Interoperation for Lazy and Eager Evaluation

William Faught

February 7, 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 incompatible 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.

Languages are normally independent, as in figure 1. Matthews and Findler [2] approached interoperation by tying languages together with boundary expressions, as in figure 2, where `ab` b and `ba` a are the boundary expressions. Expressions of one language are embedded in boundaries of another language. Embedding an expression represents the exchange of the expression from the inner language to the outer language.

Matthews and Findler used evaluation contexts to enable error states in their languages whereby on an error the entire program at that moment is discarded and

$$
\begin{aligned}
a &= x_a \mid \lambda x_a.a \mid a\ a \\
b &= x_b \mid \lambda x_b.b \mid b\ b
\end{aligned}
$$

Figure 1: Two simple languages.

$$
\begin{aligned}
a &= x_a \mid \lambda x_a.a \mid a\ a \mid \texttt{ab}\ b \\
b &= x_b \mid \lambda x_b.b \mid b\ b \mid \texttt{ba}\ a
\end{aligned}
$$

Figure 2: Two simple languages tied together with boundaries.

replaced with an error. With evaluation contexts, every reduction rule is a whole-program transformation with a focus on the reducible expression and a reference to the enclosing evaluation context (the expression outside the reducible expression). Reduction rules can either replace the focused reducible expression within the context of the containing program, or discard the evaluation context and reduce to something else like an error.

$$
\begin{aligned}
e &= x \mid v \mid e\ e \\
v &= \lambda x.e \\
(\lambda x.e)\ v &\rightarrow e[v/x] \\
e \rightarrow e' &\Rightarrow e\ e'' \rightarrow e'\ e'' \\
e \rightarrow e' &\Rightarrow v\ e \rightarrow v\ e'
\end{aligned}
$$

Figure 3: Reduction relation defined by inference rules.

The model of computation extends the work of Matthews and Findler [2] and uses the work of Kinghorn [1] as a starting point. Matthews and Findler presented [2] two approaches to resolving static and dynamic type systems for interoperation. In the first approach, called lump embedding, languages are tied together by new expressions called boundaries that embed expressions of one language within another, which represent the exchange of the embedded expressions from the inner languages to the outer languages. The embedded expressions, when exchanged, are opaque values that cannot be inspected by the outer language, and can only be returned to the inner language. In the second approach, called natural embedding, languages are tied together as with lump embedding, but instead of opaqueness, embedded expressions are reduced to values, then converted into equal values of the outer language following type annotations in the boundaries. To return a converted value back to the inner language, it must be converted back.

$$
\begin{aligned}
e &= x \mid v \mid e\ e \\
v &= \lambda x.e \\
E &= [\,] \mid E\ e \mid v\ E \\
E[(\lambda x.e)\ v] &\rightarrow E[e[v/x]]
\end{aligned}
$$

Figure 4: Reduction relation defined by evaluation contexts.

$$
\begin{aligned}
a &= x_a \mid \lambda x_a.a \mid a\ a \mid \texttt{ab}\ b \\
E_a &= [\,]_a \mid E_a\ a \mid \texttt{ab}\ E_b \\
E[(\lambda x_a.a)\ a']_a &\rightarrow E[a[a'/x_a]] \\
b &= x_b \mid v \mid b\ b \mid \texttt{ba}\ a \\
v &= \lambda x_b.b \\
E_b &= [\,]_b \mid E_b\ b \mid v\ E_b \mid \texttt{ba}\ E_a \\
E[(\lambda x_b.b)\ v]_b &\rightarrow E[b[v/x_b]]
\end{aligned}
$$

Figure 5: Naive evaluation contexts for interoperation.

The models of Matthews and Findler are defined using evaluation contexts

The systems of interoperation presented by Matthews and Findler [2] preserved the equivalence of values converted between languages that have incompatible type systems. Since the languages they used were all eager, there were no evaluation strategy incompatibilities to resolve. If a lazy language is introduced to their systems, then interoperation does not preserve the equivalence of values converted between the lazy language and the eager languages. For example, since the application of a converted function involves applications in both the outer and inner languages, the argument is subject to both the outer and inner evaluation strategies. If the outer language is lazy and the inner language is eager, then the argument may be evaluated by the inner language but not the outer language. In this case, the converted function is not equivalent to the original function. Futhermore, the conversion of composite types like lists from lazy languages to eager ones may diverge or cause an error because eager evaluation will convert the entire value, which may be of infinite size or contain expressions assumed by lazy languages not to be immediately evaluated.

Lazy and eager evaluation take opposite approaches: lazy evaluation evaluates expressions as needed, and eager evaluation evaluates expressions immediately. As such, for common expressions, lazy evaluation evaluates a proper subset of the expressions that eager evaluation does. In other words, the set of lazy evaluation strictness

points is a proper subset of that of eager evaluation. The difference between these two sets is the set of incompatible strictness points that may change the meaning of values converted from eager languages to lazy ones or may cause a divergence or an error for values converted from lazy languages to eager ones. Where boundaries that contain expressions of lazy languages are at these points, the original lazy evaluation strategy must be followed, and the guards not evaluated. This requires introducing a dual notion of values where *forced* values force the evaluation of guarded expressions of lazy languages and *unforced* values prevent their evaluation.

2 Model of Computation

The model of computation extends the model presented by Kinghorn [1] with a third language identical to the ML model except it uses lazy evaluation, and as such is named after Haskell, to which it is more similar. Hereafter, the names Haskell, ML, and Scheme refer to their corresponding models in this paper. Lists are added to all three languages. Being lazy, Haskell does not evaluate function arguments or list construction operands. These three points constitute the set of incompatible strictness points between Haskell and ML and Haskell and Scheme. At these points in ML and Scheme, reducible expressions in Haskell boundaries must not be evaluated.

Since values are irreducible at all points, and since the expressions in Haskell boundaries are irreducible at some points and not others, Haskell boundaries are a new kind of value called an *unforced value*. Like thunks, unforced values can be forced to evaluate to values. The Haskell expressions in Haskell boundaries are forced to evaluate to Haskell values, then the Haskell values are converted to ML or Scheme values. ML and Scheme values are called *forced values* because any might be the result of forcing an unforced value. Forced values are a proper subset of unforced values because unforced values can only be at points where forced values can also be, but forced values can be at points where unforced values cannot. ML and Scheme reduction rules and evaluation contexts use unforced values at the incompatible strictness points to match against Haskell boundaries, and their evaluation contexts prevent evaluation within Haskell boundaries at those points.

Figure 6 illustrates forced and unforced values at work for the cases explained in the introduction. The reductions for lines 1-4 show that the outer Haskell argument *zeroes* is not forced by the application of the inner Scheme function. The reductions for lines 4-8 show that the conversion of *zeroes* from Haskell to Scheme did not diverge, despite *zeroes* being a list of infinite size.

Theorem 1. *Evaluation Strategy Preservation*

$$
\begin{array}{ll}
\multicolumn{2}{c}{\mathit{zeroes} = \texttt{fix}\ (\lambda x_H : \{\texttt{N}\}.\texttt{cons}\ \overline{0}\ x_H)} \\
(\texttt{hs}\ (\{\texttt{N}\} \to \{\texttt{N}\})\ (\lambda x_S.x_S))\ \mathit{zeroes} & \to \\
(\lambda x'_H : \{\texttt{N}\}.\texttt{hs}\ \{\texttt{N}\}\ ((\lambda x_S.x_S)\ (\texttt{sh}\ \{\texttt{N}\}\ x'_H)))\ \mathit{zeroes} & \to \\
\texttt{hs}\ \{\texttt{N}\}\ ((\lambda x_S.x_S)\ (\texttt{sh}\ \{\texttt{N}\}\ \mathit{zeroes})) & \to \\
\texttt{hs}\ \{\texttt{N}\}\ (\texttt{sh}\ \{\texttt{N}\}\ \mathit{zeroes}) & \to \\
\texttt{hs}\ \{\texttt{N}\}\ (\texttt{sh}\ \{\texttt{N}\}\ (\texttt{cons}\ \overline{0}\ \mathit{zeroes})) & \to \\
\texttt{hs}\ \{\texttt{N}\}\ (\texttt{cons}\ (\texttt{sh}\ \texttt{N}\ \overline{0})\ (\texttt{sh}\ \{\texttt{N}\}\ \mathit{zeroes})) & \to \\
\texttt{hs}\ \{\texttt{N}\}\ (\texttt{cons}\ \overline{0}\ (\texttt{sh}\ \{\texttt{N}\}\ \mathit{zeroes})) & \to \\
\texttt{cons}\ (\texttt{hs}\ \texttt{N}\ \overline{0})\ (\texttt{hs}\ \{\texttt{N}\}\ (\texttt{sh}\ \{\texttt{N}\}\ \mathit{zeroes})) &
\end{array}
$$

Figure 6: Haskell argument and list conversions.

$e_H = \texttt{mh}\ t_M\ t_H\ e_H = \texttt{sh}\ t_H\ e_H.\ e_M = \texttt{hm}\ t_H\ t_M\ e_M = \texttt{sm}\ t_M\ e_M.\ e_S = \texttt{hs}\ t_H\ e_S = \texttt{ms}\ t_M\ e_S.$

Proof. By structural induction. □

The interoperation of Haskell and ML posed another problem: the conversion of type abstractions. The application of a converted type abstraction cannot substitute the type argument into the inner language directly, since the inner language has no notion of the types of the outer language. Instead, conversion substitutes lumps in a boundary's inner type. The application of a converted type abstraction substitutes the type argument in the boundary's outer type. Since the natural embedding [2] requires the boundary's outer and inner types to be equal, a new equality relation called lump equality is used here 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 7-9; Haskell is presented in figures 10-14; ML is presented in figures 15-19; Scheme is presented in figures 20-24; the unbrand function is presented in figure 25; and the lump equality relation is presented in figure 26.

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

Figure 7: Symbol names

Syntax	Name
$+\ e\ e$	Addition
`if0` $e\ e\ e$	Condition
`nil` t	Empty list
`nil`	Empty list
`wrong` t *string*	Error
`wrong` *string*	Error
`fix` e	Fixed-point operation
$\lambda x : t.e$	Function abstraction
$\lambda x_S.e_S$	Function abstraction
$e\ e$	Function application
`hm` $t_H\ t_M\ e_M$	Haskell-ML guard
`hs` $k_H\ e_S$	Haskell-Scheme guard
`cons` $e\ e$	List construction
`hd` e	List head
`tl` e	List tail
`mh` $t_M\ t_H\ e_H$	ML-Haskell guard
`ms` $k_M\ e_S$	ML-Scheme guard
`sh` $k_H\ e_H$	Scheme-Haskell guard
`sm` $k_M\ e_M$	Scheme-ML guard
$-\ e\ e$	Subtraction
$\Lambda u.e$	Type abstraction
$e\langle t\rangle$	Type application
`fun`? e_S	Value predicate
`list`? e_S	Value predicate
`null`? e	Value predicate
`num`? e_S	Value predicate

Figure 8: Syntax names

Syntax	Name
$b \diamond t$	Branded type
$\forall u.t$	Forall
$\forall u.k$	Forall
$t \to t$	Function abstraction
$k \to k$	Function abstraction
$\{t\}$	List
$\{k\}$	List

Figure 9: Syntax names

$$
\begin{array}{rcl}
e_H & = & x_H \mid v_H \mid e_H\ e_H \mid e_H\langle t_H\rangle \mid \texttt{fix}\ e_H \mid o\ e_H\ e_H \mid \texttt{if0}\ e_H\ e_H\ e_H \mid f\ e_H \\
 & & \texttt{null?}\ e_H \mid \texttt{wrong}\ t_H\ \mathit{string} \mid \texttt{hm}\ t_H\ t_M\ e_M \mid \texttt{hs}\ k_H\ e_S \\
v_H & = & \lambda x_H : t_H.e_H \mid \Lambda u_H.e_H \mid \overline{n} \mid \texttt{nil}\ t_H \mid \texttt{cons}\ e_H\ e_H \mid \texttt{hm}\ \texttt{L}\ t_M\ w_M \\
 & & \texttt{hs}\ \texttt{L}\ w_S \\
t_H & = & \texttt{L} \mid \texttt{N} \mid u_H \mid \{t_H\} \mid t_H \to t_H \mid \forall u_H.t_H \\
k_H & = & \texttt{L} \mid \texttt{N} \mid u_H \mid \{k_H\} \mid k_H \to k_H \mid \forall u_H.k_H \mid b \diamond t_H \\
o & = & + \mid - \\
f & = & \texttt{hd} \mid \texttt{tl} \\
E_H & = & [\,]_H \mid E_H\ e_H \mid E_H\langle t_H\rangle \mid \texttt{fix}\ E_H \mid o\ E_H\ e_H \mid o\ v_H\ E_H \\
 & & \texttt{if0}\ E_H\ e_H\ e_H \mid f\ E_H \mid \texttt{null?}\ E_H \mid \texttt{hm}\ t_H\ t_M\ E_M \mid \texttt{hs}\ k_H\ E_S
\end{array}
$$

Figure 10: Haskell syntax and evaluation contexts

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

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

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

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

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

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

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

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

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

Figure 11: Haskell typing rules

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

Figure 12: Haskell operational semantics

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

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

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

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

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

$\mathscr{E}[\texttt{hm}\ \{t_H\}\ \{t_M\}\ (\texttt{cons}\ v_M\ v'_M)]_H \to$
$\quad \mathscr{E}[\texttt{cons}\ (\texttt{hm}\ t_H\ t_M\ v_M)\ (\texttt{hm}\ \{t_H\}\ \{t_M\}\ v'_M)]$

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

$\mathscr{E}[\texttt{hm}\ (\forall u_H.t_H)\ (\forall u_M.t_M)\ (\Lambda u'_M.e_M)]_H \to \mathscr{E}[\Lambda u_H.\texttt{hm}\ t_H\ t_M[\texttt{L}/u_M]\ e_M[\texttt{L}/u'_M]]$

Figure 13: Haskell-ML operational semantics

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

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

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

$\mathscr{E}[\texttt{hs } \{k_H\} \texttt{ (cons } v_S \ v'_S)]_H \to \mathscr{E}[\texttt{cons (hs } k_H \ v_S) \texttt{ (hs } \{k_H\} \ v'_S)]$

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

$(w_S \neq \texttt{nil}$ and $w_S \neq \texttt{cons } v_S \ v'_S)$

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

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

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

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

$(w_S \neq \lambda x_S.e_S)$

$\mathscr{E}[\texttt{hs } (\forall u_H.k_H) \ w_S]_H \to \mathscr{E}[\Lambda u_H.\texttt{hs } k_H \ w_S]$

Figure 14: Haskell-Scheme operational semantics

$$
\begin{array}{rcl}
e_M & = & x_M \mid v_M \mid e_M\ e_M \mid e_M\langle t_M\rangle \mid \texttt{fix}\ e_M \mid o\ e_M\ e_M \mid \texttt{if0}\ e_M\ e_M\ e_M \\
 & & \texttt{cons}\ e_M\ e_M \mid f\ e_M \mid \texttt{null?}\ e_M \mid \texttt{wrong}\ t_M\ \mathit{string} \mid \texttt{ms}\ k_M\ e_S \\
v_M & = & w_M \mid \texttt{mh}\ t_M\ t_H\ e_H \\
w_M & = & \lambda x_M : t_M.e_M \mid \Lambda u_M.e_M \mid \overline{n} \mid \texttt{nil}\ t_M \mid \texttt{cons}\ v_M\ v_M \mid \texttt{mh}\ \texttt{L}\ t_H\ e_H \\
 & & \texttt{ms}\ \texttt{L}\ w_S \\
t_M & = & \texttt{L} \mid \texttt{N} \mid u_M \mid \{t_M\} \mid t_M \to t_M \mid \forall u_M.t_M \\
k_M & = & \texttt{L} \mid \texttt{N} \mid u_M \mid \{k_M\} \mid k_M \to k_M \mid \forall u_M.k_M \mid b \diamond t_M \\
o & = & + \mid - \\
f & = & \texttt{hd} \mid \texttt{tl} \\
E_M & = & U_M \mid \texttt{mh}\ t_M\ t_H\ E_H \\
U_M & = & [\,]_M \mid E_M\ e_M \mid w_M\ U_M \mid E_M\langle t_M\rangle \mid \texttt{fix}\ E_M \mid o\ E_M\ e_M \mid o\ w_M\ E_M \\
 & & \texttt{if0}\ E_M\ e_M\ e_M \mid \texttt{cons}\ U_M\ e_M \mid \texttt{cons}\ v_M\ U_M \mid f\ E_M \mid \texttt{null?}\ E_M \\
 & & \texttt{ms}\ k_M\ E_S
\end{array}
$$

Figure 15: ML syntax and evaluation contexts

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

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

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

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

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

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

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

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

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

Figure 16: ML typing rules

$\mathscr{E}[(\lambda x_M : t_M.e_M)\ v_M]_M \to \mathscr{E}[e_M[v_M/x_M]]$
$\mathscr{E}[(\Lambda u_M.e_M)\langle t_M\rangle]_M \to \mathscr{E}[e_M[b \diamond t_M/u_M]]$
$\mathscr{E}[\texttt{fix}\ (\lambda x_M : t_M.e_M)]_M \to \mathscr{E}[e_M[\texttt{fix}\ (\lambda x_M : t_M.e_M)/x_M]]$
$\mathscr{E}[+\ \overline{n}\ \overline{n}']_M \to \mathscr{E}[\overline{n+n'}]$
$\mathscr{E}[-\ \overline{n}\ \overline{n}']_M \to \mathscr{E}[\overline{max(n-n',0)}]$
$\mathscr{E}[\texttt{if0}\ \overline{0}\ e_M\ e'_M]_M \to \mathscr{E}[e_M]$
$\mathscr{E}[\texttt{if0}\ \overline{n}\ e_M\ e'_M]_M \to \mathscr{E}[e'_M]\ (n \neq 0)$
$\mathscr{E}[\texttt{hd}\ (\texttt{nil}\ t_M)]_M \to \mathscr{E}[\texttt{wrong}\ t_M$ "Empty list"$]$
$\mathscr{E}[\texttt{tl}\ (\texttt{nil}\ t_M)]_M \to \mathscr{E}[\texttt{wrong}\ \{t_M\}$ "Empty list"$]$
$\mathscr{E}[\texttt{hd}\ (\texttt{cons}\ v_M\ v'_M)]_M \to \mathscr{E}[v_M]$
$\mathscr{E}[\texttt{tl}\ (\texttt{cons}\ v_M\ v'_M)]_M \to \mathscr{E}[v'_M]$
$\mathscr{E}[\texttt{null?}\ (\texttt{nil}\ t_M)]_M \to \mathscr{E}[\overline{0}]$
$\mathscr{E}[\texttt{null?}\ (\texttt{cons}\ v_M\ v'_M)]_M \to \mathscr{E}[\overline{1}]$
$\mathscr{E}[\texttt{wrong}\ t_M\ \mathit{string}]_H \to$ **Error:** *string*

Figure 17: ML operational semantics

$\mathscr{E}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hm}\ \texttt{L}\ t'_M\ w_M)]_M \to \mathscr{E}[w_M]\ (t_M = t'_M \text{ and } t_M \neq \texttt{L})$

$\mathscr{E}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hm}\ \texttt{L}\ t'_M\ w_M)]_M \to \mathscr{E}[\texttt{wrong}\ t_M\ \text{“Type mismatch”}]\ (t_M \neq t'_M \text{ and } t_M \neq \texttt{L})$

$\mathscr{E}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hs}\ \texttt{L}\ w_S)]_H \to \mathscr{E}[\texttt{wrong}\ t_M\ \text{“Bad value”}]\ (t_M \neq \texttt{L})$

$\mathscr{E}[\texttt{mh}\ \texttt{N}\ \texttt{N}\ \overline{n}]_M \to \mathscr{E}[\overline{n}]$

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

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

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

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

$\mathscr{E}[\texttt{mh}\ (\forall u_M.t_M)\ (\forall u_H.t_H)\ (\Lambda u'_H.e_H)]_M \to \mathscr{E}[\Lambda u_M.\texttt{mh}\ t_M\ t_H[\texttt{L}/u_H]\ e_H[\texttt{L}/u'_H]]$

Figure 18: ML-Haskell operational semantics

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

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

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

$\mathscr{E}[\texttt{ms } \{k_M\} (\texttt{cons } v_S\ v'_S)]_M \to \mathscr{E}[\texttt{cons } (\texttt{ms } k_M\ v_S)\ (\texttt{ms } \{k_M\}\ v'_S)]$

$\mathscr{E}[\texttt{ms } \{k_M\}\ w_S]_M \to \mathscr{E}[\texttt{wrong } \lfloor \{k_M\} \rfloor \text{ “Not a list”}]$
$(w_S \neq \texttt{nil} \text{ and } w_S \neq \texttt{cons } v_S\ v'_S)$

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

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

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

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

$\mathscr{E}[\texttt{ms } (\forall u_M.k_M)\ w_S]_M \to \mathscr{E}[\Lambda u_M.\texttt{ms } k_M\ w_S]$

Figure 19: ML-Scheme operational semantics

$$
\begin{array}{lcl}
e_S & = & x_S \mid v_S \mid e_S\ e_S \mid o\ e_S\ e_S \mid p\ e_S \mid \texttt{if0}\ e_S\ e_S\ e_S \mid \texttt{cons}\ e_S\ e_S \mid f\ e_S \\
 & & \texttt{wrong}\ \mathit{string} \mid \texttt{sm}\ k_M\ e_M \\
v_S & = & w_S \mid \texttt{sh}\ k_H\ e_H \\
w_S & = & \lambda x_S.e_S \mid \overline{n} \mid \texttt{nil} \mid \texttt{cons}\ v_S\ v_S \mid \texttt{sh}\ (b \diamond t_H)\ e_H \mid \texttt{sm}\ (b \diamond t_M)\ w_M \\
o & = & + \mid - \\
f & = & \texttt{hd} \mid \texttt{tl} \\
p & = & \texttt{fun?} \mid \texttt{list?} \mid \texttt{null?} \mid \texttt{num?} \\
E_S & = & U_S \mid \texttt{sh}\ k_H\ E_H \\
U_S & = & [\,]_S \mid E_S\ e_S \mid w_S\ U_S \mid o\ E_S\ e_S \mid o\ w_S\ E_S \mid p\ E_S \mid \texttt{if0}\ E_S\ e_S\ e_S \\
 & & \texttt{cons}\ U_S\ e_S \mid \texttt{cons}\ v_S\ U_S \mid f\ E_S \mid \texttt{sm}\ k_M\ E_M
\end{array}
$$

Figure 20: Scheme syntax and evaluation contexts

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

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

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

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

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

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

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

Figure 21: Scheme typing rules

$\mathscr{E}[(\lambda x_S.e_S)\ v_S]_S \to \mathscr{E}[e_S[v_S/x_S]]$

$\mathscr{E}[w_S\ v_S]_S \to \mathscr{E}[\texttt{wrong}\ \text{“Not a function”}]\ (w_S \neq \lambda x_S.e_S)$

$\mathscr{E}[+\ \overline{n}\ \overline{n}']_S \to \mathscr{E}[\overline{n+n'}]$

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

$\mathscr{E}[o\ w_S\ w'_S]_S \to \mathscr{E}[\texttt{wrong}\ \text{“Not a number”}]\ (w_S \neq \overline{n} \text{ or } w'_S \neq \overline{n})$

$\mathscr{E}[\texttt{if0}\ \overline{0}\ e_S\ e'_S]_S \to \mathscr{E}[e_S]$

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

$\mathscr{E}[\texttt{if0}\ w_S\ e_S\ e'_S]_S \to \mathscr{E}[\texttt{wrong}\ \text{“Not a number”}]\ (w_S \neq \overline{n})$

$\mathscr{E}[f\ \texttt{nil}]_S \to \mathscr{E}[\texttt{wrong}\ \text{“Empty list”}]$

$\mathscr{E}[\texttt{hd}\ (\texttt{cons}\ v_S\ v'_S)]_S \to \mathscr{E}[v_S]$

$\mathscr{E}[\texttt{tl}\ (\texttt{cons}\ v_S\ v'_S)]_S \to \mathscr{E}[v'_S]$

$\mathscr{E}[f\ w_S]_S \to \mathscr{E}[\texttt{wrong}\ \text{“Not a list”}]\ (w_S \neq \texttt{nil} \text{ and } w_S \neq \texttt{cons}\ v_S\ v'_S)$

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

$\mathscr{E}[\texttt{fun?}\ w_S]_S \to \mathscr{E}[\overline{1}]\ (w_S \neq \lambda x_S.e_S)$

$\mathscr{E}[\texttt{list?}\ \texttt{nil}]_S \to \mathscr{E}[\overline{0}]$

$\mathscr{E}[\texttt{list?}\ (\texttt{cons}\ v_S\ v'_S)]_S \to \mathscr{E}[\overline{0}]$

$\mathscr{E}[\texttt{list?}\ w_S]_S \to \mathscr{E}[\overline{1}]\ (w_S \neq \texttt{nil} \text{ and } w_S \neq \texttt{cons}\ v_S\ v'_S)$

$\mathscr{E}[\texttt{null?}\ \texttt{nil}]_S \to \mathscr{E}[\overline{0}]$

$\mathscr{E}[\texttt{null?}\ w_S]_S \to \mathscr{E}[\overline{1}]\ (w_S \neq \texttt{nil})$

$\mathscr{E}[\texttt{num?}\ \overline{n}]_S \to \mathscr{E}[\overline{0}]$

$\mathscr{E}[\texttt{num?}\ w_S]_S \to \mathscr{E}[\overline{1}]\ (w_S \neq \overline{n})$

$\mathscr{E}[\texttt{wrong}\ \textit{string}]_S \to \textbf{Error:}\ \textit{string}$

Figure 22: Scheme operational semantics

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

$\mathscr{E}[\texttt{sh L (hs L } w_S)]_S \to \mathscr{E}[w_S]$

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

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

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

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

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

$\mathscr{E}[\texttt{sh } (\forall u_H.k_H)\ (\Lambda u'_H.e_H)]_S \to \mathscr{E}[\texttt{sh } k_H[\texttt{L}/u_H]\ e_H[\texttt{L}/u'_H]]$

Figure 23: Scheme-Haskell operational semantics

$\mathscr{E}[\texttt{sm L (mh L } k_H\ e_H)]_S \to \mathscr{E}[\texttt{wrong "Bad value"}]$

$\mathscr{E}[\texttt{sm L (ms L } w_S)]_S \to \mathscr{E}[w_S]$

$\mathscr{E}[\texttt{sm N } \overline{n}]_S \to \mathscr{E}[\overline{n}]$

$\mathscr{E}[\texttt{sm } \{k_M\}\ (\texttt{nil } t_M)]_S \to \mathscr{E}[\texttt{nil}]$

$\mathscr{E}[\texttt{sm } \{k_M\}\ (\texttt{cons } v_M\ v'_M)]_S \to \mathscr{E}[\texttt{cons (sm } k_M\ v_M) \texttt{ (sm } \{k_M\}\ v'_M)]$

$\mathscr{E}[\texttt{sm } (k_M \to k'_M)\ (\lambda x_M : t_M.e_M)]_S \to$

$\quad \mathscr{E}[\lambda x_S.\texttt{sm } k'_M\ ((\lambda x_M : t_M.e_M)\ (\texttt{ms } k_M\ x_S))]$

$\mathscr{E}[\texttt{sm } (\forall u_M.k_M)\ (\Lambda u'_M.e_M)]_S \to \mathscr{E}[\texttt{sm } k_M[\texttt{L}/u_M]\ e_M[\texttt{L}/u'_M]]$

Figure 24: Scheme-ML operational semantics

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

Figure 25: 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 26: Lump equality relation

3 Proof of Type Soundness

The proof of correctness is similar to that of Kinghorn [1], 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 $\vdash_S e_S :$ `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 $\Gamma, x_S :$ `TST` $\vdash_S e_S :$ `TST` and $\Gamma \vdash_S e'_S :$ `TST` then $\Gamma \vdash_S e_S[e'_S/x_S] :$ `TST`.

Proof. By structural induction. □

Lemma 5. *Type Substitution Preservation*
If $\Gamma, u_H \vdash_H e_H : t_H$ *and* $\Gamma \vdash_H t'_H$ *then* $\Gamma \vdash_H e_H[t'_H/u_H] : t_H[t'_H/u_H]$. *If* $\Gamma, u_M \vdash_M e_M : t_M$ *and* $\Gamma \vdash_M t'_M$ *then* $\Gamma \vdash_M e_M[t'_M/u_M] : t_M[t'_M/u_M]$.

Proof. By structural induction. □

Lemma 6. *Evaluation Context Preservation*
If $\vdash_H e_H : t_H$, $\vdash_H e'_H : t_H$, *and* $\vdash_H \mathscr{E}[e_H]_H : t'_H$ *then* $\vdash_H \mathscr{E}[e'_H]_H : t'_H$. *If* $\vdash_M e_M : t_M$, $\vdash_M e'_M : t_M$, *and* $\vdash_M \mathscr{E}[e_M]_M : t'_M$ *then* $\vdash_M \mathscr{E}[e'_M]_M : t'_M$. *If* $\vdash_S e_S : \texttt{TST}$, $\vdash_S e'_S : \texttt{TST}$, *and* $\vdash_S \mathscr{E}[e_S]_S : \texttt{TST}$ *then* $\vdash_S \mathscr{E}[e'_S]_S : \texttt{TST}$.

Proof. By structural induction. □

Theorem 5. *Haskell Preservation*
If $\Gamma \vdash_H e_H : t_H$ *and* $\mathscr{E}[e_H]_H \to \mathscr{E}[e'_H]$ *then* $\Gamma \vdash_H e'_H : t_H$.

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

Theorem 6. *ML Preservation*
If $\Gamma \vdash_M e_M : t_M$ *and* $e_M \to e'_M$ *then* $\Gamma \vdash_M e'_M : t_M$.

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

Theorem 7. *Scheme Preservation*
If $\Gamma \vdash_S e_S : \texttt{TST}$ *and* $e_S \to e'_S$ *then* $\Gamma \vdash_S e'_S : \texttt{TST}$.

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

4 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.

The approach this paper used for interoperation between three languages is not scalable. Values from each language can be directly converted to values of the other two languages and back. n languages require $n*(n-1)$ conversion mappings between them. As the number of languages increases, the number of conversion mappings

grows geometrically, which is unmaintainable. A better approach would be to make only two conversion mappings per language and chain them together to form a single path between any two languages, which would require only $n-1$ conversion mappings and grow linearly. Were this done for this model, the number of conversion mappings would be four instead of six.

References

[1] David 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.

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