1 Introduction

In this paper we show that the specification for a lambda calculus to SECD compiler originally proposed by Henderson [7] is correct. A hardware implementation of Henderson's SECD machine has recently been proven (partially) correct [6] in the Higher-Order-Logic (HOL) system [5]; [1] this paper along with the HOL proof gives us a good step towards total systems verification.

The SECD machine was designed as an abstract architecture by Landin in the 1960's when he was seeking a way to give an operational semantics for ALGOL 60 (see [10, 11, 12, 13]). Many different versions of the machine have been proposed including both lazy and eager evaluation mechanisms ([1, 4, 7, 8]).

Plotkin ([17]) proved that a "high-level" SECD machine correctly executed the call-by-name lambda calculus. In this paper we show, using more recent proof techniques, that Henderson's SECD machine is also correct. The high-level machine uses pieces of lambda calculus syntax as instructions, whereas Henderson's machine has its own instruction set. In other words, the high-level machine directly interprets lambda expressions, while the machine we deal with requires a compiler.

Henderson also describes a source language for the SECD machine called LispKit (a functional subset of LISP) and gives a description of the translation from LispKit to SECD code. We use a language very similar to LispKit in this paper, called miniSML. It consists of the untyped lambda calculus with *let, letrec* and some built-in constants, and differs from LispKit mainly in syntax (using Standard ML list notation instead of S-expression notation). There are also several different versions of miniSML we describe our version in detail. In particular, most versions of miniSML are typed although the one we use is untyped (as is LispKit).

The translation from miniSML to SECD machine code is correct if the semantics of LAM programs are preserved by the translation. Morris [15] proposed the following diagram to convey this correctness property:

$$\begin{tikzcd}
miniSML \arrow[r, "C"] \arrow[d, "miniSML\ Semantics"'] & SECD \arrow[d, "SECD\ Semantics"] \\
miniSML\ Semantic\ Domain \arrow[r, "U"'] & SECD\ Semantic\ Domain
\end{tikzcd}$$

The original idea behind this diagram was to make each corner an algebra (of the same type) and each arrow a homomorphism. The work involved in this approach

[1]Proven functionally correct (partially) to the register transfer level. There is no assurance that the fabrication is correct or that the physical hardware corresponds exactly to the specification. Nevertheless, functional correctness over all inputs provides a high degree of confidence in the design.

is in making the target language into an algebra of the same type as the source language, i.e., implementing each source language operator in terms of target operations. When this is done, the correctness of the compiler holds by the uniqueness of the homomorphisms from the (initial, term) source language algebra.

We use the same diagram to convey correctness, but our approach is somewhat different. The source and target languages are given by abstract syntaxes and the semantics are given by inductively defined relations. Given the compiler C and a representation function U, we check that the diagram commutes.

Several authors have used the latter approach to correctness, including Despeyroux [2] who gives the correctness of a (slightly different version of) miniSML to CAM machine code compiler. [2] The important difference in our proof is that we do not give meanings to non-terminating programs, either in the sense of a bottom element in denotational semantics or the (intuitively similar) approximate normal forms used by Despeyroux. This means that we can reason separately about programs which have normal forms and those that do not. It also leads to a considerable reduction in what must be proved for compiler correctness over terminating programs, and several lemmas we use may also be used to simplify Despeyroux's proof. The reason for not giving meanings to non-terminating programs is our use of operational semantics, which we believe should be non-terminating for non-terminating programs. This axiom leads to more complexity in the proof of compiler correctness for non-terminating programs. To arrive at a clean proof under this axiom we merge Plotkin's approach to this problem [17] with Despeyroux's use of natural semantics. This proof has only been done by hand and should be mechanized (preferably in HOL so that we can tie it to the machine proof) to ensure that no errors have crept in. The SECD machine is considerably more complex than the CAM machine which also adds to the complexity of the proof.

We review the ideas behind natural (operational) semantics. We then proceed by defining miniSML and SECD and giving them operational semantics. After specifying the compiler and defining a mapping between miniSML "meanings" and SECD "meanings" we prove our main theorem - that the compiler specification is correct. Finally, we change miniSML and SECD to evaluate lazily, change the compiler specification appropriately, and outline the proof of correctness for this lazy version.

2 Natural Deduction and Sequent Calculi

In this section we introduce natural deduction and sequent calculi as they provide the basis for the style of proof we will use throughout the thesis.

Natural deduction is an approach to proof which is supposed to mirror human pat-

[2]This is the work which most closely relates to our proof of correctness.

terns of reasoning. For each connective in the logic there are *introduction* rules and *elimination* rules. The introduction rules tell us how to combine terms using the connective and the elimination rules tell us take apart a term which contains the connective. The syntax of natural deduction rules is of the form $\frac{X}{t}$ which states that if every sentence $x \in X$ is true, then so is t. For instance, if our logic has some sentences, of which A and B are members, and we wish to have conjunction in the system, we could add the rules

$$(\wedge I') \quad \frac{A \qquad B}{A \wedge B} \qquad (\wedge E1') \quad \frac{A \wedge B}{A} \qquad (\wedge E2') \quad \frac{A \wedge B}{B}$$

the first of which introduces conjunction, $\wedge_I$, (if A is true and B is true then $A \wedge B$ is true) and the second two eliminate conjunction (if $A \wedge B$ is true then A alone is true (by $\wedge_{E1}$) and B alone is true (by $\wedge_{E2}$)).

Assumptions may also be kept explicitly with each rule. A sequent $\Gamma \vdash \Delta$ represents the intuition that each sentence in Δ is true when all the sentences in Γ are true. In natural deduction it is possible to perform operations on both the hypotheses Γ and the conclusions Δ. We call a natural deduction system a sequent calculus if the only operations allowed on the hypothesis are to add and discharge new sentences, and if the conclusion is always a single sentence.

Our conjunction rules from above form the sequent calculus

$$(\wedge I) \quad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \qquad (\wedge E1) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad (\wedge E2) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

where Γ, Δ is the union of the two assumption sets. It is not uncommon for there to be different sets of rules which allow the same proofs. For example, an alternative to the elimination rules given above is:

$$(\wedge E) \quad \frac{\Gamma \vdash A \wedge B \qquad \Delta, A, B \vdash C}{\Gamma, \Delta \vdash C}$$

Let us add to our conjunction rules so that we can give a small example of a proof in natural deduction. For any sentence we have an assumption axiom:

$$(\mathit{Assum}) \quad A \vdash A$$

We also add introduction and elimination for implication

$$(\Rightarrow I) \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad (\Rightarrow E) \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B}$$

Intuitively, if B is true when A is on the assumption list, then "if A then B" is true. On the other hand, if "if A then B" is known and A is true, then B follows.

If we assume $A \wedge B \vdash C$ then we can derive $B \wedge A \vdash C$ as follows

$$(\wedge E)\ \dfrac{B \wedge A \vdash B \wedge A \qquad (\Rightarrow E)\ \dfrac{(\Rightarrow I)\ \dfrac{A \wedge B \vdash C}{\vdash A \wedge B \Rightarrow C} \qquad (\wedge I)\ \dfrac{A \vdash A \quad B \vdash B}{A, B \vdash A \wedge B}}{A, B \vdash C}}{B \wedge A \vdash C}$$

Here we have labelled each step in the proof with the rule justifing it (with the exception of *Assum*). In the sequel we will label "important" steps only.

Proofs displayed in the "tree" style can grow quite large, so we introduce some notation to manage them. If c follows from p_1 and p_2 then we may write any of

$$\frac{p_1 \quad p_2}{c} \qquad \begin{array}{c} p_1 \\ \hline p_2 \\ \hline c \end{array} \qquad \begin{array}{c} (p_1) \\ p_2 \\ \hline c \end{array}$$

the last of which puts brackets around p_1 when it is a proof of more than one-line.

3 Natural Operational Semantics

In this section we develop an operational semantics which will be used to give meanings to the programming languages and to generate the compiler specification.

An operational semantics is built as a transition system following the work of Plotkin [16].

Definition 1 A *transition system (TS)* is a structure $< \Gamma, \rightarrow >$ where Γ is a set (of *configurations*), and $\rightarrow \subseteq \Gamma \times \Gamma$ is a binary relation (the transition relation).

A *terminal transition system (TTS)* is a structure $< \Gamma, \rightarrow, T >$ where the first two components form a TS and $T \subseteq \Gamma$ is a set of terminal configurations such that $\forall \gamma \in T, \gamma' \in \Gamma.(\gamma, \gamma') \notin \rightarrow$. $\Diamond$

In general we will write $a \rightarrow b$ for $(a, b) \in \rightarrow$, $a \not\rightarrow b$ for $(a, b) \notin \rightarrow$, and $a \not\rightarrow$ for $\forall b \in \Gamma, a \not\rightarrow b$.

Here is an example from Plotkin:

Example: A finite automaton M is a quintuple $< Q, \Sigma, \delta, q_0, F >$ where

- Q is a finite set of states.
- Σ is a finite set, the input alphabet.
- $\delta : Q \times \Sigma \rightarrow Q$ is the state transition relation.

- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final states.

Set $\Gamma = Q \times \Sigma^*$ and $T = \{(q, \emptyset) | q \in F\}$ and the transition relation is defined by

$$\frac{w \neq \emptyset \qquad \delta(q, hd(w)) = q'}{(q, w) \rightarrow (q', tl(w))}$$

where w is a variable ranging over sequences, $hd : \Sigma^* \rightarrow \Sigma$ takes the first element of a sequence of input characters and $tl : \Sigma^* \rightarrow \Sigma^*$ returns the input sequence without the first character. This "rule" looks like a rule from natural deduction, although the hypotheses are boolean predicates which must be satisfied and the conclusion is an element of a TS.

It is easy to define the language the automaton accepts. It is just

$$\{w | w \in \Sigma^*, \exists q \in F.(q_0, w) \rightarrow^* (q, \emptyset)\}$$

where $\rightarrow^*$ is the transitive closure of $\rightarrow$. There is no transition for a configuration with empty input. $\diamondsuit$

A typical language with variables uses a store (relating variables to their values) in order to evaluate expressions in the language. We can view our configurations as pairs (c, α) where c is a command in the language and α is a state. Then transitions such as $(c, \alpha) \rightarrow (c', \alpha')$ would indicate that evaluation of the command c in the state α results in the command c' and the state α'. In this setting, if we have any commands which halt evaluation, leaving an answer in the state, we could set $T = \{\alpha\}, \Gamma = \{(c, \alpha)\} \cup T$ and our relation could take a command and a state and result in a (final) state: $(c, \alpha) \rightarrow^* \alpha'$.

We call a TTS either *relational* (after [14]) or *natural* (after [9]) iff $\forall a \rightarrow b. b \in T$. For natural semantics with $\Gamma = \{< c, \alpha >\} \cup T$ and $T = \{t\}$ we write $\alpha \vdash c \rightarrow t$ instead of $< c, \alpha > \rightarrow t$. This notation is intended to imply the relationship with sequents in the natural deduction style of doing things. The state α is just the set of hypotheses (bindings for variables) on which the value of c relies.

Example: Consider the language

$$< e > \quad ::= \quad < n > \quad | \quad < v > \quad | \quad < e > + < e >$$

where n is a number and v is a variable. Take ρ to be a predefined function from variables to numbers (an environment), then define $\Gamma = \{(e, \rho)\} \cup T$ where $T = \{n\}$. We define $\rightarrow$ inductively as:

$$\rho \vdash n \rightarrow n$$

$$\frac{\rho v = n}{\rho \vdash v \longrightarrow n}$$

$$\frac{\rho \vdash e_1 \longrightarrow n \qquad \rho \vdash e_2 \longrightarrow n' \qquad m = n + n'}{\rho \vdash e_1 + e_2 = m}$$

The intuition behind these rules is as follows. The first rule reduces a number to itself. The second rule states that v evaluates to n just when it has the value n in the environment. The second rule evaluates each subexpression and then adds their values. The $+$ sign above the line is addition over numbers, whereas the $+$ below the line is a part of the syntax of the language. This type of overloading will be used when no confusion can arise. $\diamondsuit$

We call systems written in the style of the last example *natural semantics.* Let us be a little more precise about the elements of a natural semantics, following [9].

Variables: We allow variables to be used whose intuitive meaning is "any element of the configuration", although some variables may allow only limited configurations. For example, if a configuration comprises several components, there may be variables over entire configurations as well as over a single component. As well, a component may be something (like an abstract syntax tree) which will allow even more fine-grained variables. When variables occur in rules they must have well-defined types. In most cases it is obvious (to us, not necessarily to a machine) what the types of variables are, and they are not given. In cases where confusion can arise, types are given explicitly.

Formulae: Elements of a transition relation, possibly containing variables, will be known as formulae (for example $\rho \vdash e \longrightarrow n$). As well, we allow formulae called *conditions* which are predicates which take values from some domain such as integer (for example $w \neq n$ over variable names). For simple predicates, such as equality over integers, no reference need be made to the domain. More complicated predicates must be defined (by a set of rules).

Rules: A rule consists of a set of formulae called the *hypotheses* and a formula called the *conclusion.* Variables are universally quantified over the entire rule, so that like variables in one or more hypotheses and possibly the conclusion are all instantiated to the same value. For simplicity, we do not write the universal quantifications, unless the types of the variables cannot be inferred. The purpose of a rule is to state an "if-then" relation — namely, if there is a proof of each hypothesis, then there is a proof of the conclusion. Rules may also be conditional as discussed above.

Axioms are rules with no hypotheses.

Rule Sets: Rules are grouped together to form rule sets. Each rule set defines one relation. To distinguish different rule sets two notations may be used. First, different symbols for $\rightarrow$ may be used. These include $\Rightarrow$, $\triangleright$, $\leadsto$ and others. Second, the name of the rule set may occur over the $\vdash$ symbol. A rule set completely defines a transition relation. It may be parameterized by other rule sets, i.e., a hypothesis of a rule may be a formulae from a different rule set.

Each rule set defines a unique relation (possibly non-deterministic as more than one rule may apply at a given time). New relations may be defined in terms of old ones simply by referencing the rule set used. We will make extensive use of this throughout the rest of this paper.

We will make use of both mathematical induction (over the length of proofs) and rule induction. We assume a familiarity with the first. Rule induction is defined below.

Definition 2 The principle of Rule Induction: Let A be the least set closed under a set of rules of the form $\frac{X}{t}$. Let P be a property over elements of A. If for all rules $\frac{X}{t}$, $(\forall x \in X.P(x)) \Rightarrow P(t)$, then $\forall a \in A.P(a)$. $\diamondsuit$

The rule sets we use generate the least set closed under the rules. Thus, we simply need to show that a property holds for the axioms, and then for the other rules, assume it holds for the hypotheses and then show it it holds for the conclusion.

4 miniSML

We define the abstract syntax of miniSML and then we give it a natural semantics.

4.1 miniSML Source

The abstract syntax of miniSML programs is defined in Figure 1. miniSML is the untyped lambda calculus with constants and pairing (we consider Lisp-like lists to be made up of pairs). miniSML uses strict evaluation—arguments are evaluated before the function call.

As the standard example, the factorial function in miniSML would be written as:

```
LETREC fac = λ n.  IF n=0 THEN 1 ELSE n*(fac(n-1)) IN fac
```

An untyped list is constructed as, for example

```
CONS 3 (CONS T NIL)
```

which we write in shorthand as `(3,T)`. As is usual, `CAR` selects the first element of a list and `CDR` returns a list without the first element.

There are many pieces of abstract syntax which are not meaningfull. We assign meanings to those that are, by using a natural semantics.

```
    Const ::=   T | F | NIL | string | integer

      Var ::=   alphabetic alphanumeric*

     VarL ::=   Var | (Var, VarL)

  miniSML ::=   Var
            |   Const
            |   Op1 miniSML
            |   miniSML Op2 miniSML
            |   IF miniSML THEN miniSML ELSE miniSML
            |   λ VarL. miniSML
            |   LET VarL = miniSML IN miniSML
            |   LETREC VarL = miniSML IN miniSML
            |   miniSML miniSML
            |   (miniSML, miniSML)

      Op1 ::=   ATOM | CAR | CDR

      Op2 ::=   +| − | * | = | <= | CONS
```

Figure 1: miniSML Abstract Syntax

```
    ConstL ::=   Const | (Const, ConstL)

    L.WHNF ::=   ConstL | L.Closure

 L.Closure ::=   λ VarL. miniSML × L.Env

     L.Env ::=   () | ( Var × L.WHNF ) × L.Env | L.Env × L.Env
```

Figure 2: miniSML WHNF's and Environments

4.1.1 miniSML Operational Semantics

Our operational semantics will reduce a miniSML program to a miniSML Weak-Head-Normal-Form (L.WHNF) which is defined in Figure 2. A L.WHNF is either a constant, a tuple of constants, or a closure. A closure pairs a miniSML program with an environment.

A L.Env is an environment which gives values variables. All values in the environment are in L.WHNF as miniSML is strict. The environment is stored as a list of lists to mimic the SECD machine operation. This does not add much complexity here and makes the proof easier.

Notation: For structures of the sort $X \times Y$ we use the standard functional programming syntax $x :: y$ for x of type X and y of type Y. This notation is used when the intention for the use of the structure is as a list. If all that is intended is a pair, then (x, y) may be used. We will use $(e_1, e_2, ..., e_n)$ or even $(e_1...e_n)$ to represent $(e_1, (e_2, (...e_n)))$ where the e_i are either miniSML expressions or variables.

Updates to a L.Env ρ are written as $((e_1, x_1)...(e_n, x_n)) :: \rho$ or $\rho[e_1/x_1...e_n/x_n]$ where the e_i are constants or closures and the x_i are variable names.

Definition 3 A *proper L.Closure* is one in which all free variables in the code have a value in the environment. In our case, we require that the following rule be derivable:

$$Proper \quad \frac{e \rightarrow^{FV} s \qquad \rho \rightarrow^{DefinedBy} s' \qquad s \subseteq s'}{\rho \vdash e \rightarrow true}$$

FV extracts the free variables in a miniSML program.

FV_{Const}	$c \rightarrow \emptyset$
FV_{var}	$a \rightarrow \{a\}$
FV_{op1}	$\dfrac{e_1 \rightarrow s_1}{OP_1\ e_1 \rightarrow s_1}$
FV_{op2}	$\dfrac{e_1 \rightarrow s_1 \qquad e_2 \rightarrow s_2}{OP_2\ e_1\ e_2 \rightarrow s_1 \cup s_2}$
FV_{if}	$\dfrac{e_1 \rightarrow s_1 \qquad e_2 \rightarrow s_2 \qquad e_2 \rightarrow s_3}{IF\ e_1\ THEN\ e_2\ ELSE\ e_3 \rightarrow s_1 \cup s_2 \cup s_3}$
FV_{abs}	$\dfrac{body \rightarrow s}{\lambda(a_1...a_k).body \rightarrow s/\{a_1...a_k\}}$
FV_{let}	$\dfrac{e_1 \rightarrow s_1 \quad ... \quad e_k \rightarrow s_k \qquad body \rightarrow s}{LET\ a_1 = e_1...a_k = e_k\ IN\ body \rightarrow (s/\{a_1...a_k\}) \cup s_1 \cup ... \cup s_k}$
FV_{rec}	$\dfrac{e_1 \rightarrow s_1 \quad ... \quad e_k \rightarrow s_k \qquad body \rightarrow s}{LETREC\ a_1 = e_1...a_k = e_k\ IN\ body \rightarrow (s \cup s_1 \cup ... \cup s_k)/\{a_1...a_k\}}$
FV_{app}	$\dfrac{f \rightarrow s \qquad a \rightarrow s_1}{f\ a \rightarrow s \cup s_1}$

$DefinedBy$ extracts the set of variables defined by a given environment.

$DefinedBy_{()}$	$() \rightarrow \emptyset$
$DefinedBy_{list}$	$\dfrac{L \rightarrow s}{() :: L \rightarrow s}$
$DefinedBy_{element}$	$\dfrac{L :: L' \rightarrow s}{((x, x') :: L) :: L' \rightarrow \{x\} \cup s}$

Unless stated otherwise, we will assume that all closures in an L.Env are proper ones. This does not imply that all the closures we form in the semantics are proper, just that when they are used they will be proper. For closures of recursive functions, the original closure will not contain definitions for the recursive function names. However, before any of these closures is used the environment portion will be updated to include those definitions.

Notation: The closure of code e with environment ρ is written $\ll e, \rho \gg$.

4.2 Semantics of miniSML

Our semantics are given as a reduction of a miniSML program to a L.WHNF. The reduction is given by a relation, called $\rightarrow$ which is a subset of the set *((L.Env* $\times$ *miniSML)* $\times$ *L.WHNF)*. We give this reduction as a natural deduction in Table 1. The reduction relation is slightly complicated by the addition of "times". A time over $\rightarrow$ gives a measure of the number of steps it takes to reduce the expression to L.WHNF. This measure is not simply the number of rules applied in a particular proof; it is related to execution steps on the SECD machine. For example, note that the IF statement takes a "time" which is the sum of the "times" for the two subexpressions (either e_1 and e_2 or e_1 and e_3) plus two (as we will see, two SECD instructions are needed to control the IF statement).

To be technically correct, the "times" must be included in the set we are defining, making $\rightarrow$ a subset of *((L.Env* $\times$ *minSML)* $\times$ *(L.WHNF* $\times$ *Integer)*. Then we would have to project out the L.WHNF component in order to make comparisons between L.WHNF, as times should not affect the equality of L.WHNF's. We will ignore this (small) technicality.

The rules L_{Li} are miniSML Lookup rules (thus the two L's). They are given with a subsidiary relation $\rightarrow^L \subseteq$ *(L-env* $\times$ *Var)* $\times$ *miniSML*. The environment is kept as a list of lists of pairs *(variable-name,value)*. Although only a single list is needed, this structure does not add any great complexity and makes our proof statement easier. L_{L1} checks the "newest" pair in the environment, and if this is the variable we are looking up, returns its value. L_{L2} and L_{L3} step down the environment, ignoring values which do not match the current variable. Note that there is no case for an empty environment, which we would expect to be an error. By leaving the case out, there will be no proof of an expression which has a free variable. Therefore, expressions with free variables are not given meanings by our semantics. (Note that this is not a problem with recursive expressions as the environment in the closures will contain definitions for the recursive functions before the body of the closure is evaluated).

L_{CONST} simply reduces a constant to its value in one time step. An abstraction, L_{ABS} is mapped to a closure of the abstraction with the current environment. The hypothesis ensures that we have a proper closure. L_{VAR} makes use of $\rightarrow^L$ to find the value of a variable in the current environment.

L_{OP_2} is the rule for all binary operators. For each OP there is a corresponding *op* which performs the operation over the constants, for example addition corresponds to $+$. L_{CAR}, L_{CDR} and L_{CONS} work as expected over lists of miniSML elements. L_{ATOM} returns true if the argument is a constant and false otherwise. L_{IF} reduces the first argument to true or false, and then selects a branch.

L_{APP} handles application. Since miniSML is strict, all the arguments are evaluated before the application. As well, the function is evaluated to a closure. Then, if we add the evaluated arguments to the environment from the closure, the closure body will evaluate to the result of the application. L_{LET} works similarly, except that the body of the LET already provides the the expected closure body.

L_{REC} handles recursive function definitions. Each expression in the binding evaluates to a function closure with the same environment. We evaluate the body of the LETREC in a circular (graph) environment as shown.

5 SECD

In this section we define the SECD machine's abstract syntax and natrual semantics.

5.1 Code for Henderson's SECD machine

The abstract syntax of Henderson's SECD machine code is given in Figure 3. The SECD machine executes on four stacks, the Stack, Environment, Code, and Dump. The Stack contains arguments, the Environment values for "variables", the Code the SECD instructions to execute, and the Dump stores the other stacks while functions or arms of conditionals are being executed. Operators are evaluated postfix and expect their arguments to be on the Stack.

S.Const ::=	**integer** | **string** | **T** | **F** | **NIL**
SECD ::=	**LD (integer, integer)**
|	**LDC** S.Const
|	**LDF** SECD
|	**SEL** (SECD **JOIN**), (SECD **JOIN**)
|	S.Op1 | S.Op2 | **AP** | **DUM** | **RAP** | **RTN** | **STOP**
S.Op1 ::=	**ATOM** | **CAR** | **CDR**
S.Op2 ::=	**CONS** | **EQ** | **LEQ** | **ADD** | **SUB** | **MUL** | **DIV** | **REM**

Figure 3: SECD Abstract Syntax

5.2 SECD Natural Semantics

Our reduction for SECD code will, like miniSML, reduce to SECD WHNF's. The abstract syntax for these are given in Figure 4. S.WHNF's include contants and closures. S.Env is a list of lists of values indexed by a pair of integers ((0,0) being the first element in the top list).

L_{L1}	$((v,v')::\rho')::\rho \vdash v \rightarrow^L v'$		
L_{L2}	$\dfrac{\rho \vdash v \rightarrow^L v'}{[\,]::\rho \vdash v \rightarrow^L v'}$	L_{L3}	$\dfrac{\rho' :: \rho \vdash v \rightarrow^L v' \qquad x \neq v}{((x,x')::\rho')::\rho \vdash v \rightarrow^L v'}$
L_{CONST}	$\rho \vdash c \xrightarrow{1} c$	$c \in Const$	
L_{ABS}	$\rho \vdash (\lambda\ (x_1 ... x_k)\ b) \xrightarrow{1} \ll \lambda\ (x_1 ... x_k)\ b, \rho \gg$		
L_{VAR}	$\dfrac{\rho \vdash v \rightarrow^L v'}{\rho \vdash v \xrightarrow{1} v'}$		
L_{OP_2}	$\dfrac{\rho \vdash e_1 \xrightarrow{t_1} a_1 \qquad \rho \vdash e_2 \xrightarrow{t_2} a_2}{\rho \vdash e_1\ OP\ e_2 \xrightarrow{t_1+t_2+1} a_1\ op\ a_2}$	For $+, -, *, \leq$, $a_1, a_2 \in integer$	
L_{CAR}, L_{CDR}	$\dfrac{\rho \vdash e_1 \xrightarrow{t_1} (x_1 x_2 ... x_n)}{\rho \vdash CAR\ e_1 \xrightarrow{t_1+1} x_1}$ $\quad n \geq 1$	$\dfrac{\rho \vdash e_1 \xrightarrow{t_1} (x_1 x_2 ... x_n)}{\rho \vdash CDR\ e_1 \xrightarrow{t_1+1} (x_2 ... x_n)}$	$n \geq 1$
L_{CONS}	$\dfrac{\rho \vdash e_1 \xrightarrow{t_1} x_1 \qquad \rho \vdash e_2 \xrightarrow{t_2} (x_2 ... x_n)}{\rho \vdash CONS\ e_1\ e_2 \xrightarrow{t_1+t_2+1} (x_1 x_2 ... x_n)}$		
L_{ATOM}	$\dfrac{\rho \vdash e_1 \xrightarrow{t_1} c \quad c \in Const \cup Var}{\rho \vdash ATOM\ e_1 \xrightarrow{t_1+1} T}$	$\dfrac{\rho \vdash e_1 \xrightarrow{t_1} x}{\rho \vdash ATOM\ e_1 \xrightarrow{t_1+1} F}$	
L_{IFt}, L_{IFf}	$\dfrac{\rho \vdash e_1 \xrightarrow{t_1} T \qquad \rho \vdash e_2 \xrightarrow{t_2} e_2'}{\rho \vdash IF\ e_1\ e_2\ e_3 \xrightarrow{t_1+t_2+2} e_2'}$	$\dfrac{\rho \vdash e_1 \xrightarrow{t_1} F \qquad \rho \vdash e_3 \xrightarrow{t_3} e_3'}{\rho \vdash IF\ e_1\ e_2\ e_3 \xrightarrow{t_1+t_3+2} e_3'}$	
L_{APP}	$\dfrac{\begin{array}{c}\rho \vdash e_1 \xrightarrow{t_1} e_1' \quad ... \quad \rho \vdash e_k \xrightarrow{t_k} e_k' \\ \rho \vdash f \xrightarrow{t} \ll \lambda(x_1 ... x_k).f', \rho' \gg \quad \rho'[e_1'/x_1 ... e_k'/x_k] \vdash f' \xrightarrow{t'} r \\ \rho' \overset{Proper}{\vdash} \lambda(x_1 ... x_k).f' \rightarrow true\end{array}}{\rho \vdash f\ (e_1 ... e_k) \xrightarrow{t_1+...+t_k+t+t'+k+4} r}$		
L_{LET}	$\dfrac{\rho \vdash e_1 \xrightarrow{t_1} e_1' \quad ... \quad \rho \vdash e_k \xrightarrow{t_k} e_k' \quad \rho[e_1'/x_1 ... e_k'/x_k] \vdash b \xrightarrow{t} b'}{\rho \vdash LET\ x_1 = e_1 ... x_k = e_k\ in\ b \xrightarrow{t_1+...+t_k+k+t+4} b'}$		
L_{REC}	$\dfrac{\begin{array}{c}\rho \vdash e_1 \xrightarrow{t_1} \ll e_1', \rho \gg \quad ... \quad \rho \vdash e_k \xrightarrow{t_k} \ll e_k', \rho \gg \\ \rho' \vdash b \xrightarrow{t} b' \quad where \quad \rho' = \rho[\ll e_1', \rho' \gg /x_1 \quad ... \quad \ll e_k', \rho' \gg /x_k]\end{array}}{\rho \vdash LETREC\ x_1 = e_1 ... x_k = e_k\ in\ b \xrightarrow{t_1+...+t_k+t+k+5} b'}$		

Table 1: Reduction Rules for miniSML

```
S.ConstL ::=   S.Const | (S.Const, S.ConstL)

S.Closure ::=  SECD x S.Env

 S.WHNF ::=    S.ConstL | S.Closure

   S.Env ::=   () | S.WHNF x S.Env | S.Env x S.Env

   Stack ::=   () | S.WHNF x Stack

    Dump ::=   () | SECD x Dump>| Stack x S.Env x SECD x Dump
```

Figure 4: SECD WHNF's, Environments, and Dumps

Notation: We use | to indicate list concatenation for SECD expressions. This should not be confused with | used in the BNF notation. In order to simplify the notation, we often eliminate brackets (i.e. writing $s_1|OP$ instead of $s_1|(OP)$).

Like miniSML, not all closures are useful.

Definition 4 A S.Closure is *proper* if the following rule is derivable:

$$S.Proper \quad \frac{c \rightarrow^{VarsIn} s \qquad e \rightarrow^{Cover} s' \qquad s \subset s'}{e \vdash c \rightarrow true}$$

$VarsIn$ extracts all the free variables (pairs of integers) in a Code stream and $Cover$ extracts all variables (pairs of integers) defined in an S.Env. (In $VarsIn$ we see the first use of | as Code concatenation as opposed to BNF syntax).

$VarsIn_{()}$	$() \rightarrow \emptyset$
$VarsIn_{LD}$	$\dfrac{c \rightarrow s}{LD(m,n) \mid c \rightarrow \{(m,n)\} \cup s}$
$VarsIn_{Other}$	$\dfrac{c \neq LD(m,n) \qquad c' \rightarrow s}{c \mid c' \rightarrow s}$
$Cover$	$\dfrac{(E,0) \rightarrow^{Cover'} s}{E \rightarrow s}$
$Cover'_{()}$	$((),m) \rightarrow \emptyset$
$Cover'_{::}$	$\dfrac{(E,m+1) \rightarrow s}{((e_1,...e_n) :: E,m) \rightarrow \{(m,0),...,(m,n)\} \cup s}$

$\diamondsuit$

S.Closures differ from L.Closures in an important respect. An L.Closure contains the miniSML code for a function (i.e., a λ expression) whereas a S.Closure contains only the body of the function; the variable abstraction is assumed. Further, the return from a L.Closure is implicit whereas an S.Closure must explicitly contain the RTN instruction.

For the SECD, **replcar** is the standard name for the function which puts a loop in the environment for recursive functions. It is defined as:

Definition 5

$$\begin{array}{rcl} \textbf{replcar}\ ([\ll e_1, \rho \gg, ..., \ll e_k, \rho \gg], [] :: \rho') & = & \rho'' \\ \text{whererec}\ \rho'' = [\ll e_1, \rho'' \gg, ..., \ll e_k, \rho'' \gg] :: \rho' & & \end{array}$$

$\diamondsuit$

The reduction for SECD is now given in Table 2. The reduction is given by
$\Rightarrow \subset$ ((Stack $\times$ S-Env $\times$ Dump) $\times$ SECD) $\times$ S-WHNF.

The S_{Li} rules provide the auxilary relation $\Rightarrow^L \subset$ (S-Env $\times$ (Int $\times$ Int)) $\times$ S-WHNF. S-Env is a list of lists of S-WHNF's as described earlier. $(0,0)$ indexes the leftmost element of the topmost list, and (m,n) can be reduced to $(0,0)$ by stripping off an appropriate number of lists and elements. Again, there is no rule for expressions with free variables, and the semantics does not assign them any meaning.

S_{STOP} is the only axiom in the system. It returns the value on top of the stack.

S_{LD} finds a value in the environment using $\Rightarrow^L$ and puts this value on top of the stack. S_{LDC} puts a constant on top of the stack, and S_{LDF} creates a closure from the associated code and the current environment, and puts this closure on top of the stack.

Application, S_{AP}, expects a closure (the function) and arguments on top of the stack. It adds the arguments to the closure environment and then executes the closure code in this new environment, saving the return pointers on the dump. S_{RTN} expects these return pointers to be on the dump, and uses them to reset the stacks. S_{DUM} and S_{RAP} perform application for recursive functions. S_{DUM} puts an nil value on top of the environment so that variable accesses (except for the recursive bindings) are at least one level deep in the environment. This allows S_{RAP} to add the recursive bindings in place of the nil value without affecting other variable accesses. The **replcar** command makes sure the addition of the recursive bindings works correctly by making the environment for the new closures circular.

S_{SEL1} and S_{SEL2} handle the IF construct for the SECD machine. The code for the "then" and "else" branches are contained in the SEL command and one of them is evaluated depending on the (truth value) on top of the stack. While the branch is being evaluated the code continuation is stored on the dump, to be picked up by the S_{JOIN} rule.

S_{L1}	$((x :: e) :: e') \vdash (0,0) \Rightarrow^{L} x$		
S_{L2}	$\frac{(e' :: e) \vdash (0,n) \Rightarrow^{L} x}{((y :: e') :: e) \vdash (0,n+1) \Rightarrow^{L} x}$	S_{L3}	$\frac{e \vdash (m,n) \Rightarrow^{L} x}{(e' :: e) \vdash (m+1,n) \Rightarrow^{L} x}$
S_{STOP}	$(x :: s), e, d \vdash STOP \Rightarrow^{1} x$	S_{SEL1}	$\frac{s, e, (c.d) \vdash cT \Rightarrow^{t} r}{(T :: s), e, d \vdash SEL(cT, cF) \mid c \Rightarrow^{t+1} r}$
S_{LD}	$\frac{(x :: s), e, d \vdash c \Rightarrow^{t} r \quad e \vdash (m.n) \Rightarrow^{L} x}{s, e, d \vdash LD(m.n) \mid c \Rightarrow^{t+1} r}$	S_{SEL2}	$\frac{s, e, (c.d) \vdash cF \Rightarrow^{t} r \quad x \neq T}{(x :: s), e, d \vdash SEL(cT, cF) \mid c \Rightarrow^{t+1} r}$
S_{LDC}	$\frac{(x :: s), e, d \vdash c \Rightarrow^{t} r}{s, e, d \vdash LDC \ x \mid c \Rightarrow^{t+1} r}$	S_{JOIN}	$\frac{s, e, d \vdash c \Rightarrow^{t} r}{s, e, (c :: d) \vdash JOIN \Rightarrow^{t+1} r}$
S_{LDF}	$\frac{(\ll c'.e \gg :: s), e, d \vdash c \Rightarrow^{t} r}{s, e, d \vdash LDF \ c' \mid c \Rightarrow^{t+1} r}$	S_{CAR}	$\frac{(a :: s), e, d \vdash c \Rightarrow^{t} r}{((a :: b) :: s), e, d \vdash CAR \mid c \Rightarrow^{t+1} r}$
S_{AP}	$\frac{\begin{array}{c} e' \overset{S.Proper}{\vdash} c' \rightarrow true \\ nil, (v :: e'), ((s, e, c) :: d) \vdash c' \Rightarrow^{t} r \end{array}}{(\ll c'.e' \gg :: v :: s), e, d \vdash AP \mid c \Rightarrow^{t+1} r}$	S_{CDR}	$\frac{(b :: s), e, d \vdash c \Rightarrow^{t} r}{((a :: b) :: s), e, d \vdash CAR \mid c \Rightarrow^{t+1} r}$
S_{DUM}	$\frac{s, [] :: e, d \vdash c \Rightarrow^{t} r}{s, e, d \vdash DUM \mid c \Rightarrow^{t+1} r}$	S_{ATOM}	$\frac{(atom(a) :: s), e, d \vdash c \Rightarrow^{t} r}{(a :: s), e, d \vdash ATOM \mid c \Rightarrow^{t+1} r}$
S_{RTN}	$\frac{x :: s, e, d \vdash c \Rightarrow^{t} r}{(x), e', ((s, e, c) :: d) \vdash RTN \Rightarrow^{t+1} r}$	S_{OP}	$\frac{((b \ op \ a) :: s), e, d \vdash c \Rightarrow^{t} r}{(a :: b :: s), e, d \vdash OP \mid c \Rightarrow^{t+1} r}$
S_{CONS}	$\frac{(a :: b :: s), e, d \vdash c \Rightarrow^{t} r}{((a :: b) :: s), e, d \vdash CONS \mid c \Rightarrow^{t+1} r}$		
S_{RAP}	$\frac{nil, e'', ((s, e, c) :: d) \vdash c' \Rightarrow^{t} r \quad e'' = replcar(v, e')}{(\ll c'.e' \gg :: v :: s), [] :: e, d \vdash RAP \mid c \Rightarrow^{t+1} r}$		
S_{CONT}	$\frac{s, e, d \vdash e_1 \mid STOP \Rightarrow^{t_1} e_1' \quad e_1' :: s, e, d \vdash e_2 \Rightarrow^{t_2} r}{s, e, d \vdash e_1 \mid e_2 \Rightarrow^{t_1+t_2} r}$		

Table 2: SECD Reduction Rules

$S_{CAR}, S_{CDR}, S_{ATOM}, S_{CONS}$ [3] and S_{OP} operate on the value(s) on top of the stack and replace these value(s) by the result of the operation.

The final rule, S_{CONT} is not a part of the original Henderson specification but is necessary here as we are working in a logical system with only one axiom. Thus, to join two proofs, we must allow the axiom to be used in each proof. S_{CONT} says that if we have two complete SECD programs, we can execute the second as a continuation of the first by evaluating the first program, leaving its result on top of the stack and then evaluating the second. A STOP command is added to the first program, since STOP is the only command which is an axiom in this system (and thus must be the final command in any well-formed program or there would be no finite proof of it).

[3]In Henderson's formulation the arguments to CONS are reversed for reasons of efficiency. We do not do this here in order to keep the the exposition as simple as possible

6 Translation

We wish to translate miniSML programs to SECD programs. We want to handle two cases: (1) M reduces to a L.WHNF in a finite number of steps, and (2) M never reduces to a L.WHNF, but there is always a rule which can be applied. These two cases encompass all well-defined miniSML programs. The translation from miniSML to SECD is given in Table 3.

The translation environment is kept as a list of lists (as is the SECD environment), but we need only store variable names in the list. Translating a variable requires finding a (m, n) pair where m indicates the list the variable is in, and n indicates the variable's position in that list. The relation $\hookrightarrow^L$ mimics finding the (m, n) pair.

The first rule under $\hookrightarrow$ shows how any LAM constant is translated to a LDC instruction.

For variables, T_{VAR} uses $\hookrightarrow^L$ to find the pair (m, n) and then emits a LD instruction. T_{OP_2} and T_{OP_1} both compile to postfix application. T_{IF} translates an IF statement so that the boolean is evaluted first, leaving its result on top of the stack, and then SEL is used, in conjunction with JOIN, to evalute a given branch. T_{LAMBDA} compiles a function into a LDF where the body has a RTN appended to it. Function application T_{APP} is translated to a sequence which creates a list of the arguments on the stack, loads the function and then performs the AP. The arguments are in reverse order because the list is built up from the end. The LET is translated to the equivalent of a λ and an application. T_{REC} translates recursive functions by using DUM and RAP instead of the AP which occurs for the LET. Notice that for the body of the LET and for all components of the REC, the translation environment is updated with the new variable locations.

We can now define a translation, in terms of $\hookrightarrow$ between L.WHNF's and S.WHNF's. The translation is given in table 4. The function $\triangleright$ performs the mapping between WHNF's while $\triangleright_E$ maps $\triangleright$ throughout an environment.

In order to define this, we must be able to "extract" a translation environment from a L.Env. This amounts to keeping the variable names, while throwing away the value. This is given by the $\triangleright_T$ transitions.

$\triangleright_{const}$ indicates that constants are the same in L.WHNF as in S-WHNF.

Mapping closures is more involved. We have broken this down into two cases, one where the environment is circular, and one where it is not. The non-recursive $\triangleright_{clos}$ is quite straightforward. If the L.Env ρ maps to the S.Env E, and ρ gives the translation environment ϱ, and compiling the body of the function in ϱ with the variables from the LAMBDA gives b', then the L.Closure maps to a S-Closure with code $(b'|RTN)$ and environment E.

$\triangleright_{recclos}$ requires a little more work, as we change a circular L.Env to a circular

T_{L1}	$(x :: \varrho') :: \varrho \vdash x \hookrightarrow^{L} (0,0)$
T_{L2}	$\dfrac{\varrho' :: \varrho \vdash x \hookrightarrow^{L} (0,n) \quad x \neq y}{(y :: \varrho') :: \varrho \vdash x \hookrightarrow^{L} (0,n+1)}$
T_{L3}	$\dfrac{\varrho \vdash x \hookrightarrow^{L} (m,n)}{([\,] :: \varrho) \vdash x \hookrightarrow^{L} (m+1,n)}$
T_{L4}	$\dfrac{\varrho' :: \varrho \vdash x \hookrightarrow^{L} (m,n) \qquad x \neq y \qquad m \neq 0}{(y :: \varrho' :: \varrho) \vdash x \hookrightarrow^{L} (m,n)}$
T_{CONST}	$\varrho \vdash n \hookrightarrow LDC\ n$
T_{VAR}	$\dfrac{\varrho \vdash x \hookrightarrow^{L} (m,n)}{\varrho \vdash x \hookrightarrow LD(m,n)}$
T_{OP2}	$\dfrac{\varrho \vdash e_1 \hookrightarrow s_1 \quad \varrho \vdash e_2 \hookrightarrow s_2}{\varrho \vdash OP^{L}\ e_1\ e_2 \hookrightarrow s_1 \mid s_2 \mid OP^{S}}$
T_{OP1}	$\dfrac{\varrho \vdash e_1 \hookrightarrow s_1}{\varrho \vdash OP\ e_1 \hookrightarrow s_1 \mid OP}$
T_{IF}	$\dfrac{\varrho \vdash e_1 \hookrightarrow s_1 \quad \varrho \vdash e_2 \hookrightarrow s_2 \quad \varrho \vdash e_3 \hookrightarrow s_3}{\varrho \vdash IF\ e_1 e_2 e_3 \hookrightarrow s_1 \mid SEL((s_2 \mid JOIN, (s_3 \mid JOIN)))}$
T_{LAMBDA}	$\dfrac{[x_1, ..., x_k] :: \varrho \vdash b \hookrightarrow s}{\varrho \vdash \lambda(x_1 ... x_k) b \hookrightarrow LDF \mid (s \mid RTN)}$
T_{APP}	$\dfrac{\varrho \vdash e_1 \hookrightarrow s_1 ... \varrho \vdash e_k \hookrightarrow s_k \quad \varrho \vdash f \hookrightarrow s_f}{\varrho \vdash f\ e_1 ... e_k \hookrightarrow (LDC\ NIL) \mid s_k \mid CONS \mid ... \mid s_1 \mid CONS \mid s_f \mid AP}$
T_{LET}	$\dfrac{\varrho \vdash e_1 \hookrightarrow s_1 ... \varrho \vdash e_k \hookrightarrow s_k \quad [x_1, ..., x_k] :: \varrho \vdash b \hookrightarrow s_b}{\begin{array}{c} \varrho \vdash LET\ x_1 = e_1 ... x_k = e_k\ in\ b \\ \hookrightarrow (LDC\ NIL) \mid s_k \mid CONS \mid ... \mid s_1 \mid CONS \mid (LDF(s_b \mid RTN)) \mid AP \end{array}}$
T_{REC}	$\dfrac{[x_1, ..., x_k] :: \varrho \vdash e_1 \hookrightarrow s_1 ... [x_1, ..., x_k] :: \varrho \vdash e_k \hookrightarrow s_k \quad [x_1, ..., x_k] :: \varrho \vdash b \hookrightarrow s_b}{\begin{array}{c} \varrho \vdash LETREC\ x_1 = e_1 ... x_k = e_k\ in\ b \\ \hookrightarrow DUM \mid (LDC\ NIL) \mid s_k \mid CONS \mid ... \mid s_1 \mid CONS \mid (LDF(s_b \mid RTN)) \mid RAP \end{array}}$

Table 3: miniSML translation schema

$\triangleright_{const}$	$c \triangleright c$ where $c \in$ Const
$\triangleright_{clos}$	$\dfrac{\begin{array}{c}\rho \triangleright_E E \\ \rho \triangleright_T \varrho \\ ([x_1 ... x_k] :: \varrho) \vdash b \hookrightarrow b'\end{array}}{\ll LAMBDA(x_1 ... x_k)\ b, \rho) \gg\ \triangleright \ll (b' \vert RTN), E \gg}$
$\triangleright_{recclos}$	$\dfrac{\begin{array}{c}\rho \triangleright_E E \\ \rho \triangleright_T \varrho \\ [y_1, ..., y_j] :: \varrho \vdash b \hookrightarrow b' \\ \rho' = [(\ll e_1, \rho' \gg, y_1), ..., (\ll e_j, \rho' \gg, y_j)] :: \rho \\ \ll e_1, \rho'' \gg\ \triangleright e'_1 ... \ll e_j, \rho'' \gg\ \triangleright e'_j \qquad where\ \ \rho'' = [(0, y_1), ..., (0, y_j)] :: \rho\end{array}}{\ll b, \rho' \gg\ \triangleright \ll (b' \vert RTN), replcar([e'_1, ..., e'_j], ([] :: E)) \gg}$
$\triangleright_{E0}$	$[] \triangleright_E []$
$\triangleright_{E1}$	$\dfrac{\begin{array}{c}\rho \triangleright_E E \\ w_1 \triangleright w'_1 ... w_n \triangleright w'_n\end{array}}{([w_1, ..., w_n] :: \rho) \triangleright_E ([w'_1, ..., w'_n] :: E)}$
$\triangleright_{T0}$	$[] \triangleright_T []$
$\triangleright_{T1}$	$\dfrac{\rho \triangleright_T \varrho}{[(x_1, x'_1), ..., (x_n, x'_n)] :: \rho \triangleright_T [x_1, ..., x_n] :: \varrho}$

Table 4: miniSML WHNF's to SECD WHNF's

S.Env. Again, ρ is equivalent to E, and our translation environment is ϱ, and the body compiles to b'. Further, each (recursive) closure in ρ must be changed to the equivalent SECD closure. These closures contain circular environments which we must eliminate if our mapping is to terminate. Since we only require the positions of the recursive bindings in order to define this mapping (and the values associated with these bindings are eliminated by $\triangleright_T$ anyway), we can put arbitrary non-recursive values in for the recursive bindings before we translate them. This is why ρ'' is defined with the arbitrary value zero in for the recursive bindings.

Before continuing it is worth noting that the translation halts for all well-formed miniSML expressions. We simply outline this proof as it is quite standard.

Theorem 1 *For any miniSML program M such that there exists a ρ such that $\rho \vdash M \longrightarrow W$ for some L.WHNF W, the translator applied to M in environment ϱ, where $\rho \triangleright_T \varrho$, halts in finitely many steps.*

Proof: By rule induction.

Base Cases: Translation of constants obviously halts as there is no recursive call. Translation of variables halts if $\hookrightarrow^L$ halts. By the assumption that M has a WHNF we know that ϱ will be defined over any variable in M, say x. The base case for x, when it is the top element of the environment halts with value $(0, 0)$. For each of the other rules, we assume by the inductive hypothesis that $\hookrightarrow^L$ halts for each premiss, in which case it obviously halts for the conclusions as these only perform simple additions.

Inductive cases: We assume that $\hookrightarrow$ halts for the premisses. Since each conclusion simply rearranges results from its premisses, these also halt. $\Box$.

7 Proof of the Translation

Several more definitions are needed before we can give a satisfactory proof statement.

Definition 6 A TS $< \Gamma, \longrightarrow >$ is *deterministic* if $\gamma \rightarrow \gamma' \wedge \gamma \rightarrow \gamma''$ implies $\gamma' = \gamma''$. $\Diamond$

A relation is deterministic if it gives at most one value to every input.

Definition 7 Define $\Rightarrow_{run}$ as

$$\frac{\rho \rhd_E E \qquad S, E, D \vdash M|STOP \Rightarrow^t x}{\rho \vdash M \Rightarrow^t_{run} x}$$

$\Diamond$

Thie relation allows us to run an SECD program and extract the result.

We can now state what we wish to prove, namely that the following diagram commutes (the underscores represent the positions the items at the corners take in each sequent):

$$\begin{tikzcd} M \arrow[r, "\rho\vdash\_\hookrightarrow\_"] \arrow[d, "\rho\vdash\_\rightarrow^t\_"'] & M' \arrow[d, "\rho\vdash\_\Rightarrow^t_{run}\_"] \\ S \arrow[r, "\_\rhd\_"'] & S' \end{tikzcd}$$

(Here we use ρ for the translation environment instead of ϱ where $\rho \rhd_T \varrho$ in order to simplify the proofs to follow).

If M has a L.WHNF ($\rightarrow$ terminates) then t is a fixed positive integer, and the diagram will hold with the t on $\Rightarrow$ being the same positive integer. On the other hand (M has no L.WHNF) t will never become fixed, either on $\rightarrow$ or $\Rightarrow$.

7.1 Proof Obligations

How do we go about proving that this diagram commutes? Informally we require that

$$\Rightarrow \circ \hookrightarrow \;\equiv\; \rhd \circ \longrightarrow$$

or in other words, that chasing around the diagram from M to S' in either direction gives the same result. We can break this equivalence into implications in which case we get

$$\exists M'.\frac{\rho \vdash M \longrightarrow^t S \qquad S \rhd S'}{\rho \vdash M \hookrightarrow M' \qquad \rho \vdash M' \Rightarrow^t_{run} S'} \tag{1}$$

$$\exists S.\frac{\rho \vdash M \hookrightarrow M' \qquad \rho \vdash M' \Rightarrow^t_{run} S'}{\rho \vdash M \longrightarrow^t S \qquad S \rhd S'} \tag{2}$$

(1) represents *completeness* and (2) *soundness.* Taking a closer look at completeness and remembering that we are trying to show the compiler correct, it is obvious that some M' exists such that $\rho \vdash M \hookrightarrow M'$, and that this M' must meet the restriction that $\rho \vdash M' \Rightarrow^r_{run} S'$. We can rewrite completeness in a simpler form as

$$\frac{\rho \vdash M \longrightarrow^t S \qquad S \rhd S' \qquad \rho \vdash M \hookrightarrow M'}{\rho \vdash M' \Rightarrow^r_{run} S'} \tag{3}$$

For some cases soundness is equivalent to completeness ([2])

Theorem 2 *If* $\longrightarrow$ *and* $\Rightarrow$ *are deterministic and* $\rhd$ *is one-to-one then soundness is equivalent to completeness.*

Proof: Soundness implies completeness since M' and S' are uniquely determined, and since $\rhd$ is one-to-one, the S in the soundness proof must be the one uniquely determined by $\longrightarrow$ in the completeness proof. Completeness implies soundness as we can choose the S uniquely determined by $\longrightarrow$ in the completeness proof as our S for the soundness proof, and since $\rhd$ is one-to-one, $S \rhd S'$ must hold. □.

In our case, and in many others, $\rhd$ is not one-to-one because of closures. In particular, since closures contain arbitrary expressions within them, two different L.WHNF's may convert to the same S.WHNF (for example, consider a *let* construct and its equivalent application representation).

Since each of soundness and completeness require a large proof effort, we should invest some time to find a way to eliminate one or the other. Here we explore several ways of eliminating one of the above proofs by proving (simpler) properties of our functions.

Lemma 1 *If* $\longrightarrow, \Rightarrow$, *and* $\rhd$ *are deterministic, and* $\Rightarrow$ *is total, then soundness implies completeness.*

Proof: Assume we have a proof of soundness. Then S has been shown to exist such that $\rho \vdash M \rightarrow S$ and $S \triangleright S'$. Since $\rightarrow$ is deterministic, S is unique. Since $\triangleright$ is deterministic, S' is unique. Since $\Rightarrow$ is total, some S'' exists such that $\rho \vdash M' \Rightarrow_{run} S''$, but since $\Rightarrow$ is also deterministic, $S' = S''$. $\Box$.

Lemma 2 *If $\rightarrow, \Rightarrow$, and $\triangleright$ are deterministic, and $\rightarrow$ and $\triangleright$ are total, then completeness implies soundness.*

Proof: Assume we have a proof of completeness. Then we know that we have a unique S, S' by the determinism and totality of $\rightarrow$, $\triangleright$. Since $\Rightarrow$ is also deterministic we know that it too yields S' in the soundness proof. $\Box$.

Theorem 3 *If $\rightarrow, \Rightarrow$ and $\triangleright$ are deterministic* **and** *total, then completeness is equivalent to soundness.*

Proof: From previous two lemmas $\Box$.

At first sight this work does not buy us anything as $\rightarrow$ and $\Rightarrow$ are obviously not total, being undefined for nonterminating programs. But, consider splitting the proof into two cases—one where we only consider programs with L.WHNF's, and the other where we only consider those without L.WHNF's. In the case were a L.WHNF is assumed, the subset of $\rightarrow$ which we are using is, by definition, total. It turns out that $\triangleright$ is total (being defined in terms of $\hookrightarrow$), and thus we could use lemma 2 and only prove completeness when M has a L.WHNF.

In the case where M has no L.WHNF we can rewrite *completeness* and *soundness* as

$$\forall t. \frac{\rho \vdash M \not\rightarrow^{t} \qquad \rho \vdash M \hookrightarrow M'}{\rho \vdash M' \not\Rightarrow^{t}_{run}} \tag{4}$$

$$\forall t. \frac{\rho \vdash M \hookrightarrow M' \qquad \rho \vdash M' \not\Rightarrow^{t}_{run}}{\rho \vdash M \not\rightarrow} \tag{5}$$

both of which we must prove.

We first have to show that the preconditions for Lemma 2 hold.

Lemma 3 *The $\rightarrow$ and $\Rightarrow$ relations are deterministic.*

Proof: For each relation there is one, and only one, rule for each language construct. Thus, only one rule can apply to a given piece of syntax. $\Box$

Lemma 4 *$\triangleright$ is deterministic.*

Proof: $\triangleright$ is defined over L.WHNF's. There is only one case for constants. The two cases for closures are distinct (depending on whether the environment is recursive of not) and are deterministic, as $\hookrightarrow$ (previous lemma) and $\triangleright_E$ and $\triangleright_T$ are deterministic. □

Lemma 5 *$\hookrightarrow$ is total over well-formed miniSML expressions.*

Proof: $\hookrightarrow$ is defined over all possible cases in the abstract syntax for miniSML, except those with free variables. Well-formed miniSML expressions have no free variables. □

Lemma 6 *$\triangleright$ is total (all closures are assumed to be well-formed).*

Proof: Since $\hookrightarrow$ is total, and $\triangleright_E$ and $\triangleright_T$ are obviously total, it follows that $\triangleright$, defined over both constants and closures (which completely cover L.WHNF's) is total. □

Thus we see that for well-formed miniSML programs, $\rightarrow$ is deterministic and $\triangleright$ is deterministic and total.

7.2 WHNF's

Lemma 7 Completeness for Programs with WHNF's:
If x has L-WHNF x' and $x' \triangleright x''$ and x compiles to c, then running c on the SECD machine gives us x''.

$$\boxed{\rho \vdash x \rightarrow^t x'} \; \boxed{x' \triangleright x''} \; \boxed{\rho \vdash x \hookrightarrow c}$$

$$\boxed{\rho \vdash c \Rightarrow^t_{run} x''}$$

Proof: By induction on the length of the proofs over the translation rules.

Base Cases:

T

$\textbf{T}_{\textbf{QUOTE}}$: Quotation is the easiest case. Any constant has the same SECD normal form and miniSML normal form, and runs simply by putting its value on top of the stack.

$$\boxed{\rho \vdash QUOTE\ s \xrightarrow{1} s} \; \boxed{s \triangleright s} \; \boxed{\varrho \vdash QUOTE\ s \hookrightarrow LDC\ s}$$

$$\boxed{\dfrac{\dfrac{s.S, E, D \vdash STOP \Rightarrow^0 s}{S, E, D \vdash LDC\ s | STOP \Rightarrow^1 s}}{\rho \vdash LDC\ s \Rightarrow^1_{run} s}}$$

T$_{\text{VAR}}$: Variables take considerably more work. We use an application of rule induction over the lookup rules. These rules give cases on how an environment can be added to, namely, we can add a single new definition to the current "top-level", or, we can add a series of (possibly recursive) definitions, as the new top-level.

Base Cases:

L_{L1}: This case handles new values added to the top of the environment. We prove a more general form of the rule, namely that we are looking up any one of the values in the top level of the environment. This allows us to break down the proof into two cases, depending on where the environment is recursive or not.

Non-recursive binding: Let $\rho' = \rho[e_1/x_1...e_k/x_k]$. Look up $x_i \in \{x_1...x_k\}$.

$$\frac{\rho' \vdash x_i \rightarrow^L e_i}{\rho' \vdash x_i \rightarrow^1 e_i} \qquad \frac{\begin{matrix}\rho \rhd_E E \\ e_i \rhd e'_i\end{matrix}}{\rho' \rhd_E (e'_1...e'_k) :: E} \qquad \frac{\rho' \vdash x_i \hookrightarrow^L (0,i)}{\rho' \vdash x \hookrightarrow (0,i)}$$

$$\cfrac{\cfrac{(e'_i.S), ((e'_0, ..., e'_n) :: E), D \vdash STOP \Rightarrow^0 e'_i \quad ((e'_0...e'_n) :: E) \vdash (0,i) \Rightarrow^L e'_i}{S, ((e'_0...e'_n) :: E), D \vdash LD(0,i)|STOP \Rightarrow^1 e'_i}}{\rho' \vdash LD(0,i) \Rightarrow^1_{run} e'_i}$$

Recursive binding: Let $\rho' = \rho[\ll e_1, \rho' \gg /x_1, ..., \ll e_1, \rho' \gg /x_1]$. Look up $x_i \in \{x_1...x_k\}$.

$$\frac{\rho' \vdash x_i \rightarrow^L \ll e_i, \rho' \gg}{\rho' \vdash x_i \rightarrow^1 \ll e_i, \rho' \gg} \qquad \begin{matrix}\cfrac{\cfrac{\begin{matrix}(l+1, \sigma[x_0...x_n]) \vdash e_i \hookrightarrow LDF(e^i_c|RTN) \\ \ll e_i, \rho[0/x_0...0/x_n] \gg \rhd e'_i\end{matrix}}{\ll e_i, \rho' \gg \rhd e^i_{repl}}}{\delta[\ll e_0, \rho' \gg ... \ll e_n, \rho' \gg] \rhd_E (e^0_{repl}...e^n_{repl}) :: E} \\ where\ e^i_{repl} = \ll (e^i_c|RTN), replcar(((e'_0...e'_n) :: E)) \gg\end{matrix} \qquad \frac{\rho' \vdash x_i \hookrightarrow^L (0,i)}{\rho' \vdash x \hookrightarrow (0,i)}$$

$$\cfrac{\cfrac{(e^i_{repl}.S), ((e^0_{repl}, ..., e^n_{repl}) :: E), D \vdash STOP \Rightarrow^1 e^i_{repl} \quad ((e^0_{repl}...e^n_{repl}) :: E) \vdash (0,i) \Rightarrow^L e^i_{repl}}{S, ((e^0_{repl}...e^n_{repl}) :: E), D \vdash LD(0,i)|STOP \Rightarrow^1 e^i_{repl}}}{\rho' \vdash LD(0,i) \Rightarrow^1_{run} e^i_{repl}}$$

Inductive Cases for variables: Assume it holds for ρ.

L_{L2}, where we add an empty list to the environment ((I) indicates where induction is used).

$$\begin{array}{|c||c||c|}\hline \dfrac{\dfrac{\rho \vdash x \rightarrow^{L} e_x}{\rho[\,] \vdash x \rightarrow^{L} e_x}}{\rho[\,] \vdash x \rightarrow^{1} e_x} & e_x \rhd e'_x \qquad \dfrac{\rho \rhd E}{\rho[\,] \rhd_E ([\,] :: E)} & \dfrac{\dfrac{\rho \vdash x \hookrightarrow^{L} (a,b)}{\rho[\,] \vdash x \hookrightarrow^{L} (a+1,b)}}{\rho[\,] \vdash x \hookrightarrow LD(a+1,b)} \\ \hline \multicolumn{3}{|c|}{\dfrac{\dfrac{(e'_x :: S), ([\,] :: E), D \vdash STOP \Rightarrow^{0} e'_x \qquad \dfrac{E \vdash (a,b) \Rightarrow^{L} e'_x \quad (I)}{([\,] :: E) \vdash (a+1,b) \Rightarrow^{L} e'_x}}{S, ([\,] :: E), D \vdash LD(a+1,b) | STOP \Rightarrow^{1} e'_x}}{\rho[\,] \vdash LD(a+1,b) \Rightarrow^{1}_{run} e'_x}} \\ \hline \end{array}$$

L_{L3}, where the value we are looking for is not the top element, and so we must skip by it. Let $\rho' = ((y, e_y) :: \rho'') :: \rho$ and the inductive hypothesis holds for $\rho'' :: \rho$.

$$\begin{array}{|c||c||c|}\hline \dfrac{\dfrac{x \neq y \quad \rho'' :: \rho \vdash x \rightarrow^{L} e_x}{\rho' \vdash x \rightarrow^{L} e_x}}{\rho' \vdash x \rightarrow^{1} e_x} & e_x \rhd e'_x \qquad \dfrac{\begin{array}{c} e_y \rhd e'_y \\ \rho \rhd_E E \quad \rho'' \rhd_E E'' \end{array}}{\rho' \rhd_E (e'_y :: E'') :: E} & \dfrac{\dfrac{x \neq y \quad \rho'' :: \rho \vdash x \hookrightarrow^{L} (a,b)}{\rho' \vdash x \hookrightarrow^{L} (a,b+1)}}{\rho' \vdash x \hookrightarrow (a,b+1)} \\ \hline \multicolumn{3}{|c|}{\dfrac{\dfrac{(e'_x.S), (E'' :: E), D \vdash STOP \Rightarrow^{0} e'_x \qquad \dfrac{(E'' :: E) \vdash (a,b) \Rightarrow^{L} e'_x \quad (I)}{((e'_y :: E'') :: E) \vdash (a,b+1) \Rightarrow^{L} e'_x}}{S, ((e_y :: E') :: E), D \vdash LD(a,b+1) | STOP \Rightarrow^{1} e'_x}}{\rho' \vdash LD(a,b+1) \Rightarrow^{1}_{run} e'_x}} \\ \hline \end{array}$$

And that ends T_{VAR}.

T$\mathbf{T_{LAMBDA}}$: Lambda expressions evaluate in one step to closures, and thus form the final base case.

$$\begin{array}{|c||c||c|}\hline \begin{array}{c} \rho \vdash \lambda\ x_1 ... x_k\ bod \rightarrow \\ \ll \lambda\ x_1 ... x_k\ bod, \rho \gg \end{array} & \dfrac{\begin{array}{c} \delta \rhd_E E \\ \rho[x_1 ... x_k] \vdash bod \hookrightarrow b' \end{array}}{\begin{array}{c} \ll \lambda\ x_1 ... x_k\ bod, \rho \gg \\ \rhd \ll (b' | RTN), E \gg \end{array}} & \dfrac{\rho[x_1 ... x_k] \vdash bod \rightarrow b'}{\begin{array}{c} \rho \vdash \ll \lambda\ x_1 ... x_k\ bod, \rho \gg \\ \hookrightarrow LDF(b' | RTN) \end{array}} \\ \hline \multicolumn{3}{|c|}{\dfrac{\dfrac{\ll b' | RTN, E \gg .S, E, D \vdash STOP \Rightarrow^{0} \ll b' | RTN, E \gg}{S, E, D \vdash LDF(b' | RTN) | STOP \Rightarrow^{1} \ll b' | RTN, E \gg}}{\rho \vdash LDF(b' | RTN) \Rightarrow^{1}_{run} \ll b' | RTN, E \gg}} \\ \hline \end{array}$$

Inductive Cases:

Now we consider the inductive cases. The inductive hypothesis states that the lemma holds over values e_i, e'_i, e''_i, c_i where:

$$\begin{array}{|c||c||c|}\hline \rho \vdash e_i \rightarrow^{t_i} e'_1 & e'_i \rhd e''_i & \rho \vdash e_i \hookrightarrow c_i \\ \hline \multicolumn{3}{|c|}{\rho \vdash c_i \Rightarrow^{t_i}_{run} e''_i} \\ \hline \end{array}$$

T

T$_{\mathbf{OP1}}$: Operators which require just one argument, CAR, CDR, and ATOM, are the easiest inductive case. The inductive hypothesis is over e_1.

$$\frac{\rho \vdash e_1 \rightarrow^{t} e_1'}{\rho \vdash OP\ e_1 \rightarrow^{t+1} \mathit{op}\ e_1'} \quad\Big\|\quad \frac{e_1' \rhd e_1''}{\mathit{op}\ e_1' \rhd \mathit{op}\ e_1''} \quad\Big\|\quad \frac{\rho \vdash e_1 \hookrightarrow c_1}{\rho \vdash OP\ e_1 \hookrightarrow c_1|OP}$$

$$\dfrac{\dfrac{S,E,D \vdash c_1|STOP \Rightarrow^{t} e_1'' \quad (I) \qquad \dfrac{\mathit{op}\ e_1''.S,E,D \vdash STOP \Rightarrow^{0} \mathit{op}\ e_1''}{e_1''.S,E,D \vdash OP|STOP \Rightarrow^{1} \mathit{op}\ e_1''}}{S,E,D \vdash c_1|OP|STOP \Rightarrow^{t+1} \mathit{op}\ e_1''}}{\rho \vdash c_1|OP \Rightarrow_{run}^{t+1} \mathit{op}\ e_1''}$$

T$_{\mathbf{OP2}}$: Two argument operators require the use of induction twice, over e_1 and e_2.

$$\frac{\rho \vdash e_1 \rightarrow^{t_1} e_1' \qquad \rho \vdash e_2 \rightarrow^{t_2} e_2'}{\rho \vdash e_1\ OP\ e_2 \rightarrow^{t_1+t_2+1} e_1'\ \mathit{op}\ e_2'} \quad\Big\|\quad \frac{e_1' \rhd e_1'' \qquad e_2' \rhd e_2''}{e_1'\ \mathit{op}\ e_2' \rhd e_1''\ \mathit{op}\ e_2''} \quad\Big\|\quad \frac{\rho \vdash e_1 \hookrightarrow c_1 \qquad \rho \vdash e_2 \hookrightarrow c_2}{\rho \vdash e_1\ OP\ e_2 \hookrightarrow c_2|c_1|OP}$$

$$\dfrac{\dfrac{S,E,D \vdash c_2|STOP \Rightarrow^{t_2} e_2'' \quad (I) \qquad \dfrac{\left(\dfrac{e_1''\ \mathit{op}\ e_2''.S,E,D \vdash STOP \Rightarrow^{0} e_1''\ \mathit{op}\ e_2''}{e_2''.e_1''.S,E,D \vdash OP|STOP \Rightarrow^{1} e_1''\ \mathit{op}\ e_2''}\right) \qquad e_2''.S,E,D \vdash c_1|STOP \Rightarrow^{t_1} e_1'' \quad (I)}{e_2''.S,E,D \vdash c_1|OP|STOP \Rightarrow^{t_1+1} e_1''\ \mathit{op}\ e_2''}}{S,E,D \vdash c_2|c_1|OP|STOP \Rightarrow^{t_1+t_2+1} e_1''\ \mathit{op}\ e_2''}}{\rho \vdash c_2|c_1|OP \Rightarrow_{run}^{t_1+t_2+1} e_1''\ \mathit{op}\ e_2''}$$

T$_{\mathbf{IF}}$: The IF statement also requires two uses of induction, one over e_1, and one over either e_2 or e_3 depending on the value of e_1. Below we show the case where the value of e_1 is true; the other case is similar.

$$\frac{\rho \vdash e_1 \rightarrow^{t_1} T \qquad \rho \vdash e_2 \rightarrow^{t_2} e_2'}{\rho \vdash IF\ e_1 e_2 e_3 \rightarrow^{t_1+t_2+2} e_2'} \quad\Big\|\quad e_2' \rhd e_2'' \quad\Big\|\quad \frac{\rho \vdash e_1 \hookrightarrow c_1 \qquad \rho \vdash e_2 \hookrightarrow c_2 \qquad \rho \vdash e_3 \hookrightarrow c_3}{\rho \vdash IF\ e_1 e_2 e_3 \hookrightarrow e_2'}$$

$$\dfrac{\dfrac{S,E,D \vdash c_1|STOP \Rightarrow^{t_1} T \quad (I) \qquad \dfrac{\dfrac{\left(\dfrac{e_2''.S,E,D \vdash STOP \Rightarrow^{0} e_2''}{e_2''.S,E,(STOP.D) \vdash JOIN \Rightarrow^{1} e_2''}\right) \qquad S,E,(STOP.D) \vdash c_2|STOP \Rightarrow^{t_2} e_2'' \quad (I)}{S,E,(STOP.D) \vdash (c_2|JOIN) \Rightarrow^{t_2+1} e_2''}}{T.S,E,D \vdash (SEL(c_2|JOIN, c_3|JOIN))|STOP \Rightarrow^{t_2+2} e_2''}}{S,E,D \vdash c_1|(SEL(c_2|JOIN, c_3|JOIN))|STOP \Rightarrow^{t_1+t_2+2} e_2''}}{\rho \vdash c_1|(SEL(c_2|JOIN, c_3|JOIN)) \Rightarrow_{run}^{t_1+t_2+2} e_2''}$$

T$_{\mathbf{APP}}$: For application, we need $k+2$ uses of induction; k for the arguments, and one for the reduction of the function in the environment with the reduced arguments.

Structural and rule induction are not strong enough to carry out the proof for application. These induction principles would give us the hypothesis

$$\begin{array}{|c||c||c|}\hline rho \vdash f \rightarrow^{t_f} \ll \lambda x_1 \ldots x_k.f',\rho' \gg & \dfrac{\rho' \rhd_E E' \qquad \rho'[x_1\ldots x_k] \vdash f' \hookrightarrow s'_f}{\ll \lambda x_1\ldots x_k.f',\rho' \gg \;\rhd\; \ll s'_f|RTN,E' \gg} & \rho \vdash f \hookrightarrow s_f \\ \hline \multicolumn{3}{|c|}{\rho \vdash s_f \Rightarrow^{t_f}_{run} \ll s'_f|RTN,E' \gg} \\ \hline \end{array}$$

and the k hypotheses:

$$\begin{array}{|c||c||c|}\hline \rho \vdash e_i \rightarrow^{t_i} e'_i & e'_i \rhd r_i & \rho \vdash e_i \hookrightarrow s_i \\ \hline \multicolumn{3}{|c|}{\rho \vdash s_i \Rightarrow^{t_i}_{run} r_i} \\ \hline \end{array}$$

But, as we see in the proof of application, we also need a hypothesis for the *body of the abstraction*, which is not given by structural of rule induction (since the compiler does not generate code for the closure produced for the function f), namely

$$\begin{array}{|c||c||c|}\hline \rho'[e'_1/x_1\ldots e'_k/x_k] \vdash f' \rightarrow^{t} f'' & f'' \rhd r & \rho'[x_0\ldots x_k] \vdash f' \hookrightarrow s'_f \\ \hline \multicolumn{3}{|c|}{\rho'[e'_1/x_1\ldots e'_k/x_k] \vdash s'_f \Rightarrow^{t}_{run} r} \\ \hline \end{array}$$

To simplify the notation in the following proof, we use $t_{n,m}$ for the sum of the times t_n to t_m, that is $t_{n,m} = \sum_{i=n}^{m} t_i$. We do not rewrite the hypothesis just given above.

$$\left(\begin{array}{c}\left(\dfrac{r.S,E,D \vdash STOP \Rightarrow^{0} r}{r,r_1\ldots r_k.E,(S,E,STOP.D) \vdash RTN \Rightarrow^{1} r}\right)\\[2ex] \dfrac{\dfrac{nil,r_1\ldots r_k.E,(S,E,STOP.D) \vdash s'_f|STOP \Rightarrow^{t} r \quad (I)}{nil,r_1\ldots r_k.E,(S,E,STOP.D) \vdash s'_f|RTN \Rightarrow^{t+1} r}}{((s'_f|RTN,E).r_1\ldots r_k.S),E,D \vdash AP|STOP \Rightarrow^{t+2} r}\end{array}\right)$$

$$\dfrac{(r_1\ldots r_k.S),E,D \vdash s_f|STOP \Rightarrow^{t_f} (s'_f|RTN,E) \quad (I)}{\underline{(r_1\ldots r_k.S),E,D \vdash s_f|AP|STOP \Rightarrow^{t_f+t+3} r}}$$

$$\vdots$$

$$\dfrac{\dfrac{\dfrac{().S,E,D \vdash s_k|STOP \Rightarrow^{t_k+1} r_k \quad (I) \qquad \dfrac{(r_k).S,E,D \vdash s_{k-1}|\ldots|s_1|CONS|(s_f)|AP|STOP \Rightarrow^{t_{1,k-1}+t_f+t+(k-1)+3} r}{r_k.().S,E,D \vdash CONS|\ldots|s_1|CONS|(s_f)|AP|STOP \Rightarrow^{t_{1,k-1}+t_f+t+k+3} r}}{().S,E,D \vdash s_k|\ldots|s_1|CONS|(s_f)|AP|STOP \Rightarrow^{t_{1,k}+t_f+t+k+3} r}}{S,E,D \vdash LDC\ NIL|s_k|\ldots|s_1|CONS|(s_f)|AP|STOP \Rightarrow^{t_{1,k}+t_f+t+k+4} r}}{\rho \vdash LDC\ NIL|s_k|\ldots|s_1|CONS|(s_f)|AP \Rightarrow^{t_{1,k}+t_f+t+k+4}_{run} r}$$

$\mathrm{T_{LET}}$: The LET construct is so similar to the next case that we leave it out (the only difference being DUM is not executed and AP is used instead of RAP).

TREC: Let $E' = replcar((r'_1...r'_k).E)$. Here we need $k+1$ inductive arguments, k for the arguments and 1 for the body.

$$\dfrac{\begin{array}{c}\rho \vdash e_1 \rightarrow^{t_1} \ll e'_1, \rho \gg ... \rho \vdash e_k \rightarrow^{t_k} \ll e'_k, \rho \gg \\ \rho' = \rho[\ll e'_1, \rho'/x_1 \gg ... \ll e'_k, \rho'/x_k \gg] \\ \rho' \vdash b \rightarrow^{t_b} b'\end{array}}{\rho \vdash letrec\ (x_1...e_k) = (e_1...e_k)\ in\ b \rightarrow^{t_{1,k}+t_b+k+5} b'}$$

$$\begin{array}{c} e_1 \rhd r_1 \\ \vdots \\ e_k \rhd r_k \\ b' \rhd r \end{array}$$

$$\dfrac{\begin{array}{c}\rho \vdash e_1 \hookrightarrow s_1 ... \rho \vdash e_k \hookrightarrow s_k \\ \rho[x_1...x_k] \vdash b \hookrightarrow s_b\end{array}}{\begin{array}{c}\rho \vdash letrec\ (x_1...x_k) = (e_1...e_k)\ in\ b \hookrightarrow \\ DUM|LDC\ NIL|s_k|CONS|...|s_1| \\ CONS|(LDF(s_b|RTN))|RAP\end{array}}$$

$$\left(\dfrac{r.S,E,D \vdash STOP \Rightarrow^{0} r}{r,E',(S,E,STOP.D) \vdash RTN \Rightarrow^{1} r}\right)$$

$$\dfrac{\dfrac{\dfrac{nil,E',(S,E,STOP.D) \vdash s_b|STOP \Rightarrow^{t_b} r \quad (I)}{nil,E',(S,E,STOP.D) \vdash s_b|RTN \Rightarrow^{t_b+1} r}}{((s_b|RTN,E).r_1...r_k.().S),E,D \vdash RAP|STOP \Rightarrow^{t_b+2} r}}{(r_1...r_k.().S),E,D \vdash LDF(s_b|RTN)|RAP|STOP \Rightarrow^{t_b+3} r}$$

$$\vdots$$

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}().S,E,D \vdash s_k|STOP \Rightarrow^{t_k} \\ r_k \quad (I)\end{array} \qquad \dfrac{(r_k).().S,E,D \vdash s_{k-1}|...|s_1|CONS|(LDF(s_b|RTN))|RAP|STOP \Rightarrow^{t_{1,k-1}+t_b+(k-1)+3} r}{r_k.().().S,E,D \vdash CONS|...|s_1|CONS|(LDF(s_b|RTN))|RAP|STOP \Rightarrow^{t_{1,k-1}+t_b+k+3} r}}{().().S,E,D \vdash s_k|...|s_1|CONS|(LDF(s_b|RTN))|RAP|STOP \Rightarrow^{t_{1,k}+t_b+k+3} r}}{\dfrac{().S,E,D \vdash LDC\ NIL|s_k|...|s_1|CONS|(LDF(s_b|RTN))|RAP|STOP \Rightarrow^{t_{1,k}+t_b+k+4} r}{S,E,D \vdash DUM|LDC\ NIL|s_k|...|s_1|CONS|(LDF(s_b|RTN))|RAP|STOP \Rightarrow^{t_{1,k}+t_b+k+5} r}}}{\rho \vdash DUM|LDC\ NIL|s_k|...|s_1|CONS|(LDF(s_b|RTN))|RAP \Rightarrow_{run}^{t_{1,k}+t_b+k+5} r}$$

□

7.3 Non-terminating Programs

Now we consider the case where M has no WHNF. We have to show both completeness and soundness in this case, but they are so alike that not all the detail is given.

Lemma 8 Completeness of Non-terminating Programs:

$$\forall t. \frac{\rho \vdash M \not\rightarrow^{t} \qquad \rho \vdash M \hookrightarrow M'}{\rho \vdash M' \not\Rightarrow^{t}_{run}}$$

Proof: Assume that M has no L.WHNF at time t. We need to show that the SECD machine has not stopped at any time $\leq t$.

By induction over the length of the proofs.

Base cases: M cannot be a variable, a constant or a lambda abstraction as these reach WHNF at time 1.

Inductive cases: Assume the lemma holds for each rule hypothesis.

For these proofs we leave out the $STOP$ and $\Rightarrow_{run}$ lines as by now these are standard.

T_{OP_1}:

$$\frac{\rho \vdash e_1 \not\rightarrow^{t}}{\rho \vdash OP_1\, e_1 \not\rightarrow^{t+1}} \qquad \frac{\rho \vdash e_1 \rightarrow c_1}{\rho \vdash OP\, e_1 \hookrightarrow c_1|OP'}$$

$$\frac{S, E, D \vdash c_1|STOP \not\Rightarrow^{t} \quad (I)}{S, E, D \vdash c_1|OP \not\Rightarrow^{t+1}}$$

T_{OP_2}: This is the last case we will go through in detail, as the rest follow the same idea. There are three cases: (1) e_1 does not terminate but e_2 does.

$$\frac{\begin{array}{c}\rho \vdash e_1 \not\rightarrow^{t_1}\\ \rho \vdash e_2 \rightarrow^{t_2} S_2\end{array}}{\rho \vdash e_1\ OP_2\ e_2 \not\rightarrow^{t_1+t_2+1}} \qquad \frac{\begin{array}{c}\rho \vdash e_1 \hookrightarrow c_1\\ \rho \vdash e_2 \hookrightarrow c_2\end{array}}{\rho \vdash e_1\ OP_2\ e_1 \hookrightarrow c_2|c_1|OP'}$$

$$\frac{S, E, D \vdash c_2|STOP \Rightarrow^{t_2} c_2' \quad (I) \qquad c_2' :: S, E, D \vdash c_1|STOP \not\Rightarrow^{t_1} \quad (I)}{S, E, D \vdash c_2|c_1|OP \not\Rightarrow^{t_1+t_2+1}}$$

(2) e_1 does terminate but e_2 does, and (3) both e_1 and e_2 fail to terminate follow the pattern of case (1).

All of the remaining proofs have the same form as the one just given. For example, the IF command would simply have more cases, depending on how many (1,2, or 3) and which subexpressions are assumed not to terminate. For LET and REC expressions the same idea applies, as one or more of the arguments or the body are assumed not to terminate, breaking the proof down into cases like the one given above. □

Lemma 9 Soundness of Non-terminating Programs:

$$\forall t. \frac{\rho \vdash M' \not\Rightarrow^{t}_{run} \qquad \rho \vdash M \hookrightarrow M'}{\rho \vdash M \not\rightarrow}$$

Proof: Assume M' has no S.WHNF at time t. We need to show that M has no L.WHNF at time $\leq t$.

Base cases: M' cannot be LD, LDC, or LDF as these reach S.WHNF at time 1.

Indictive cases: For example T_{OP_1}:

$$\frac{S, E, D \vdash c_1|STOP \not\Rightarrow^{t}}{S, E, D \vdash c_1|OP \not\Rightarrow^{t+1}} \qquad \frac{\rho \vdash e_1 \hookrightarrow c_1}{\rho \vdash OP\, e_1 \hookrightarrow c_1|OP'}$$

$$\frac{\rho \vdash e_1 \not\rightarrow^{t} \quad (I)}{\rho \vdash OP_1\, e_1 \not\rightarrow^{t+1}}$$

This proof is easily seen to be just a "mirror image" of the previous one, where here we assume that an SECD (sub-) expression does not terminate, which implies, by the inductive hypothesis, that a miniSML (sub-) expression did not terminate. □.

7.4 Finally, the Theorem

Theorem 4

$$\begin{tikzcd}
M \arrow[r, "\rho\vdash \_ \hookrightarrow \_"] \arrow[d, "\rho\vdash \_ \longrightarrow^{t} \_"'] & M' \arrow[d, "\rho\vdash \_ \Rightarrow^{t}_{run} \_"] \\
S \arrow[r, "\triangleright"'] & S'
\end{tikzcd}$$

Proof: By the previous lemmas. □

7.5 Comments on the Proof

Although this proof is quite long, it is straightforward. It is not as long as it would have been in the style of Despeyroux, where soundness of terminating programs has to be proved separately. The proof of soundness is (somewhat) the same length as the proof of completeness (several pages long). In contrast, proving that relations are deterministic is very easy and the proof for non-terminating programs is also very straightforward.

8 Adding Lazy Evaluation

Lazy evaluation (call-by-name), although not part of Henderson's original formulation, is an important evaluation strategy in that it will give a result if one is possible. Here we change miniSML to evaluate lazily and extend the SECD machine so that it also can handle this. The hardware which has been proven correct in HOL is for the strict SECD used above. Nevertheless, changing to lazy evaluation is not overly complicated and the microcode for the hardware could be reprogrammed to handle several new operations without invalidating the proof of the existing operations.

To add lazy evaluation to our miniSML language we have to change the values that can be stored in the environment slightly. As well as WHNF's the environment will have to store "suspensions" of arguments with their environments.

To support call-by-name evaluation we have to change the application rule (so that it does not evaluate the argument eagerly but instead creates a suspension of the argument) and the variable lookup rule (which may have to evaluate a suspension). Since *LET* and *LETREC* involve applications (in pure lambda) they must be changed as well. *CONS* must also be lazy, although the other built-in operators retain their strict semantics since they definitely use their arguments. This gives us a call-by-name semantics.

$$L_{APP'} \quad \frac{\rho \vdash f \to^{t} \ll \lambda x.f', \rho' \gg \qquad (x, \ll a, \rho \gg) :: \rho' \vdash f' \to^{t'} r}{\rho \vdash (f\ a) \to^{t+t'+5} r}$$

$$L_{VAR2} \quad \frac{\rho \vdash x \to^{L} \ll a, \rho' \gg \qquad \rho' \vdash a \to^{t} a'}{\rho \vdash x \to^{t+2} a'}$$

$$L_{LET'} \quad \frac{\rho[\ll e_1, \rho \gg /x_1 \ldots \ll e_n, \rho \gg /x_n] \vdash b \to^{t} b'}{\rho \vdash LET\ x_1 = e_1 \ldots x_n = e_n\ in\ b \to^{t+n+3} b'}$$

$$L_{REC'} \quad \frac{\rho' \vdash b \to^{t} b' \ \ where\ \ \rho' = \rho[\ll e_1, \rho' \gg /x_1 \ldots \ll e_n, \rho' \gg /x_n]}{\rho \vdash LETREC\ x_1 = e_1 \ldots x_n = e_n\ in\ b \to^{t+n+4} b'}$$

$$L_{CONS'} \quad CONS\ e_1\ e_2 \to^{1} (e_1, e_2)$$

Figure 5: Changes to miniSML semantics for lazy evaluation

8.1 Lazy Instructions for the SECD machine

It is possible to delay evaluation of arguments on the SECD machine without adding any new instructions. Specifically, each argument to a function can be made into a nullary function (a becomes $\lambda().a$) and each reference to a variable in the body of the function would become an application. This technique does not allow sharing (every time an argument is used it must be reevaluated) and so is not a likable solution. Instead, we can add a couple more operations to the machine to directly handle lazy evaluation. Evaluation with sharing is known as call-by-need.

Henderson has proposed three instructions for the SECD machine to evaluate call-by-need. *LDE* creates a suspension and puts in on the stack, *AP0* applies a function to something which may be a suspension, and *UPD* updates the environment with the value of a suspension in order to facilitate sharing. In order to achieve this, we must have *pointers* to suspensions. If p is such a pointer, we will write $\downarrow p$ for the value p points to. The transitions for the instructions are:

$$\frac{\downarrow p = \ll c, e \gg :: s, e, d \vdash c' \Rightarrow r}{s, e, d \vdash (LDE\ c)|c' \Rightarrow r}$$

$$\frac{x :: s, e, d \vdash c \Rightarrow r}{\downarrow p = x :: s, e, d \vdash AP0|c \Rightarrow r}$$

$$\frac{nil, e, ((p :: s, e', c') :: d) \vdash c \Rightarrow r}{\downarrow p = \ll c, e \gg .s, e, d \vdash AP0|c \Rightarrow r}$$

$$\frac{x :: [], e', d \vdash c' \Rightarrow r \qquad \downarrow p := x}{x :: s, e, ((p :: s, e', c') :: d) \vdash UPD \Rightarrow r}$$

Before we attempt to explain these commands, let us add on to our compiler so that we can see from whence they came. Assume that all our applications are lazy.

$$\frac{\varrho \vdash v \hookrightarrow^{L} (m.n)}{\varrho \vdash v \hookrightarrow LD(m.n)|AP0}$$

$$\frac{\varrho \vdash f \hookrightarrow f' \qquad \varrho \vdash a \hookrightarrow a'}{\varrho \vdash (f\ a) \hookrightarrow LDC\ \ NIL|LDE(a'|UPD)|CONS|LDF(f'|RTN)|AP}$$

As well, obvious changes are made to LET and $LETREC$.

If we look at LDE as delaying evaluation of an argument by converting a to $\lambda().a$ and $AP0$ as applying () to the argument, then it is clear what Henderson's commands are doing. Here is an example evaluation of $(\lambda x.x + x)(3 + 3)$.

$$\begin{array}{c}
18, [p], [] \vdash STOP \Rightarrow 18 \\ \hline
9 :: 9, [p], STOP \vdash ADD|RTN \Rightarrow 18 \\ \hline
p :: 9, [p], STOP \vdash AP0|ADD|RTN \Rightarrow 18 \\ \hline
9, [p], STOP \vdash LD(0,0)|AP0|ADD|RTN \Rightarrow 18 \qquad \downarrow p = 9 \\ \hline
9, e, ((p, [], [p]) :: STOP) \vdash UPD \Rightarrow 18 \\ \hline
[], e, (p, [], [p] :: (STOP)) \vdash LDC\ 3|LDC\ 3\ ADD\ UPD \Rightarrow 18 \\
p, [p], STOP \vdash AP0|LD(0,0)|AP0|ADD|RTN \Rightarrow 18 \\ \hline
[], [p], STOP \vdash LD(0,0)|AP0|LD(0,0)|AP0|ADD|RTN \Rightarrow 18 \\ \hline
p, [], [] \vdash CONS|LDF(...)|AP \Rightarrow 18 \qquad \downarrow p = \ll LDC\ 3|LDC\ 3|ADD|UPD \gg \\ \hline
[], [], [] \vdash LDE(LDC\ 3...)|CONS|LDF... \Rightarrow 18 \\ \hline
[], [], [] \vdash LDC\ \ NIL|LDE(LDC\ 3|LDC\ 3|ADD|UPD)|CONS \\
|LDF(LD(0.0)|AP0|LD(0.0)|AP0|ADD|RTN)|AP|STOP \Rightarrow 18
\end{array}$$

Unfortunately, we no longer have only one rule per SECD command (as we must do pattern matching on what p points to above in the $AP0$ command). Although there is no easy way to get back to a single rule per instruction, if we push the pattern matching into the LD rules we need add only two instructions:

$$\boxed{\begin{array}{ll}
S_{SUSP} & \dfrac{(p :: s), e, d \vdash c' \Rightarrow r \qquad p = \ll c, e \gg}{s, e, d \vdash (SUSP\ \ c)|c' \Rightarrow r} \\[2ex]
S_{UPD} & \dfrac{(x :: s), e, d \vdash c \Rightarrow r \qquad \downarrow p := x}{(x :: s'), e', ((p, s, e, c) :: d) \vdash UPD(p) \Rightarrow r}
\end{array}}$$

where as well as the previous LD rule we have (when a suspension is looked up):

$$\boxed{S_{LD2} \quad \dfrac{e \vdash (m.n) \Rightarrow^{L} \ll c.e \gg \qquad [], e, ((p, s', e', c) :: d) \vdash c \Rightarrow r}{s', e', d \vdash LD(m, n)|c' \Rightarrow r}}$$

The compilation for these is

$$\boxed{T_{APP'} \quad \frac{\varrho \vdash f \hookrightarrow f' \qquad \varrho \vdash x_1 \hookrightarrow x_1' ... \varrho \vdash x_n \hookrightarrow x_n'}{\begin{array}{c}\varrho \vdash (f\, x_1 ... x_n) \hookrightarrow LDC\ NIL|SUSP(x_n'|UPD)|CONS|... \\ |SUSP(x_1'|UPD)|CONS|LDF(f'|RTN)|AP\end{array}}}$$

Here is the same example using these commands.

$$\begin{array}{c}
18, [], []| - STOP \Rightarrow 18 \\ \hline
9 :: 9, p, STOP| - ADD|RTN \Rightarrow 18 \\ \hline
9, p, STOP| - LD(0.0)|ADD|RTN \Rightarrow 18 \\ \hline
9, [], (p, [], p, LD(0.0)|ADD|RTN) :: STOP| - UPD \Rightarrow 18 \qquad p = 9 \\ \hline
[], [], (p, [], p, LD(0.0)|ADD|RTN) :: STOP| - LDC\ 3|LDC\ 3|ADD|UPD \Rightarrow 18 \\ \hline
[], p, STOP \vdash LD(0.0)|LD(0.0)|ADD|RTN \Rightarrow 18 \qquad \downarrow p = \ll LDC\ 3|LDC\ 3|ADD|UPD, [] \gg \\ \hline
[], [], []| - LDCNIL|SUSP(LDC\ 3|LDC\ 3|ADD|APD)|CONS|LDF(LD(0.0)|LD(0.0)|ADD|RTN)|AP|STOP \Rightarrow 18
\end{array}$$

Moving the pattern matching in this manner seems to simplify the code considerably, although more extensive testing would be required to ensure this. Nevertheless, we use these $SUSP$ and UPD along with $T_{APP'}$ to define our lazy machine and compiler.

8.2 Compiler correctness for the Lazy SECD

The proof of the lazy compiler follows quite closely the proof of the previous compiler, differing only for the rules noted above. Since the setup is the same, we simply offer these portions of the completeness proof.

T_{VAR2}: This shows the case where a variable is used for the first time and the environment contains a suspension.

$$\begin{array}{|c||c||c|}
\hline
\dfrac{\begin{array}{c}\rho \vdash x \rightarrow^{L} \ll a, \rho' \gg \\ \rho' \vdash a \rightarrow^{t} a'\end{array}}{\rho \vdash x \rightarrow^{t+2} a'} & \begin{array}{c} a' \rhd a'' \\ \rho' \rhd E' \end{array} & \begin{array}{c} \rho \vdash x \hookrightarrow LD(m.n) \\ \rho' \vdash a \hookrightarrow c_a \end{array} \\
\hline
\end{array}$$

$$\boxed{\begin{array}{c}
a'' :: S, E, D \vdash STOP \Rightarrow^{0} a'' \qquad \downarrow p := a'' \\ \hline
a'', E', (p, S, E, STOP.D) \vdash UPD \Rightarrow^{t} a'' \quad (I) \\ \hline
[], E', (p, S, E, STOP.D) \vdash c_a|UPD \Rightarrow^{t+1} a'' \\ \hline
S, E, D \vdash LD(m.n)|STOP \Rightarrow^{t+2} op\ e_1'' \\ \hline
\rho \vdash LD(m.n) \Rightarrow_{run}^{t+2} op\ e_1''
\end{array}}$$

$T_{APP'}$: We consider the syntactically easier case of only one argument. Again, our inductive hypotheses are not available from structural or rule induction. They are:

$$\begin{array}{|c||c||c|}
\hline
\rho \vdash f \rightarrow^{t} \ll \lambda x.f', \rho' \gg & \dfrac{\rho'[x] \vdash f' \hookrightarrow s_f' \quad \rho' \rhd_E E'}{\ll \lambda x.f', \rho' \gg \rhd \ll s_f'|RTN, E' \gg} & \rho \vdash f \hookrightarrow s_f \\
\hline
\end{array}$$

$$\boxed{\rho \vdash s_f \Rightarrow_{run}^{t} \ll s_f'|RTN, E' \gg}$$

$$\frac{\boxed{\rho'[\ll a,\rho \gg /x] \vdash f' \rightarrow^{t'} f''} \;\boxed{f'' \rhd r}\; \boxed{\rho'[x] \vdash f' \hookrightarrow s'_f}}{\boxed{\rho'[\ll a,\rho \gg /x] \vdash s'_f \Rightarrow^{t'}_{run} r}}$$

The second hypothesis is on the body of a lambda expression, not on the argument. Here is the proof under these assumptions as well as the fact that when $\rho \vdash a \hookrightarrow s_a$ we have:

$$\rho \vdash fa \hookrightarrow LDC\ NIL|SUSP(s_a|UPD)|CONS|LDF(s_f|RTN)|AP|STOP$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}(\ll s_a|UPD,E \gg).S,E,D \vdash s_f|STOP \Rightarrow^{t} \\ \ll s'_f|RTN,E' \gg \quad (I)\end{array} \qquad \dfrac{\begin{array}{c}\left(\dfrac{(r,S.E,D) \vdash STOP \Rightarrow^{0} r}{r,(\ll s_a|UPD,E \gg).E',(S,E,STOP.D) \vdash RTN \Rightarrow^{1} r}\right) \\ \dfrac{[],(\ll s_a|UPD,E \gg).E',(S,E,STOP.D) \vdash s'_f|STOP \Rightarrow^{t'} r \quad (I)}{[],(\ll s_a|UPD,E \gg).E',(S,E,STOP.D) \vdash s'_f|RTN \Rightarrow^{t'+1} r}\end{array}}{\ll s'_f|RTN,E' \gg .(\ll s_a|UPD,E \gg).S,E,D \vdash AP|STOP \Rightarrow^{t'+2} r}}{(\ll s_a|UPD,E \gg).S,E,D \vdash s_f|AP|STOP \Rightarrow^{t+t'+2} r}}{\ll s_a|UPD,E \gg .().S,E,D \vdash CONS|s_f|AP|STOP \Rightarrow^{t+t'+3} r}}{().S,E,D \vdash SUSP(s_a|UPD)|CONS|s_f|AP|STOP \Rightarrow^{t+t'+4} r}}{S,E,D \vdash LDC\ NIL|SUSP(s_a|UPD)|CONS|s_f|AP|STOP \Rightarrow^{t+t'+5} r}}{\rho \vdash LDC\ NIL|SUSP(s_a|UPD)|CONS|s_f|AP \Rightarrow^{t+t'+5}_{run} r}$$

This is the case for only one argument, but it generalizes easily. *LET* and *REC* follow this pattern as well.

Timing seems a little more difficult in the lazy case as it is dependent on how often (if at all) each variable is executed. The first use requires time to evaluate the suspension, but each subsequent use is a constant time of 1. Luckily, both the lazy miniSML and the LSECD both handle these the same way so that the inductive hypothesis takes care of timing for us.

9 Conclusion

We have proved the specification of a compiler from miniSML to SECD to be correct. This proof builds on the natural semantics style used by Despeyroux. Although the proof is quite long, no given part is very difficult and it follows closely our intuition on how such a proof should proceed.

Hand proofs are prone to errors. This proof should be mechanically checked using a theorem prover such as HOL. This would also tie it in with the SECD hardware proof.

We have not said anything about implementing the compiler to meet the specification, or better yet, deriving the implementation from the specification. Much work has been done in this area. Instead we concentrated on simplifying the notation for specifications and the proof effort needed to show them correct.

Natural semantics rules can be directly implemented, for example by TYPOL [3]. This would allow one to execute the specification directly. This does not preclude other implementations, as a system such as TYPOL is quite complex and would be difficult to implement on correct hardware, such as the SECD chip. A better first step would be to correctly implement the specification in miniSML itself.

10 Acknowledgments

This work has been supported by the Alberta MicroElectronics Center, the Natural Sciences and Engineering Research Council, the Izaak Walton Killam Memorial Fund, and the Alberta Heritage Scholarship Fund. Thanks also to Graham Birtwistle, Brian Graham and Konrad Slind who have worked on the SECD project and read drafts of this report.

References

[1] W. Burge. *Recursive Programming Techniques.* Addison-Wesley, New York, 1975.

[2] J. Despeyroux. Proof of Translation in Natural Semantics. In *Proceedings of the 1986 Symposium on Logic in Computer Science*, pages 193–205, Cambridge MA, 1986.

[3] T. Despeyroux. Executable Specification of Static Semantics. In *Semantics of Data Types*, LNCS 173, 1984.

[4] A. J. Field and P. G. Harrison. *Functional programming.* Addison–Wesley, New York, 1988.

[5] M. J. C. Gordon. HOL: A Proof Generating System for Higher Oorder Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128, Norwell, Massachusetts, 1988. Kluwer.

[6] B. Graham. SECD: The Design and Verification of a Functional Microprocessor. Research Report 90/395/19, University of Calgary, 1990.

[7] P. Henderson. *Functional programming; applications and implementation.* Prentice Hall, London, 1980.

[8] M. J. Hermann, G. Birtwistle, B. Graham, and T. Simpson. The Architecture of Henderson's SECD Machine. Research Report 89/340/02, Computer Science Department, University of Calgary, 1989.

[9] G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258, Amsterdam, 1988. Elsevier Science Publishers (North Holland).

[10] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[11] P. J. Landin. The correspondence between ALGOL60 and Church's lambda calculus, Part 1. *Communications of the ACM*, 8(2):89–101, 1965.

[12] P. J. Landin. The correspondence between ALGOL60 and Church's lambda calculus, Part 2. *Communications of the ACM*, 8(3):158–165, 1965.

[13] P. J. Landin. An abstract machine for designers of computing languages. In *Proceedings of the IFIP Congress 65, volume 2*, pages 438–439, Washington, 1966. Spartan Books.

[14] K. Mitchell. Course notes. Edinburgh, 1988.

[15] F. Morris. Advice on structuring compilers and proving them correct. In *Proc. ACM Symposium on POPL*, pages 144–152, 1973.

[16] G. D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.

[17] G. D. Plotkin. A Structural Approach to Operational Semantics. Daimi, University of Aarhus, Denmark, 1979.