Interoperation for Lazy and Eager Evaluation

William Faught

May 10, 2011

Abstract

Programmers forgo existing solutions to problems in other programming languages where interoperation proves too cumbersome; they remake solutions, rather than reuse them. To facilitate reuse, interoperation must resolve incompatible programming language features transparently at the boundaries between languages. To address part of this problem, this paper presents a model of computation that resolves lazy and eager evaluation strategies. Unforced values act as thunks that are used and forced where appropriate by the languages themselves and do not require programmer forethought.

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 lazy and eager evaluation strategies.

Matthews and Findler presented a method of safe interoperation between languages with incompatible polymorphic static and dynamic type systems [1]. We observe that their method is insufficient for safe interoperation between languages with incompatible lazy and eager evaluation strategies, then explain the underlying problem, and then finally 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. Hereafter, we use the

$\texttt{sh}\ \{\texttt{N}\}\ (\texttt{cons}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”})\ (\texttt{nil}\ \texttt{N})) \rightarrow$

$\texttt{cons}\ (\texttt{sh}\ \texttt{N}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}))\ (\texttt{sh}\ \{\texttt{N}\}\ (\texttt{nil}\ \texttt{N})) \rightarrow$

Error: “Not a number”

Figure 1: Scheme forces the conversion of list construction operands.

$\texttt{sh}\ \{\texttt{N}\}\ (\texttt{cons}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”})\ (\texttt{nil}\ \texttt{N})) \rightarrow$

$\texttt{cons}\ (\texttt{sh}\ \texttt{N}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}))\ (\texttt{sh}\ \{\texttt{N}\}\ (\texttt{nil}\ \texttt{N})) \rightarrow$

$\texttt{cons}\ (\texttt{sh}\ \texttt{N}\ (\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}))\ (\texttt{nil}\ \texttt{N})$

Figure 2: Scheme does not force the conversion of list construction operands.

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 strictness points 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 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 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 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

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

Figure 3: Scheme forces the conversion of arguments.

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

Figure 4: Scheme does not force the conversion of arguments.

have been used, but it was not. Figure 4 demonstrates Scheme not forcing the evaluation of the Haskell argument, which allows the Scheme function to produce a number.

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 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 evalation 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. $e_H \equiv \texttt{mh}\ t_M\ t_H\ e_H \equiv \texttt{sh}\ t_H\ e_H$
2. $e_M \equiv \texttt{hm}\ t_H\ t_M\ e_M \equiv \texttt{sm}\ t_M\ e_M$
3. $e_S \equiv \texttt{hs}\ t_H\ e_S \equiv \texttt{ms}\ t_M\ e_S$

where $\equiv$ *denotes extensional equality.*

Proof. By structural induction. □

value conversion

Another complication with introducing Haskell to the model of computation is how to convert type abstractions between Haskell and ML. 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 [1], 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.

Legends of symbol and syntax names are presented in figures 5-7; Haskell is presented in figures 8-12; ML is presented in figures 13-17; Scheme is presented in figures 18-22; the unbrand function is presented in figure 23; and lump equality is presented in figure 24.

Symbol	**Name**
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 5: Symbol names

Syntax	**Name**
$+\ e\ e$	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
$\texttt{fix}\ e$	Fixed-point operation
$\lambda x : t.e$	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 6: Syntax names

Syntax	**Name**
$b \diamond t$	Branded type
$\forall y.t$	Universally quantified type
$\forall y.k$	Universally quantified conversion scheme
$t \to t$	Function abstraction
$k \to k$	Function abstraction
$\{t\}$	List
$\{k\}$	List

Figure 7: Syntax names

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

Figure 8: Haskell syntax and evaluation contexts

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

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

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

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

$$\frac{\Gamma \vdash_H e_H : t_H \to t'_H \quad \Gamma \vdash_H e'_H : t_H}{\Gamma \vdash_H e_H\ e'_H : t'_H} \qquad \frac{\Gamma \vdash_H e_H : t_H \to t_H}{\Gamma \vdash_H \mathtt{fix}\ e_H : t_H}$$

$$\frac{\Gamma \vdash_H t_H \quad \Gamma \vdash_H e_H : \forall y_H.t'_H}{\Gamma \vdash_H e_H\langle t_H\rangle : t'_H[t_H/y_H]} \qquad \frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathtt{hd}\ e_H : t_H} \qquad \frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathtt{tl}\ e_H : \{t_H\}}$$

$$\frac{\Gamma \vdash_H e_H : \mathtt{N} \quad \Gamma \vdash_H e'_H : \mathtt{N}}{\Gamma \vdash_H a\ e_H\ e'_H : \mathtt{N}} \qquad \frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathtt{null?}\ e_H : \mathtt{N}} \qquad \frac{\Gamma \vdash_H \lfloor k_H \rfloor \quad \Gamma \vdash_S e_S : \mathtt{TST}}{\Gamma \vdash_H \mathtt{hs}\ k_H\ e_S : \lfloor k_H \rfloor}$$

$$\frac{\Gamma \vdash_H e_H : \mathtt{N} \quad \Gamma \vdash_H e'_H : t_H \quad \Gamma \vdash_H e''_H : t_H}{\Gamma \vdash_H \mathtt{if0}\ e_H\ e'_H\ e''_H : t_H} \qquad \frac{\Gamma \vdash_H t_H}{\Gamma \vdash_H \mathtt{wrong}\ t_H\ \mathit{string} : t_H}$$

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

Figure 9: Haskell typing rules

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

Figure 10: Haskell operational semantics

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

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

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

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

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

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

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

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

Figure 11: Haskell-ML operational semantics

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

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

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

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

$\mathscr{F}[\texttt{hs}\ \{k_H\}\ f_S]_H \to \mathscr{F}[\texttt{wrong}\ \lfloor \{k_H\} \rfloor\ \text{“Not a list”}]$

$\quad (f_S \neq \texttt{nil}\ \text{and}\ f_S \neq \texttt{cons}\ u_S\ u'_S)$

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

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

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

$\mathscr{F}[\texttt{hs}\ (k_H \to k'_H)\ f_S]_H \to \mathscr{F}[\texttt{wrong}\ \lfloor k_H \to k'_H \rfloor\ \text{“Not a function”}]$

$\quad (f_S \neq \lambda x_S.e_S)$

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

Figure 12: Haskell-Scheme operational semantics

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

Figure 13: ML syntax and evaluation contexts

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

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

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

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

$$
\frac{\Gamma \vdash_M e_M : t_M \to t'_M \quad \Gamma \vdash_M e'_M : t_M}{\Gamma \vdash_H e_M\ e'_M : t'_M} \qquad \frac{\Gamma \vdash_M e_M : t_M \to t_M}{\Gamma \vdash_M \mathtt{fix}\ e_M : t_M}
$$

$$
\frac{\Gamma \vdash_M t_M \quad \Gamma \vdash_M e_M : \forall y_M . t'_M}{\Gamma \vdash_M e_M \langle t_M \rangle : t'_M[t_M / y_M]} \qquad \frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathtt{hd}\ e_M : t_M} \qquad \frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathtt{tl}\ e_M : \{t_M\}}
$$

$$
\frac{\Gamma \vdash_M e_M : \mathtt{N} \quad \Gamma \vdash_M e'_M : \mathtt{N}}{\Gamma \vdash_M a\ e_M\ e'_M : \mathtt{N}} \qquad \frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathtt{null?}\ e_M : \mathtt{N}} \qquad \frac{\Gamma \vdash_M \lfloor k_M \rfloor \quad \Gamma \vdash_S e_S : \mathtt{TST}}{\Gamma \vdash_M \mathtt{ms}\ k_M\ e_S : \lfloor k_M \rfloor}
$$

$$
\frac{\Gamma \vdash_M e_M : \mathtt{N} \quad \Gamma \vdash_M e'_M : t_M \quad \Gamma \vdash_M e''_M : t_M}{\Gamma \vdash_M \mathtt{if0}\ e_M\ e'_M\ e''_M : t_M} \qquad \frac{\Gamma \vdash_M t_M}{\Gamma \vdash_M \mathtt{wrong}\ t_M\ \mathit{string} : t_M}
$$

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

Figure 14: ML typing rules

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

Figure 15: ML operational semantics

$$\mathscr{F}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hm}\ \texttt{L}\ t'_M\ f_M)]_M \to \mathscr{F}[f_M]\ (t_M = t'_M \text{ and } t_M \neq \texttt{L})$$
$$\mathscr{F}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hm}\ \texttt{L}\ t'_M\ f_M)]_M \to \mathscr{F}[\texttt{wrong}\ t_M\ \text{“Type mismatch”}]\ (t_M \neq t'_M \text{ and } t_M \neq \texttt{L})$$
$$\mathscr{F}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hs}\ \texttt{L}\ f_S)]_H \to \mathscr{F}[\texttt{wrong}\ t_M\ \text{“Bad value”}]\ (t_M \neq \texttt{L})$$
$$\mathscr{F}[\texttt{mh}\ \texttt{N}\ \texttt{N}\ \overline{n}]_M \to \mathscr{F}[\overline{n}]$$
$$\mathscr{F}[\texttt{mh}\ \{t_M\}\ \{t_H\}\ (\texttt{nil}\ t'_H)]_M \to \mathscr{F}[\texttt{nil}\ t_M]$$
$$\mathscr{F}[\texttt{mh}\ \{t_M\}\ \{t_H\}\ (\texttt{cons}\ e_H\ e'_H)]_M \to \mathscr{F}[\texttt{cons}\ (\texttt{mh}\ t_M\ t_H\ e_H)\ (\texttt{mh}\ \{t_M\}\ \{t_H\}\ e'_H)]$$
$$\mathscr{F}[\texttt{mh}\ (t_M \to t'_M)\ (t_H \to t'_H)\ (\lambda x_H : t''_H.e_H)]_M \to$$
$$\qquad \mathscr{F}[\lambda x_M : t_M.\texttt{mh}\ t'_M\ t'_H\ ((\lambda x_H : t''_H.e_H)\ (\texttt{hm}\ t_H\ t_M\ x_M))]$$
$$\mathscr{F}[\texttt{mh}\ (\forall y_M.t_M)\ (\forall y_H.t_H)\ (\Lambda y'_H.e_H)]_M \to \mathscr{F}[\Lambda y_M.\texttt{mh}\ t_M\ t_H[\texttt{L}/y_H]\ e_H[\texttt{L}/y'_H]]$$

Figure 16: ML-Haskell operational semantics

$$
\begin{array}{l}
\mathscr{F}[\texttt{ms N } \overline{n}]_M \rightarrow \mathscr{F}[\overline{n}] \\
\mathscr{F}[\texttt{ms N } f_S]_M \rightarrow \mathscr{F}[\texttt{wrong N } \text{“Not a number”}]\ (f_S \neq \overline{n}) \\
\mathscr{F}[\texttt{ms } \{k_M\} \texttt{ nil}]_M \rightarrow \mathscr{F}[\texttt{nil } \lfloor k_M \rfloor] \\
\mathscr{F}[\texttt{ms } \{k_M\} \ (\texttt{cons } u_S\ u'_S)]_M \rightarrow \mathscr{F}[\texttt{cons } (\texttt{ms } k_M\ u_S)\ (\texttt{ms } \{k_M\}\ u'_S)] \\
\mathscr{F}[\texttt{ms } \{k_M\}\ f_S]_M \rightarrow \mathscr{F}[\texttt{wrong } \lfloor \{k_M\} \rfloor \text{ “Not a list”}] \\
\qquad (f_S \neq \texttt{nil} \text{ and } f_S \neq \texttt{cons } u_S\ u'_S) \\
\mathscr{F}[\texttt{ms } (b \diamond t_M)\ (\texttt{sm } (b \diamond t_M)\ u_M)]_M \rightarrow \mathscr{F}[u_M] \\
\mathscr{F}[\texttt{ms } (b \diamond t_M)\ f_S]_M \rightarrow \mathscr{F}[\texttt{wrong } \lfloor b \diamond t_M \rfloor \text{ “Brand mismatch”}] \\
\qquad (f_S \neq \texttt{sm } (b \diamond t_M)\ e_M) \\
\mathscr{F}[\texttt{ms } (k_M \rightarrow k'_M)\ (\lambda x_S.e_S)]_M \rightarrow \\
\qquad \mathscr{F}[\lambda x_M : \lfloor k_M \rfloor.\texttt{ms } k'_M\ ((\lambda x_S.e_S)\ (\texttt{sm } k_M\ x_M))] \\
\mathscr{F}[\texttt{ms } (k_M \rightarrow k'_M)\ f_S]_M \rightarrow \mathscr{F}[\texttt{wrong } \lfloor k_M \rightarrow k'_M \rfloor \text{ “Not a function”}] \\
\qquad (f_S \neq \lambda x_S.e_S) \\
\mathscr{F}[\texttt{ms } (\forall y_M.k_M)\ f_S]_M \rightarrow \mathscr{F}[\Lambda y_M.\texttt{ms } k_M\ f_S]
\end{array}
$$

Figure 17: ML-Scheme operational semantics

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

Figure 18: Scheme syntax and evaluation contexts

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

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

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

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

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

$$\frac{\Gamma \vdash_S e_S : \texttt{TST} \quad \Gamma \vdash_S e'_S : \texttt{TST} \quad \Gamma \vdash_S e''_S : \texttt{TST}}{\Gamma \vdash_S \texttt{if0}\ e_S\ e'_S\ e''_S : \texttt{TST}} \qquad \frac{}{\vdash_S \texttt{wrong}\ \mathit{string} : \texttt{TST}}$$

$$\frac{\Gamma \vdash_H \lfloor k_H \rfloor \quad \Gamma \vdash_H e_H : t_H \quad \lfloor k_H \rfloor = t_H}{\Gamma \vdash_S \texttt{sh}\ k_H\ e_H : \texttt{TST}} \qquad \frac{\Gamma \vdash_M \lfloor k_M \rfloor \quad \Gamma \vdash_M e_M : t_M \quad \lfloor k_M \rfloor = t_M}{\Gamma \vdash_S \texttt{sm}\ k_M\ e_M : \texttt{TST}}$$

Figure 19: Scheme typing rules

$$
\begin{aligned}
&\mathscr{F}[(\lambda x_S.e_S)\ u_S]_S \to \mathscr{F}[e_S[u_S/x_S]]\\
&\mathscr{F}[f_S\ u_S]_S \to \mathscr{F}[\texttt{wrong}\ \text{“Not a function”}]\ (f_S \neq \lambda x_S.e_S)\\
&\mathscr{F}[+\ \overline{n}\ \overline{n}']_S \to \mathscr{F}[\overline{n+n'}]\\
&\mathscr{F}[-\ \overline{n}\ \overline{n}']_S \to \mathscr{F}[\overline{max(n-n',0)}]\\
&\mathscr{F}[a\ f_S\ f'_S]_S \to \mathscr{F}[\texttt{wrong}\ \text{“Not a number”}]\ (f_S \neq \overline{n} \text{ or } f'_S \neq \overline{n})\\
&\mathscr{F}[\texttt{if0}\ \overline{0}\ e_S\ e'_S]_S \to \mathscr{F}[e_S]\\
&\mathscr{F}[\texttt{if0}\ \overline{n}\ e_S\ e'_S]_S \to \mathscr{F}[e'_S]\ (n \neq 0)\\
&\mathscr{F}[\texttt{if0}\ f_S\ e_S\ e'_S]_S \to \mathscr{F}[\texttt{wrong}\ \text{“Not a number”}]\ (f_S \neq \overline{n})\\
&\mathscr{F}[c\ \texttt{nil}]_S \to \mathscr{F}[\texttt{wrong}\ \text{“Empty list”}]\\
&\mathscr{F}[\texttt{hd}\ (\texttt{cons}\ u_S\ u'_S)]_S \to \mathscr{F}[u_S]\\
&\mathscr{F}[\texttt{tl}\ (\texttt{cons}\ u_S\ u'_S)]_S \to \mathscr{F}[u'_S]\\
&\mathscr{F}[c\ f_S]_S \to \mathscr{F}[\texttt{wrong}\ \text{“Not a list”}]\ (f_S \neq \texttt{nil} \text{ and } f_S \neq \texttt{cons}\ u_S\ u'_S)\\
&\mathscr{F}[\texttt{fun}?\ (\lambda x_S.e_S)]_S \to \mathscr{F}[\overline{0}]\\
&\mathscr{F}[\texttt{fun}?\ f_S]_S \to \mathscr{F}[\overline{1}]\ (f_S \neq \lambda x_S.e_S)\\
&\mathscr{F}[\texttt{list}?\ \texttt{nil}]_S \to \mathscr{F}[\overline{0}]\\
&\mathscr{F}[\texttt{list}?\ (\texttt{cons}\ u_S\ u'_S)]_S \to \mathscr{F}[\overline{0}]\\
&\mathscr{F}[\texttt{list}?\ f_S]_S \to \mathscr{F}[\overline{1}]\ (f_S \neq \texttt{nil} \text{ and } f_S \neq \texttt{cons}\ u_S\ u'_S)\\
&\mathscr{F}[\texttt{null}?\ \texttt{nil}]_S \to \mathscr{F}[\overline{0}]\\
&\mathscr{F}[\texttt{null}?\ f_S]_S \to \mathscr{F}[\overline{1}]\ (f_S \neq \texttt{nil})\\
&\mathscr{F}[\texttt{num}?\ \overline{n}]_S \to \mathscr{F}[\overline{0}]\\
&\mathscr{F}[\texttt{num}?\ f_S]_S \to \mathscr{F}[\overline{1}]\ (f_S \neq \overline{n})\\
&\mathscr{F}[\texttt{wrong}\ \textit{string}]_S \to \textbf{Error: } \textit{string}
\end{aligned}
$$

Figure 20: Scheme operational semantics

$$\mathscr{F}[\texttt{sh}\ \texttt{L}\ (\texttt{hm}\ \texttt{L}\ k_M\ f_M)]_S \to \mathscr{F}[\texttt{wrong}\ \text{“Bad value”}]$$
$$\mathscr{F}[\texttt{sh}\ \texttt{L}\ (\texttt{hs}\ \texttt{L}\ f_S)]_S \to \mathscr{F}[f_S]$$
$$\mathscr{F}[\texttt{sh}\ \texttt{N}\ \overline{n}]_S \to \mathscr{F}[\overline{n}]$$
$$\mathscr{F}[\texttt{sh}\ \{k_H\}\ (\texttt{nil}\ t_H)]_S \to \mathscr{F}[\texttt{nil}]$$
$$\mathscr{F}[\texttt{sh}\ \{k_H\}\ (\texttt{cons}\ e_H\ e'_H)]_S \to \mathscr{F}[\texttt{cons}\ (\texttt{sh}\ k_H\ e_H)\ (\texttt{sh}\ \{k_H\}\ e'_H)]$$
$$\mathscr{F}[\texttt{sh}\ (k_H \to k'_H)\ (\lambda x_H : t_H.e_H)]_S \to$$
$$\qquad \mathscr{F}[\lambda x_S.\texttt{sh}\ k'_H\ ((\lambda x_H : t_H.e_H)\ (\texttt{hs}\ k_H\ x_S))]$$
$$\mathscr{F}[\texttt{sh}\ (\forall y_H.k_H)\ (\Lambda y'_H.e_H)]_S \to \mathscr{F}[\texttt{sh}\ k_H[\texttt{L}/y_H]\ e_H[\texttt{L}/y'_H]]$$

Figure 21: Scheme-Haskell operational semantics

$$\mathscr{F}[\texttt{sm}\ \texttt{L}\ (\texttt{mh}\ \texttt{L}\ k_H\ e_H)]_S \rightarrow \mathscr{F}[\texttt{wrong}\ \text{“Bad value”}]$$
$$\mathscr{F}[\texttt{sm}\ \texttt{L}\ (\texttt{ms}\ \texttt{L}\ f_S)]_S \rightarrow \mathscr{F}[f_S]$$
$$\mathscr{F}[\texttt{sm}\ \texttt{N}\ \overline{n}]_S \rightarrow \mathscr{F}[\overline{n}]$$
$$\mathscr{F}[\texttt{sm}\ \{k_M\}\ (\texttt{nil}\ t_M)]_S \rightarrow \mathscr{F}[\texttt{nil}]$$
$$\mathscr{F}[\texttt{sm}\ \{k_M\}\ (\texttt{cons}\ u_M\ u'_M)]_S \rightarrow \mathscr{F}[\texttt{cons}\ (\texttt{sm}\ k_M\ u_M)\ (\texttt{sm}\ \{k_M\}\ u'_M)]$$
$$\mathscr{F}[\texttt{sm}\ (k_M \rightarrow k'_M)\ (\lambda x_M : t_M.e_M)]_S \rightarrow$$
$$\quad \mathscr{F}[\lambda x_S.\texttt{sm}\ k'_M\ ((\lambda x_M : t_M.e_M)\ (\texttt{ms}\ k_M\ x_S))]$$
$$\mathscr{F}[\texttt{sm}\ (\forall y_M.k_M)\ (\Lambda y'_M.e_M)]_S \rightarrow \mathscr{F}[\texttt{sm}\ k_M[\texttt{L}/y_M]\ e_M[\texttt{L}/y'_M]]$$

Figure 22: Scheme-ML operational semantics

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

Figure 23: Unbrand function

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

Figure 24: Lump equality

3 Conclusion

Lazy and eager evaluation can be resolved transparently for common expressions at the boundaries between languages with unforced and forced values. This is more convenient than an explicit force operator that programmers must use manually by anticipating which expressions must be forced.

References

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