long as u_1 is τ_a. But this is true by the structure of the typing rules, since (13) holds.

Rule $[ref_{ts}]$: We are given $\Gamma \rhd \mathbf{ref}\, v : \tau_a$, so by the structure of the typing rules, τ_a is of the form $u\,\mathrm{ref}$, and $\Gamma \rhd v : u$. If we further extend the type inference system with the axiom $\Gamma \rhd \ell : u\,\mathrm{ref}$, then, since e'_a is ℓ, it only remains to show

$$\Gamma \rhd \sigma[\ell \mapsto v](\ell_i) : u_i$$

for all $\ell_i \in \mathrm{dom}(\sigma[\ell \mapsto v])$. Since ℓ is new, this follows for all $\ell_i \in \mathrm{dom}(\sigma)$ from (12), and since $\Gamma \rhd v : u$, $\Gamma \rhd \sigma[\ell \mapsto v](\ell) : u$.

Rule $[let^1_{ts}]$: We are given $\Gamma \rhd \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2, : \tau_a$, so by the structure of the typing rules $\Gamma \rhd e_1 : \tau'$ and either

- e_1 is non-expansive and $\Gamma[x \mapsto \mathrm{Close}_\Gamma(\tau')] \rhd e_2 : \tau_a$; or
- e_1 is expansive and $\Gamma[x \mapsto \mathrm{AppClose}_\Gamma(\tau')] \rhd e_2 : \tau_a$.

By the premise $\langle e_1, \sigma\rangle \longrightarrow \langle e'_1, \sigma'\rangle$ of Rule $[let^1_{ts}]$ we know that e_1 is not a value, and so is expansive; thus we do not need to consider the non-expansive case any further. Thus, by the induction hypothesis on the premise, $\Gamma \rhd e'_1 : \tau'$, $\Gamma \rhd \sigma'(\ell_i) : u_i$, and

$$\Gamma[x \mapsto \mathrm{AppClose}_\Gamma(\tau')] \rhd e_2 : \tau_a$$

are provable, for all $\ell_i \in \mathrm{dom}(\sigma')$, in the type inference system as extended for σ'. This latter judgement implies

$$\Gamma[x \mapsto \mathrm{Close}_\Gamma(\tau')] \rhd e_2 : \tau_a$$

Therefore, whether e'_1 is expansive or non-expansive,

$$\Gamma \rhd \mathbf{let}\, x = e'_1 \,\mathbf{in}\, e_2 : \tau_a$$

by either Rule $[let^{non}_{type}]$ or Rule $[let^{exp}_{type}]$.

Rule $[let^2_{ts}]$: We are given $\Gamma \rhd \mathbf{let}\, x = v_1 \,\mathbf{in}\, e_2 : \tau_a$, so by the structure of the typing rules $\Gamma \rhd v_1 : \tau'$ and either

- v_1 is non-expansive and $\Gamma[x \mapsto \mathrm{Close}_\Gamma(\tau')] \rhd e_2 : \tau_a$; or
- v_1 is expansive and $\Gamma[x \mapsto \mathrm{AppClose}_\Gamma(\tau')] \rhd e_2 : \tau_a$.

Since v_1 is a value, it is non-expansive and we need only consider that case. Thus by Lemma 9, $\Gamma \rhd e_2[v_1/x] : \tau_a$. Since this rule does not change the state, $\Gamma \rhd \sigma(\ell_i) : u_i$ holds for all $\ell_i \in \text{dom}(\sigma)$ by (12). $\Box$

Now we prove type soundness for the natural semantics.

Theorem 12 If $\rhd\langle e, \sigma\rangle : \tau$ and $\langle e, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$, then $\rhd\langle v, \sigma'\rangle : \tau$.

Proof: Type soundness with respect to the natural semantics is an immediate corollary of Lemma 13. $\Box$

Lemma 13 If $\Gamma \rhd \langle e_a, \sigma\rangle : \tau_a$ and $\langle e_a, \sigma\rangle \Downarrow \langle v_a, \sigma'\rangle$, then $\Gamma \rhd \langle v_a, \sigma'\rangle : \tau_a$. If, furthermore, $\Gamma \rhd \langle e_b, \sigma\rangle : \tau_b$, then $\Gamma \rhd \langle e_b, \sigma'\rangle : \tau_b$.

Proof: By Definition 8, the type inference system extended by

$$\Gamma \rhd \ell_i : u_i \, \text{ref} \tag{14}$$

for all $\ell_i \in \text{dom}(\sigma)$ proves $\Gamma \rhd e_a : \tau_a$ and $\Gamma \rhd \sigma(\ell_i) : u_i$ for all $\ell_i \in \text{dom}(\sigma)$. We claim that if we further extend the type inference system with an axiom of the form of (14) for each $\ell_i \in \text{dom}(\sigma') \setminus \text{dom}(\sigma)$, then $\Gamma \rhd v_a : \tau_a$ and $\Gamma \rhd \sigma'(\ell_i) : u_i$ are provable in this system, for all $\ell_i \in \text{dom}(\sigma')$. Once we have proved this claim, it only remains to show that if $\Gamma \rhd e_b : \tau_b$ is provable in the type inference system as extended for σ, then it is also provable using the system as further extended for σ'. But since the type inference system as extended for σ' includes all rules in the system as extended for σ, this is immediate.

We prove the above claim by induction on the height of the proof of $\langle e_a, \sigma\rangle \Downarrow \langle v_a, \sigma'\rangle$. We perform a case analysis of the last rule in the proof, again showing only the more difficult or illustrative cases.

Rule $[asg_{ns}]$: We are given $\Gamma \rhd (:= e_1)e_2 : \tau_a$, so by the structure of the typing rules $\Gamma \rhd e_1 : \tau_a \, \text{ref}$ and $\Gamma \rhd e_2 : \tau_a$. By induction on the first premise $\langle e_1, \sigma\rangle \Downarrow \langle \ell_1, \sigma_1\rangle$ of Rule $[asg_{ns}]$, the judgements $\Gamma \rhd \ell_1 : \tau_a \, \text{ref}$, $\Gamma \rhd e_2 : \tau_a$, and $\Gamma \rhd \sigma_1(\ell_i) : u_i$ are provable, for all $\ell_i \in \text{dom}(\sigma_1)$, using the type inference system as extended for σ_1. Induction on the second premise $\langle e_2, \sigma_1\rangle \Downarrow \langle v_2, \sigma_2\rangle$ now yields that

$$\Gamma \rhd v_2 : \tau_a \tag{15}$$

$$\Gamma \rhd \ell_1 : \tau_a \, \text{ref} \tag{16}$$

and

$$\Gamma \rhd \sigma_2(\ell_i) : u_i \tag{17}$$

are provable, for all $\ell_i \in \text{dom}(\sigma_2)$, using the type inference system as extended for σ_2. Since $\ell_1 \in \text{dom}(\sigma_2)$, we do not further extend the system for $\sigma_2[\ell_1 \mapsto v_2]$. Since v_a is just v_2, it only remains to show that $\Gamma \rhd \sigma_2[\ell_1 \mapsto v_2](\ell_i) : u_i$ for all $\ell_i \in \text{dom}(\sigma_2)$. This follows from (15) and (17) as long as u_1 is τ_a. But this is true by the structure of the typing rules, since (16) holds.

Rule $[\mathit{ref}_{ns}]$**:** We are given $\Gamma \rhd \mathbf{ref}\, e : \tau_a$, so by the structure of the typing rules τ_a is of the form $u\,\text{ref}$, and $\Gamma \rhd e : u$. By the induction hypothesis on the premise $\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$, the judgements

$$\Gamma \rhd v : u \tag{18}$$

and

$$\Gamma \rhd \sigma'(\ell_i) : u_i \tag{19}$$

are provable, for all $\ell_i \in \text{dom}(\sigma')$, in the type inference system as extended for σ'. If we further extend the type inference system with the axiom $\Gamma \rhd \ell : u\,\text{ref}$, then, since v_a is ℓ, it only remains to show

$$\Gamma \rhd \sigma'[\ell \mapsto v](\ell_i) : u_i$$

for all $\ell_i \in \text{dom}(\sigma'[\ell \mapsto v])$. Since ℓ is new, this follows from (18) and (19).

Rule $[\mathit{let}_{ns}]$**:** We are given $\Gamma \rhd \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \tau_a$, so by the structure of the typing rules $\Gamma \rhd e_1 : \tau'$ and either

- e_1 is non-expansive and $\Gamma[x \mapsto \text{Close}_\Gamma(\tau')] \rhd e_2 : \tau_a$; or
- e_1 is expansive and $\Gamma[x \mapsto \text{AppClose}_\Gamma(\tau')] \rhd e_2 : \tau_a$.

$$\Gamma[x \mapsto \text{AppClose}_\Gamma(\tau')] \rhd e_2 : \tau_a$$

implies

$$\Gamma[x \mapsto \text{Close}_\Gamma(\tau')] \rhd e_2 : \tau_a$$

so the latter holds in either case. Thus by the induction hypothesis on the first premise $\langle e_1, \sigma\rangle \Downarrow \langle v_1, \sigma_1\rangle$, the judgements $\Gamma \triangleright v_1 : \tau'$, $\Gamma \triangleright \sigma_1(\ell_i) : u_i$, and

$$\Gamma[x \mapsto \mathrm{Close}_\Gamma(\tau')] \triangleright e_2 : \tau_a$$

are provable for all $\ell_i \in \mathrm{dom}(\sigma_2)$ using the type inference system as extended for σ_1. By Lemma 9, $\Gamma \triangleright e_2[v_1/x] : \tau_a$. The induction hypothesis on the second premise $\langle e_2[v_1/x], \sigma_1\rangle \Downarrow \langle v_2, \sigma_2\rangle$ then yields $\Gamma \triangleright v_2 : \tau_a$ and $\Gamma \triangleright \sigma_2(\ell_i) : u_i$ for all $\ell_i \in \mathrm{dom}(\sigma_2)$ using the type inference system as extended for σ_2. □

Finally, we present a type soundness proof for the reduction semantics.

Theorem 14 If $\triangleright\langle e, \sigma\rangle : \tau$ and $\langle e, \sigma\rangle \longmapsto^* \langle v, \sigma'\rangle$, then $\triangleright\langle v, \sigma'\rangle : \tau$.

Proof: This theorem follows from Lemma 15 by a simple induction on the length of the derivation sequence $\langle e, \sigma\rangle \longmapsto^* \langle v, \sigma'\rangle$. □

Lemma 15 If $\Gamma \triangleright \langle e_a, \sigma\rangle : \tau_a$ and $\langle e_a, \sigma\rangle \longmapsto \langle e'_a, \sigma'\rangle$, then $\Gamma \triangleright \langle e'_a, \sigma'\rangle : \tau_a$. Furthermore, if $\Gamma \triangleright \langle e_b, \sigma\rangle : \tau_b$, then $\Gamma \triangleright \langle e_b, \sigma'\rangle : \tau_b$.

Proof: By Definition 8, the type inference system extended by

$$\Gamma \triangleright \ell_i : u_i\,\mathrm{ref} \tag{20}$$

for all $\ell_i \in \mathrm{dom}(\sigma)$ proves $\Gamma \triangleright e_a : \tau_a$ and $\Gamma \triangleright \sigma(\ell_i) : u_i$ for all $\ell_i \in \mathrm{dom}(\sigma)$. We claim that if we further extend the type inference system with an axiom of the form of (20) for each $\ell_i \in \mathrm{dom}(\sigma') \setminus \mathrm{dom}(\sigma)$, then $\Gamma \triangleright e'_a : \tau_a$ and $\Gamma \triangleright \sigma'(\ell_i) : u_i$ are provable in this system, for all $\ell_i \in \mathrm{dom}(\sigma')$. Once we have proved this claim, it only remains to show that if $\Gamma \triangleright e_b : \tau_b$ is provable in the type inference system as extended for σ, then it is also provable using the system as further extended for σ'. But since the type inference system as extended for σ' includes all rules in the system as extended for σ, this is immediate.

We prove the above claim by induction on the structure of the evaluation context E[ ] used in the reduction step $\langle e_a, \sigma\rangle \longmapsto \langle e'_a, \sigma'\rangle$. As is our custom, we show only the more difficult and illustrative cases in this proof.

E = [ ]: The proof proceeds by a case analysis of the rule used to prove $\langle e_a, \sigma\rangle \rightsquigarrow \langle e'_a, \sigma'\rangle$. The proof for Rules $[fn_{rs}]$, $[ref_{rs}]$, $[deref_{rs}]$, $[asg_{rs}]$, and $[let_{rs}]$ are identical to the proofs for the corresponding rules $[fn_{ts}]$, $[ref_{ts}]$, $[deref_{ts}]$, $[asg_{ts}]$, and $[let^2_{ts}]$ in Lemma 11 for transition semantics.

$\mathbf{E} = \mathbf{let}\, x = \mathrm{E}'\, \mathbf{in}\, e$: Let $e_a = \mathrm{E}[e_1]$ and $e'_a = \mathrm{E}[e'_1]$. We are given $\Gamma \rhd e_a : \tau_a$, so by the structure of the typing rules $\Gamma \rhd \mathrm{E}'[e_1] : \tau'$ and either

- $\mathrm{E}'[e_1]$ is non-expansive and $\Gamma[x \mapsto \mathrm{Close}_\Gamma(\tau')] \rhd e : \tau_a$; or
- $\mathrm{E}'[e_1]$ is expansive and $\Gamma[x \mapsto \mathrm{AppClose}_\Gamma(\tau')] \rhd e : \tau_a$.

Since $\langle \mathrm{E}[e_1], \sigma\rangle \longmapsto \langle \mathrm{E}[e'_1], \sigma'\rangle$,

$$\langle e_1, \sigma\rangle \rightsquigarrow \langle e'_1, \sigma'\rangle \tag{21}$$

Thus e_1 is not a value, and so neither is $\mathrm{E}'[e_1]$; $\mathrm{E}'[e_1]$ is therefore expansive and we do not need to consider the non-expansive case any further. By (21) and Rule $[context_{rs}]$, $\langle \mathrm{E}'[e_1], \sigma\rangle \longmapsto \langle \mathrm{E}'[e'_1], \sigma'\rangle$. By the induction hypothesis, $\Gamma \rhd \mathrm{E}'[e'_1] : \tau'$, $\Gamma \rhd \sigma'(\ell_i) : u_i$, and

$$\Gamma[x \mapsto \mathrm{AppClose}_\Gamma(\tau')] \rhd e : \tau_a$$

are provable for all $\ell_i \in \mathrm{dom}(\sigma')$ using the type inference system as extended for σ'. This latter judgement implies

$$\Gamma[x \mapsto \mathrm{Close}_\Gamma(\tau')] \rhd \langle e, \sigma'\rangle : \tau_a$$

Therefore, whether $\mathrm{E}'[e'_1]$ is expansive or non-expansive,

$$\Gamma \rhd \mathbf{let}\, x = \mathrm{E}'[e'_1]\, \mathbf{in}\, e : \tau_a$$

holds by either Rule $[let^{non}_{type}]$ or Rule $[let^{exp}_{type}]$. □

It is clear that the three type soundness proofs are very similar. The proofs for the transition and reduction semantics are almost identical, although the cases are arranged differently in the two proofs. Each case in the proof for natural semantics makes more appeals to the induction hypothesis, since the rules in the natural semantics have more premises. However, this is counter-balanced by correspondingly more rules and/or evaluation contexts in the transition and reduction semantics, resulting in more cases for those proofs. On the other hand, the natural semantics does hold a slight edge over the transition and reduction semantics in that the proof for natural semantics avoids the induction over the length of the derivation sequence, albeit a simple one. It is significant to note that the vast difference in complexity between Tofte's proof and Wright and Felleisen's proof is not due to Tofte's use of natural semantics or Wright and Felleisen's use of reduction semantics. Rather, it is a consequence of Wright and Felleisen's use of value substitution rather than environments.

4 Conclusion

This paper has compared three forms of operations semantics from the perspectives of expressiveness and utility. We found that the expressive power of the three varied with the language. Natural semantics was very good at specifying "large granularity" features such as blocks, but is apparently unable to capture interleaving because of its "small granularity". On the other hand, transition semantics and reduction semantics easily express "small granularity" features but have difficulty with "large granularity" features. Reduction semantics provided especially concise specifications of non-sequential control constructs such as abortion and interleaving. In terms of utility, the natural semantics seemed to allow simpler proofs for the two applications we looked at.

There is certainly much more investigation that can be done in this matter. Additional and more advanced features can be used to further explore the comparative expressiveness of the approaches to semantics. Certainly other applications can be used to get a better idea of the utility of the semantics. Finally, the scope of the comparison can be expanded to include other criteria and other forms of semantics. But hopefully the start in this paper will prove enlightening and provide a basis for further investigation.

A In Pursuit of a Natural Semantics for Parallelism

In this appendix we make further attempts to construct a natural semantics for interleaving. Through this endeavor we hope to gain a better understanding of why this form of semantics cannot express this feature.

Consider the extension to the natural semantics of **While** given in Table 23.[25] Since a natural semantics deals with the execution of a statement as an atomic unit, we explicitly decompose compound statements in the branches of the **par** statement. This is a bit unusual, but the semantics is still valid.

Also note that several rules may be applicable for a given statement. Some of this is inherent in the non-deterministic interleaving of statements; however, even for a given interleaving, multiple proofs are possible with these semantics. For example, both of the following proofs execute the statements in the order s_1, s_2, s_3, s_4.

$$\dfrac{\dfrac{\langle s_1,\sigma_0\rangle \Downarrow \sigma_1 \quad \langle s_2,\sigma_1\rangle \Downarrow \sigma_2}{\langle s_1; s_2,\sigma_0\rangle \Downarrow \sigma_2}\ \mathit{Rule}[comp_{ns}] \quad \dfrac{\langle s_3,\sigma_2\rangle \Downarrow \sigma_3 \quad \langle s_4,\sigma_3\rangle \Downarrow \sigma_4}{\langle s_3; s_4,\sigma_2\rangle \Downarrow \sigma_4}\ \mathit{Rule}[comp_{ns}]}{\langle (s_1; s_2)\ \mathbf{par}\ (s_3; s_4),\sigma_0\rangle \Downarrow \sigma_4}\ \mathit{Rule}[par^5_{ns}]$$

$$\dfrac{\langle s_1,\sigma_0\rangle \Downarrow \sigma_1 \quad \dfrac{\langle s_2,\sigma_1\rangle \Downarrow \sigma_2 \quad \dfrac{\langle s_3,\sigma_2\rangle \Downarrow \sigma_3 \quad \langle s_4,\sigma_3\rangle \Downarrow \sigma_4}{\langle s_3; s_4,\sigma_2\rangle \Downarrow \sigma_4}\ \mathit{Rule}[comp_{ns}]}{\langle s_2\ \mathbf{par}\ (s_3; s_4),\sigma_2\rangle \Downarrow \sigma_4}\ \mathit{Rule}[par^5_{ns}]}{\langle (s_1; s_2)\ \mathbf{par}\ (s_3; s_4),\sigma_0\rangle \Downarrow \sigma_4}\ \mathit{Rule}[par^1_{ns}]$$

As a result, if a program property is defined in terms of the structure of the derivation of the program's semantics, it is necessary to show that this definition is consistent across different derivations of the same meaning.

But even if we are willing to accept this extra non-determinism, there are more serious problems with this semantics. Consider the statement

$$((s_1; s_2); (s_3; s_4))\ \mathbf{par}\ (s_5; s_6)$$

With the given rules, we can either:

[25] A naive semantics would leave out the rules $[par^2_{ns}]$ and $[par^4_{ns}]$. As Jon Riecke pointed out, the resulting semantics would assume that the ';' operator associates to the right [Rie91]. Thus the naive semantics would not work for all possible parsings of a compound statement.

$$\frac{\langle s_1, \sigma\rangle \Downarrow \sigma' \quad \langle s_2 \,\mathbf{par}\, s_3, \sigma'\rangle \Downarrow \sigma''}{\langle (s_1; s_2) \,\mathbf{par}\, s_3, \sigma\rangle \Downarrow \sigma''} \qquad [par^1_{ns}]$$

$$\frac{\langle s_1 \,\mathbf{par}\, s_3, \sigma\rangle \Downarrow \sigma' \quad \langle s_2, \sigma'\rangle \Downarrow \sigma''}{\langle (s_1; s_2) \,\mathbf{par}\, s_3, \sigma\rangle \Downarrow \sigma''} \qquad [par^2_{ns}]$$

$$\frac{\langle s_2, \sigma\rangle \Downarrow \sigma' \quad \langle s_1 \,\mathbf{par}\, s_3, \sigma'\rangle \Downarrow \sigma''}{\langle s_1 \,\mathbf{par}\, (s_2; s_3), \sigma\rangle \Downarrow \sigma''} \qquad [par^3_{ns}]$$

$$\frac{\langle s_1 \,\mathbf{par}\, s_2, \sigma\rangle \Downarrow \sigma' \quad \langle s_3, \sigma'\rangle \Downarrow \sigma''}{\langle s_1 \,\mathbf{par}\, (s_2; s_3), \sigma\rangle \Downarrow \sigma''} \qquad [par^4_{ns}]$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \sigma' \quad \langle s_2, \sigma'\rangle \Downarrow \sigma''}{\langle s_1 \,\mathbf{par}\, s_2, \sigma\rangle \Downarrow \sigma''} \qquad [par^5_{ns}]$$

$$\frac{\langle s_2, \sigma\rangle \Downarrow \sigma' \quad \langle s_1, \sigma'\rangle \Downarrow \sigma''}{\langle s_1 \,\mathbf{par}\, s_2, \sigma\rangle \Downarrow \sigma''} \qquad [par^6_{ns}]$$

Table 23: Extension to Natural Semantics for Parallelism

$$\frac{\langle (s_1; (s_2; s_3)) \,\mathbf{par}\, s_4, \sigma \rangle \Downarrow \sigma'}{\langle ((s_1; s_2); s_3) \,\mathbf{par}\, s_4, \sigma \rangle \Downarrow \sigma'} \qquad [par^7_{ns}]$$

$$\frac{\langle s_1 \,\mathbf{par}\, (s_2; (s_3; s_4)), \sigma \rangle \Downarrow \sigma'}{\langle s_1 \,\mathbf{par}\, ((s_2; s_3); s_4), \sigma \rangle \Downarrow \sigma'} \qquad [par^8_{ns}]$$

Table 24: Another Extension to Natural Semantics for Parallelism

- execute $(s_1; s_2)$ first, followed by the execution of $(s_3; s_4) \,\mathbf{par}\, (s_5; s_6)$
- execute $(s_1; s_2) \,\mathbf{par}\, (s_5; s_6)$ first, followed by the execution of $(s_3; s_4)$
- execute s_5 first, followed by the execution of $((s_1; s_2); (s_3; s_4)) \,\mathbf{par}\, s_6$
- execute $((s_1; s_2); (s_3; s_4)) \,\mathbf{par}\, s_5$ first, followed by the execution of s_6
- execute $((s_1; s_2); (s_3; s_4))$ first and then $(s_5; s_6)$
- execute $(s_5; s_6)$ first and then $((s_1; s_2); (s_3; s_4))$

These rules allow us to only decompose one side of the **par** at a time and only decompose a single level of composition before making an interleaving commitment. Thus we cannot capture, for example, the interleaving s_1, s_5, s_2, s_3, s_6, s_4.

If we add the rules of Table 24, which allow us to associate the composition operator to the right, we can overcome these limitations.[26] The following proof uses the previously excluded interleaving order.

$$\frac{\dfrac{\langle s_1, \sigma_0 \rangle \Downarrow \sigma_1 \quad \dfrac{\langle s_5, \sigma_1 \rangle \Downarrow \sigma_2 \quad \dfrac{\langle s_2, \sigma_2 \rangle \Downarrow \sigma_3 \quad \dfrac{\langle s_3, \sigma_3 \rangle \Downarrow \sigma_4 \quad \dfrac{\langle s_4, \sigma_4 \rangle \Downarrow \sigma_5 \quad \langle s_6, \sigma_5 \rangle \Downarrow \sigma_6}{\langle s_4 \,\mathbf{par}\, s_6, \sigma_4 \rangle \Downarrow \sigma_6} \, Rule[par^6_{ns}]}{\langle (s_3; s_4) \,\mathbf{par}\, s_6, \sigma_3 \rangle \Downarrow \sigma_6} \, Rule[par^1_{ns}]}{\langle (s_2; (s_3; s_4)) \,\mathbf{par}\, s_6, \sigma_2 \rangle \Downarrow \sigma_6} \, Rule[par^1_{ns}]}{\langle (s_2; (s_3; s_4)) \,\mathbf{par}\, (s_5; s_6), \sigma_1 \rangle \Downarrow \sigma_6} \, Rule[par^3_{ns}]}{\langle (s_1; (s_2; (s_3; s_4))) \,\mathbf{par}\, (s_5; s_6), \sigma_0 \rangle \Downarrow \sigma_6} \, Rule[par^1_{ns}]}{\langle ((s_1; s_2); (s_3; s_4)) \,\mathbf{par}\, (s_5; s_6), \sigma_0 \rangle \Downarrow \sigma_6} \, Rule[par^7_{ns}]$$

[26]As previously observed, once we can associate composition to the right, rules $[par^2_{ns}]$ and $[par^4_{ns}]$ become unnecessary.

It seems that we are now able to arbitrarily interleave sequences of compound statements. However, we still treat the **if** and **while** statements as monolithic entities; we cannot interleave within the statements in the branches of an **if** statement or the body of a **while** statement. For example, if the statement $\mathbf{while}\, e\, \mathbf{do}(s_1; s_2)$ was in one branch of a **par** statement, we could not interleave a statement from the other branch between s_1 and s_2 in an execution of the loop body. Thus we must add additional rules that allow the decomposition of **if** and **while** statements in the branches of a **par** statement. Table 25 shows such rules.

We appear to be very close to a solution now. But our rules do not allow us to decompose **par** statements that occur in the branches of another **par** statement. In the case of **if** and **while** statements, we just duplicated the usual rules for these statements in the context of a **par** statement. To do that for nested **par** statements, we would need to add a large number of rules such as

$$\frac{\langle (s_1; ((s_2\, \mathbf{par}\, s_3); s_4))\, \mathbf{par}\, s_5, \sigma' \rangle \Downarrow \sigma''}{\langle (((s_1; s_2)\mathbf{par}\, s_3); s_4)\, \mathbf{par}\, s_5, \sigma \rangle \Downarrow \sigma''} \qquad [par^{21}_{ns}]$$

Now each of these new rules is also a **par** rule that we need to duplicate in the context of a **par** statement:

$$\frac{\langle (((s_1; ((s_2\, \mathbf{par}\, s_3); s_4))\, \mathbf{par}\, s_5); s_6)\, \mathbf{par}\, s_7, \sigma' \rangle \Downarrow \sigma''}{\langle (((((s_1; s_2)\mathbf{par}\, s_3); s_4)\, \mathbf{par}\, s_5); s_6)\, \mathbf{par}\, s_7, \sigma \rangle \Downarrow \sigma''} \qquad [par^{22}_{ns}]$$

If we leave out Rule $[par^{22}_{ns}]$, for example, we can't achieve the interleaving s_1, s_7, s_2, s_3, s_4, s_5, s_6 for the statement in the conclusion of that rule.

It is clear that this approach requires an infinite number of rules, and this seems to be the problem in general with constructing a natural semantics for parallelism. Since natural semantics constrains us to only be able to denote the total execution of a statement, we need to be able to specify the atomic statement (**skip** or assignment) that is to be executed first. But there are an unbounded number of possibilities for how this statement will be nested in the structure of an arbitrary statement, so we cannot give a rule for each case. This leads us to conclude that natural semantics cannot express arbitrary interleaving.

$$\frac{\langle (s_1; s_3)\,\mathbf{par}\,s_4, \sigma\rangle \Downarrow \sigma'}{\langle (((\mathbf{if}\,e\,\mathbf{then}\,s_1\,\mathbf{else}\,s_2); s_3)\,\mathbf{par}\,s_4, \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{true} \qquad [par^{9}_{ns}]$$

$$\frac{\langle (s_2; s_3)\,\mathbf{par}\,s_4, \sigma\rangle \Downarrow \sigma'}{\langle (((\mathbf{if}\,e\,\mathbf{then}\,s_1\,\mathbf{else}\,s_2); s_3)\,\mathbf{par}\,s_4, \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{false} \qquad [par^{10}_{ns}]$$

$$\frac{\langle s_1\,\mathbf{par}\,(s_2; s_4), \sigma\rangle \Downarrow \sigma'}{\langle s_1\,\mathbf{par}\,((\mathbf{if}\,e\,\mathbf{then}\,s_2\,\mathbf{else}\,s_3); s_4), \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{true} \qquad [par^{11}_{ns}]$$

$$\frac{\langle s_1\,\mathbf{par}\,(s_3; s_4), \sigma\rangle \Downarrow \sigma'}{\langle s_1\,\mathbf{par}\,((\mathbf{if}\,e\,\mathbf{then}\,s_2\,\mathbf{else}\,s_3); s_4), \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{false} \qquad [par^{12}_{ns}]$$

$$\frac{\langle s_1\,\mathbf{par}\,s_3, \sigma\rangle \Downarrow \sigma'}{\langle (\mathbf{if}\,e\,\mathbf{then}\,s_1\,\mathbf{else}\,s_2)\,\mathbf{par}\,s_3, \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{true} \qquad [par^{13}_{ns}]$$

$$\frac{\langle s_2\,\mathbf{par}\,s_3, \sigma\rangle \Downarrow \sigma'}{\langle (\mathbf{if}\,e\,\mathbf{then}\,s_1\,\mathbf{else}\,s_2)\,\mathbf{par}\,s_3, \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{false} \qquad [par^{14}_{ns}]$$

$$\frac{\langle s_1\,\mathbf{par}\,s_2, \sigma\rangle \Downarrow \sigma'}{\langle s_1\,\mathbf{par}\,(\mathbf{if}\,e\,\mathbf{then}\,s_2\,\mathbf{else}\,s_3), \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{true} \qquad [par^{15}_{ns}]$$

$$\frac{\langle s_1\,\mathbf{par}\,s_3, \sigma\rangle \Downarrow \sigma'}{\langle s_1\,\mathbf{par}\,(\mathbf{if}\,e\,\mathbf{then}\,s_2\,\mathbf{else}\,s_3), \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{false} \qquad [par^{16}_{ns}]$$

$$\frac{\langle (s_1; ((\mathbf{while}\,e\,\mathbf{do}\,s_1); s_2))\,\mathbf{par}\,s_3, \sigma\rangle \Downarrow \sigma'}{\langle ((\mathbf{while}\,e\,\mathbf{do}\,s_1); s_2)\,\mathbf{par}\,s_3, \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{true} \qquad [par^{17}_{ns}]$$

$$\frac{\langle s_1\,\mathbf{par}\,(s_2; ((\mathbf{while}\,e\,\mathbf{do}\,s_2); s_3)), \sigma\rangle \Downarrow \sigma'}{\langle s_1\,\mathbf{par}\,((\mathbf{while}\,e\,\mathbf{do}\,s_2); s_3), \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{true} \qquad [par^{18}_{ns}]$$

$$\frac{\langle (s_1; \mathbf{while}\,e\,\mathbf{do}\,s_1)\,\mathbf{par}\,s_2, \sigma\rangle \Downarrow \sigma'}{\langle (\mathbf{while}\,e\,\mathbf{do}\,s_1)\,\mathbf{par}\,s_2, \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{true} \qquad [par^{19}_{ns}]$$

$$\frac{\langle s_1\,\mathbf{par}\,(s_2; \mathbf{while}\,e\,\mathbf{do}\,s_2), \sigma\rangle \Downarrow \sigma'}{\langle s_1\,\mathbf{par}\,(\mathbf{while}\,e\,\mathbf{do}\,s_2), \sigma\rangle \Downarrow \sigma'} \quad \text{if } \mathcal{E}[\![e]\!]\sigma = \text{true} \qquad [par^{20}_{ns}]$$

Table 25: Yet One More Extension to Natural Semantics for Parallelism

References

[FH89] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report COMP TR89-100, Rice University, Department of Computer Science, June 1989.

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, 1987.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Press, 1992.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, September 1981.

[Rie91] Jon Riecke, October 1991. Private communication.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report COMP TR91-160, Rice University, Department of Computer Science, April 1991.