INTEROPERATION FOR INCOMPATIBLE EVALUATION STRATEGIES

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

June 2008

APPROVAL PAGE

TITLE: Interoperation for Incompatible Evaluation Strategies

AUTHOR: William Faught

DATE SUBMITTED: June 2008

Dr. John Clements	
Advisor or Committee Chair	Signature
Dr. Gene Fisher	
Committee Member	Signature
Dr. Aaron Keen	
Committee Member	Signature

Abstract

Interoperation for Incompatible Evaluation Strategies

by

William Faught

Software components written in different programming languages can cooperate through interoperation. Differences between languages—incompatibilities—complicate interoperation. This paper explores and resolves incompatible type systems, support for parametricity, and evaluation strategies with a model of computation, gives a thorough proof of its type soundness, and describes an implementation of it. The model uses contracts for higher-order functions and lump types to resolve incompatible type systems, label types to resolve incompatible support for parametricity, and delayed conversions for list constructions to resolve incompatible evaluation strategies. These mechanisms enable the interoperation of Haskell, ML, and Scheme without compromising their semantics.

Acknowledgements

I want to thank my parents, Jerry and Jo Ann, for their encouragement, advice, and support, without which this would not have been possible.

I want to thank my adviser, John Clements, for helping me along the way. I very much appreciate the time he set aside for me and his advice.

Contents

List of Figures

Chapter 1

Introduction

The complexities of software interoperation in part engender the proverbial reinvention of the wheel. Programmers forgo preexisting solutions to problems where interoperation proves too cumbersome; they reimplement software components, rather than reuse them. Disparate programming language features complicate the conversion of values exchanged between components of different languages. Resolving language incompatibilities transparently at boundaries between component languages facilitates interoperation by unburdening programmers. This paper explores and resolves two such incompatibilities with a model of computation and then proves its type soundness and describes its implementation.

The first incompatibility is type systems. Static type systems calculate and validate the types of expressions before run time, thereby ensuring that well-typed programs do not encounter type errors during run time. Dynamic type systems detect invalid operations on values using value predicates during run time and do not calculate or validate the types of expressions at compile time. Statically-typed languages—languages that use static type systems—that use values from dynamically-typed languages must verify that the values match their expected types. Languages are assumed to exchange a common set of values that can be checked straightforwardly without coercion. Mismatched values and expected types could cause type errors during run time and violate type soundness. Ad-hoc polymorphism in dynamically-typed languages enables argument types to determine polymorphic function behavior. Since determining function behavior is undecidable [2], actual types for these functions cannot be reliably calculated at language boundaries and compared to expected types. Instead, they are wrapped in contracts [4] that defer the checking of their parameter and result types until they are used during run time. If they are never used, their types cannot be checked, but neither can they cause type errors.

The second incompatibility is parametricity. Parametric polymorphism in statically-typed languages enables function types to be abstracted with type variables and then instantiated into concrete types. Parametricity constrains the behavior of parametric polymorphic functions by ensuring that they behave the same regardless of the types and values of their arguments, and that functions with instantiated result types produce as their results the arguments associated with the same instantiated types. Functions from dynamically-typed languages that use value predicates or conditions on arguments and are used as parametric polymorphic functions by languages that have parametricity can violate their parametricity. Arguments for these functions must be obscured such that value predicates and conditions cannot examine them and annotated to ensure the correct ones are produced as results.

The third incompatibility is evaluation strategies. Evaluation strategies determine the order in which languages evaluate expressions. Eager evaluation evaluates expressions regardless of necessity, and lazy evaluation evaluates expressions only where necessary. Lazy languages—languages that use lazy evaluation—can construct infinite streams as lists because they do not evaluate list elements when lists are constructed, but eager languages cannot because they do. Since there exist lazy lists—lists in lazy languages—for which no naturally equivalent eager lists exist, lazy lists crossing to eager languages are not converted to eager lists. Instead, elements of lazy lists are converted when accessed by eager languages if they are not lazy lists too.

The languages in the model must be able to express programs in which the aforementioned three incompatibilities arise. Haskell, ML, and Scheme each possess a unique combination of properties that together are sufficient for this purpose: Haskell and ML use static type systems and have parametricity, Scheme uses a dynamic type system, ML and Scheme use eager evaluation, and Haskell uses lazy evaluation.

The rest of the paper is organized as follows: Chapter 2 defines the model of computation, Chapter 3 proves the type soundness of the model, Chapter 4 describes an implementation of the model, Chapter 5 discusses related work, Chapter 6 discusses future work, and Chapter 7 discusses the conclusions.

Chapter 2

Model of Computation

The model of computation comprises three dependent models of computation, based on that of Matthews and Findler (((CITATION))). The Haskell and ML models are based on System F, extended with a fixed-point operation. The Scheme model is based on lambda calculus, having a simple type system to ensure no free variables, and extended with type predicates. All models have natural numbers, arithmetic, conditions, lists, and errors. The Haskell model has a call-by-name evaluation strategy, and ML and Scheme have call-by-value evaluation strategies. The models are presented with grammars and operational semantics in the style of (((CITATION))) and typing rules.

The Haskell model has a (((STRICT?))) subset of the strictness points of the ML and Scheme models, and hence forces the reduction of fewer expressions where those expressions are not used. When a function from the Haskell model is converted to a function in the ML or Scheme models, this same subset must be preserved or the meaning of the function will change and parametricity will not hold. Concretely, this means that function arguments and list construction operands must not be reduced. Thus evaluation contexts for the ML and Scheme models are made aware of whether Haskell language boundary guards are in these places, and if so, to make them irreducible.

Since Haskell language boundary guards are forced in some places but not others, they must be considered a value sometimes, but not others. Thus there are two kinds of values: all values, called unforced values, which include imported Haskell expressions, and forced values, which exclude imported Haskell expressions. Unforced values occur in the evaluation contexts and reduction rules where Haskell importations should not be forced, namely function arguments and list construction operands, and forced values occur everywhere else a value is required.

Evaluation contexts are split into two: forced, E, and unforced, U. Only forced evaluation contexts can reduce anything, including Haskell importations, and unforced evaluation contexts restrict where expressions are forced. Where U appears in an evaluation context, any Haskell importation matching that expression is not forced because only E can reduce it.

Since the Haskell and ML models have their own types, type abstractions from one imported into the other cannot be easily converted, because any type the conversion is applied to cannot be substituted into the other language. Instead, and like the importation of parametric polymorphic types into the Scheme model, the lump type is subsituted into the imported type abstraction, and a lump equality relation, (((LUMP EQUALITY REL))) asserts that corresponding parts of the inner and outer types of the importation must be equal, or one of them must be a lump.

——

The model of computation is based on that of Matthews and Findler **??**. Their model consists of two simple models, one representing ML and the other

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

Figure 2.1: Function and list conversions with unforced points.

Scheme.

The ML model is a simply-typed lambda calculus extended with parametric polymorphism called System F. The substitution semantics by which type abstractions are applied means the ML model has parametricity, which is a property that ensures that programs behave the same regardless of the types applied to by type abstractions. The ML model introduces new expressions, type abstractions and type applications, to express parametric polymorphism, and new types, type variables and forall types, for them. The ML model uses an eager evaluation strategy.

The Scheme model is an extended untyped lambda calculus using an eager evaluation strategy. Value predicates enable ad-hoc polymorphism. Uses a simple type system to check for free variables.

To this mix we introduce a Haskell model identical to the ML model, except it uses a lazy evaluation strategy.

The definitions of the Haskell, ML, and Scheme models begin in figures 2.2, 2.7, and 2.12.

Expressions are written e, types are written t, forced values are written w, unforced values are written v, forced evaluation contexts are written E, and unforced evaluation contexts are written U. Symbols that represent grammar non-terminals or relations typically have letter subscripts that specify a model.

The Haskell and ML models have static type systems that use typing environments, written Γ, and typing relations, written $\vdash$. Typing judgements for expressions are written $\Gamma \vdash e : t$, where e is bound in Γ and has the type t. Typing judgements for types are written $\Gamma \vdash t$ and mean t is bound in Γ. Extended typing environments are written $\Gamma, x : t$ for variables and Γ, u for type variables. Typing environments are omitted where empty. The Scheme model uses a simple type system to ensure no free variables. Every well-typed Scheme model expression has the type `TST`. Type substitution within types is written $x[y/z]$, where the type y is substituted for free occurrences of the type variable z in the type x.

TODO: opsem

Expression and type substitutions within expressions are written like type substitutions within types.

2.1 Natural Numbers

Natural numbers are written $\overline{n}$, which syntactically represents the natural number n. Natural numbers have the type `N` for the Haskell and ML models. Addition and subtraction are written $+\ e\ e$ and $-\ e\ e$. In the Scheme model, if either arithmetic operand is not a natural number, then the result is an error, written `wrong` "Not a number", which discards the evaluation context and halts the computation. Conditions, written `if0` e e e, test whether a value is the natural number $\overline{0}$. If it is, then it reduces to the first alternative; otherwise, it reduces to the second. The Scheme model has predicates that determine whether values are functions, lists, empty lists, and natural numbers, written `fun`? e_S, `list`? e_S, `null`? e_S, and `num`? e_S. Predicates reduce to $\overline{0}$ if true and $\overline{1}$ if false.

2.2 Lists

Empty lists are written `nil` t in the Haskell and ML models and `nil` in the Scheme model. List constructions are written `cons` e e in the Haskell, ML, and Scheme models. Lists of elements that have the type t in the Haskell and ML models have the type $\{t\}$. The Scheme model has predicates that determine whether values are lists and empty lists, written `list`? e and `null`? e.

2.3 Functions

TODO

2.4 Types

TODO

2.5 Interoperation

TODO

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

Figure 2.2: Haskell grammar 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 2.3: Haskell typing rules

$$\mathscr{E}[(\lambda x_H : t_H.e_H)\ e'_H]_H \rightarrow \mathscr{E}[e_H[e'_H/x_H]]$$

$$\mathscr{E}[(\Lambda u_H.e_H)\langle t_H\rangle]_H \rightarrow \mathscr{E}[e_H[b \diamond t_H/u_H]]$$

$$\mathscr{E}[\texttt{fix}\ (\lambda x_H : t_H.e_H)]_H \rightarrow \mathscr{E}[e_H[\texttt{fix}\ (\lambda x_H : t_H.e_H)/x_H]]$$

$$\mathscr{E}[+\ \overline{n}\ \overline{n}']_H \rightarrow \mathscr{E}[\overline{n+n'}]$$

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

$$\mathscr{E}[\texttt{if0}\ \overline{0}\ e_H\ e'_H]_H \rightarrow \mathscr{E}[e_H]$$

$$\mathscr{E}[\texttt{if0}\ \overline{n}\ e_H\ e'_H]_H \rightarrow \mathscr{E}[e'_H]\ (n \neq 0)$$

$$\mathscr{E}[\texttt{hd}\ (\texttt{nil}\ t_H)]_H \rightarrow \mathscr{E}[\texttt{wrong}\ t_H\ \text{“Empty list”}]$$

$$\mathscr{E}[\texttt{tl}\ (\texttt{nil}\ t_H)]_H \rightarrow \mathscr{E}[\texttt{wrong}\ \{t_H\}\ \text{“Empty list”}]$$

$$\mathscr{E}[\texttt{hd}\ (\texttt{cons}\ e_H\ e'_H)]_H \rightarrow \mathscr{E}[e_H]$$

$$\mathscr{E}[\texttt{tl}\ (\texttt{cons}\ e_H\ e'_H)]_H \rightarrow \mathscr{E}[e'_H]$$

$$\mathscr{E}[\texttt{null?}\ (\texttt{nil}\ t_H)]_H \rightarrow \mathscr{E}[\overline{0}]$$

$$\mathscr{E}[\texttt{null?}\ (\texttt{cons}\ e_H\ e'_H)]_H \rightarrow \mathscr{E}[\overline{1}]$$

$$\mathscr{E}[\texttt{wrong}\ t_H\ \mathit{string}]_H \rightarrow \textbf{Error: } \mathit{string}$$

Figure 2.4: Haskell operational semantics

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

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

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

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

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

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

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

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

Figure 2.5: Haskell-ML operational semantics

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.6: Haskell-Scheme operational semantics

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

Figure 2.7: ML grammar 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 2.8: ML typing rules

$$\mathscr{E}[(\lambda x_M : t_M.e_M)\ v_M]_M \rightarrow \mathscr{E}[e_M[v_M/x_M]]$$

$$\mathscr{E}[(\Lambda u_M.e_M)\langle t_M\rangle]_M \rightarrow \mathscr{E}[e_M[b \diamond t_M/u_M]]$$

$$\mathscr{E}[\texttt{fix}\ (\lambda x_M : t_M.e_M)]_M \rightarrow \mathscr{E}[e_M[\texttt{fix}\ (\lambda x_M : t_M.e_M)/x_M]]$$

$$\mathscr{E}[+\ \overline{n}\ \overline{n}']_M \rightarrow \mathscr{E}[\overline{n+n'}]$$

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

$$\mathscr{E}[\texttt{if0}\ \overline{0}\ e_M\ e'_M]_M \rightarrow \mathscr{E}[e_M]$$

$$\mathscr{E}[\texttt{if0}\ \overline{n}\ e_M\ e'_M]_M \rightarrow \mathscr{E}[e'_M]\ (n \neq 0)$$

$$\mathscr{E}[\texttt{hd}\ (\texttt{nil}\ t_M)]_M \rightarrow \mathscr{E}[\texttt{wrong}\ t_M\ \text{“Empty list”}]$$

$$\mathscr{E}[\texttt{tl}\ (\texttt{nil}\ t_M)]_M \rightarrow \mathscr{E}[\texttt{wrong}\ \{t_M\}\ \text{“Empty list”}]$$

$$\mathscr{E}[\texttt{hd}\ (\texttt{cons}\ v_M\ v'_M)]_M \rightarrow \mathscr{E}[v_M]$$

$$\mathscr{E}[\texttt{tl}\ (\texttt{cons}\ v_M\ v'_M)]_M \rightarrow \mathscr{E}[v'_M]$$

$$\mathscr{E}[\texttt{null?}\ (\texttt{nil}\ t_M)]_M \rightarrow \mathscr{E}[\overline{0}]$$

$$\mathscr{E}[\texttt{null?}\ (\texttt{cons}\ v_M\ v'_M)]_M \rightarrow \mathscr{E}[\overline{1}]$$

$$\mathscr{E}[\texttt{wrong}\ t_M\ \mathit{string}]_H \rightarrow \textbf{Error: } \mathit{string}$$

Figure 2.9: ML operational semantics

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

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

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

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

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

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

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

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

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

Figure 2.10: ML-Haskell operational semantics

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

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

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

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

$\mathscr{E}[\texttt{ms}\ \{k_M\}\ w_S]_M \rightarrow \mathscr{E}[\texttt{wrong}\ \lfloor \{k_M\} \rfloor\ \text{“Not a list”}]$

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

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

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

$\quad (w_S \neq \texttt{sm}\ (b \diamond t_M)\ e_M)$

$\mathscr{E}[\texttt{ms}\ (k_M \rightarrow k'_M)\ (\lambda x_S.e_S)]_M \rightarrow$

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

$\mathscr{E}[\texttt{ms}\ (k_M \rightarrow k'_M)\ w_S]_M \rightarrow \mathscr{E}[\texttt{wrong}\ \lfloor k_M \rightarrow k'_M \rfloor\ \text{“Not a function”}]$

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

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

Figure 2.11: ML-Scheme operational semantics

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

Figure 2.12: Scheme grammar 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 2.13: Scheme typing rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.14: Scheme operational semantics

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

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

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

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

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

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

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

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

Figure 2.15: Scheme-Haskell operational semantics

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

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

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

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

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

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

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

Figure 2.16: Scheme-ML operational semantics

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

Figure 2.17: Unbrand function

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

Figure 2.18: Lump equality relation

Chapter 3

Conclusion

This work resolved three language incompatibilities in a system of interoperation for three diverse languages. It resolved incompatible type systems with contracts for higher-order functions and lump types. It resolved incompatible support for parametricity with label types. It resolved incompatible evaluation strategies with delayed conversions for list constructions. It defined a model of computation that can express interoperation where the aforementioned incompatibilities arise and resolve them, provided a proof of its type soundness, and described an implementation of it that supported additional language features.

Bibliography

[1] N. Benton. Embedded interpreters. *J. Funct. Program.*, 15(4):503–542, 2005.

[2] M. Blume and D. McAllester. A sound (and complete) model of contracts. *SIGPLAN Not.*, 39(9):189–200, 2004.

[3] ECMA. *Common Language Infrastructure (CLI)*, 4th edition, June 2006.

[4] R. B. Findler and M. Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, 2002.

[5] A. Guha, J. Matthews, R. B. Findler, and S. Krishnamurthi. Relationally-parametric polymorphic contracts. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 29–40, New York, NY, USA, 2007. ACM.

[6] F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: translating scheme to ml. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 192–203, New York, NY, USA, 1995. ACM.

[7] 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.

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

[9] Microsoft. *Microsoft Interface Definition Language*, November 2007.

[10] Object Management Group. *Common Object Request Broker Architecture (CORBA) Specification*, 3.1 edition, August 2004.

[11] B. C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[12] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 964–974, New York, NY, USA, 2006. ACM.