Interoperation for Incompatible Evaluation Strategies

William Faught

October 4, 2010

Abstract

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.

1 Introduction

The complexities of software interoperation in part engender the proverbial reinvention of the wheel. Programmers forgo existing solutions to problems in other languages 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 reuse by unburdening programmers. To address part of this problem, this paper presents a model of computation that resolves incompatible evaluation strategies.

For an expression imported from one language to another, if its conversion requires any evaluation within the expression, then the evaluation must follow the order of evaluation defined by the source language's evaluation strategy. Were the order of evaluation to differ, the meaning of the expression may change, as expressions that were assumed to never be evaluated, or evaluated a finite number of times, may in fact be evaluated.

Interoperable languages must preserve strictness points within converted expressions.

Call-by-name and call-by-value evaluation strategies use orders of evaluation that take opposite approaches. Call-by-name evaluates expressions needed only by primitive operations, whereas call-by-value evaluates all expressions. As such, call-by-name evaluates a proper subset of the expressions that call-by-value does. In other words, the set of call-by-name strictness points is a proper subset of that of call-by-value. The exclusive disjunction between these two sets is the set of incompatible strictness points that may change the meaning of expressions that are converted from call-by-name to call-by-value. In call-by-name, expressions at these points may be assumed to never be evaluated, or assumed to be evaluated only a finite number of times. Call-by-value may violate these assumptions, and hence interoperation may change the meaning of programs.

In general, the exclusive disjunction of the strictness points of two interoperable languages is their set of incompatible strictness points. If this set is not empty, then interoperation must preserve those strictness points in the other language and apply them to converted expressions.

The language that has an incompatible strictness point must delay evaluating imported expressions from the other language

Interoperation requires preserving these strictness points for each evaluation strategy, even after a call-by-name value is converted to a call-by-value value. Otherwise, expressions may be evaluated where they were not before and cause errors or diverge. This means deferring the evaluation of converted call-by-name expressions in these incompatible points using a dual notion of values and evaluation contexts in call-by-value languages that handles both call-by-value values and converted call-by-name expressions, called *forced* and *unforced* values and evaluation contexts.

2 Model of Computation

The model of computation comprises three dependent models of computation, based on that of Matthews and Findler [3]. 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.

2.1 Notation

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 environ-

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

Figure 1: Function and list conversions with unforced points.

ments, 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.

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

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

Figure 2: 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 3: Haskell typing rules

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

Figure 4: Haskell operational semantics

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

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

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

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

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

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

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

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

Figure 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 6: Haskell-Scheme operational semantics

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

Figure 7: ML syntax and evaluation contexts

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

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

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

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

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

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

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

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

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

Figure 8: ML typing rules

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

Figure 9: ML operational semantics

$$\mathscr{E}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hm}\ \texttt{L}\ t'_M\ w_M)]_M \to \mathscr{E}[w_M]\ (t_M = t'_M \text{ and } t_M \neq \texttt{L})$$
$$\mathscr{E}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hm}\ \texttt{L}\ t'_M\ w_M)]_M \to \mathscr{E}[\texttt{wrong}\ t_M\ \text{``Type mismatch''}]\ (t_M \neq t'_M \text{ and } t_M \neq \texttt{L})$$
$$\mathscr{E}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hs}\ \texttt{L}\ w_S)]_H \to \mathscr{E}[\texttt{wrong}\ t_M\ \text{``Bad value''}]\ (t_M \neq \texttt{L})$$
$$\mathscr{E}[\texttt{mh}\ \texttt{N}\ \texttt{N}\ \overline{n}]_M \to \mathscr{E}[\overline{n}]$$
$$\mathscr{E}[\texttt{mh}\ \{t_M\}\ \{t_H\}\ (\texttt{nil}\ t'_H)]_M \to \mathscr{E}[\texttt{nil}\ t_M]$$
$$\mathscr{E}[\texttt{mh}\ \{t_M\}\ \{t_H\}\ (\texttt{cons}\ e_H\ e'_H)]_M \to \mathscr{E}[\texttt{cons}\ (\texttt{mh}\ t_M\ t_H\ e_H)\ (\texttt{mh}\ \{t_M\}\ \{t_H\}\ e'_H)]$$
$$\mathscr{E}[\texttt{mh}\ (t_M \to t'_M)\ (t_H \to t'_H)\ (\lambda x_H : t''_H.e_H)]_M \to$$
$$\mathscr{E}[\lambda x_M : t_M.\texttt{mh}\ t'_M\ t'_H\ ((\lambda x_H : t''_H.e_H)\ (\texttt{hm}\ t_H\ t_M\ x_M))]$$
$$\mathscr{E}[\texttt{mh}\ (\forall u_M.t_M)\ (\forall u_H.t_H)\ (\Lambda u'_H.e_H)]_M \to \mathscr{E}[\Lambda u_M.\texttt{mh}\ t_M\ t_H[\texttt{L}/u_H]\ e_H[\texttt{L}/u'_H]]$$

Figure 10: ML-Haskell operational semantics

$$\mathscr{E}[\texttt{ms}\ \texttt{N}\ \overline{n}]_M \rightarrow \mathscr{E}[\overline{n}]$$
$$\mathscr{E}[\texttt{ms}\ \texttt{N}\ w_S]_M \rightarrow \mathscr{E}[\texttt{wrong}\ \texttt{N}\ \text{“Not a number”}]\ (w_S \neq \overline{n})$$
$$\mathscr{E}[\texttt{ms}\ \{k_M\}\ \texttt{nil}]_M \rightarrow \mathscr{E}[\texttt{nil}\ \lfloor k_M \rfloor]$$
$$\mathscr{E}[\texttt{ms}\ \{k_M\}\ (\texttt{cons}\ v_S\ v'_S)]_M \rightarrow \mathscr{E}[\texttt{cons}\ (\texttt{ms}\ k_M\ v_S)\ (\texttt{ms}\ \{k_M\}\ v'_S)]$$
$$\mathscr{E}[\texttt{ms}\ \{k_M\}\ w_S]_M \rightarrow \mathscr{E}[\texttt{wrong}\ \lfloor \{k_M\} \rfloor\ \text{“Not a list”}]$$
$$(w_S \neq \texttt{nil} \text{ and } w_S \neq \texttt{cons}\ v_S\ v'_S)$$
$$\mathscr{E}[\texttt{ms}\ (b \diamond t_M)\ (\texttt{sm}\ (b \diamond t_M)\ v_M)]_M \rightarrow \mathscr{E}[v_M]$$
$$\mathscr{E}[\texttt{ms}\ (b \diamond t_M)\ w_S]_M \rightarrow \mathscr{E}[\texttt{wrong}\ \lfloor b \diamond t_M \rfloor\ \text{“Brand mismatch”}]$$
$$(w_S \neq \texttt{sm}\ (b \diamond t_M)\ e_M)$$
$$\mathscr{E}[\texttt{ms}\ (k_M \rightarrow k'_M)\ (\lambda x_S.e_S)]_M \rightarrow$$
$$\mathscr{E}[\lambda x_M : \lfloor k_M \rfloor.\texttt{ms}\ k'_M\ ((\lambda x_S.e_S)\ (\texttt{sm}\ k_M\ x_M))]$$
$$\mathscr{E}[\texttt{ms}\ (k_M \rightarrow k'_M)\ w_S]_M \rightarrow \mathscr{E}[\texttt{wrong}\ \lfloor k_M \rightarrow k'_M \rfloor\ \text{“Not a function”}]$$
$$(w_S \neq \lambda x_S.e_S)$$
$$\mathscr{E}[\texttt{ms}\ (\forall u_M.k_M)\ w_S]_M \rightarrow \mathscr{E}[\Lambda u_M.\texttt{ms}\ k_M\ w_S]$$

Figure 11: ML-Scheme operational semantics

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

Figure 12: Scheme syntax and evaluation contexts

$$\overline{\vdash_S \texttt{TST}}$$

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

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

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

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

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

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

Figure 13: Scheme typing rules

$$\mathscr{E}[(\lambda x_S.e_S)\ v_S]_S \to \mathscr{E}[e_S[v_S/x_S]]$$
$$\mathscr{E}[w_S\ v_S]_S \to \mathscr{E}[\texttt{wrong}\ \text{“Not a function”}]\ (w_S \neq \lambda x_S.e_S)$$
$$\mathscr{E}[+\ \overline{n}\ \overline{n}']_S \to \mathscr{E}[\overline{n+n'}]$$
$$\mathscr{E}[-\ \overline{n}\ \overline{n}']_S \to \mathscr{E}[\overline{max(n-n',0)}]$$
$$\mathscr{E}[o\ w_S\ w'_S]_S \to \mathscr{E}[\texttt{wrong}\ \text{“Not a number”}]\ (w_S \neq \overline{n}\ \text{or}\ w'_S \neq \overline{n})$$
$$\mathscr{E}[\texttt{if0}\ \overline{0}\ e_S\ e'_S]_S \to \mathscr{E}[e_S]$$
$$\mathscr{E}[\texttt{if0}\ \overline{n}\ e_S\ e'_S]_S \to \mathscr{E}[e'_S]\ (n \neq 0)$$
$$\mathscr{E}[\texttt{if0}\ w_S\ e_S\ e'_S]_S \to \mathscr{E}[\texttt{wrong}\ \text{“Not a number”}]\ (w_S \neq \overline{n})$$
$$\mathscr{E}[f\ \texttt{nil}]_S \to \mathscr{E}[\texttt{wrong}\ \text{“Empty list”}]$$
$$\mathscr{E}[\texttt{hd}\ (\texttt{cons}\ v_S\ v'_S)]_S \to \mathscr{E}[v_S]$$
$$\mathscr{E}[\texttt{tl}\ (\texttt{cons}\ v_S\ v'_S)]_S \to \mathscr{E}[v'_S]$$
$$\mathscr{E}[f\ w_S]_S \to \mathscr{E}[\texttt{wrong}\ \text{“Not a list”}]\ (w_S \neq \texttt{nil}\ \text{and}\ w_S \neq \texttt{cons}\ v_S\ v'_S)$$
$$\mathscr{E}[\texttt{fun?}\ (\lambda x_S.e_S)]_S \to \mathscr{E}[\overline{0}]$$
$$\mathscr{E}[\texttt{fun?}\ w_S]_S \to \mathscr{E}[\overline{1}]\ (w_S \neq \lambda x_S.e_S)$$
$$\mathscr{E}[\texttt{list?}\ \texttt{nil}]_S \to \mathscr{E}[\overline{0}]$$
$$\mathscr{E}[\texttt{list?}\ (\texttt{cons}\ v_S\ v'_S)]_S \to \mathscr{E}[\overline{0}]$$
$$\mathscr{E}[\texttt{list?}\ w_S]_S \to \mathscr{E}[\overline{1}]\ (w_S \neq \texttt{nil}\ \text{and}\ w_S \neq \texttt{cons}\ v_S\ v'_S)$$
$$\mathscr{E}[\texttt{null?}\ \texttt{nil}]_S \to \mathscr{E}[\overline{0}]$$
$$\mathscr{E}[\texttt{null?}\ w_S]_S \to \mathscr{E}[\overline{1}]\ (w_S \neq \texttt{nil})$$
$$\mathscr{E}[\texttt{num?}\ \overline{n}]_S \to \mathscr{E}[\overline{0}]$$
$$\mathscr{E}[\texttt{num?}\ w_S]_S \to \mathscr{E}[\overline{1}]\ (w_S \neq \overline{n})$$
$$\mathscr{E}[\texttt{wrong}\ \mathit{string}]_S \to \textbf{Error:}\ \mathit{string}$$

Figure 14: Scheme operational semantics

$$\mathscr{E}[\texttt{sh L (hm L } k_M\ w_M)]_S \rightarrow \mathscr{E}[\texttt{wrong} \text{ ``Bad value''}]$$
$$\mathscr{E}[\texttt{sh L (hs L } w_S)]_S \rightarrow \mathscr{E}[w_S]$$
$$\mathscr{E}[\texttt{sh N } \overline{n}]_S \rightarrow \mathscr{E}[\overline{n}]$$
$$\mathscr{E}[\texttt{sh } \{k_H\} \texttt{ (nil } t_H)]_S \rightarrow \mathscr{E}[\texttt{nil}]$$
$$\mathscr{E}[\texttt{sh } \{k_H\} \texttt{ (cons } e_H\ e'_H)]_S \rightarrow \mathscr{E}[\texttt{cons (sh } k_H\ e_H) \texttt{ (sh } \{k_H\}\ e'_H)]$$
$$\mathscr{E}[\texttt{sh } (k_H \rightarrow k'_H)\ (\lambda x_H : t_H.e_H)]_S \rightarrow$$
$$\mathscr{E}[\lambda x_S.\texttt{sh } k'_H\ ((\lambda x_H : t_H.e_H)\ (\texttt{hs } k_H\ x_S))]$$
$$\mathscr{E}[\texttt{sh } (\forall u_H.k_H)\ (\Lambda u'_H.e_H)]_S \rightarrow \mathscr{E}[\texttt{sh } k_H[\texttt{L}/u_H]\ e_H[\texttt{L}/u'_H]]$$

Figure 15: Scheme-Haskell operational semantics

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

Figure 16: Scheme-ML operational semantics

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

Figure 17: 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 18: Lump equality relation

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
$\longrightarrow$	Reduction relation
t	Type
u	Type variable
Γ	Typing environment
$\vdash$	Typing relation
U	Unforced evaluation context
v	Unforced value
x	Variable

Figure 19: Symbol names.

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

Figure 20: 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 21: Syntax names.

3 Conclusion

Evaluation strategy incompatibilities can be resolved transparently at language boundaries. Where two interoperable languages do not share a strictness point, if an expression crosses from the language without the strictness point to the one with, then the conversion of the expression must be delayed until the value is needed. Otherwise, the expression may diverge or reduce to an error, and thus reduce differently due to the interoperation.

Statically-typed languages with parametric polymorphism can interoperate through lump equality. Normally, expressions have equivalent types on both sides of the language boundary, but in the case of type abstractions, the outer type argument cannot be substituted into the inner language's type abstraction. A lump is substituted into the inner language's type of the guard and the applied to the type abstraction, and the lump equality relation allows for a notion of type equivalence where the substituted lump type can match the outer type instantiated for the outer type variable.

In an interoperable system of n languages, there must be $n * (n - 1)$ language mappings, two for every pair of languages to convert to and from one another. As this model of computation demonstrates, the geometric growth of the interoperation model is almost too much to manage. In general, for a sizable group of languages, this approach of interface bridging is unmaintainable. A better approach is to make language mappings between a language and only one other language that is most similar to it. As long as there is a spanning tree for the graph of languages, the number of languages mappings in the best case is $n - 1$, linear growth.

References

[1] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, 2002.

[2] 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.

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

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