INTEROPERATION FOR LAZY AND EAGER EVALUATION

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

William Faught

May 2011

AUTHORIZATION FOR REPRODUCTION OF MASTER'S THESIS

Signature

Date

APPROVAL PAGE

TITLE: Interoperation for Lazy and Eager Evaluation

AUTHOR: William Faught

DATE SUBMITTED: May 2011

Dr. John Clements ______

Advisor or Committee Chair   Signature

Dr. Gene Fisher ______

Committee Member   Signature

Dr. Phillip Nico ______

Committee Member   Signature

Abstract

Interoperation for Lazy and Eager Evaluation

by

William Faught

Programmers forgo existing solutions to problems in other programming languages where software interoperation proves too cumbersome; they remake solutions, rather than reuse them. To facilitate reuse, interoperation must resolve language incompatibilities transparently. To address part of this problem, we present a model of computation that resolves incompatible lazy and eager evaluation strategies using dual notions of evaluation contexts and values to mirror the lazy evaluation strategy in the eager one. This method could be extended to resolve incompatible evaluation strategies for any pair of languages with common expressions.

Acknowledgements

I am grateful for the support of my parents, Jerry and Jo Ann, and my uncle, Tim. I could not have done this without them.

Contents

List of Figures

Chapter 1

Introduction

Programmers forgo existing solutions to problems in other programming languages where software interoperation proves too cumbersome; they remake solutions, rather than reuse them. To facilitate reuse, interoperation must resolve language incompatibilities transparently. To address part of this problem, we present a model of computation that resolves incompatible lazy and eager evaluation strategies.

Matthews and Findler presented a method of type-safe interoperation between languages with incompatible polymorphic static and dynamic type systems [3]. We observe that their method is insufficient for transparent interoperation between languages with incompatible lazy and eager evaluation strategies. We explain the underlying problem and present a method of interoperation that resolves this incompatibility.

The model of computation of Matthews and Findler comprises two eager languages based on ML and Scheme. We extend their model of computation with a third language that is based on Haskell and identical to their ML-like language, except it is lazy. We introduce lists to all three languages. Types are equal up to alpha equivalence. Letter subscripts denote languages. Languages do not share variables or type variables. Hereafter, we use the names of Haskell, ML, and Scheme to refer to their counterparts in our model of computation.

Unlike ML and Scheme, Haskell does not evaluate function arguments or list construction operands. These three evaluation contexts comprise the set of incompatible evaluation contexts between Haskell and ML, and Haskell and Scheme. Since Haskell permits unused erroneous or divergent expressions in these evaluation contexts and ML and Scheme do not, there are Haskell values that have no counterpart in ML and Scheme. Attempting to convert such values to ML and Scheme forces the evaluation of such expressions and breaks the transparency of interoperation.

Figure 1.1 demonstrates how a straightforward introduction of Haskell to the model of Matthews and Findler breaks the transparency of interoperation when converting a list construction from Haskell to Scheme. The Haskell list construction contains an erroneous operand that Scheme forces to evaluate in the process of converting the Haskell list construction. Figure 1.2 demonstrates Scheme correctly deferring the evaluation of the erroneous Haskell list construction operand and producing as a result the counterpart Scheme list construction.

Moreover, since the conversion of functions from ML and Scheme to Haskell requires the application of the original function to the converted Haskell argument, ML and Scheme always force the evaluation of the converted Haskell argument, even if it is never used. The application of such converted functions effectively changes the order of evaluation of Haskell and breaks the transparency of interoperation.

Likewise, figure 1.3 demonstrates the conversion of a function from Haskell to Scheme. Scheme forces the evaluation of the erroneous Haskell argument in the process of applying the Scheme function, even though the Haskell argument is never used. From the perspective of the outermost Haskell application, the argument must have been used, but it was not. Figure 1.4 demonstrates Scheme not forcing the evaluation of the Haskell argument, which allows the Scheme function to produce a number.

Figure 1.1: Transparency broken for list construction operands.

$$
\begin{array}{ll}
\texttt{sh}\ \{\texttt{N}\}\ (\texttt{cons}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”})\ (\texttt{nil}\ \texttt{N})) & \rightarrow \\
\texttt{cons}\ (\texttt{sh}\ \texttt{N}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}))\ (\texttt{sh}\ \{\texttt{N}\}\ (\texttt{nil}\ \texttt{N})) & \rightarrow \\
\textbf{Error:}\ \text{“Not a number”} &
\end{array}
$$

Figure 1.2: Transparency not broken for list construction operands.

$$
\begin{array}{ll}
\texttt{sh}\ \{\texttt{N}\}\ (\texttt{cons}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”})\ (\texttt{nil}\ \texttt{N})) & \rightarrow \\
\texttt{cons}\ (\texttt{sh}\ \texttt{N}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}))\ (\texttt{sh}\ \{\texttt{N}\}\ (\texttt{nil}\ \texttt{N})) & \rightarrow \\
\texttt{cons}\ (\texttt{sh}\ \texttt{N}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}))\ (\texttt{nil}\ \texttt{N}) &
\end{array}
$$

Figure 1.3: Transparency broken for function arguments.

$$
\begin{array}{ll}
(\texttt{hs}\ (\texttt{N} \rightarrow \texttt{N})\ (\lambda x_S.\overline{0}))\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}) & \rightarrow \\
(\lambda x_H : \texttt{N}.\texttt{hs}\ \texttt{N}\ ((\lambda x_S.\overline{0})\ (\texttt{sh}\ \texttt{N}\ x_H)))\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}) & \rightarrow \\
\texttt{hs}\ \texttt{N}\ ((\lambda x_S.\overline{0})\ (\texttt{sh}\ \texttt{N}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}))) & \rightarrow \\
\textbf{Error:}\ \text{“Not a number”} &
\end{array}
$$

Figure 1.4: Transparency not broken for function arguments.

$$
\begin{array}{ll}
(\texttt{hs}\ (\texttt{N} \rightarrow \texttt{N})\ (\lambda x_S.\overline{0}))\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}) & \rightarrow \\
(\lambda x_H : \texttt{N}.\texttt{hs}\ \texttt{N}\ ((\lambda x_S.\overline{0})\ (\texttt{sh}\ \texttt{N}\ x_H)))\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}) & \rightarrow \\
\texttt{hs}\ \texttt{N}\ ((\lambda x_S.\overline{0})\ (\texttt{sh}\ \texttt{N}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}))) & \rightarrow \\
\texttt{hs}\ \texttt{N}\ \overline{0} & \rightarrow \\
\overline{0} &
\end{array}
$$

Chapter 2

Model of Computation

To preserve the transparency of interoperation, ML and Scheme must not force Haskell to evaluate reducible expressions in Haskell boundaries in the incompatible evaluation contexts, and must force their evaluation in all other evaluation contexts. Haskell boundaries must be a new kind of value that ML and Scheme can force to become a reducible expression in certain evaluation contexts, and thereby force the evaluation of the inner Haskell reducible expressions to Haskell values and the conversion of those values to ML or Scheme.

Since ML and Scheme do not force Haskell to evaluate in some evaluation contexts, we must factor Haskell boundaries out of ML and Scheme evaluation context nonterminals, E, into new evaluation context nonterminals. We name these new nonterminals F because they allow ML and Scheme to force Haskell to evaluate, and we rename the primary evaluation context nonterminals from E to U (unforced) because they do not. Likewise, we factor Haskell boundaries out of ML and Scheme value nonterminals, v, into new value nonterminals. We name these new nonterminals f (forced) and rename the old value nonterminals from v to u (unforced). We rename Haskell evaluation contexts and values to F and

f, respectively.

In ML and Scheme, we tie F and U together by replacing U with F in the syntax and operational semantics in all evaluation contexts except the incompatible ones. Likewise, we tie f and u together by replacing u with f in the syntax and operational semantics in those same evaluation contexts. F evaluation contexts produce f values, and U evaluation contexts produce u values. U only applies to incompatible evaluation contexts, and F applies to all others. ML and Scheme use F to evaluate expressions. We rename the meta evaluation context from $\mathscr{E}$ to $\mathscr{F}$.

Transparency is restored for interoperation in all cases with our changes to the model of computation of Matthews and Findler.

Theorem 1. *Interoperation is transparent:*

1. $e_H \simeq \texttt{hm}\ t_H\ t_M\ (\texttt{mh}\ t_M\ t_H\ e_H) \simeq \texttt{hs}\ k_H\ (\texttt{sh}\ k_H\ e_H)$

2. $e_M \simeq \texttt{mh}\ t_M\ t_H\ (\texttt{hm}\ t_H\ t_M\ e_M) \simeq \texttt{ms}\ k_M\ (\texttt{sm}\ k_M\ e_M)$

3. $e_S \simeq \texttt{sh}\ k_H\ (\texttt{hs}\ k_H\ e_S) \simeq \texttt{sm}\ k_M\ (\texttt{ms}\ k_M\ e_S)$

where $\simeq$ *denotes observational equivalence [1].*

Proof. By structural induction. □

The conversion of type abstractions between Haskell and ML was not straightforward. The application of a converted type abstraction cannot substitute the type argument into the nested expression because the type argument is meaningless in the nested expression's language. Instead, the application substitutes the type argument and a lump into the boundary's outer and inner types, respectively. Since the natural embedding requires the boundary's outer and inner types to be equal [3], we use a new notion of equality called lump equality that allows lumps within the boundary's inner type to match any corresponding type in the boundary's outer type.

Figures 2.1–2.3 present legends of symbol and syntax names; figure 2.4 presents the unbrand function; figure 2.5 presents the lump equality relation; figures 2.6–2.10 present Haskell; figures 2.11–2.15 present ML; and figures 2.16–2.20 present Scheme.

Figure 2.1: Symbol names

b	Brand
k	Conversion scheme
e	Expression
F	Forced evaluation context
f	Forced value
$\mathtt{L}$	Lump
$\doteq$	Lump equality relation
$\mathscr{F}$	Meta evaluation context
$\overline{n}$	Natural number
$\mathtt{N}$	Natural number
$\rightarrow$	Reduction relation
t	Type
y	Type variable
Γ	Typing environment
$\vdash$	Typing relation
U	Unforced evaluation context
u	Unforced value
x	Variable

Figure 2.2: Syntax names

$+\ e\ e$	Addition
$\texttt{if0}\ e\ e\ e$	Condition
$\texttt{nil}\ t$	Empty list
$\texttt{nil}$	Empty list
$\texttt{null}?\ e$	Empty list predicate
$\texttt{wrong}\ t\ \textit{string}$	Error
$\texttt{wrong}\ \textit{string}$	Error
$\lambda x : t.e$	Function abstraction
$\lambda x_S.e_S$	Function abstraction
$\texttt{fun}?\ e_S$	Function abstraction predicate
$e\ e$	Function application
$\texttt{hm}\ t_H\ t_M\ e_M$	Haskell-ML guard
$\texttt{hs}\ k_H\ e_S$	Haskell-Scheme guard
$\texttt{cons}\ e\ e$	List construction
$\texttt{hd}\ e$	List head
$\texttt{list}?\ e_S$	List predicate
$\texttt{tl}\ e$	List tail
$\texttt{mh}\ t_M\ t_H\ e_H$	ML-Haskell guard
$\texttt{ms}\ k_M\ e_S$	ML-Scheme guard
$\texttt{num}?\ e_S$	Number predicate
$\texttt{sh}\ k_H\ e_H$	Scheme-Haskell guard
$\texttt{sm}\ k_M\ e_M$	Scheme-ML guard
$-\ e\ e$	Subtraction
$\Lambda y.e$	Type abstraction
$e\ \langle t \rangle$	Type application

Figure 2.3: Syntax names

$b \diamond t$	Branded type
$\forall y.t$	Forall type
$\forall y.k$	Forall conversion scheme
$t \to t$	Function abstraction
$k \to k$	Function abstraction
$\{t\}$	List
$\{k\}$	List

Figure 2.4: The unbrand function.

$$
\begin{array}{rcl}
\lfloor \mathtt{L} \rfloor & = & \mathtt{L} \\
\lfloor \mathtt{N} \rfloor & = & \mathtt{N} \\
\lfloor y_H \rfloor & = & y_H \\
\lfloor y_M \rfloor & = & y_M \\
\lfloor \{k_H\} \rfloor & = & \{\lfloor k_H \rfloor\} \\
\lfloor \{k_M\} \rfloor & = & \{\lfloor k_M \rfloor\} \\
\lfloor k_H \rightarrow k_H \rfloor & = & \lfloor k_H \rfloor \rightarrow \lfloor k_H \rfloor \\
\lfloor k_M \rightarrow k_M \rfloor & = & \lfloor k_M \rfloor \rightarrow \lfloor k_M \rfloor \\
\lfloor \forall y_H . k_H \rfloor & = & \forall y_H . \lfloor k_H \rfloor \\
\lfloor \forall y_M . k_M \rfloor & = & \forall y_M . \lfloor k_M \rfloor \\
\lfloor b \diamond t_H \rfloor & = & t_H \\
\lfloor b \diamond t_M \rfloor & = & t_M
\end{array}
$$

Figure 2.5: The lump equality relation.

$$x \doteq x$$
$$x \doteq y \Rightarrow y \doteq x$$
$$x \doteq y \text{ and } y \doteq z \Rightarrow x \doteq z$$
$$t_H \doteq \mathtt{L}$$
$$t_M \doteq \mathtt{L}$$
$$t_H = t_M \Rightarrow t_H \doteq t_M$$

Figure 2.6: The Haskell syntax and evaluation contexts.

$$
\begin{array}{rcl}
e_H & = & x_H \mid f_H \mid e_H\ e_H \mid e_H\ \langle t_H \rangle \mid a\ e_H\ e_H \mid \texttt{if0}\ e_H\ e_H\ e_H \mid c\ e_H \\
 & & \texttt{null?}\ e_H \mid \texttt{wrong}\ t_H\ \mathit{string} \mid \texttt{hm}\ t_H\ t_M\ e_M \mid \texttt{hs}\ k_H\ e_S \\
f_H & = & \lambda x_H : t_H.e_H \mid \Lambda y_H.e_H \mid \overline{n} \mid \texttt{nil}\ t_H \mid \texttt{cons}\ e_H\ e_H \mid \texttt{hm}\ \texttt{L}\ t_M\ f_M \\
 & & \texttt{hs}\ \texttt{L}\ f_S \\
t_H & = & \texttt{L} \mid \texttt{N} \mid y_H \mid \{t_H\} \mid t_H \to t_H \mid \forall y_H.t_H \\
k_H & = & \texttt{L} \mid \texttt{N} \mid y_H \mid \{k_H\} \mid k_H \to k_H \mid \forall y_H.k_H \mid b \diamond t_H \\
a & = & + \mid - \\
c & = & \texttt{hd} \mid \texttt{tl} \\
F_H & = & [\,]_H \mid F_H\ e_H \mid F_H\ \langle t_H \rangle \mid a\ F_H\ e_H \mid a\ f_H\ F_H \mid \texttt{if0}\ F_H\ e_H\ e_H \\
 & & c\ F_H \mid \texttt{null?}\ F_H \mid \texttt{hm}\ t_H\ t_M\ F_M \mid \texttt{hs}\ k_H\ F_S
\end{array}
$$

Figure 2.7: The Haskell typing rules.

$$\frac{}{\vdash_H \mathtt{L}} \quad \frac{}{\vdash_H \mathtt{N}} \quad \frac{}{\Gamma, y_H \vdash_H y_H}$$

$$\frac{\Gamma \vdash_H t_H}{\Gamma \vdash_H \{t_H\}} \quad \frac{\Gamma \vdash_H t_H \quad \Gamma \vdash_H t'_H}{\Gamma \vdash_H t_H \to t'_H} \quad \frac{\Gamma, y_H \vdash_H t_H}{\Gamma \vdash_H \forall y_H.t_H}$$

$$\frac{\Gamma \vdash_H t_H \quad \Gamma, x_H : t_H \vdash_H e_H : t'_H}{\Gamma \vdash_H (\lambda x_H : t_H.e_H) : t_H \to t'_H} \quad \frac{\Gamma, y_H \vdash_H e_H : t_H}{\Gamma \vdash_H \Lambda y_H.e_H : \forall y_H.t_H} \quad \frac{}{\vdash_H \overline{n} : \mathtt{N}}$$

$$\frac{\Gamma \vdash_H t_H}{\Gamma \vdash_H \mathtt{nil}\ t_H : \{t_H\}} \quad \frac{\Gamma \vdash_H e_H : t_H \quad \Gamma \vdash_H e'_H : \{t_H\}}{\Gamma \vdash_H \mathtt{cons}\ e_H\ e'_H : \{t_H\}} \quad \frac{}{\Gamma, x_H : t_H \vdash_H x_H : t_H}$$

$$\frac{\Gamma \vdash_H e_H : t_H \to t'_H \quad \Gamma \vdash_H e'_H : t_H}{\Gamma \vdash_H e_H\ e'_H : t'_H} \quad \frac{\Gamma \vdash_H t_H \quad \Gamma \vdash_H e_H : \forall y_H.t'_H}{\Gamma \vdash_H e_H\ \langle t_H \rangle : t'_H[t_H/y_H]}$$

$$\frac{\Gamma \vdash_H e_H : \mathtt{N} \quad \Gamma \vdash_H e'_H : \mathtt{N}}{\Gamma \vdash_H a\ e_H\ e'_H : \mathtt{N}} \quad \frac{\Gamma \vdash_H e_H : \mathtt{N} \quad \Gamma \vdash_H e'_H : t_H \quad \Gamma \vdash_H e''_H : t_H}{\Gamma \vdash_H \mathtt{if0}\ e_H\ e'_H\ e''_H : t_H}$$

$$\frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathtt{null?}\ e_H : \mathtt{N}} \quad \frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathtt{hd}\ e_H : t_H} \quad \frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathtt{tl}\ e_H : \{t_H\}}$$

$$\frac{\Gamma \vdash_H t_H}{\Gamma \vdash_H \mathtt{wrong}\ t_H\ \mathit{string} : t_H} \quad \frac{\Gamma \vdash_H \lfloor k_H \rfloor \quad \Gamma \vdash_S e_S : \mathtt{TST}}{\Gamma \vdash_H \mathtt{hs}\ k_H\ e_S : \lfloor k_H \rfloor}$$

$$\frac{\Gamma \vdash_H t_H \quad \Gamma \vdash_M t_M \quad \Gamma \vdash_M e_M : t'_M \quad t_H \doteq t_M \quad t_M = t'_M}{\Gamma \vdash_H \mathtt{hm}\ t_H\ t_M\ e_M : t_H}$$

Figure 2.8: The Haskell operational semantics.

$$\mathscr{F}[(\lambda x_H : t_H.e_H)\ e'_H]_H \rightarrow \mathscr{F}[e_H[e'_H/x_H]]$$
$$\mathscr{F}[(\Lambda y_H.e_H)\ \langle t_H\rangle]_H \rightarrow \mathscr{F}[e_H[b \diamond t_H/y_H]]$$
$$\mathscr{F}[+\ \overline{n}\ \overline{n}']_H \rightarrow \mathscr{F}[\overline{n+n'}]$$
$$\mathscr{F}[-\ \overline{n}\ \overline{n}']_H \rightarrow \mathscr{F}[\overline{max(n-n',0)}]$$
$$\mathscr{F}[\texttt{if0}\ \overline{0}\ e_H\ e'_H]_H \rightarrow \mathscr{F}[e_H]$$
$$\mathscr{F}[\texttt{if0}\ \overline{n}\ e_H\ e'_H]_H \rightarrow \mathscr{F}[e'_H]\ (n \neq 0)$$
$$\mathscr{F}[\texttt{hd}\ (\texttt{nil}\ t_H)]_H \rightarrow \mathscr{F}[\texttt{wrong}\ t_H\ \text{“Empty list”}]$$
$$\mathscr{F}[\texttt{tl}\ (\texttt{nil}\ t_H)]_H \rightarrow \mathscr{F}[\texttt{wrong}\ \{t_H\}\ \text{“Empty list”}]$$
$$\mathscr{F}[\texttt{hd}\ (\texttt{cons}\ e_H\ e'_H)]_H \rightarrow \mathscr{F}[e_H]$$
$$\mathscr{F}[\texttt{tl}\ (\texttt{cons}\ e_H\ e'_H)]_H \rightarrow \mathscr{F}[e'_H]$$
$$\mathscr{F}[\texttt{null?}\ (\texttt{nil}\ t_H)]_H \rightarrow \mathscr{F}[\overline{0}]$$
$$\mathscr{F}[\texttt{null?}\ (\texttt{cons}\ e_H\ e'_H)]_H \rightarrow \mathscr{F}[\overline{1}]$$
$$\mathscr{F}[\texttt{wrong}\ t_H\ \mathit{string}]_H \rightarrow \textbf{Error: } \mathit{string}$$

Figure 2.9: The Haskell-ML operational semantics.

$$\mathscr{F}[\texttt{hm}\ t_H\ t_M\ (\texttt{mh}\ t'_M\ t'_H\ e_H)]_H \to \mathscr{F}[e_H]\quad (t_H \neq \texttt{L} \wedge t_H = t'_H)$$

$$\mathscr{F}[\texttt{hm}\ t_H\ t_M\ (\texttt{mh}\ t'_M\ t'_H\ e_H)]_H \to$$
$$\qquad \mathscr{F}[\texttt{wrong}\ t_H\ \text{“Type mismatch”}]\quad (t_H \neq \texttt{L} \wedge t_H \neq t'_H)$$

$$\mathscr{F}[\texttt{hm}\ t_H\ \texttt{L}\ (\texttt{ms}\ \texttt{L}\ f_S)]_H \to \mathscr{F}[\texttt{wrong}\ t_H\ \text{“Bad value”}]\quad (t_H \neq \texttt{L})$$

$$\mathscr{F}[\texttt{hm}\ \texttt{N}\ \texttt{N}\ \overline{n}]_H \to \mathscr{F}[\overline{n}]$$

$$\mathscr{F}[\texttt{hm}\ \{t_H\}\ \{t_M\}\ (\texttt{nil}\ t'_M)]_H \to \mathscr{F}[\texttt{nil}\ t_H]$$

$$\mathscr{F}[\texttt{hm}\ \{t_H\}\ \{t_M\}\ (\texttt{cons}\ u_M\ u'_M)]_H \to$$
$$\qquad \mathscr{F}[\texttt{cons}\ (\texttt{hm}\ t_H\ t_M\ u_M)\ (\texttt{hm}\ \{t_H\}\ \{t_M\}\ u'_M)]$$

$$\mathscr{F}[\texttt{hm}\ (t_H \to t'_H)\ (t_M \to t'_M)\ (\lambda x_M : t''_M.e_M)]_H \to$$
$$\qquad \mathscr{F}[\lambda x_H : t_H.\texttt{hm}\ t'_H\ t'_M\ ((\lambda x_M : t''_M.e_M)\ (\texttt{mh}\ t_M\ t_H\ x_H))]$$

$$\mathscr{F}[\texttt{hm}\ (\forall y_H.t_H)\ (\forall y_M.t_M)\ (\Lambda y'_M.e_M)]_H \to \mathscr{F}[\Lambda y_H.\texttt{hm}\ t_H\ t_M[\texttt{L}/y_M]\ e_M[\texttt{L}/y'_M]]$$

Figure 2.10: The Haskell-Scheme operational semantics.

$\mathscr{F}[\texttt{hs}\ t_H\ (\texttt{sh}\ t'_H\ e_H)]_H \rightarrow \mathscr{F}[e_H]\ (t_H = t'_H)$

$\mathscr{F}[\texttt{hs}\ \texttt{N}\ \overline{n}]_H \rightarrow \mathscr{F}[\overline{n}]$

$\mathscr{F}[\texttt{hs}\ \texttt{N}\ f_S]_H \rightarrow \mathscr{F}[\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}]\ (f_S \neq \overline{n})$

$\mathscr{F}[\texttt{hs}\ \{k_H\}\ \texttt{nil}]_H \rightarrow \mathscr{F}[\texttt{nil}\ \lfloor k_H \rfloor]$

$\mathscr{F}[\texttt{hs}\ \{k_H\}\ (\texttt{cons}\ u_S\ u'_S)]_H \rightarrow \mathscr{F}[\texttt{cons}\ (\texttt{hs}\ k_H\ u_S)\ (\texttt{hs}\ \{k_H\}\ u'_S)]$

$\mathscr{F}[\texttt{hs}\ \{k_H\}\ f_S]_H \rightarrow \mathscr{F}[\texttt{wrong}\ \lfloor \{k_H\} \rfloor\ \text{“Not a list”}]$
$\quad (f_S \neq \texttt{nil} \text{ and } f_S \neq \texttt{cons}\ u_S\ u'_S)$

$\mathscr{F}[\texttt{hs}\ (b \diamond t_H)\ (\texttt{sh}\ (b \diamond t_H)\ e_H)]_H \rightarrow \mathscr{F}[e_H]$

$\mathscr{F}[\texttt{hs}\ (b \diamond t_H)\ f_S]_H \rightarrow \mathscr{F}[\texttt{wrong}\ t_H\ \text{“Brand mismatch”}]\ (f_S \neq \texttt{sh}\ (b \diamond t_H)\ e_H)$

$\mathscr{F}[\texttt{hs}\ (k_H \rightarrow k'_H)\ (\lambda x_S.e_S)]_H \rightarrow \mathscr{F}[\lambda x_H : \lfloor k_H \rfloor.\texttt{hs}\ k'_H\ ((\lambda x_S.e_S)\ (\texttt{sh}\ k_H\ x_H))]$

$\mathscr{F}[\texttt{hs}\ (k_H \rightarrow k'_H)\ f_S]_H \rightarrow \mathscr{F}[\texttt{wrong}\ \lfloor k_H \rightarrow k'_H \rfloor\ \text{“Not a function”}]$
$\quad (f_S \neq \lambda x_S.e_S)$

$\mathscr{F}[\texttt{hs}\ (\forall y_H.k_H)\ f_S]_H \rightarrow \mathscr{F}[\Lambda y_H.\texttt{hs}\ k_H\ f_S]$

Figure 2.11: The ML syntax and evaluation contexts.

$$
\begin{array}{rcl}
e_M & = & x_M \mid u_M \mid e_M\ e_M \mid e_M\ \langle t_M \rangle \mid a\ e_M\ e_M \mid \texttt{if0}\ e_M\ e_M\ e_M \\
& & \texttt{cons}\ e_M\ e_M \mid c\ e_M \mid \texttt{null?}\ e_M \mid \texttt{wrong}\ t_M\ \mathit{string} \mid \texttt{ms}\ k_M\ e_S \\
u_M & = & f_M \mid \texttt{mh}\ t_M\ t_H\ e_H \\
f_M & = & \lambda x_M : t_M . e_M \mid \Lambda y_M . e_M \mid \overline{n} \mid \texttt{nil}\ t_M \mid \texttt{cons}\ u_M\ u_M \mid \texttt{mh}\ \texttt{L}\ t_H\ e_H \\
& & \texttt{ms}\ \texttt{L}\ f_S \\
t_M & = & \texttt{L} \mid \texttt{N} \mid y_M \mid \{t_M\} \mid t_M \to t_M \mid \forall y_M . t_M \\
k_M & = & \texttt{L} \mid \texttt{N} \mid y_M \mid \{k_M\} \mid k_M \to k_M \mid \forall y_M . k_M \mid b \diamond t_M \\
a & = & + \mid - \\
c & = & \texttt{hd} \mid \texttt{tl} \\
F_M & = & U_M \mid \texttt{mh}\ t_M\ t_H\ F_H \\
U_M & = & [\,]_M \mid F_M\ e_M \mid f_M\ U_M \mid F_M\ \langle t_M \rangle \mid a\ F_M\ e_M \mid a\ f_M\ F_M \\
& & \texttt{if0}\ F_M\ e_M\ e_M \mid \texttt{cons}\ U_M\ e_M \mid \texttt{cons}\ u_M\ U_M \mid c\ F_M \mid \texttt{null?}\ F_M \\
& & \texttt{ms}\ k_M\ F_S
\end{array}
$$

Figure 2.12: The ML typing rules.

$$\frac{}{\vdash_M \mathtt{L}} \quad \frac{}{\vdash_M \mathtt{N}} \quad \frac{}{\Gamma, y_M \vdash_M y_M}$$

$$\frac{\Gamma \vdash_M t_M}{\Gamma \vdash_M \{t_M\}} \quad \frac{\Gamma \vdash_M t_M \quad \Gamma \vdash_M t'_M}{\Gamma \vdash_M t_M \to t'_M} \quad \frac{\Gamma, y_M \vdash_M t_M}{\Gamma \vdash_M \forall y_M.t_M}$$

$$\frac{\Gamma \vdash_M t_M \quad \Gamma, x_M : t_M \vdash_M e_M : t'_M}{\Gamma \vdash_M (\lambda x_M : t_M.e_M) : t_M \to t'_M} \quad \frac{\Gamma, y_M \vdash_M e_M : t_M}{\Gamma \vdash_M \Lambda y_M.e_M : \forall y_M.t_M} \quad \frac{}{\vdash_M \overline{n} : \mathtt{N}}$$

$$\frac{\Gamma \vdash_M t_M}{\Gamma \vdash_M \mathtt{nil}\ t_M : \{t_M\}} \quad \frac{\Gamma \vdash_M e_M : t_M \quad \Gamma \vdash_M e'_M : \{t_M\}}{\Gamma \vdash_M \mathtt{cons}\ e_M\ e'_M : \{t_M\}} \quad \frac{}{\Gamma, x_M : t_M \vdash_M x_M : t_M}$$

$$\frac{\Gamma \vdash_M e_M : t_M \to t'_M \quad \Gamma \vdash_M e'_M : t_M}{\Gamma \vdash_H e_M\ e'_M : t'_M} \quad \frac{\Gamma \vdash_M t_M \quad \Gamma \vdash_M e_M : \forall y_M.t'_M}{\Gamma \vdash_M e_M\ \langle t_M \rangle : t'_M[t_M/y_M]}$$

$$\frac{\Gamma \vdash_M e_M : \mathtt{N} \quad \Gamma \vdash_M e'_M : \mathtt{N}}{\Gamma \vdash_M a\ e_M\ e'_M : \mathtt{N}} \quad \frac{\Gamma \vdash_M e_M : \mathtt{N} \quad \Gamma \vdash_M e'_M : t_M \quad \Gamma \vdash_M e''_M : t_M}{\Gamma \vdash_M \mathtt{if0}\ e_M\ e'_M\ e''_M : t_M}$$

$$\frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathtt{null?}\ e_M : \mathtt{N}} \quad \frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathtt{hd}\ e_M : t_M} \quad \frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathtt{tl}\ e_M : \{t_M\}}$$

$$\frac{\Gamma \vdash_M t_M}{\Gamma \vdash_M \mathtt{wrong}\ t_M\ \mathit{string} : t_M} \quad \frac{\Gamma \vdash_M \lfloor k_M \rfloor \quad \Gamma \vdash_S e_S : \mathtt{TST}}{\Gamma \vdash_M \mathtt{ms}\ k_M\ e_S : \lfloor k_M \rfloor}$$

$$\frac{\Gamma \vdash_M t_M \quad \Gamma \vdash_H t_H \quad \Gamma \vdash_H e_H : t'_H \quad t_M \doteq t_H \quad t_H = t'_H}{\Gamma \vdash_M \mathtt{mh}\ t_M\ t_H\ e_H : t_M}$$

Figure 2.13: The ML operational semantics.

$$\mathscr{F}[(\lambda x_M : t_M.e_M)\ u_M]_M \rightarrow \mathscr{F}[e_M[u_M/x_M]]$$
$$\mathscr{F}[(\Lambda y_M.e_M)\ \langle t_M \rangle]_M \rightarrow \mathscr{F}[e_M[b \diamond t_M/y_M]]$$
$$\mathscr{F}[+\ \overline{n}\ \overline{n}']_M \rightarrow \mathscr{F}[\overline{n+n'}]$$
$$\mathscr{F}[-\ \overline{n}\ \overline{n}']_M \rightarrow \mathscr{F}[\overline{max(n-n',0)}]$$
$$\mathscr{F}[\texttt{if0}\ \overline{0}\ e_M\ e'_M]_M \rightarrow \mathscr{F}[e_M]$$
$$\mathscr{F}[\texttt{if0}\ \overline{n}\ e_M\ e'_M]_M \rightarrow \mathscr{F}[e'_M]\ (n \neq 0)$$
$$\mathscr{F}[\texttt{hd}\ (\texttt{nil}\ t_M)]_M \rightarrow \mathscr{F}[\texttt{wrong}\ t_M\ \text{“Empty list”}]$$
$$\mathscr{F}[\texttt{tl}\ (\texttt{nil}\ t_M)]_M \rightarrow \mathscr{F}[\texttt{wrong}\ \{t_M\}\ \text{“Empty list”}]$$
$$\mathscr{F}[\texttt{hd}\ (\texttt{cons}\ u_M\ u'_M)]_M \rightarrow \mathscr{F}[u_M]$$
$$\mathscr{F}[\texttt{tl}\ (\texttt{cons}\ u_M\ u'_M)]_M \rightarrow \mathscr{F}[u'_M]$$
$$\mathscr{F}[\texttt{null?}\ (\texttt{nil}\ t_M)]_M \rightarrow \mathscr{F}[\overline{0}]$$
$$\mathscr{F}[\texttt{null?}\ (\texttt{cons}\ u_M\ u'_M)]_M \rightarrow \mathscr{F}[\overline{1}]$$
$$\mathscr{F}[\texttt{wrong}\ t_M\ \mathit{string}]_H \rightarrow \textbf{Error: } \mathit{string}$$

Figure 2.14: The ML-Haskell operational semantics.

$$\mathscr{F}[\texttt{mh}\ t_M\ t_H\ (\texttt{hm}\ t'_H\ t'_M\ f_M)]_M \rightarrow \mathscr{F}[f_M]\ (t_M \neq \texttt{L} \wedge t_M = t'_M)$$

$$\mathscr{F}[\texttt{mh}\ t_M\ t_H\ (\texttt{hm}\ t'_H\ t'_M\ f_M)]_M \rightarrow$$

$$\quad \mathscr{F}[\texttt{wrong}\ t_M\ \text{“Type mismatch”}]\ (t_M \neq \texttt{L} \wedge t_M \neq t'_M)$$

$$\mathscr{F}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hs}\ \texttt{L}\ f_S)]_H \rightarrow \mathscr{F}[\texttt{wrong}\ t_M\ \text{“Bad value”}]\ (t_M \neq \texttt{L})$$

$$\mathscr{F}[\texttt{mh}\ \texttt{N}\ \texttt{N}\ \overline{n}]_M \rightarrow \mathscr{F}[\overline{n}]$$

$$\mathscr{F}[\texttt{mh}\ \{t_M\}\ \{t_H\}\ (\texttt{nil}\ t'_H)]_M \rightarrow \mathscr{F}[\texttt{nil}\ t_M]$$

$$\mathscr{F}[\texttt{mh}\ \{t_M\}\ \{t_H\}\ (\texttt{cons}\ e_H\ e'_H)]_M \rightarrow$$

$$\quad \mathscr{F}[\texttt{cons}\ (\texttt{mh}\ t_M\ t_H\ e_H)\ (\texttt{mh}\ \{t_M\}\ \{t_H\}\ e'_H)]$$

$$\mathscr{F}[\texttt{mh}\ (t_M \rightarrow t'_M)\ (t_H \rightarrow t'_H)\ (\lambda x_H : t''_H.e_H)]_M \rightarrow$$

$$\quad \mathscr{F}[\lambda x_M : t_M.\texttt{mh}\ t'_M\ t'_H\ ((\lambda x_H : t''_H.e_H)\ (\texttt{hm}\ t_H\ t_M\ x_M))]$$

$$\mathscr{F}[\texttt{mh}\ (\forall y_M.t_M)\ (\forall y_H.t_H)\ (\Lambda y'_H.e_H)]_M \rightarrow \mathscr{F}[\Lambda y_M.\texttt{mh}\ t_M\ t_H[\texttt{L}/y_H]\ e_H[\texttt{L}/y'_H]]$$

Figure 2.15: The ML-Scheme operational semantics.

$\mathscr{F}[\texttt{ms N}\ \overline{n}]_M \to \mathscr{F}[\overline{n}]$

$\mathscr{F}[\texttt{ms N}\ f_S]_M \to \mathscr{F}[\texttt{wrong N}\ \text{“Not a number”}]\ (f_S \neq \overline{n})$

$\mathscr{F}[\texttt{ms}\ \{k_M\}\ \texttt{nil}]_M \to \mathscr{F}[\texttt{nil}\ \lfloor k_M \rfloor]$

$\mathscr{F}[\texttt{ms}\ \{k_M\}\ (\texttt{cons}\ u_S\ u'_S)]_M \to \mathscr{F}[\texttt{cons}\ (\texttt{ms}\ k_M\ u_S)\ (\texttt{ms}\ \{k_M\}\ u'_S)]$

$\mathscr{F}[\texttt{ms}\ \{k_M\}\ f_S]_M \to \mathscr{F}[\texttt{wrong}\ \lfloor \{k_M\} \rfloor\ \text{“Not a list”}]$
$\quad (f_S \neq \texttt{nil}\ \text{and}\ f_S \neq \texttt{cons}\ u_S\ u'_S)$

$\mathscr{F}[\texttt{ms}\ (b \diamond t_M)\ (\texttt{sm}\ (b \diamond t_M)\ u_M)]_M \to \mathscr{F}[u_M]$

$\mathscr{F}[\texttt{ms}\ (b \diamond t_M)\ f_S]_M \to \mathscr{F}[\texttt{wrong}\ \lfloor b \diamond t_M \rfloor\ \text{“Brand mismatch”}]$
$\quad (f_S \neq \texttt{sm}\ (b \diamond t_M)\ e_M)$

$\mathscr{F}[\texttt{ms}\ (k_M \to k'_M)\ (\lambda x_S.e_S)]_M \to$
$\quad \mathscr{F}[\lambda x_M : \lfloor k_M \rfloor.\texttt{ms}\ k'_M\ ((\lambda x_S.e_S)\ (\texttt{sm}\ k_M\ x_M))]$

$\mathscr{F}[\texttt{ms}\ (k_M \to k'_M)\ f_S]_M \to \mathscr{F}[\texttt{wrong}\ \lfloor k_M \to k'_M \rfloor\ \text{“Not a function”}]$
$\quad (f_S \neq \lambda x_S.e_S)$

$\mathscr{F}[\texttt{ms}\ (\forall y_M.k_M)\ f_S]_M \to \mathscr{F}[\Lambda y_M.\texttt{ms}\ k_M\ f_S]$

Figure 2.16: The Scheme syntax and evaluation contexts.

$$
\begin{array}{lcl}
e_S & = & x_S \mid u_S \mid e_S\ e_S \mid a\ e_S\ e_S \mid p\ e_S \mid \texttt{if0}\ e_S\ e_S\ e_S \mid \texttt{cons}\ e_S\ e_S \mid c\ e_S \\
 & & \texttt{wrong}\ \mathit{string} \mid \texttt{sm}\ k_M\ e_M \\
u_S & = & f_S \mid \texttt{sh}\ k_H\ e_H \\
f_S & = & \lambda x_S.e_S \mid \overline{n} \mid \texttt{nil} \mid \texttt{cons}\ u_S\ u_S \mid \texttt{sh}\ (b \diamond t_H)\ e_H \mid \texttt{sm}\ (b \diamond t_M)\ f_M \\
a & = & + \mid - \\
c & = & \texttt{hd} \mid \texttt{tl} \\
p & = & \texttt{fun}? \mid \texttt{list}? \mid \texttt{null}? \mid \texttt{num}? \\
F_S & = & U_S \mid \texttt{sh}\ k_H\ F_H \\
U_S & = & [\,]_S \mid F_S\ e_S \mid f_S\ U_S \mid a\ F_S\ e_S \mid a\ f_S\ F_S \mid p\ F_S \mid \texttt{if0}\ F_S\ e_S\ e_S \\
 & & \texttt{cons}\ U_S\ e_S \mid \texttt{cons}\ u_S\ U_S \mid c\ F_S \mid \texttt{sm}\ k_M\ F_M
\end{array}
$$

Figure 2.17: The Scheme typing rules.

$$\frac{}{\vdash_S \texttt{TST}}$$

$$\frac{\Gamma, x_S : \texttt{TST} \vdash_S e_S : \texttt{TST}}{\Gamma \vdash_S \lambda x_S . e_S : \texttt{TST}} \quad \frac{}{\vdash_S \overline{n} : \texttt{TST}} \quad \frac{}{\vdash_S \texttt{nil} : \texttt{TST}}$$

$$\frac{\Gamma \vdash_S e_S : \texttt{TST} \quad \Gamma \vdash_S e'_S : \texttt{TST}}{\Gamma \vdash_S \texttt{cons}\ e_S\ e'_S : \texttt{TST}} \quad \frac{}{\Gamma, x_S : \texttt{TST} \vdash_S x_S : \texttt{TST}}$$

$$\frac{\Gamma \vdash_S e_S : \texttt{TST} \quad \Gamma \vdash_S e'_S : \texttt{TST}}{\Gamma \vdash_H e_S\ e'_S : \texttt{TST}} \quad \frac{\Gamma \vdash_S e_S : \texttt{TST} \quad \Gamma \vdash_S e'_S : \texttt{TST}}{\Gamma \vdash_S a\ e_S\ e'_S : \texttt{TST}}$$

$$\frac{\Gamma \vdash_S e_S : \texttt{TST} \quad \Gamma \vdash_S e'_S : \texttt{TST} \quad \Gamma \vdash_S e''_S : \texttt{TST}}{\Gamma \vdash_S \texttt{if0}\ e_S\ e'_S\ e''_S : \texttt{TST}} \quad \frac{\Gamma \vdash_S e_S : \texttt{TST}}{\Gamma \vdash_S p\ e_S : \texttt{TST}}$$

$$\frac{\Gamma \vdash_S e_S : \texttt{TST}}{\Gamma \vdash_S c\ e_S : \texttt{TST}} \quad \frac{}{\vdash_S \texttt{wrong}\ \mathit{string} : \texttt{TST}}$$

$$\frac{\Gamma \vdash_H \lfloor k_H \rfloor \quad \Gamma \vdash_H e_H : t_H \quad \lfloor k_H \rfloor = t_H}{\Gamma \vdash_S \texttt{sh}\ k_H\ e_H : \texttt{TST}}$$

$$\frac{\Gamma \vdash_M \lfloor k_M \rfloor \quad \Gamma \vdash_M e_M : t_M \quad \lfloor k_M \rfloor = t_M}{\Gamma \vdash_S \texttt{sm}\ k_M\ e_M : \texttt{TST}}$$

Figure 2.18: The Scheme operational semantics.

$\mathscr{F}[(\lambda x_S.e_S)\ u_S]_S \rightarrow \mathscr{F}[e_S[u_S/x_S]]$

$\mathscr{F}[f_S\ u_S]_S \rightarrow \mathscr{F}[\texttt{wrong}$ "Not a function"$]$ $(f_S \neq \lambda x_S.e_S)$

$\mathscr{F}[+\ \overline{n}\ \overline{n}']_S \rightarrow \mathscr{F}[\overline{n+n'}]$

$\mathscr{F}[-\ \overline{n}\ \overline{n}']_S \rightarrow \mathscr{F}[\overline{max(n-n',0)}]$

$\mathscr{F}[a\ f_S\ f'_S]_S \rightarrow \mathscr{F}[\texttt{wrong}$ "Not a number"$]$ $(f_S \neq \overline{n}$ or $f'_S \neq \overline{n})$

$\mathscr{F}[\texttt{if0}\ \overline{0}\ e_S\ e'_S]_S \rightarrow \mathscr{F}[e_S]$

$\mathscr{F}[\texttt{if0}\ \overline{n}\ e_S\ e'_S]_S \rightarrow \mathscr{F}[e'_S]$ $(n \neq 0)$

$\mathscr{F}[\texttt{if0}\ f_S\ e_S\ e'_S]_S \rightarrow \mathscr{F}[\texttt{wrong}$ "Not a number"$]$ $(f_S \neq \overline{n})$

$\mathscr{F}[c\ \texttt{nil}]_S \rightarrow \mathscr{F}[\texttt{wrong}$ "Empty list"$]$

$\mathscr{F}[\texttt{hd}\ (\texttt{cons}\ u_S\ u'_S)]_S \rightarrow \mathscr{F}[u_S]$

$\mathscr{F}[\texttt{tl}\ (\texttt{cons}\ u_S\ u'_S)]_S \rightarrow \mathscr{F}[u'_S]$

$\mathscr{F}[c\ f_S]_S \rightarrow \mathscr{F}[\texttt{wrong}$ "Not a list"$]$ $(f_S \neq \texttt{nil}$ and $f_S \neq \texttt{cons}\ u_S\ u'_S)$

$\mathscr{F}[\texttt{fun?}\ (\lambda x_S.e_S)]_S \rightarrow \mathscr{F}[\overline{0}]$

$\mathscr{F}[\texttt{fun?}\ f_S]_S \rightarrow \mathscr{F}[\overline{1}]$ $(f_S \neq \lambda x_S.e_S)$

$\mathscr{F}[\texttt{list?}\ \texttt{nil}]_S \rightarrow \mathscr{F}[\overline{0}]$

$\mathscr{F}[\texttt{list?}\ (\texttt{cons}\ u_S\ u'_S)]_S \rightarrow \mathscr{F}[\overline{0}]$

$\mathscr{F}[\texttt{list?}\ f_S]_S \rightarrow \mathscr{F}[\overline{1}]$ $(f_S \neq \texttt{nil}$ and $f_S \neq \texttt{cons}\ u_S\ u'_S)$

$\mathscr{F}[\texttt{null?}\ \texttt{nil}]_S \rightarrow \mathscr{F}[\overline{0}]$

$\mathscr{F}[\texttt{null?}\ f_S]_S \rightarrow \mathscr{F}[\overline{1}]$ $(f_S \neq \texttt{nil})$

$\mathscr{F}[\texttt{num?}\ \overline{n}]_S \rightarrow \mathscr{F}[\overline{0}]$

$\mathscr{F}[\texttt{num?}\ f_S]_S \rightarrow \mathscr{F}[\overline{1}]$ $(f_S \neq \overline{n})$

$\mathscr{F}[\texttt{wrong}\ \textit{string}]_S \rightarrow$ **Error:** *string*

Figure 2.19: The Scheme-Haskell operational semantics.

$\mathscr{F}[\texttt{sh}\ k_H\ (\texttt{hs}\ k'_H\ f_S)]_S \to \mathscr{F}[f_S]$

$\mathscr{F}[\texttt{sh}\ \texttt{L}\ (\texttt{hm}\ \texttt{L}\ k_M\ f_M)]_S \to \mathscr{F}[\texttt{wrong}\ \text{“Bad value”}]$

$\mathscr{F}[\texttt{sh}\ \texttt{N}\ \overline{n}]_S \to \mathscr{F}[\overline{n}]$

$\mathscr{F}[\texttt{sh}\ \{k_H\}\ (\texttt{nil}\ t_H)]_S \to \mathscr{F}[\texttt{nil}]$

$\mathscr{F}[\texttt{sh}\ \{k_H\}\ (\texttt{cons}\ e_H\ e'_H)]_S \to \mathscr{F}[\texttt{cons}\ (\texttt{sh}\ k_H\ e_H)\ (\texttt{sh}\ \{k_H\}\ e'_H)]$

$\mathscr{F}[\texttt{sh}\ (k_H \to k'_H)\ (\lambda x_H : t_H.e_H)]_S \to$

$\quad \mathscr{F}[\lambda x_S.\texttt{sh}\ k'_H\ ((\lambda x_H : t_H.e_H)\ (\texttt{hs}\ k_H\ x_S))]$

$\mathscr{F}[\texttt{sh}\ (\forall y_H.k_H)\ (\Lambda y'_H.e_H)]_S \to \mathscr{F}[\texttt{sh}\ k_H[\texttt{L}/y_H]\ e_H[\texttt{L}/y'_H]]$

Figure 2.20: The Scheme-ML operational semantics.

$$\mathscr{F}[\texttt{sm}\ \texttt{L}\ (\texttt{ms}\ \texttt{L}\ f_S)]_S \rightarrow \mathscr{F}[f_S]$$

$$\mathscr{F}[\texttt{sm}\ \texttt{L}\ (\texttt{mh}\ \texttt{L}\ k_H\ e_H)]_S \rightarrow \mathscr{F}[\texttt{wrong}\ \text{“Bad value”}]$$

$$\mathscr{F}[\texttt{sm}\ \texttt{N}\ \overline{n}]_S \rightarrow \mathscr{F}[\overline{n}]$$

$$\mathscr{F}[\texttt{sm}\ \{k_M\}\ (\texttt{nil}\ t_M)]_S \rightarrow \mathscr{F}[\texttt{nil}]$$

$$\mathscr{F}[\texttt{sm}\ \{k_M\}\ (\texttt{cons}\ u_M\ u'_M)]_S \rightarrow \mathscr{F}[\texttt{cons}\ (\texttt{sm}\ k_M\ u_M)\ (\texttt{sm}\ \{k_M\}\ u'_M)]$$

$$\mathscr{F}[\texttt{sm}\ (k_M \rightarrow k'_M)\ f_M]_S \rightarrow \mathscr{F}[\lambda x_S.\texttt{sm}\ k'_M\ (f_M\ (\texttt{ms}\ k_M\ x_S))]$$

$$\mathscr{F}[\texttt{sm}\ (\forall y_M.k_M)\ (\Lambda y'_M.e_M)]_S \rightarrow \mathscr{F}[\texttt{sm}\ k_M[\texttt{L}/y_M]\ e_M[\texttt{L}/y'_M]]$$

Chapter 3

Proof of Type Soundness

The proof of correctness is similar to that of Kinghorn [2], mutatis mutandis.

Lemma 1. *Inversion of the Typing Relation*

The syntactic forms of well-typed expressions determine the types of their subexpressions.

Proof. Immediate from the typing rules. □

Lemma 2. *Uniqueness of Types*

If e_H, e_M, and e_S are well-typed then they have only one type.

Proof. By structural induction on e_H, e_M, and e_S and lemma 1. □

Lemma 3. *Canonical Forms*

The syntactic forms of unforced values for each type.

Proof. Immediate from the definitions of unforced values and the typing relations. □

Theorem 2. *Haskell Progress*

If $\vdash_H e_H : t_H$ *then* e_H *is an unforced value or* $e_H \to e'_H$ *or* $e_H \to$ **Error:** *string.*

Proof. By structural induction on e_H and theorems 3 and 4. □

Theorem 3. *ML Progress*

If $\vdash_M e_M : t_M$ *then* e_M *is an unforced value or* $e_M \to e'_M$ *or* $e_M \to$ **Error:** *string.*

Proof. By structural induction on e_M and theorems 2 and 4. □

Theorem 4. *Scheme Progress*

If $\vdash_S e_S : \texttt{TST}$ *then* e_S *is an unforced value or* $e_S \to e'_S$ *or* $e_S \to$ **Error:** *string.*

Proof. By structural induction on e_S and theorems 2 and 3. □

Lemma 4. *Expression Substitution Preservation*

If $\Gamma, x_H : t_H \vdash_H e_H : t'_H$ *and* $\Gamma \vdash_H e'_H : t_H$ *then* $\Gamma \vdash_H e_H[e'_H/x_H] : t'_H$. *If* $\Gamma, x_M : t_M \vdash_M e_M : t'_M$ *and* $\Gamma \vdash_M e'_M : t_M$ *then* $\Gamma \vdash_M e_M[e'_M/x_M] : t'_M$. *If* $\Gamma, x_S : \texttt{TST} \vdash_S e_S : \texttt{TST}$ *and* $\Gamma \vdash_S e'_S : \texttt{TST}$ *then* $\Gamma \vdash_S e_S[e'_S/x_S] : \texttt{TST}$.

Proof. By structural induction. □

Lemma 5. *Type Substitution Preservation*

If $\Gamma, y_H \vdash_H e_H : t_H$ *and* $\Gamma \vdash_H t'_H$ *then* $\Gamma \vdash_H e_H[t'_H/y_H] : t_H[t'_H/y_H]$. *If* $\Gamma, y_M \vdash_M e_M : t_M$ *and* $\Gamma \vdash_M t'_M$ *then* $\Gamma \vdash_M e_M[t'_M/y_M] : t_M[t'_M/y_M]$.

Proof. By structural induction. □

Lemma 6. *Evaluation Context Preservation*

If $\vdash_H e_H : t_H$, $\vdash_H e'_H : t_H$, *and* $\vdash_H \mathscr{F}[e_H]_H : t'_H$ *then* $\vdash_H \mathscr{F}[e'_H]_H : t'_H$. *If* $\vdash_M e_M : t_M$, $\vdash_M e'_M : t_M$, *and* $\vdash_M \mathscr{F}[e_M]_M : t'_M$ *then* $\vdash_M \mathscr{F}[e'_M]_M : t'_M$. *If* $\vdash_S e_S : \texttt{TST}$, $\vdash_S e'_S : \texttt{TST}$, *and* $\vdash_S \mathscr{F}[e_S]_S : \texttt{TST}$ *then* $\vdash_S \mathscr{F}[e'_S]_S : \texttt{TST}$.

Proof. By structural induction. □

Theorem 5. *Haskell Preservation*

If $\Gamma \vdash_H e_H : t_H$ *and* $\mathscr{F}[e_H]_H \rightarrow \mathscr{F}[e'_H]$ *then* $\Gamma \vdash_H e'_H : t_H$.

Proof. By cases on the reduction $\mathscr{F}[e_H]_H \rightarrow \mathscr{F}[e'_H]$, lemma 6, and theorems 6 and 7. □

Theorem 6. *ML Preservation*

If $\Gamma \vdash_M e_M : t_M$ *and* $e_M \rightarrow e'_M$ *then* $\Gamma \vdash_M e'_M : t_M$.

Proof. By cases on the reduction $\mathscr{F}[e_M]_H \rightarrow \mathscr{F}[e'_M]$, lemma 6, and theorems 5 and 7. □

Theorem 7. *Scheme Preservation*

If $\Gamma \vdash_S e_S : \texttt{TST}$ *and* $e_S \rightarrow e'_S$ *then* $\Gamma \vdash_S e'_S : \texttt{TST}$.

Proof. By cases on the reduction $\mathscr{F}[e_S]_S \rightarrow \mathscr{F}[e'_S]$, lemma 6, and theorems 5 and 6. □

Chapter 4

Conclusion

Our method of interoperation resolves incompatible lazy and eager evaluation strategies transparently. The eager evaluation strategy mirrors the lazy one for reducible expressions inside the lazy language's boundaries in incompatible evaluation contexts common to both languages. Forced and unforced evaluation contexts and values comprise a simple framework that implements such a system. This method could be extended to resolve incompatible evaluation strategies for any pair of languages with common expressions.

Bibliography

[1] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[2] D. L. Kinghorn. Preserving parametricity while sharing higher-order, polymorphic functions between scheme and ml. Master's thesis, California Polytechnic State University, San Luis Obispo, June 2007.

[3] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *SIGPLAN Not.*, 42(1):3–10, 2007.