Operational Semantics for Multi-Language Programs

Jacob Matthews
University of Chicago
and
Robert Bruce Findler
University of Chicago

Abstract

Interoperability is big business, a fact to which .NET, the JVM, and COM can attest. Language designers are well aware of this, and they are designing programming languages that reflect it — for instance, SML#, Mondrian, and Scala all treat interoperability as a central design feature. Still, current multi-language research tends not to focus on the semantics of these features, but only on how to implement them efficiently. In this paper, we attempt to rectify that by giving a technique for specifying the operational semantics of a multi-language system as a composition of the models of its constituent languages. Our technique abstracts away the low-level details of interoperability like garbage collection and representation coherence, and lets us focus on semantic properties like type-safety, equivalence, and termination behavior. In doing so it allows us to adapt standard theoretical techniques such as subject-reduction, logical relations, and operational equivalence for use on multilanguage systems. Generally speaking, our proofs of properties in a multi-language context are mutually-referential versions of their single language counterparts.

We demonstrate our technique with a series of strategies for embedding a Scheme-like language into an ML-like language. We start by connecting very simple languages with a very simple strategy, and work our way up to languages that interact in sophisticated ways and have sophisticated features such as polymorphism and effects. Along the way, we prove relevant results such as type-soundness and termination for each system we present using adaptations of standard techniques.

Beyond giving simple expressive models, our studies have uncovered several interesting facts about interoperability. For example, higher-order function contracts naturally emerge as the glue to ensure that interoperating languages respect each other's type systems. Our models also predict that the embedding strategy where foreign values are opaque is as expressive as the embedding strategy where foreign values are translated to corresponding values in the other language, and we were able to experimentally verify this behavior using PLT Scheme's foreign function interface.

1. INTRODUCTION

A modern large-scale software system is likely written in a variety of languages: its core might be written in Java, while it has specialized system interaction routines written in C and a web-based user interface written in PHP. And even academic languages have caught multi-language programming fever, due perhaps to temptingly large numbers of libraries available for other languages. This has prompted language implementors to target COM [Finne et al. 1999; Steckler 1999], Java Virtual Machine bytecode [Benton and Kennedy 1999; Odersky et al. 2005], and most recently Microsoft's Common Language

Runtime [Benton et al. 2004; Meijer et al. 2001; Pinto 2003]. Furthermore, where foreign function interfaces have historically been used in practice to allow high-level safe languages to call libraries written in low-level unsafe languages like C (as was the motivation for the popular wrapper generator SWIG [Beazley 1996]), these new foreign function interfaces are built to allow high-level, safe languages to interoperate with other high-level, safe languages, such as Python with Scheme [Meunier and Silva 2003] and Lua with OCaml [Ramsey 2003].

Since these embeddings are driven by practical concerns, the research that accompanies them rightly focuses on the bits and bytes of interoperability — how to represent data in memory, how to call a foreign function efficiently, and so on. But an important theoretical problem arises, independent of these implementation-level concerns: how can we reason formally about multi-language programs? This is a particularly important question for systems that involve typed languages, because we have to show that the embeddings respect their constituents' type systems.

In this paper we present a simple method for giving operational semantics for multi-language systems that are rich enough to model a wide variety of multi-language embedding strategies, and powerful enough that we have been able to use them for type soundness proofs, proofs by logical relation, and contextual equivalence proofs. Our technique is based on simple constructs we call *boundaries*, which regulate both control flow and value conversion between languages. We introduce boundaries through series of calculi in which we extend a simple ML-like language with the ability to interact in various ways with a simple Scheme-like language.

In section 2, we introduce those two constituent languages formally and connect them using a primitive embedding where values in one language are opaque to the other. In section 3, we enrich that embedding into an embedding where boundaries use type information to transform values in one language into counterparts in the other language, and we show that this embedding has an interesting connection to higher-order contracts. Section 4 shows two surprising relationships between the expressive power of these two embeddings in section. In section 5, we argue that our technique can model more realistic languages by adding two different exceptions systems, each of which corresponds to existing programming language implementations. In section 6, we consider non-parametric and parametric polymorphism, and in section 7 we show how our treatment of parametric polymorphism also applies to a more general class of non-type-directed conversions. Section 8 summarizes related work and section 9 concludes.

2. THE LUMP EMBEDDING

To begin, we pick two languages, give them formal models, and then tie those formal models together. In the interest of focusing on interoperation rather than the special features of particular languages, we have chosen two simple calculi: an extended model of the untyped call-by-value lambda calculus, which we use as a stand-in for Scheme, and an extended model of the simply-typed lambda calculus, which we use as a stand-in for ML (though it more closely resembles Plotkin's PCF without fixpoint operators [Plotkin 1977]). Figures 1 and 2 present these languages in an abstract manner that we instantiate multiple ways to model different forms of interoperability. One goal of this section is to explain that figure's peculiarities, but for now notice that aside from unusual subscripts and font choices, the two language models look pretty much as they would in a normal Felleisen-and-Hieb-style presentation [1992].

$$\begin{array}{lcl}
\mathbf{e} & = & \mathbf{x} \mid \mathbf{v} \mid (\mathbf{e}\ \mathbf{e}) \mid (op\ \mathbf{e}\ \mathbf{e}) \mid (\texttt{if0}\ \mathbf{e}\ \mathbf{e}\ \mathbf{e}) \\
\mathbf{v} & = & (\lambda \mathbf{x} : \tau .\ \mathbf{e}) \mid \overline{n} \\
op & = & + \mid - \\
\tau & = & \iota \mid \tau \rightarrow \tau \\
\mathbf{x} & = & \text{ML variables [distinct from Scheme variables]} \\
\mathbf{E} & = & [\,]_M \mid (\mathbf{E}\ \mathbf{e}) \mid (\mathbf{v}\ \mathbf{E}) \mid (op\ \mathbf{E}\ \mathbf{e}) \mid (op\ \mathbf{v}\ \mathbf{E}) \mid (\texttt{if0}\ \mathbf{E}\ \mathbf{e}\ \mathbf{e})
\end{array}$$

$$\frac{}{\Gamma, \mathbf{x} : \tau \vdash_M \mathbf{x} : \tau} \quad \frac{\Gamma, \mathbf{x} : \tau_1 \vdash_M \mathbf{e} : \tau_2}{\Gamma \vdash_M (\lambda \mathbf{x} : \tau_1 .\ \mathbf{e}) : \tau_1 \rightarrow \tau_2} \quad \frac{}{\Gamma \vdash_M \overline{n} : \iota}$$

$$\frac{\Gamma \vdash_M \mathbf{e}_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_M \mathbf{e}_2 : \tau_1}{\Gamma \vdash_M (\mathbf{e}_1\ \mathbf{e}_2) : \tau_2} \quad \frac{\Gamma \vdash_M \mathbf{e}_1 : \iota \quad \Gamma \vdash_M \mathbf{e}_2 : \iota}{\Gamma \vdash_M (op\ \mathbf{e}_1\ \mathbf{e}_2) : \iota}$$

$$\frac{\Gamma \vdash_M \mathbf{e}_1 : \iota \quad \Gamma \vdash_M \mathbf{e}_2 : \tau \quad \Gamma \vdash_M \mathbf{e}_3 : \tau}{\Gamma \vdash_M (\texttt{if0}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3) : \tau}$$

$$\begin{array}{lcl}
\mathscr{E}[((\lambda \mathbf{x} : \tau .\ \mathbf{e})\ \mathbf{v})]_M & \rightarrow & \mathscr{E}[\mathbf{e}[\mathbf{x} \,/\, \mathbf{v}]] \\
\mathscr{E}[(+\ \overline{n_1}\ \overline{n_2})]_M & \rightarrow & \mathscr{E}[\overline{n_1 + n_2}] \\
\mathscr{E}[(-\ \overline{n_1}\ \overline{n_2})]_M & \rightarrow & \mathscr{E}[\overline{max(n_1 - n_2, 0)}] \\
\mathscr{E}[(\texttt{if0}\ \overline{0}\ \mathbf{e}_1\ \mathbf{e}_2)]_M & \rightarrow & \mathscr{E}[\mathbf{e}_1] \\
\mathscr{E}[(\texttt{if0}\ \overline{n}\ \mathbf{e}_1\ \mathbf{e}_2)]_M & \rightarrow & \mathscr{E}[\mathbf{e}_2]\ (\text{where } n \neq 0)
\end{array}$$

Fig. 1. Core calculus for ML, primed for interoperability

To make the preparation more concrete, as we explain our presentation of the core calculi we also simultaneously develop our first interoperation calculus, which we call the lump embedding. In the lump embedding, ML can call Scheme functions with ML values as arguments and receive Scheme values as results. However, ML sees Scheme values as opaque lumps that cannot be used directly, only returned to Scheme; likewise ML values are opaque lumps to Scheme. For instance, we allow ML to pass a function to Scheme and then use it again as a function if Scheme returns it; but we do *not* allow Scheme to use that same value as a function directly or vice versa.

The lump embedding is a conveniently simple example, but it is worth attention for other reasons as well. First, it represents a particularly easy-to-implement useful multi-language system, achievable more or less automatically for any pair of programming languages so long as both languages have some notion of expressions that yield values. Second, it corresponds to real multi-language systems that can be found "in the wild": many foreign function interfaces give C programs access to native values as pointers that C can only return to the host language. For instance this is how stable pointers in the Haskell foreign function interface behave [Chakravarty 2002]. Third, even this embedding can add expressive power to a pair of programming languages, as we show in section 4.

Where possible, we have typeset all of the fragments of our ML language (and in particular the nonterminals) using a **bold font with serifs**, and all the fragments of our Scheme language with a light sans-serif font. For instance, **e** means the ML expression nonterminal and e means the Scheme expression nonterminal. These distinctions are meaningful, and throughout this paper we use them implicitly. We have not generally given language terminals this treatment, because in our judgment it makes things less rather than more

$$
\begin{array}{lcl}
\mathsf{e} & = & \mathsf{v} \mid (\mathsf{e}\ \mathsf{e}) \mid \mathsf{x} \mid (op\ \mathsf{e}\ \mathsf{e}) \mid (\texttt{if0}\ \mathsf{e}\ \mathsf{e}\ \mathsf{e}) \mid (pr\ \mathsf{e}) \mid (\texttt{wrong}\ str) \\
\mathsf{v} & = & (\lambda \mathsf{x}.\ \mathsf{e}) \mid \overline{n} \\
op & = & + \mid - \\
pr & = & \texttt{proc?} \mid \texttt{nat?} \\
\mathsf{x} & = & \text{Scheme variables [distinct from ML variables]} \\
\mathsf{E} & = & [\,]_S \mid (\mathsf{E}\ \mathsf{e}) \mid (\mathsf{v}\ \mathsf{E}) \mid (op\ \mathsf{E}\ \mathsf{e}) \mid (op\ \mathsf{v}\ \mathsf{E}) \mid (\texttt{if0}\ \mathsf{E}\ \mathsf{e}\ \mathsf{e}) \mid (pr\ \mathsf{E})
\end{array}
$$

$$
\frac{}{\Gamma, \mathsf{x} : \mathbf{TST} \vdash_S \mathsf{x} : \mathbf{TST}} \quad \frac{\Gamma, \mathsf{x} : \mathbf{TST} \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_S (\lambda \mathsf{x}.\ \mathsf{e}) : \mathbf{TST}} \quad \cdots
$$

$$
\begin{array}{lcl}
\mathscr{E}[((\lambda \mathsf{x}.\ \mathsf{e})\ \mathsf{v})]_S & \rightarrow & \mathscr{E}[\mathsf{e}[\mathsf{x} / \mathsf{v}]] \\
\mathscr{E}[(\mathsf{v}_1\ \mathsf{v}_2)]_S & \rightarrow & \mathscr{E}[\texttt{wrong}\ \text{“non-procedure”}] \\
 & & \qquad \text{(where } \mathsf{v}_1 \neq \lambda \mathsf{x}.\mathsf{e}) \\
\mathscr{E}[(+\ \overline{n_1}\ \overline{n_2})]_S & \rightarrow & \mathscr{E}[\overline{n_1 + n_2}] \\
\mathscr{E}[(-\ \overline{n_1}\ \overline{n_2})]_S & \rightarrow & \mathscr{E}[\overline{max(n_1 - n_2, 0)}] \\
\mathscr{E}[(op\ \mathsf{v}_1\ \mathsf{v}_2)]_S & \rightarrow & \mathscr{E}[\texttt{wrong}\ \text{“non-number”}] \\
 & & \qquad \text{(where } \mathsf{v}_1 \neq \overline{n} \text{ or } \mathsf{v}_2 \neq \overline{n}) \\
\mathscr{E}[(\texttt{if0}\ \overline{0}\ \mathsf{e}_1\ \mathsf{e}_2)]_S & \rightarrow & \mathscr{E}[\mathsf{e}_1] \\
\mathscr{E}[(\texttt{if0}\ \mathsf{v}\ \mathsf{e}_1\ \mathsf{e}_2)]_S & \rightarrow & \mathscr{E}[\mathsf{e}_2] \text{ (where } \mathsf{v} \neq \overline{0}) \\
\mathscr{E}[(\texttt{proc?}\ (\lambda \mathsf{x}.\ \mathsf{e}))]_S & \rightarrow & \mathscr{E}[\overline{0}] \\
\mathscr{E}[(\texttt{proc?}\ \mathsf{v})]_S & \rightarrow & \mathscr{E}[\overline{1}] \text{ (where } \mathsf{v} \neq (\lambda \mathsf{x}.\mathsf{e}) \text{ for any } \mathsf{x}, \mathsf{e}) \\
\mathscr{E}[(\texttt{nat?}\ \overline{n})]_S & \rightarrow & \mathscr{E}[\overline{0}] \\
\mathscr{E}[(\texttt{nat?}\ \mathsf{v})]_S & \rightarrow & \mathscr{E}[\overline{1}] \text{ (where } \mathsf{v} \neq \overline{n} \text{ for any } n) \\
\mathscr{E}[(\texttt{wrong}\ str)]_S & \rightarrow & \textbf{Error}\text{: } str
\end{array}
$$

Fig. 2. Core calculus for Scheme, primed for interoperability

clear. Occasionally we use a subscript instead of a font distinction in cases where the font difference would be too subtle.

Figure 3 summarizes the extensions to figures 1 and 2 to support the lump embedding, and the next four subsections describe its syntax, type system, and operational semantics.

2.1 Syntax

The syntaxes of the two languages we use as our starting point are shown in figures 1 and 2. On the ML side, we have taken the explicitly-typed lambda calculus and added numbers (where $\overline{n}$ indicates the syntactic term representing the number n) and a few built-in primitives, including an `if0` form. On the Scheme side, we have taken an untyped lambda calculus and added the same extensions plus some useful predicates and a `wrong` form that takes a literal error message string.

To extend that base syntax with the ability to interoperate, we need a way of writing down a program that contains both ML and Scheme code. While real systems typically do this by somehow allowing the programmer to call predefined foreign-language functions from shared libraries, we would rather keep our system more abstract than that. Instead, we introduce syntactic boundaries between ML and Scheme, cross-language casts that indicate a switch of languages. We will use boundaries like these in all the systems we present.

Concretely we represent a boundary as a new kind of expression in each language. To

$$
\begin{array}{lll}
\mathbf{e} = \cdots \mid ({}^{\tau}\!MS\, \mathsf{e}) & \mathsf{e} = \cdots \mid (SM^{\tau}\, \mathbf{e}) & \\
\mathbf{v} = \cdots \mid ({}^{\mathbf{L}}\!MS\, \mathsf{v}) & \mathsf{v} = \cdots \mid (SM^{\tau}\, \mathbf{v}) \text{ where } \tau \neq \mathbf{L} & \\
\tau = \cdots \mid \mathbf{L} & & \\
\mathbf{E} = \cdots \mid ({}^{\tau}\!MS\, \mathsf{E}) & \mathsf{E} = \cdots \mid (SM^{\tau}\, \mathbf{E}) & \\
 & \mathscr{E} = \mathbf{E} &
\end{array}
$$

$$
\frac{\Gamma \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_M ({}^{\tau}\!MS\, \mathsf{e}) : \tau} \qquad \frac{\Gamma \vdash_M \mathbf{e} : \tau}{\Gamma \vdash_S (SM^{\tau}\, \mathbf{e}) : \mathbf{TST}}
$$

$$
\begin{array}{rcl}
\mathscr{E}[({}^{\tau}\!MS (SM^{\tau}\mathbf{v}))]_M & \rightarrow & \mathscr{E}[\mathbf{v}] \\
\mathscr{E}[({}^{\tau}\!MS\, \mathsf{v})]_M & \rightarrow & \mathscr{E}[{}^{\tau}\!MS\, (\texttt{wrong} \text{ “Bad value”})] \\
 & & \text{if } \tau \neq \mathbf{L} \text{ and } \mathsf{v} \neq (SM^{\tau}\mathbf{v}) \text{ for any } \mathbf{v} \\
\mathscr{E}[(SM^{\mathbf{L}}\, ({}^{\mathbf{L}}\!MS\, \mathsf{v}))]_S & \rightarrow & \mathscr{E}[\mathsf{v}]
\end{array}
$$

Fig. 3. Extensions to figures 1 and 2 to form the lump embedding

---

the ML grammar, we add

$$\mathbf{e} = \cdots \mid ({}^{\tau}\!MS\ \mathsf{e})$$

(think of MS as “ML-outside, Scheme-inside”) and to the Scheme grammar we add

$$\mathsf{e} = \cdots \mid (SM^{\tau}\ \mathbf{e})$$

(think of SM as “Scheme outside, ML inside”) where the τ on the ML side of each boundary indicates the type ML will consider its side of the boundary to be. These grammar extensions are in figure 3.[1]

2.2 Typing rules

In figure 1, ML has a standard type system with the typing judgment $\vdash_M$ where numbers have type ι and functions have arrow types. Similarly, in figure 2, Scheme has a trivial type system with the judgment $\vdash_S$ that gives all closed terms the type **TST** (“the Scheme type”). (We have omitted several typing rules from the Scheme side; every Scheme expression has a rule that gives it type **TST** if its subparts have type **TST**.)

In our lump embedding extension, we add a new type **L** (for “lump”) to ML and we add a new rule to each typing judgment corresponding to the new syntactic forms. The new Scheme judgment says that an $(SM^{\tau}\ \mathbf{e})$ boundary is well-typed if ML's type system proves $\mathbf{e}$ has type τ — that is, a Scheme program type-checks if it is closed and all its ML subterms have the types the program claims they have. The new ML judgment says that $({}^{\tau}\!MS\ \mathsf{e})$ has type τ if e type-checks under Scheme's typing system. In both cases, τ can be any type, not just **L** as one might expect. If $\tau = \mathbf{L}$ we are sending a native Scheme value

[1]Our choice of representation allows a program term to represent finer-grained interoperability than real multi-language systems typically let programmers write down (although intermediate states in a computation can typically reach all of the states our boundaries can express). We could impose restrictions on initial program terms to make them correspond more directly to programs a programmer could type in, but this would just encumber us without fundamentally changing the system.

across the boundary (which will be a lump in ML); if $\tau \neq \mathbf{L}$ we are sending an ML value across the boundary (which will be a lump in Scheme).

In these typing judgments, Scheme rules use the same type environment that ML does. We do that to allow ML expressions to refer to variables bound by ML (or vice versa) even if they are separated by a sequence of boundaries (this is necessary to give types to functions that use foreign arguments; we give an example of such a function in section 2.4). We assume that Scheme and ML variables are drawn from disjoint sets.

2.3 Evaluation contexts

We use Felleisen-and-Hieb-style context-sensitive reduction semantics to specify the operational semantics for our systems. In figure 1, we define an evaluation context for ML ($\mathbf{E}$) and one for Scheme (E), and we use a third, unspecified evaluation context ($\mathscr{E}$) for the definitions of the rewriting rules. In a single-language setting, we would instantiate $\mathscr{E}$ to $\mathbf{E}$ for ML and to E for Scheme. In a multi-language setting, though, we must pick just one or the other, and the choice determines in which language programs begin and what kind of values they produce. Since our programs begin and end with ML, we have $\mathscr{E} = \mathbf{E}$ in figure 3.

To allow evaluation of Scheme expressions embedded in ML expressions (and vice versa), we must let ML evaluation contexts contain Scheme evaluation contexts and vice versa. We do that by adding new context productions for boundaries:

$$\begin{aligned} \mathbf{E} &= \cdots \mid ({}^{\tau}MS\ \mathsf{E}) \\ \mathsf{E} &= \cdots \mid (SM^{\tau}\ \mathbf{E}) \end{aligned}$$

Evaluation of an ML boundary proceeds by reducing its embedded Scheme expression in a Scheme context, and evaluation of a Scheme boundary proceeds by reducing its embedded ML expression in an ML context.

2.4 Reduction rules

The reduction rules in the core model are all reasonably standard, with a few peculiarities. On the ML side, we allow subtraction of two numbers but floor all results at zero; this is because we only allow natural numbers in the language. The Scheme side has a bit more going on dynamically. Since Scheme has only a trivial type system, we add dynamic checks to every appropriate form that reduce to `wrong` if they receive an illegal value. The reduction rule for `wrong` itself discards the entire program context and aborts the program with an error message.

To combine the languages, we might hope to just merge their sets of reductions together. That does not quite work. For instance, the ML term $((\lambda \mathbf{x} : \iota . \mathbf{x})\ (\overline{1}\ \overline{1}))$ would reduce to (`wrong` "Non-procedure"), rather than getting stuck, since Scheme's reduction for a non-procedure application would apply to $(\overline{1}\ \overline{1})$ even though it is an ML fragment. To remedy this, we extend Felleisen and Hieb's context-sensitive rewriting framework by differentiating holes generated by ML's evaluation contexts, named $[\,]_M$ holes, and holes generated by Scheme's evaluation contexts, named $[\,]_S$ holes. Scheme's rewriting rules only apply to evaluation contexts with $[\,]_S$ holes, (and similarly for ML); we indicate that restriction by writing an S (or M) subscript next to the bracket on each evaluation context on the left-hand side of Scheme's (or ML's) rewriting rules.[2]

---

[2]This is the same extension we made in earlier work to model Scheme's multiple values [Matthews and Findler

With this extension, the example above decomposes into the context $((\lambda \mathbf{x} : \iota.\mathbf{x})[\,]_M)$ but not $((\lambda \mathbf{x} : \iota.\mathbf{x})[\,]_S)$. Because the hole is named $[\,]_M$ and the Scheme application rule only applies to holes named $[\,]_S$, it does not apply and the term remains stuck. However, if instead we had written $((\lambda \mathbf{x} : \iota.\mathbf{x})\ ({}^{\iota}\mathit{MS}\ (\overline{1}\ \overline{1})))$ it would decompose into an ML context with a Scheme hole: $((\lambda \mathbf{x} : \iota.\mathbf{x})\ ({}^{\iota}\mathit{MS}\ [\,]_S))$ with the erroneous application $(\overline{1}\ \overline{1})$ inside. Since that redex is in a Scheme hole, the term reduces to an error.

To finish the lump embedding, all that remains is to specify the reduction rules and values for the boundaries between languages. If an *MS* boundary of type **L** has a Scheme value inside, we consider the entire expression to be an ML value. Similarly, when an *SM* boundary of a non-lump type has an ML value inside, we consider the whole expression to be a Scheme value. In contrast, if an *MS* boundary with a non-lump type has a Scheme value in it, or when an *SM* boundary of a lump type has an ML value inside, we expect that inner value to be the foreign representation of a native value, and our reduction rules should turn it back into a native value. We do that by cancelling matching boundaries, shown in figure 3's reduction rules.

ML's typing rules guarantee that values that appear inside $(\mathit{SM}^{\mathbf{L}}\ \mathbf{v})$ expressions will in fact be lump values, so the $\mathit{SM}^{\mathbf{L}}$ reduction can safely restrict its attention to values of the correct form. On the other hand, Scheme offers no guarantees, so the rule for eliminating an ${}^{\tau}\mathit{MS}$ boundary must apply whenever the Scheme expression is a value at all.

These additions give us a precise account of the behavior for lump embedding we described at the beginning of this section. To get a sense of how the calculus works, consider this example:

$$
\begin{array}{ll}
 & ((\lambda \mathbf{fa} : \mathbf{L} \rightarrow \mathbf{L} \rightarrow \mathbf{L}. \\
 & \quad ((\mathbf{fa}\ ({}^{\mathbf{L}}\mathit{MS}\ (\lambda \mathsf{x}.\ (+\ \mathsf{x}\ \overline{1})))) \\
 & \quad\quad ({}^{\mathbf{L}}\mathit{MS}\ \overline{3}))) \\
 & \ (\lambda \mathbf{f} : \mathbf{L}.\ \lambda \mathbf{x} : \mathbf{L}.\ ({}^{\mathbf{L}}\mathit{MS}\ ((\mathit{SM}^{\mathbf{L}}\ \mathbf{f})\ (\mathit{SM}^{\mathbf{L}}\ \mathbf{x}))))) \\
\rightarrow & (((\lambda \mathbf{f} : \mathbf{L}.\ \lambda \mathbf{x} : \mathbf{L}.\ ({}^{\mathbf{L}}\mathit{MS}\ ((\mathit{SM}^{\mathbf{L}}\ \mathbf{f})\ (\mathit{SM}^{\mathbf{L}}\ \mathbf{x})))) \\
 & ({}^{\mathbf{L}}\mathit{MS}\ (\lambda \mathsf{x}.\ (+\ \mathsf{x}\ \overline{1})))) \\
 & ({}^{\mathbf{L}}\mathit{MS}\ \overline{3})) \\
\rightarrow^2 & {}^{\mathbf{L}}\mathit{MS}\ ((\mathit{SM}^{\mathbf{L}}({}^{\mathbf{L}}\mathit{MS}(\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))))(\mathit{SM}^{\mathbf{L}}\ ({}^{\mathbf{L}}\mathit{MS}\ \overline{3})))) \\
\rightarrow & {}^{\mathbf{L}}\mathit{MS}\ ((\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))\ (\mathit{SM}^{\mathbf{L}}\ ({}^{\mathbf{L}}\mathit{MS}\ \overline{3})))) \\
\rightarrow & {}^{\mathbf{L}}\mathit{MS}\ ((\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))\ \overline{3}) \\
\rightarrow & {}^{\mathbf{L}}\mathit{MS}\ (+\ \overline{3}\ \overline{1}) \\
\rightarrow & {}^{\mathbf{L}}\mathit{MS}\ \overline{4}
\end{array}
$$

In the initial term of this reduction sequence, we use a "left-left-lambda" encoding of **let** to bind the name **fa** (for "foreign-apply") to a curried ML function that takes two foreign values, applies the first to the second in Scheme, and returns the result as a foreign value. The program uses **fa** to apply a Scheme add-one function to the Scheme number $\overline{3}$. In two computation steps, we plug in the Scheme function and its argument into the body of **fa**. In that term there are two instances of $(\mathit{SM}^{\mathbf{L}}\ ({}^{\mathbf{L}}\mathit{MS}\ \mathsf{v}))$ subterms, both of which are cancelled in the next two computation steps. After those cancellations, the term is just a Scheme application of the add-one function to $\overline{3}$, which reduces to the Scheme value $\overline{4}$.

2005]. We could alternately introduce two different notations for Scheme and ML application, but we find it inelegant; it suggests that a multi-language implementation would decide how to evaluate each term by inspecting it, when real systems decide how to evaluate a term based on the language in which the term is being evaluated — *i.e.*, its context.

If we try to apply the ML add-one function to the Scheme number $\overline{3}$ instead (and adjust **fa**'s type to make that possible), we will end up with an intermediate term like this:

$$\begin{array}{ll} & ({}^{\iota}MS\ ((SM^{\iota\rightarrow\iota}\,(\lambda\mathbf{x}:\iota.(+\ \mathbf{x}\ \overline{1})))\ \overline{3})) \\ \rightarrow & ({}^{\iota}MS\ (\texttt{wrong}\ \text{“non-procedure”})) \\ \rightarrow & \textbf{Error}\text{: non-procedure} \end{array}$$

Here, Scheme tries to apply the ML function directly, which leads to a runtime error since it is illegal for Scheme to apply ML functions. We cannot make the analogous mistake and try to apply a Scheme function in ML, since terms like $({}^{\mathbf{L}}MS\ (\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1})))\ \overline{3}$ are ill-typed.

The formulation of the lump embedding in figure 3 allows us to prove type soundness using the standard technique of preservation and progress.

THEOREM 1. *If $\Gamma \vdash_M \boldsymbol{e} : \tau$, then either $\boldsymbol{e} \rightarrow^* \boldsymbol{v}$, $\boldsymbol{e} \rightarrow^*$ **Error**: str, or $\boldsymbol{e}$ diverges.*

Before we can proceed to establishing preservation and progress, we need a few technical lemmas, all of which are standard: uniqueness of types, inversion, and replacement. The proofs of the first two are entirely standard, but the replacement lemma requires a slight generalization from its presentation in Wright and Felleisen [1994].

LEMMA 2. *If $\Gamma \vdash_M \boldsymbol{e} : \tau$, then:*

—*If $\boldsymbol{e}'$ is a subterm of $\boldsymbol{e}$ and $\Gamma' \vdash_M \boldsymbol{e}' : \tau'$, then for all terms $\boldsymbol{e}''$ where $\Gamma' \vdash_M \boldsymbol{e}'' : \tau'$, $\Gamma \vdash_M \boldsymbol{e}[\boldsymbol{e}'/\boldsymbol{e}''] : \tau$.*

—*If e' is a subterm of $\boldsymbol{e}$ and $\Gamma' \vdash_S \mathsf{e}' : \boldsymbol{TST}$, then for all terms e'' where $\Gamma' \vdash_S \mathsf{e}'' : \boldsymbol{TST}$, $\Gamma \vdash_M \boldsymbol{e}[\mathsf{e}'/\mathsf{e}''] : \tau$.*

Given these we can show preservation:

LEMMA 3.
If $\vdash_M \boldsymbol{e} : \tau$ and $\boldsymbol{e} \rightarrow \boldsymbol{e}'$, then $\vdash_M \boldsymbol{e}' : \tau$.

PROOF. By cases on the reduction $\mathbf{e} \rightarrow \mathbf{e}'$. With the above lemmas, the cases are entirely standard except for the boundary reductions. We present only those.

Case $\mathscr{E}[{}^{\tau'}MS\ (SM^{\tau'}\ \mathbf{v})] \rightarrow \mathscr{E}[\mathbf{v}]$

By premise and uniqueness of types, we have that $\Gamma \vdash_M ({}^{\tau'}MS\ (SM^{\tau'}\ \mathbf{v})) : \tau'$. By inversion we have that $\Gamma \vdash_S (SM^{\tau'}\ \mathbf{v}) : \mathbf{TST}$, and by inversion again we have that $\Gamma \vdash_M \mathbf{v} : \tau'$. Thus by replacement, $\mathscr{E}[\mathbf{v}]$ has type τ.

Case $\mathscr{E}[({}^{\tau'}MS\ \mathsf{v})]_M \rightarrow \mathscr{E}[({}^{\tau'}MS\ (\texttt{wrong}\ \text{“Bad value”}))]$ where $\mathsf{v} \neq (SM^{\tau'}\mathbf{v})$

By premise and uniqueness of types, $\Gamma \vdash_M ({}^{\tau'}MS\ \mathsf{v}) : \tau'$. A calculation shows that

$$({}^{\tau'}MS\ (\texttt{wrong}\ \text{“Bad value”})$$

also has type τ' in Γ, so by replacement

$$\mathscr{E}[({}^{\tau'}MS\ (\texttt{wrong}\ \text{“Bad value”}))]$$

has type τ.

Case $\mathscr{E}[(SM^{\mathbf{L}}\ ({}^{\mathbf{L}}MS\,\mathsf{v}))]_S \rightarrow \mathscr{E}[\mathsf{v}]$

By premise and uniqueness of types, we have that $\Gamma \vdash_S (SM^{\mathbf{L}}\ ({}^{\mathbf{L}}MS\,\mathsf{v})) : \mathbf{TST}$. By inversion we have that $\Gamma \vdash_M ({}^{\mathbf{L}}MS\,\mathsf{v}) : \mathbf{L}$, and by inversion again we have that $\Gamma \vdash_S \mathsf{v} : \mathbf{TST}$. Thus by replacement, $\mathscr{E}[\mathsf{v}]$ has type τ. □

To prove progress, we have to strengthen the statement of the lemma; we need to be able to use induction on both ML's and Scheme's typing judgments, and we need to prove the statement for all evaluation contexts because we have no notion of evaluating Scheme programs in empty contexts.

LEMMA 4. *For all ML expressions* $\boldsymbol{e}$ *and Scheme expressions* e*, both of the following hold:*

(1) If $\vdash_M \boldsymbol{e} : \tau$*, then either* $\boldsymbol{e}$ *is an ML value or for all top-level evaluation contexts* $\mathscr{E}[\ ]_M$ *either there exists an* $\boldsymbol{e}'$ *such that* $\mathscr{E}[\boldsymbol{e}] \rightarrow \boldsymbol{e}'$ *or* $\mathscr{E}[\boldsymbol{e}] \rightarrow$ **Error***: str for some error message str.*

(2) If $\vdash_S \mathsf{e} : \boldsymbol{TST}$*, then either* e *is a Scheme value or for all top-level evaluation contexts* $\mathscr{E}[\ ]_S$ *either there exists an* $\boldsymbol{e}'$ *such that* $\mathscr{E}[\mathsf{e}] \rightarrow \boldsymbol{e}'$ *or* $\mathscr{E}[\mathsf{e}] \rightarrow$ **Error***: str for some error message str.*

PROOF. By simultaneous induction on the structure of the typing derivation. Cases generally make use of the the fact that we can compose contexts where the hole in the outer context corresponds to the outermost language of the inner context, but are otherwise straightforward. We show the most interesting case.

Case $\dfrac{\vdash_S \mathsf{e} : \mathbf{TST}}{\vdash_M ({}^{\tau}\!MS\ \mathsf{e}) : \tau}$:

We must show that either $({}^{\tau}\!MS\ \mathsf{e})$ is a value or that for an arbitrary top-level evaluation context $\mathscr{E}[\ ]_M$, $\mathscr{E}[({}^{\tau}\!MS\ \mathsf{e})]$ reduces. If e is a Scheme value, then depending on τ either the entire expression is a value or one of the two reduction rules for ${}^{\tau}\!MS$ boundaries directly applies. If e is not a Scheme value, then by induction on (2) we have that for all evaluation contexts with Scheme holes, e can reduce. The context $\mathscr{E}[(SM^{\tau}\ [\ ]_S)]$ is an ML evaluation context with a Scheme hole; thus $\mathscr{E}[(SM^{\tau}\ \mathsf{e})]$ reduces as required. □

With these two lemmas established, theorem 1 is nearly immediate:

PROOF. Combination of lemmas 3 and 4. □

We should point out that because of the way we have combined the two languages, type soundness entails that both languages are type-sound with respect to *their own* type systems — in other words, both single-language type soundness proofs are special cases. So theorem 1 makes a stronger claim than the claim that an interpreter written in ML automatically forms a "type-sound" embedding because all ML programs must well-typed, since the latter only establishes type-soundness with respect to one of the two involved languages.

3. THE NATURAL EMBEDDING

The lump embedding is a useful starting point, but realistic multi-language systems offer richer cross-language communication primitives. A more useful way to pass values between our Scheme and ML models, suggested many times in the literature (*e.g.*, [Benton 2005; Ramsey 2003; Ohori and Kato 1993]) is to use a type-directed strategy to convert ML numbers to equivalent Scheme numbers and ML functions to equivalent Scheme functions (for some suitable notion of equivalence) and vice versa. We call this the natural embedding.

We can quickly get at the essence of this strategy by extending the core calculi from figures 1 and 2, just as we did before to form the lump embedding. Again, we add new syntax reduction rules to figures 1 and 2 that pertain to boundaries. In this section we will add ${}^{\tau}MS_N$ and SM_N^{τ} boundaries, adding the subscript N (for "natural") only to distinguish these new boundaries from lump boundaries from section 2.

We will assume we can translate numbers from one language to the other, and give reduction rules for boundary-crossing numbers based on that assumption:

$$\begin{aligned}\mathscr{E}[(SM_N^{\iota}\,\overline{n})]_S &\rightarrow \mathscr{E}[\overline{n}]\\ \mathscr{E}[({}^{\iota}MS_N\,\overline{n})]_M &\rightarrow \mathscr{E}[\overline{n}]\end{aligned}$$

In some multi-language settings, differing byte representations might complicate this task. Worse, some languages may have more expansive notions of numbers than others — for instance, the actual Scheme language treats many different kinds of numbers uniformly (e.g., integers, floating-point numbers, arbitrary precision rationals, and complex numbers), whereas in the actual ML language these numbers are distinguished. More sophisticated versions of the above rules would address these problems.

We must be more careful with procedures, though. We cannot get away with just moving the text of a Scheme procedure into ML or vice versa; aside from the obvious problem that their grammars generate different sets of terms, ML does not even have a reasonable equivalent for every Scheme procedure. Instead, for this embedding we represent a foreign procedure with a proxy. We represent a Scheme procedure in ML at type $\tau_1 \rightarrow \tau_2$ by a new procedure that takes an argument of type τ_1, converts it to a Scheme equivalent, runs the original Scheme procedure on that value, and then converts the result back to ML at type τ_2. For example, $({}^{\tau_1\rightarrow\tau_2}MS_N\,\lambda \mathsf{x}.\mathsf{e})$ becomes $(\lambda\mathbf{x}:\tau_1.\,{}^{\tau_2}MS_N\,((\lambda \mathsf{x}.\mathsf{e})\,(SM_N^{\tau_1}\,\mathbf{x})))$ and vice versa for Scheme to ML. Note that the boundary that converts the argument is an $SM_N^{\tau_1}$ boundary, not an ${}^{\tau_1}MS_N$ boundary. The direction of conversion reverses for function arguments.

This would complete the natural embedding, but for one important problem: the system has stuck states, since a boundary might receive a value of an inappropriate shape. Stuck states violate type-soundness, and in an implementation they might correspond to segmentation faults or other undesirable behavior. As it turns out, higher-order contracts [Findler and Felleisen 2002; Findler and Blume 2006] arise naturally as the checks required to protect against these stuck states. We show that in the next three sections: first we add dynamic guards directly to boundaries to provide a baseline, then show how to separate them, and finally observe that these separated guards are precisely contracts between ML and Scheme, and that since ML statically guarantees that it always lives up to its contracts, we can eliminate their associated dynamic checks.

3.1 A simple method for adding error transitions

In the lump embedding, we can always make a single, immediate check that would tell us if the value Scheme provided ML was consistent with the type ML ascribed to it. This is no longer possible, since we cannot know if a Scheme function always produces a value that can be converted to the appropriate type. Still, we can perform an optimistic check that preserves ML's type safety: when a Scheme value crosses a boundary, we only check its first-order qualities — *i.e.*, whether it is a number or a procedure. If it has the appropriate first-order behavior, we assume the type ascription is correct and perform the conversion, distributing into domain and range conversions as before. If it does not, we immediately signal an error. This method works to catch all errors that would lead to stuck states; even though it only checks first-order properties, the program can only reach a stuck state if

$$\mathbf{e} = \cdots \mid (MSG^{\tau}\ \mathsf{e})$$
$$\mathsf{e} = \cdots \mid (GSM^{\tau}\ \mathbf{e})$$

$$\mathbf{E} = \cdots \mid (MSG^{\tau}\ \mathsf{E})$$
$$\mathsf{E} = \cdots \mid (GSM^{\tau}\ \mathbf{E})$$

$$\frac{\Gamma \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_M (MSG^{\tau}\ \mathsf{e}) : \tau} \quad \frac{\Gamma \vdash_M \mathbf{e} : \tau}{\Gamma \vdash_S (GSM^{\tau}\ \mathbf{e}) : \mathbf{TST}}$$

$$\begin{aligned}
\mathscr{E}[(GSM^{\iota}\ \overline{n})]_S &\rightarrow \mathscr{E}[\overline{n}] \\
\mathscr{E}[(GSM^{\tau_1 \rightarrow \tau_2}\ \mathbf{v})]_S &\rightarrow \mathscr{E}[(\lambda \mathsf{x}.\,(GSM^{\tau_2}(\mathbf{v}\,(MSG^{\tau_1}\ \mathsf{x}))))] \\
\mathscr{E}[(MSG^{\iota}\ \overline{n})]_M &\rightarrow \mathscr{E}[\overline{n}] \\
\mathscr{E}[(MSG^{\iota}\ \mathsf{v})]_M &\rightarrow \mathscr{E}[MSG^{\iota}\ (\texttt{wrong}\ \text{“Non-number”})] \\
&\qquad \mathsf{v} \neq \overline{n} \text{ for any } n \\
\mathscr{E}[(MSG^{\tau_1 \rightarrow \tau_2}(\lambda \mathsf{x}.\mathsf{e}))]_M &\rightarrow \mathscr{E}[(\lambda \mathbf{x} : \tau_1.(MSG^{\tau_2}\ ((\lambda \mathsf{x}.\mathsf{e})\,(GSM^{\tau_1}\ \mathbf{x}))))] \\
\mathscr{E}[(MSG^{\tau_1 \rightarrow \tau_2}\ \mathsf{v})]_M &\rightarrow \mathscr{E}[(MSG^{\tau_1 \rightarrow \tau_2}\ (\texttt{wrong}\ \text{“Non-procedure”}))] \\
&\qquad \mathsf{v} \neq \lambda \mathsf{x}.\mathsf{e} \text{ for any } \mathsf{x}, \mathsf{e}
\end{aligned}$$

Fig. 4. Extensions to figure 1 and 2 to form the simple natural embedding

a value is used in such a way that it does not have the appropriate first-order properties anyway.

To model this method, rather than adding the SM_N^{τ} and ${}^{\tau}MS_N$ constructs to our core languages from figures 1 and 2, we instead add "guarded" versions GSM^{τ} and MSG^{τ} shown in figure 4. These rules translate values in the same way that SM_N^{τ} and ${}^{\tau}MS_N$ did before, but also detect concrete, first-order witnesses to an invalid type ascription (*i.e.*, numbers for procedures or procedures for numbers) and abort the program if one is found. We call the language formed by these rules the simple natural embedding. We give its rules in figure 4, but it may be easier to understand how it works by reconsidering the examples we gave at the end of section 2. Each of those examples, modified to use the natural embedding rather than the lump embedding, successfully evaluates to the ML number $\overline{4}$. Here is the reduction sequence produced by the last of those examples, which was ill-typed before:

$$\begin{aligned}
&((MSG^{\iota \rightarrow \iota}\ (\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1})))\ \overline{3}) \\
\rightarrow\ &((\lambda \mathbf{x}' : \iota.MSG^{\iota}\ ((\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))\ (GSM^{\iota}\ \mathbf{x}')))\ \overline{3}) \\
\rightarrow\ &(MSG^{\iota}\ ((\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))\ (GSM^{\iota}\ \overline{3}))) \\
\rightarrow\ &(MSG^{\iota}\ ((\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))\ \overline{3})) \\
\rightarrow\ &(MSG^{\iota}\ (+\ \overline{3}\ \overline{1})) \\
\rightarrow\ &(MSG^{\iota}\ \overline{4}) \\
\rightarrow\ &\overline{4}
\end{aligned}$$

ML converts the Scheme add-one function to an ML function with type $\iota \rightarrow \iota$ by replacing it with a function that converts its argument to a Scheme number, feeds that number to the original Scheme function, and then converts the result back to an ML number. Then it applies this new function to the ML number $\overline{3}$, which gets converted to the Scheme number

$$\mathbf{e} = \cdots \mid ({}^{\tau}MS_N\ \mathsf{e})$$
$$\mathsf{e} = \cdots \mid (\mathscr{G}^{\tau}\ \mathsf{e}) \mid (SM_N^{\tau}\ \mathbf{e})$$

$$\mathbf{E} = \cdots \mid ({}^{\tau}MS_N\ \mathsf{E})$$
$$\mathsf{E} = \cdots \mid (\mathscr{G}^{\tau}\ \mathsf{E}) \mid (SM_N^{\tau}\ \mathbf{E})$$

$$\frac{\Gamma \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_S (\mathscr{G}^{\tau}\ \mathsf{e}) : \mathbf{TST}}$$

$$\frac{\Gamma \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_M ({}^{\tau}MS_N\ \mathsf{e}) : \tau} \quad \frac{\Gamma \vdash_M \mathbf{e} : \tau}{\Gamma \vdash_S (SM_N^{\tau}\ \mathbf{e}) : \tau}$$

$$\begin{array}{rcl}
\mathscr{E}[(SM_N^{\iota}\,\overline{n})]_S & \to & \mathscr{E}[\overline{n}] \\
\mathscr{E}[({}^{\iota}MS_N\,\overline{n})]_M & \to & \mathscr{E}[\overline{n}] \\
\mathscr{E}[(SM_N^{\tau_1\to\tau_2}\,\mathbf{v})]_S & \to & \mathscr{E}[(\lambda\mathsf{x}.(SM_N^{\tau_2}\,(\mathbf{v}\,({}^{\tau_1}MS_N\,\mathsf{x}))))] \\
\mathscr{E}[({}^{\tau_1\to\tau_2}MS_N\,(\lambda\mathsf{x}.\mathsf{e}))]_M & \to & \mathscr{E}[(\lambda\mathbf{x}:\tau_1.({}^{\tau_2}MS_N((\lambda\mathsf{x}.\mathsf{e})\,(SM_N^{\tau_1}\,\mathbf{x}))))] \\
\mathscr{E}[(\mathscr{G}^{\iota}\,\overline{n})]_S & \to & \mathscr{E}[\overline{n}] \\
\mathscr{E}[(\mathscr{G}^{\iota}\mathsf{v})]_S & \to & \mathscr{E}[\texttt{wrong}\text{ “Non-number”}]\ (\mathsf{v} \neq \overline{n} \text{ for any } n) \\
\mathscr{E}[(\mathscr{G}^{\tau_1\to\tau_2}(\lambda\mathsf{x}.\mathsf{e}))]_S & \to & \mathscr{E}[(\lambda\mathsf{x}'.(\mathscr{G}^{\tau_2}((\lambda\mathsf{x}.\mathsf{e})(\mathscr{G}^{\tau_1}\mathsf{x}'))))] \\
\mathscr{E}[(\mathscr{G}^{\tau_1\to\tau_2}\mathsf{v})]_S & \to & \mathscr{E}[\texttt{wrong}\text{ “Non-procedure”}]\ (\mathsf{v} \neq (\lambda\mathsf{x}.\mathsf{e}) \text{ for any } \mathsf{x}, \mathsf{e})
\end{array}$$

Fig. 5. Extensions to figures 1 and 2 to form the separated-guards natural embedding

---

$\overline{3}$, run through the Scheme function, and finally converted back to the ML number $\overline{4}$, which is the program's final answer.

The method works at higher-order types because it applies type conversions recursively. Consider this expression:

$$({}^{(\iota\to\iota)\to\iota}MS_N\ (\lambda\mathsf{f}.\,(\texttt{if0}\ (\mathsf{f}\ \overline{1})\ \overline{2}\ \mathsf{f})))$$

Depending on the behavior of its arguments, the Scheme procedure may or may not produce a number. ML treats it as though it definitely had type $(\iota \to \iota) \to \iota$, and wraps it to the ML value

$$(\lambda\mathbf{x} : \iota \to \iota.({}^{\iota}MS_N\ ((\lambda\mathsf{f}.\ (\texttt{if0}\ (\mathsf{f}\ \overline{1})\ \overline{2}\ \mathsf{f}))\ (SM_N^{\iota\to\iota}\ \mathbf{x}))))$$

Whenever this value is applied to a function, that function is converted to a Scheme value at type $\iota \to \iota$ and the result is converted from Scheme to ML at type ι. Thus conversion in either direction works at a given type if it works in *both* directions at all smaller types.

THEOREM 5. *If* $\vdash_M \mathbf{e} : \tau$*, then either* $\mathbf{e} \to^* \mathbf{v}$*,* $\mathbf{e} \to^* \textbf{Error}: str$*, or* $\mathbf{e}$ *diverges.*

PROOF. By a standard argument along the lines of theorem 1. □

3.2 A refinement: guards

Adding dynamic checks to boundaries themselves is an expedient way to ensure type soundness, but we find it a little troublesome. For one thing, boundaries are necessarily the core of any multi-language system, so they should be as small and simple as possible. For another, coupling the task of value conversion with the conceptually unrelated task of detecting and signalling errors means that changing the method of signalling errors requires modifying the internals of value conversion.

To decouple error-handling from value conversion, we separate the guarded boundaries of the previous subsection into their constituent parts: boundaries and guards. These separated boundaries have the semantics of the ${}^{\tau}MS_N$ and SM_N^{τ} boundaries we introduced at the beginning of this section. Guards will be new expressions of the form $(\mathscr{G}^{\tau}\ \mathsf{e})$ that perform all dynamic checks necessary to ensure that their arguments behave as τ in the sense of the previous subsection. In all initial terms, we will wrap every boundary with an appropriate guard: $({}^{\tau}MS_N\ (\mathscr{G}^{\tau}\ \mathsf{e}))$ instead of $(MSG^{\tau}\ \mathsf{e})$ and $(\mathscr{G}^{\tau}\ (SM^{\tau}\ \mathbf{e}))$ instead of $(GSM^{\tau}\ \mathbf{e})$.

Figure 5 shows the rules for guards. An ι guard applied to a number reduces to that number, and the same guard applied to a procedure aborts the program. A $\tau_1 \rightarrow \tau_2$ guard aborts the program if given a number, and if given a procedure reduces to a new procedure that applies a τ_1 guard to its input, runs the original procedure on that value, and then applies a τ_2 guard to the original procedure's result. This is just like the strategy we use to convert functions in the first place, but it doesn't perform any foreign-language translation by itself; it just distributes the guards in preparation for conversion later on.

The guard distribution rules for functions can move a guard arbitrarily far away from the boundary it protects; if this motion ever gave a value the opportunity to get to a boundary without first being blessed by the appropriate guard, the system would get stuck. We can prove this never happens by defining a combined language that has *both* guarded boundaries GSM^{τ} and MSG^{τ} and unguarded boundaries with separated guards ${}^{\tau}MS_N$, SM_N^{τ}, and $\mathscr{G}^{\tau}$; *i.e.* the language formed by combining figures 1 and 2 with figures 4 and 5. In this combined language, an argument by induction shows that guarded boundaries are observably equivalent to guards combined with unguarded boundaries.

We start by defining an evaluation function for the combined language. For all closed ML terms $\mathbf{e}$ such that $\vdash_M \mathbf{e} : \tau$,

$$\begin{aligned} eval_N(\mathbf{e}) &= \texttt{proc} \text{ if } \mathbf{e} \rightarrow^* \lambda \mathbf{x} : \tau.\mathbf{e} \\ eval_N(\mathbf{e}) &= n \text{ if } \mathbf{e} \rightarrow^* \overline{n} \\ eval_N(\mathbf{e}) &= \textbf{Error}\!: s \text{ if } \mathbf{e} \rightarrow^* \textbf{Error}\!: s \end{aligned}$$

Now we define program contexts $\mathbf{C}$ and C as the compatible closures of $\mathbf{e}$ and e, respectively. Holes in these contexts have names as in section 2. Given those, we can define an operational equivalence relation $\simeq$ in terms of the evaluator and contexts:

$$\begin{aligned} \mathbf{e}_1 \simeq \mathbf{e}_2 &\overset{\text{def}}{\Leftrightarrow} \forall \mathbf{C}[\,]_M.\ eval_N(\mathbf{C}[\mathbf{e}_1]) = eval_N(\mathbf{C}[\mathbf{e}_2]) \\ \mathsf{e}_1 \simeq \mathsf{e}_2 &\overset{\text{def}}{\Leftrightarrow} \forall \mathbf{C}[\,]_S.\ eval_N(\mathbf{C}[\mathsf{e}_1]) = eval_N(\mathbf{C}[\mathsf{e}_2]) \end{aligned}$$

where $\mathbf{C}[\,]$ is a ML program context and the subscripts M and S indicate what type of hole appears in it. We use ML contexts in both cases here because they are the top-level context for programs.

THEOREM 6. *For all Scheme expressions* e *and ML expressions* $\mathbf{e}$*, the following propositions hold:*

$$\begin{aligned} &(1)\ \ (MSG^{\tau}\ \mathsf{e}) \simeq ({}^{\tau}MS_N(\mathscr{G}^{\tau}\ \mathsf{e})) \\ &(2)\ \ (GSM^{\tau}\ \mathbf{e}) \simeq (\mathscr{G}^{\tau}(SM_N^{\tau}\ \mathbf{e})) \end{aligned}$$

The proof of this claim requires some technical lemmas. The first states that if two terms are equivalent in any evaluation context, then they are equivalent in any context at all:

LEMMA 7. *If* $eval_N(\mathscr{E}[e_1]) = eval_N(\mathscr{E}[e_2])$ *for all* $\mathscr{E}$*, then* $e_1 \simeq e_2$.

PROOF. As in Felleisen *et al* [1987]. □

Note that this also corresponds to Mason and Talcott's "closed instantiations of uses" (ciu) equivalence notion [1991].

The second states that if two terms uniquely reduce to terms that are observationally equivalent, then they are equivalent themselves:

LEMMA 8. *If for all evaluation contexts* $\mathscr{E}[\,]$, $\mathscr{E}[e_1] \rightarrow^* \mathscr{E}[e'_1]$ *(uniquely) and* $\mathscr{E}[e_2] \rightarrow^* \mathscr{E}[e'_2]$ *(uniquely) and* $e'_1 \simeq e'_2$*, then* $e_1 \simeq e_2$.

PROOF. Assume $e_1 \not\simeq e_2$. Then there is some context $\mathscr{E}'[\,]$ such that $eval_N(\mathscr{E}'[e_1])$ produces a different result from $eval_N(\mathscr{E}'[e_2])$. But by premises, $\mathscr{E}'[e_1] \rightarrow^* \mathscr{E}'[e'_1]$ and $\mathscr{E}'[e_2] \rightarrow^* \mathscr{E}'[e'_2]$, and $e'_1 \simeq e'_2$. By definition the evaluations of these two terms are the same. This is a contradiction. □

With these lemmas established, we can prove the main theorem of interest.

PROOF. By lemma 7, for each proposition it suffices to show that the two forms are equivalent in any evaluation context $\mathscr{E}[\,]$. If $\mathbf{e}$ in (1) or e in (2) diverges or signals an error when evaluated, then both sides of each equivalence do so as well. Thus it suffices to show that the evaluation function produces identical results when $\mathbf{e}$ (or e) is a value $\mathbf{v}$ (or v) and the entire expression appears in an evaluation context. We prove that by induction on τ.
Case $\tau = \iota$: For (1): if v is the number $\overline{n}$:

$$\mathscr{E}[(MSG^{\iota}\ \overline{n})] \rightarrow \mathscr{E}[\overline{n}]$$
$$\mathscr{E}[({}^{\iota}MS_N\ (\mathscr{G}^{\iota}\ \overline{n}))] \rightarrow \mathscr{E}[({}^{\iota}MS_N\ \overline{n})] \rightarrow \mathscr{E}[\overline{n}]$$

If v is $(\lambda \mathsf{x}.\mathsf{e}')$, then

$$\mathscr{E}[(MSG^{\iota}\,(\lambda \mathsf{x}.\mathsf{e}'))] \rightarrow \mathscr{E}[(MSG^{\iota}\,(\texttt{wrong}\ \text{"Non-number"}))] \rightarrow \textbf{Error}\text{: Non-number}$$
$$\mathscr{E}[({}^{\iota}MS_N\,(\mathscr{G}^{\iota}\,(\lambda \mathsf{x}.\mathsf{e}')))] \rightarrow \mathscr{E}[({}^{\iota}MS_N\,(\texttt{wrong}\ \text{"Non-number"}))] \rightarrow \textbf{Error}\text{: Non-number}$$

For (2): the analogous argument applies, but by inversion we know that $\mathbf{v}$ must be a natural number so the error case is impossible.
Case $\tau = \tau_1 \rightarrow \tau_2$: For (1): if v is $(\lambda \mathsf{x}.\mathsf{e}')$, then the left-hand side expression reduces as follows:

$$\mathscr{E}[(MSG^{\tau_1 \rightarrow \tau_2}\,(\lambda \mathsf{x}.\mathsf{e}'))] \rightarrow \mathscr{E}[(\lambda \mathbf{x} : \tau.(MSG^{\tau_2}\,((\lambda \mathsf{x}.\mathsf{e}')\,(GSM^{\tau_1}\,\mathbf{x}))))]$$

For the right-hand side:

$$
\begin{array}{lr}
\mathscr{E}[({}^{\tau_1 \rightarrow \tau_2}MS_N(\mathscr{G}^{\tau_1 \rightarrow \tau_2}(\lambda \mathsf{x}.\mathsf{e}')))] & \\
\rightarrow & \text{(by guard reduction)} \\
\mathscr{E}[({}^{\tau_1 \rightarrow \tau_2}MS_N(\lambda \mathsf{x}'.\mathscr{G}^{\tau_2}((\lambda \mathsf{x}.\mathsf{e}')(\mathscr{G}^{\tau_1}\mathsf{x}'))))] & \\
\rightarrow & \text{(by boundary reduction)} \\
\mathscr{E}[(\lambda \mathbf{x}'' : \tau_1.({}^{\tau_2}MS_N((\lambda \mathsf{x}'.\mathscr{G}^{\tau_2}((\lambda \mathsf{x}.\mathsf{e}')(\mathscr{G}^{\tau_1}\mathsf{x}')))(SM_N^{\tau_1}\mathbf{x}''))))] & \\
\simeq & \text{(by } \beta_\omega \text{[Sabry and Felleisen 1993])} \\
\mathscr{E}[(\lambda \mathbf{x}'' : \tau_1.({}^{\tau_2}MS_N(\mathscr{G}^{\tau_2}((\lambda \mathsf{x}.\mathsf{e}')(\mathscr{G}^{\tau_1}(SM_N^{\tau_1}\mathbf{x}''))))))] & \\
\simeq & \text{(by induction hypothesis 1)} \\
\mathscr{E}[(\lambda \mathbf{x}'' : \tau_1.({}^{\tau_2}MS_N(\mathscr{G}^{\tau_2}((\lambda \mathsf{x}.\mathsf{e}')(GSM^{\tau_1}\mathbf{x}'')))))] & \\
\simeq & \text{(by induction hypothesis 2)} \\
\mathscr{E}[(\lambda \mathbf{x}'' : \tau_1.(MSG^{\tau_2}((\lambda \mathsf{x}.\mathsf{e}')(GSM^{\tau_1}\mathbf{x}''))))] &
\end{array}
$$

If v is a number $\overline{n}$, then

$$\mathscr{E}[(MSG^{\tau_1 \to \tau_2}\, \overline{n})] \to \mathscr{E}[(MSG^{\tau_1 \to \tau_2}\, (\texttt{wrong}\text{ “Non-procedure”}))] \to \textbf{Error}\text{: Non-procedure}$$

and

$$\mathscr{E}[({}^{\tau_1 \to \tau_2}MS_N\, (\mathscr{G}^{\tau_1 \to \tau_2}\, \overline{n}))] \to \mathscr{E}[({}^{\tau_1 \to \tau_2}MS_N\, (\texttt{wrong}\text{ “Non-procedure”}))] \to \textbf{Error}\text{: Non-procedure}$$

For (2): the analogous argument applies, with boundaries swapped in the final steps. Also, again by inversion, we can rule out the case in which an error occurs. □

From theorem 6, we know that we can freely exchange any guarded boundary for an unguarded boundary that is wrapped with an appropriate guard without affecting the program's result. It follows that all programs in the separated-guard language that properly wrap their boundaries with guards are well-behaved. The function $elab_M()$ performs that wrapping:

$$\begin{array}{rcl} elab_M() & : & \mathbf{e} \to \mathbf{e} \\ & \vdots & \\ elab_M(({}^{\tau}MS_N\ \mathsf{e})) & = & ({}^{\tau}MS_N\ (\mathscr{G}^{\tau}\ elab_S(\mathsf{e}))) \\ & & \\ elab_S() & : & \mathsf{e} \to \mathsf{e} \\ & \vdots & \\ elab_S((SM_N^{\tau}\ \mathsf{e})) & = & (\mathscr{G}^{\tau}\ (SM_N^{\tau}\ elab_M(\mathbf{e}))) \end{array}$$

(The missing cases in these definitions simply recur structurally on their inputs and otherwise leave them intact.)

Using $elab_M()$ and $elab_S()$ we can formulate a type soundness result for the natural, separated guard embedding as a corollary of theorem 6.

COROLLARY 9. *If* $\vdash_M \boldsymbol{e} : \tau$ *in the language formed by combining figures 1, 2, and 5 (i.e., the natural, separated guard language) and* $\boldsymbol{e}$ *contains no guards, then either* $elab_M(\boldsymbol{e}) \to^* \boldsymbol{v}$, $elab_M(\boldsymbol{e}) \to^*$ **Error***: str, or* $elab_M(\boldsymbol{e})$ *diverges.*

PROOF. If $\vdash_M \mathbf{e} : \tau$, then $\vdash_M elab_M(\mathbf{e}) : \tau$ as well. By theorem 6, $elab_M(\mathbf{e})$ is equivalent to a program in the simple natural embedding, and by theorem 5 that program is well-behaved. Thus, so is $elab_M(\mathbf{e})$. □

The restriction that **e** have no guards in corollary 9 is only necessary because theorem 5 does not account for guards, so we need to be able to guarantee that we can eliminate every guard using the equivalence of theorem 6. This is purely technical — if we wanted to allow programmers to use guards directly, we could reprove theorem 5 with that extension.

3.3 A further refinement: contracts

While the guard strategy of the last subsection works, an implementation based on it would perform many dynamic checks that are guaranteed to succeed. For instance, the term $({}^{\iota \to \iota}MS_N\, (\mathscr{G}^{\iota \to \iota}\, (\lambda \mathsf{x}.\mathsf{x})))$ reduces to $(\lambda \mathbf{x} : \iota.({}^{\iota}MS_N(\mathscr{G}^{\iota}\, ((\lambda \mathsf{x}.\mathsf{x})\, (\mathscr{G}^{\iota}\, (SM_N^{\iota}\, \mathbf{x}))))))$. The check performed by the leftmost guard is necessary, but the check performed by the rightmost guard could be omitted: since the value is coming directly from ML, ML's type system guarantees that the conversion will succeed.

We can refine our guarding strategy to eliminate those unnecessary checks. We split guards into two varieties: positive guards, written $\mathscr{G}_+^{\tau}$, that apply to values going from

Scheme to ML, and negative guards, written $\mathscr{G}^{\tau}_{-}$, that apply to values going from ML to Scheme. Their reduction rules are:

$$\begin{array}{lcl}
\mathscr{E}[(\mathscr{G}^{\iota}_{+}\ \overline{n})]_S & \rightarrow & \mathscr{E}[\overline{n}] \\
\mathscr{E}[(\mathscr{G}^{\iota}_{+}\mathsf{v})]_S & \rightarrow & \mathscr{E}[(\mathtt{wrong}\ \text{“Non-number”})]\ (\mathsf{v} \neq \overline{n} \text{ for any} n) \\
\mathscr{E}[(\mathscr{G}^{\tau_1\rightarrow\tau_2}_{+}\mathsf{v})]_S & \rightarrow & \mathscr{E}[\mathscr{G}^{\tau_2}_{+}\ (\mathsf{v}\ (\mathscr{G}^{\tau_1}_{-}\ \mathsf{x}))]\ (\mathsf{v} = (\lambda \mathsf{x}.\mathsf{e})) \\
\mathscr{E}[(\mathscr{G}^{\tau_1\rightarrow\tau_2}_{+}\mathsf{v})]_S & \rightarrow & \mathscr{E}[(\mathtt{wrong}\ \text{“Non-function”})]\ (\mathsf{v} \neq (\lambda \mathsf{x}.\mathsf{e}) \text{ for any } \mathsf{x}, \mathsf{e}) \\
 & & \\
\mathscr{E}[(\mathscr{G}^{\iota}_{-}\mathsf{v})]_S & \rightarrow & \mathscr{E}[\mathsf{v}] \\
\mathscr{E}[(\mathscr{G}^{\tau_1\rightarrow\tau_2}_{-}\mathsf{v})]_S & \rightarrow & \mathscr{E}[(\lambda \mathsf{x}.(\mathscr{G}^{\tau_2}_{-}(\mathsf{v}(\mathscr{G}^{\tau_1}_{+}\mathsf{x}))))]
\end{array}$$

The function cases are the interesting rules here. Since functions that result from positive guards are bound for ML, we check the inputs that ML will supply them using a negative guard; since the result of those functions will be Scheme values going to ML, they must be guarded with positive guards. Negative guards never directly signal an error; they exist only to protect ML functions from erroneous Scheme inputs. They put positive guards on the arguments to ML functions but use negative guards on their results because those values will have come from ML.

This new system eliminates half of the first-order checks associated with the first separated-guard system, but maintains equivalence with the simple natural embedding system.

THEOREM 10. *For all ML expressions* $\boldsymbol{e}$ *and Scheme expressions* e*, both of the following propositions hold:*

$$\begin{array}{ll}
(1) & (MSG^{\tau}\ \mathsf{e}) \simeq ({}^{\tau}MS_N\ (\mathscr{G}^{\tau}_{+}\ \mathsf{e})) \\
(2) & (GSM^{\tau}\ \boldsymbol{e}) \simeq (\mathscr{G}^{\tau}_{-}\ (SM^{\tau}_N\ \boldsymbol{e}))
\end{array}$$

PROOF. As the proof of theorem 6, *mutatis mutandis*. □

Similarly by defining $elab_{C,M}()$ and $elab_{C,S}()$ functions by analogy to the $elab_M()$ and $elab_S()$ functions of the last section, inserting positive guards around ${}^{\tau}MS_N$ boundaries and negative guards around SM^{τ}_N boundaries, we can obtain the same type-soundness result for this system.

COROLLARY 11. *If* $\vdash_M \boldsymbol{e} : \tau$ *where* $\boldsymbol{e}$ *is a term in the natural, separated guard language, then either* $elab_{C,M}(\boldsymbol{e}) \rightarrow^{*} \boldsymbol{v}$, $elab_{C,M}(\boldsymbol{e}) \rightarrow^{*}$ **Error***: str, or* $elab_{C,M}(\boldsymbol{e})$ *diverges.*

PROOF. Combine theorem 10 and theorem 5 as in the proof of corollary 9. □

As it happens, programmers can implement $\mathscr{G}^{\tau}_{+}$ and $\mathscr{G}^{\tau}_{-}$ directly in the language we have defined so far:

$$\begin{array}{rcl}
\mathtt{G}^{\iota}_{+} & \stackrel{\text{def}}{=} & (\lambda \mathsf{x}.\ (\mathtt{if0}\ (\mathtt{nat?}\ \mathsf{x}) \\
 & & \qquad \mathsf{x} \\
 & & \qquad (\mathtt{wrong}\ \text{“Non-number”}))) \\
\mathtt{G}^{\tau_1\rightarrow\tau_2}_{+} & \stackrel{\text{def}}{=} & (\lambda \mathsf{x}.\ (\mathtt{if0}\ (\mathtt{proc?}\ \mathsf{x}) \\
 & & \qquad (\lambda \mathsf{y}.\ (\mathtt{G}^{\tau_2}_{+}\ (\mathsf{x}\ (\mathtt{G}^{\tau_1}_{-}\ \mathsf{y})))) \\
 & & \qquad (\mathtt{wrong}\ \text{“Non-procedure”}))) \\
\mathtt{G}^{\iota}_{-} & \stackrel{\text{def}}{=} & (\lambda \mathsf{x}.\ \mathsf{x}) \\
\mathtt{G}^{\tau_1\rightarrow\tau_2}_{-} & \stackrel{\text{def}}{=} & (\lambda \mathsf{x}.\ \lambda \mathsf{y}.\ (\mathtt{G}^{\tau_2}_{+}\ (\mathsf{x}\ (\mathtt{G}^{\tau_1}_{-}\ \mathsf{y}))))
\end{array}$$

Each of these implementations is operationally equivalent to the guard it implements.

THEOREM 12. *For all* e *and* τ*, both of the following hold:*

$$(1)\ \mathtt{G}^{\tau}_{+}\, e \simeq \mathscr{G}^{\tau}_{+}\, e$$
$$(2)\ \mathtt{G}^{\tau}_{-}\, e \simeq \mathscr{G}^{\tau}_{-}\, e$$

PROOF. As in the proof of theorem 6, we show equivalence by induction on τ where e is a value and the term appears in an evaluation context; this suffices to prove theorem as stated.

Case $\tau = \iota$:

For (1): Suppose $\mathscr{E}[\,]$ is some evaluation context in which $\mathsf{e} \rightarrow^* \mathsf{v}$. The term $\mathscr{E}[(\mathtt{G}^{\iota}_{+}\ \mathsf{v})]$ reduces as follows:

```
    𝓔[(G^ι_+ v)]
    𝓔[((λx.
          (if0 (nat? x)
               x
               (wrong “non-number”)))
       v)]
 →  𝓔[(if0 (nat? v) v (wrong “Non-number”))]
```

If v is a natural number, then:

$$\mathscr{E}[(\mathtt{if0}\ (\mathtt{nat?}\ \mathsf{v})\ \mathsf{v}\ (\mathtt{wrong}\ \text{“Non-number”}))] \rightarrow \mathscr{E}[\mathsf{v}]$$

and the latter reduces directly to $\mathscr{E}[\mathsf{v}]$, so in this case the two are indistinguishable. Similarly, if v is not a number, then

```
    𝓔[(if0 (nat? v)
            v
            (wrong “Non-number”))]
 →  𝓔[wrong “Non-number”]
```

and $\mathscr{E}[(\mathscr{G}^{\iota}_{+}\mathsf{v})] \rightarrow \mathscr{E}[\mathtt{wrong}\ \text{“Non-number”}]$. Therefore the proposition holds for the base case.

For (2): The same argument applies, but there is no possibility of the computation ending with an error.

Case $\tau = \tau_1 \rightarrow \tau_2$:

For (1): Assume $\mathscr{E}$ is some evaluation context where $\mathscr{E}[\mathsf{e}] \rightarrow^* \mathscr{E}[\mathsf{v}]$. Then $\mathscr{E}[(\mathtt{G}^{\tau_1\rightarrow\tau_2}_{+}\ \mathsf{e})] \rightarrow^* \mathscr{E}[(\mathtt{G}^{\tau_1\rightarrow\tau_2}_{+}\ \mathsf{v})]$ and $\mathscr{E}[(\mathscr{G}^{\tau_1\rightarrow\tau_2}_{+}\ \mathsf{e})] \rightarrow^* \mathscr{E}[(\mathscr{G}^{\tau_1\rightarrow\tau_2}_{+}\ \mathsf{v})]$. The former then reduces as follows:

```
    𝓔[G^{τ1→τ2}_+ v]
    𝓔[((λx.
          (if0 (proc? x)
               (λy. (G^{τ2}_+ (x (G^{τ1}_- y))))
               (wrong “Non-procedure”)))
       v)]
 →  𝓔[(if0 (proc? v)
            (λy. (G^{τ2}_+ (v (G^{τ1}_- y))))
            (wrong “Non-procedure”))]
```

If v is not a procedure, then that term aborts the program with error message “Non-procedure”, as does the term $\mathscr{E}[(\mathscr{G}^{\tau_1\rightarrow\tau_2}_{+}\mathsf{v})]$. If v is a procedure, then that term reduces

$$
\begin{array}{ll}
\mathscr{T}^{\iota}_{M} & \stackrel{\text{def}}{=} (\lambda \mathbf{x} : \mathbf{L}. \\
& \quad ({}^{\iota}MS_L\ ((\mathsf{Y}\ (\lambda \mathsf{f}.\ \lambda \mathsf{n}. \\
& \qquad\qquad (\texttt{if0}\ \mathsf{n} \\
& \qquad\qquad\qquad (SM^{\iota}_{L}\ \overline{0}) \\
& \qquad\qquad\qquad (SM^{\iota}_{L}\ (+\ \overline{1}\ ({}^{\iota}MS_L\ (\mathsf{f}\ (-\ \mathsf{n}\ \overline{1})))))))) \\
& \qquad\quad (SM^{\mathbf{L}}_{L}\ \mathbf{x})))) \\
\mathscr{T}^{\iota}_{S} & \stackrel{\text{def}}{=} (\lambda \mathsf{x}.\ (\mathsf{Y}\ (\lambda \mathsf{f}.\ \lambda \mathsf{n}. \\
& \qquad\qquad (SM^{\mathbf{L}}_{L}\ (\texttt{if0}\ ({}^{\iota}MS_L\ \mathsf{n}) \\
& \qquad\qquad\qquad ({}^{\mathbf{L}}MS_L\ \overline{0}) \\
& \qquad\qquad\qquad ({}^{\mathbf{L}}MS_L\ (+\ \overline{1}\ (\mathsf{f}\ (SM^{\iota}_{L}\ (-\ ({}^{\iota}MS_L\ \mathsf{n})\ \overline{1})))))))) \\
& \qquad \mathsf{x}))
\end{array}
$$

$$
\begin{array}{lll}
\mathscr{T}^{\tau_1 \to \tau_2}_{M} & \stackrel{\text{def}}{=} & (\lambda \mathbf{x} : \mathbf{L}.\lambda \mathbf{y} : \tau_1.(\mathscr{T}^{\tau_2}_{M}({}^{\mathbf{L}}MS_L((SM^{\mathbf{L}}_{L}\mathbf{x})\mathscr{T}^{\tau_1}_{S}(SM^{\tau_1}_{L}\mathbf{y}))))) \\
\mathscr{T}^{\tau_1 \to \tau_2}_{S} & \stackrel{\text{def}}{=} & (\lambda \mathsf{x}.\lambda \mathsf{y}.(\mathscr{T}^{\tau_2}_{S}(SM^{\tau_2}_{L}(({}^{\tau_1 \to \tau_2}MS_L\mathsf{x})(\mathscr{T}^{\tau_1}_{M}({}^{\mathbf{L}}MS_L\mathsf{y}))))))
\end{array}
$$

Fig. 6. Translation functions for lump values

to

$$\mathscr{E}[(\lambda \mathsf{y}.(\mathtt{G}^{\tau_2}_{+}(\mathsf{v}(\mathtt{G}^{\tau_1}_{-}\mathsf{y}))))]$$

(where y is not free in v), and

$$\mathscr{E}[(\mathscr{G}^{\tau_1 \to \tau_2}_{+}\mathsf{v})] \to \mathscr{E}[(\lambda \mathsf{y}.(\mathscr{G}^{\tau_2}_{+}(\mathsf{v}(\mathscr{G}^{\tau_1}_{-}\mathsf{y}))))]$$

(where y is not free in v). These two terms in the hole are observationally equivalent by the induction hypothesis.

For (2): The same argument applies, without possibility of the term reducing to an error. □

Given these implementations, we can observe that $\mathtt{G}^{\tau}_{+}$ and $\mathtt{G}^{\tau}_{-}$ are a pair of projections in the sense of Findler and Blume [2006] where the two parties are + and - and we have prior knowledge from ML's type system that - never violates its contract. From a practical perspective, this means that rather than implementing separate, special-purpose error-containment code for our multi-language systems, we can use an off-the-shelf mechanism with confidence instead. This adds to our confidence in the fine-grained interoperability scheme in Gray *et al* [2005]. More theoretically, it means we can use the simple system of section 3.1 for our models and be more confident that our soundness results apply to actual multi-language embeddings that we write with contracts. From the contract perspective, it also shows one way of using mechanized reasoning to statically eliminate dynamic assertions from annotated programs. In this light it can be seen as a hybrid type system [Flanagan 2006].

4. WHAT HAVE WE GAINED?

The natural embedding fits our intuitive notions of what a useful interoperability system ought to look like much more than the lump embedding does, so it seems like it should give us the power to write more programs. In fact the exact opposite is true: the natural-embedding ${}^{\tau}MS_N$ and SM_N^{τ} boundaries are macro-expressible in the lump embedding (in the sense of Felleisen [1991]), meaning that any natural-embedding program can be translated using local transformations into a lump-embedding program. Furthermore, for some pairs of languages the lump embedding actually admits *more* programs than the natural embedding does. The next two subsections make this precise.

4.1 The lump embedding simulates the natural embedding

As it turns out, two sufficiently determined parties using the lump embedding of section 2 can simulate the natural embedding of section 3. To show that, we define two type-indexed functions, $\mathscr{T}_S^{\tau}$ and $\mathscr{T}_M^{\tau}$, that can be written in the lump embedding. These functions translate values whose ML type is τ from Scheme to ML and from ML to Scheme, respectively; they are shown in figure 6. (In that figure and below, for clarity we use the notation ${}^{\tau}MS_L$ and SM_L^{τ} rather than ${}^{\tau}MS$ and SM^{τ} to refer to the lump embedding's boundaries.) The translation functions for higher-order types use essentially the same method we presented in section 3 for converting a procedure value — they translate a procedure by left- and right-composing appropriate translators for its input and output types. That leaves us with nothing but the base types, in our case just numbers.

The key insight required for those is that the ML number $\overline{3}$ represents not just an opaque datum but the ability to perform some specified action three times to some specified base value; so to convert it to the equivalent Scheme number $\overline{3}$ we can choose to perform the Scheme **add1** function three times to the Scheme $\overline{0}$ base value. Informally, we could define an ML-to-Scheme number conversion function like so:

$$\begin{array}{l}(\lambda \mathbf{x} : \iota .\ (\mathbf{iterate\ x} \\ \qquad\qquad (\lambda \mathbf{y} : \mathbf{L}.\ ({}^{\mathbf{L}}MS\ (\mathbf{add1}\ (SM^{\mathbf{L}}\ \mathbf{y})))) \\ \qquad\qquad ({}^{\mathbf{L}}MS\ \overline{0})))\end{array}$$

where **iterate** is a "fold for numbers" function that loops for the specified number of times, applying the given function successively to results starting with the given base value. Figure 6 presents the complete translation, further encoding **iterate** as a direct use of the Y fix-point combinator. (The two translators look different from each other, but that difference is superficial. It comes from the fact that programmers can only directly write Y in Scheme, so we only give it to ourselves as a Scheme function.)

LEMMA 13. *In the language formed by extending the language of figures 1 and 2 with both figure 3 and figure 4, both of the following propositions hold:*

$$\begin{array}{rcl}(GSM^{\tau}\boldsymbol{e}) & \simeq & \mathscr{T}_S^{\tau}(\mathtt{G}_-^{\tau}(SM_L^{\tau}\boldsymbol{e})) \\ (MSG^{\tau}\mathsf{e}) & \simeq & \mathscr{T}_M^{\tau}({}^{\tau}MS_L(\mathtt{G}_+^{\tau}\ \mathsf{e}))\end{array}$$

PROOF. As before, it is sufficient to consider the case where $\mathbf{e} = \mathbf{v}$ (for the first proposition) or $\mathsf{e} = \mathsf{v}$ (for the second) and the overall expression is in an evaluation context.
Case $\tau = \iota$: Lemma 14 (below) establishes that the translated form of a ι boundary reduces to the corresponding number in the lump embedding. This coincides with the natural-embedding reduction.
Case $\tau = \tau_1 \to \tau_2$: By induction as in the proof of theorem 6. □

LEMMA 14. *Both of the following propositions hold:*

—*For any Scheme value* $(SM_L^{\iota}\,\overline{n})$ *and for any top-level evaluation context* $\mathscr{E}[\ ]_S$, $\mathscr{E}[\mathscr{T}_S^{\iota}(\mathtt{G}_-^{\iota}(SM_L^{\iota}\,\overline{n}))]_S \to^* \mathscr{E}[\overline{n}]$.

—For any ML value ($^{L}MS_L$ v) and for any top-level evaluation context $\mathscr{E}[\ ]_M$:
—If $\mathsf{v} = \overline{n}$ for some n, then $\mathscr{E}[\mathscr{T}^{\iota}_{M}(^{L}MS_L\,(\mathtt{G}^{\iota}_{+}\,\mathsf{v}))]_M \rightarrow^* \mathscr{E}[\overline{n}]$.
*—If $\mathsf{v} \neq \overline{n}$ for any n, then $\mathscr{E}[\mathscr{T}^{\iota}_{M}(^{L}MS_L\,(\mathtt{G}^{\iota}_{+}\,\mathsf{v}))]_M \rightarrow^*$ **Error**: Non-number.*

PROOF. The error cases hold by calculation of the reductions; the non-error cases require induction on the number *n*. □

Given lemma 13, we can run natural-embedding programs using the lump-embedding evaluator by preprocessing them with these program conversion functions:

$$\begin{aligned} trans_M() &: \mathbf{e} \rightarrow \mathbf{e} \\ trans_M((\mathbf{e}_1\ \mathbf{e}_2)) &= (trans_M(\mathbf{e}_1)\ trans_M(\mathbf{e}_2)) \\ &\vdots \\ trans_M((MSG^{\tau}\mathsf{e})) &= (\mathscr{T}^{\tau}_{M}\ (^{\tau}MS_L\,(\mathtt{G}^{\tau}_{+}\,trans_S(\mathsf{e})))) \end{aligned}$$

$$\begin{aligned} trans_S() &: \mathsf{e} \rightarrow \mathsf{e} \\ trans_S((\mathsf{e}_1\ \mathsf{e}_2)) &= (trans_S(\mathsf{e}_1)\ trans_S(\mathsf{e}_2)) \\ &\vdots \\ trans_S((GSM^{\tau}\mathbf{e})) &= (\mathscr{T}^{\tau}_{S}(\mathtt{G}^{\tau}_{-}\,(SM^{\tau}_{L}\,trans_M(\mathbf{e})))) \end{aligned}$$

COROLLARY 15. *If **e** is a well-typed closed program in the natural embedding of section 3, then $eval_N(\boldsymbol{e}) = eval_L(trans_M(\boldsymbol{e}))$.*

PROOF. Combine theorems 10 and 12 and lemma 13. □

COROLLARY 16. *Both the SM^{τ}_{N} and $^{\tau}MS_N$ boundaries are macro-expressible in the lump embedding.*

Based on these translation rules, we were able to implement a program using PLT Scheme and C interacting through PLT Scheme's foreign interface [Barzilay and Orlovsky 2004] but maintaining a strict lump discipline; we were able to build base-value converters in that setting.

The given conversion algorithms are quite inefficient, running in time proportional to the magnitude of the number, but we can do better. Rather than having the sender simply transmit a unary "keep going" or "stop" signal to the receiver, the receiver could give the sender representations of 0 and 1 and let the sender repeatedly send the least significant bit of the number to be converted. This approach would run in time proportional to the base-2 log of the converted number assuming that each language had constant-time bit-shift left and bit-shift right operations.

4.2 The lump embedding admits non-termination

By the argument above, we have shown that going from the lump embedding to the natural embedding has not bought us any expressive power. That's okay; we have not lost any power either. But there are language-embedding pairs for which we would have. In particular, if we embed our ML stand-in into another copy of itself using the lump embedding, we gain the ability to write non-terminating programs in the resulting language even though both constituent languages are terminating. If we move to the natural embedding, we lose that ability.

To make that more precise, consider the ML-to-ML lump-embedding language defined in figure 7, which shows an embedding analogous to the ML-to-Scheme lump embedding

$$
\begin{array}{lcl}
\mathbf{e} & = & \mathbf{x} \mid \mathbf{v} \mid (\mathbf{e}\ \mathbf{e}) \mid (op\ \mathbf{e}\ \mathbf{e}) \mid (\mathtt{if0}\ \mathbf{e}\ \mathbf{e}\ \mathbf{e}) \mid ({}^{\tau}M_1M_2^{\sigma}\ \mathsf{e}) \\
\mathbf{v} & = & (\lambda \mathbf{x} : \tau .\ \mathbf{e}) \mid \overline{n} \mid ({}^{\mathbf{L}_1}M_1M_2^{\sigma}\ \mathsf{v}) \\
\tau & = & \iota \mid \tau \rightarrow \tau \mid \mathbf{L}_1 \\
\mathbf{E} & = & [\ ]_{M_1} \mid (\mathbf{E}\ \mathbf{e}) \mid (\mathbf{v}\ \mathbf{E}) \mid (op\ \mathbf{E}\ \mathbf{e}) \mid (op\ \mathbf{v}\ \mathbf{E}) \mid \\
 & & (\mathtt{if0}\ \mathbf{E}\ \mathbf{e}\ \mathbf{e}) \mid ({}^{\tau}M_1M_2^{\sigma}\ \mathsf{E})
\end{array}
$$

$$
\frac{\Gamma \vdash_{M_2} \mathsf{e} : \sigma\ (\sigma \neq \mathbf{L}_2)}{\Gamma \vdash_{M_1} ({}^{\mathbf{L}_1}M_1M_2^{\sigma}\ \mathsf{e}) : \mathbf{L}_1}
\qquad
\frac{\Gamma \vdash_{M_2} \mathsf{e} : \mathbf{L}_2}{\Gamma \vdash_{M_1} ({}^{\tau}M_1M_2^{\mathbf{L}_2}\ \mathsf{e}) : \tau}
$$

$$
\begin{array}{lcl}
\mathscr{E}[({}^{\tau}M_1M_2^{\mathbf{L}_2}\ ({}^{\mathbf{L}_2}M_2M_1^{\tau}\ \mathbf{v}))]_{M_1} & \rightarrow & \mathscr{E}[\mathbf{v}] \\
\mathscr{E}[({}^{\tau}M_1M_2^{\mathbf{L}_2}\ ({}^{\mathbf{L}_2}M_2M_1^{\tau'}\ \mathbf{v}))]_{M_1} & \rightarrow & \textbf{Error}\text{: Bad conversion} \\
 & & \qquad\text{(where } \tau \neq \tau'\text{)}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{e} & = & \mathsf{x} \mid \mathsf{v} \mid (\mathsf{e}\ \mathsf{e}) \mid (op\ \mathsf{e}\ \mathsf{e}) \mid (\mathtt{if0}\ \mathsf{e}\ \mathsf{e}\ \mathsf{e}) \mid ({}^{\sigma}M_2M_1^{\tau}\ \mathbf{e}) \\
\mathsf{v} & = & (\lambda \mathsf{x} : \sigma .\ \mathsf{e}) \mid \overline{n} \mid ({}^{\mathbf{L}_2}M_2M_1^{\tau}\ \mathbf{v}) \\
\sigma & = & \iota \mid \sigma \rightarrow \sigma \mid \mathbf{L}_2 \\
\mathsf{E} & = & [\ ]_{M_2} \mid (\mathsf{E}\ \mathsf{e}) \mid (\mathsf{v}\ \mathsf{E}) \mid (op\ \mathsf{E}\ \mathsf{e}) \mid (op\ \mathsf{v}\ \mathsf{E}) \mid \\
 & & (\mathtt{if0}\ \mathsf{E}\ \mathsf{e}\ \mathsf{e}) \mid ({}^{\sigma}M_2M_1^{\tau}\ \mathbf{E})
\end{array}
$$

$$
\frac{\Gamma \vdash_{M_1} \mathbf{e} : \tau\ (\tau \neq \mathbf{L}_1)}{\Gamma \vdash_{M_2} ({}^{\mathbf{L}_2}M_2M_1^{\tau}\ \mathsf{e}) : \mathbf{L}_2}
\qquad
\frac{\Gamma \vdash_{M_1} \mathbf{e} : \mathbf{L}_1}{\Gamma \vdash_{M_2} ({}^{\sigma}M_2M_1^{\mathbf{L}_1}\ \mathbf{e}) : \sigma}
$$

$$
\begin{array}{lcl}
\mathscr{E}[({}^{\sigma}M_2M_1^{\mathbf{L}_1}\ ({}^{\mathbf{L}_1}M_1M_2^{\sigma}\ \mathsf{v}))]_{M_2} & \rightarrow & \mathscr{E}[\mathsf{v}] \\
\mathscr{E}[({}^{\sigma}M_2M_1^{\mathbf{L}_1}\ ({}^{\mathbf{L}_1}M_1M_2^{\sigma'}\ \mathsf{v}))]_{M_2} & \rightarrow & \textbf{Error}\text{: Bad conversion} \\
 & & \qquad\text{(where } \sigma \neq \sigma'\text{)}
\end{array}
$$

Fig. 7. An ML-in-ML lump embedding

---

from section 2 where both sides are copies of our ML stand-in (figure 7 elides the standard rules for abstractions, application, numeric operators and so on, which are as in figure 1). That language admits nonterminating programs: if the two constituent languages conspire they can allow one language to "pack" a value of type $\mathbf{L}_1 \rightarrow \mathbf{L}_1$ into a value of type $\mathbf{L}_1$ and later "unpack" that value back into a value of type $\mathbf{L}_1 \rightarrow \mathbf{L}_1$. This gives us the ability to build something akin to Abadi *et al*'s type `dynamic` [1991], and enough power to build a nonterminating term.

THEOREM 17. *The language of figure 7 is non-terminating.*

PROOF. By construction. Given the following definitions:

$\mathrm{P} \stackrel{\text{def}}{=} (\lambda \mathbf{x} : \mathbf{L}_1 \rightarrow \mathbf{L}_1 .\ ({}^{\mathbf{L}_1}M_1M_2^{\iota\rightarrow\mathbf{L}_2}\ (\lambda \mathsf{y} : \iota .\ ({}^{\mathbf{L}_2}M_2M_1^{\mathbf{L}_1\rightarrow\mathbf{L}_1}\ \mathbf{x}))))$

$\mathrm{U} \stackrel{\text{def}}{=} (\lambda \mathbf{x} : \mathbf{L}_1 .\ ({}^{\mathbf{L}_1\rightarrow\mathbf{L}_1}M_1M_2^{\mathbf{L}_2}\ (({}^{\iota\rightarrow\mathbf{L}_2}M_2M_1^{\mathbf{L}_1}\ \mathbf{x}))\ \overline{0}))$

consider the following term:

$\Omega \stackrel{\text{def}}{=} ((\lambda \mathbf{x} : \mathbf{L}_1 .\ ((\mathrm{U}\ \mathbf{x})\ \mathbf{x}))\ (\mathrm{P}\ (\lambda \mathbf{x} : \mathbf{L}_1 .\ ((\mathrm{U}\ \mathbf{x})\ \mathbf{x}))))$

It is a simple calculation to verify that Ω type-checks as a closed M_1M_2 program, and that its reduction graph contains a cycle. □

$$\begin{array}{lcl} \mathbf{e} & = & \mathbf{x} \mid \mathbf{v} \mid (\mathbf{e}\ \mathbf{e}) \mid (op\ \mathbf{e}\ \mathbf{e}) \mid (\texttt{if0}\ \mathbf{e}\ \mathbf{e}\ \mathbf{e}) \mid ({}^{\tau}M_1M_2^{\sigma}\ \mathsf{e}) \\ \mathbf{v} & = & (\lambda \mathbf{x} : \tau.\ \mathbf{e}) \mid \overline{n} \\ \tau & = & \iota \mid \tau \to \tau \\ \mathbf{E} & = & [\ ]_{M_1} \mid (\mathbf{E}\ \mathbf{e}) \mid (\mathbf{v}\ \mathbf{E}) \mid (op\ \mathbf{E}\ \mathbf{e}) \mid (op\ \mathbf{v}\ \mathbf{E}) \mid \\ & & (\texttt{if0}\ \mathbf{E}\ \mathbf{e}\ \mathbf{e}) \mid ({}^{\tau}M_1M_2^{\sigma}\ \mathsf{E}) \end{array}$$

$$\frac{\Gamma \vdash_{M_2} \mathsf{e} : \sigma \quad \tau = \sigma}{\Gamma \vdash_{M_1} {}^{\tau}M_1M_2^{\sigma}\ \mathsf{e} : \tau}$$

$$\begin{array}{lcl} \mathscr{E}[({}^{\iota}M_1M_2^{\iota}\ \overline{n})]_{M_1} & \to & \mathscr{E}[\overline{n}] \\ \mathscr{E}[({}^{\tau_1\to\tau_2}M_1M_2^{\sigma_1\to\sigma_2}\ \mathsf{v})]_{M_1} & \to & \mathscr{E}[(\lambda \mathbf{x} : \tau_1.\ ({}^{\tau_2}M_1M_2^{\sigma_2}\ (\mathsf{v}\ ({}^{\sigma_1}M_2M_1^{\tau_1}\ \mathbf{x}))))] \\ & & \text{(where } \tau_1 = \sigma_1, \tau_2 = \sigma_2) \end{array}$$

$$\begin{array}{lcl} \mathsf{e} & = & \mathsf{x} \mid \mathsf{v} \mid (\mathsf{e}\ \mathsf{e}) \mid (op\ \mathsf{e}\ \mathsf{e}) \mid (\texttt{if0}\ \mathsf{e}\ \mathsf{e}\ \mathsf{e}) \mid ({}^{\sigma}M_2M_1^{\tau}\ \mathbf{e}) \\ \mathsf{v} & = & (\lambda \mathsf{x} : \sigma.\ \mathsf{e}) \mid \overline{n} \\ \sigma & = & \iota \mid \sigma \to \sigma \\ \mathsf{E} & = & [\ ]_{M_2} \mid (\mathsf{E}\ \mathsf{e}) \mid (\mathsf{v}\ \mathsf{E}) \mid (op\ \mathsf{E}\ \mathsf{e}) \mid (op\ \mathsf{v}\ \mathsf{E}) \mid \\ & & (\texttt{if0}\ \mathsf{E}\ \mathsf{e}\ \mathsf{e}) \mid ({}^{\sigma}M_2M_1^{\tau}\ \mathbf{E}) \end{array}$$

$$\frac{\Gamma \vdash_{M_1} \mathbf{e} : \tau \quad \tau = \sigma}{\Gamma \vdash_{M_2} ({}^{\sigma}M_1M_2^{\tau}\ \mathbf{e}) : \sigma}$$

$$\begin{array}{lcl} \mathscr{E}[({}^{\iota}M_2M_1^{\iota}\ \overline{n})]_{M_2} & \to & \mathscr{E}[\overline{n}] \\ \mathscr{E}[({}^{\sigma_1\to\sigma_2}M_2M_1^{\tau_1\to\tau_2}\ \mathbf{v})]_{M_2} & \to & \mathscr{E}[(\lambda\ \mathsf{x} : \sigma_1\ .\ ({}^{\sigma_2}M_2M_1^{\tau_2}\ (\mathbf{v}\ ({}^{\tau_1}M_1M_2^{\sigma_1}\ \mathsf{x}))))] \\ & & \text{(where } \tau_1 = \sigma_1, \tau_2 = \sigma_2) \end{array}$$

Fig. 8. An ML-in-ML natural embedding

The P and U functions make the construction work by creating a way to convert a value of type $\mathbf{L}_1$ to a value of type $\mathbf{L}_1 \to \mathbf{L}_1$ and vice versa — the same type conversion ability that the isorecursive type $\mu L.L \to L$ would give us, except that our conversion uses a dynamic check that could fail (but does not in the Ω term). We could extend this technique to write a fixed-point combinator and from there we could implement variants of the conversion functions from figure 6, giving us a full natural embedding along with Y.

If we go to the natural embedding, we lose the crucial ability to hide functions as lumps, so we regain termination and lose the ability to write Y or Ω.

We define the natural ML-to-ML embedding in figure 8. It makes changes analogous the changes made our ML-and-Scheme system between sections 2 and 3: we remove the lump type, and instead we define conversions at all between languages using direct conversion at base types and wrappers at higher-order types. As in figure 7, we have omitted several standard rules for language features that do not directly bear on interoperation.

THEOREM 18. *If* $\boldsymbol{e}$ *is a program in the language of figure 8 and* $\vdash_M \boldsymbol{e} : \tau$, *then* $\boldsymbol{e} \to^* \boldsymbol{v}$.

To show that this language terminates, we use a generalized form of Tait's method for proving termination by use of logical relations [Tait 1967] (our presentation is based on

Pierce [2002], chapter 12). We define two logical relations, one for M_1 and one for M_2.

$$\begin{array}{lcl}
\mathcal{C}_1[\![\tau]\!] & \stackrel{\text{def}}{=} & \{\mathbf{e} \mid \forall \mathcal{E}[\,]_M.\ \mathcal{E}[\mathbf{e}]_M \rightarrow^* \mathcal{E}[\mathbf{v}] \text{ and } \mathbf{v} \in \mathcal{V}_1[\![\tau]\!]\} \\
\mathcal{V}_1[\![\iota]\!] & \stackrel{\text{def}}{=} & \{\overline{n} \mid n \in \mathbb{N}\} \\
\mathcal{V}_1[\![\tau_1 \rightarrow \tau_2]\!] & \stackrel{\text{def}}{=} & \{\mathbf{v} \mid \forall \mathbf{v}' \in \mathcal{V}_1[\![\tau_1]\!].(\mathbf{v}\ \mathbf{v}') \in \mathcal{C}_1[\![\tau_2]\!]\} \\
\\
\mathcal{C}_2[\![\sigma]\!] & \stackrel{\text{def}}{=} & \{\mathsf{e} \mid \forall \mathcal{E}[\,]_S.\ \mathcal{E}[\mathsf{e}]_S \rightarrow^* \mathcal{E}[\mathsf{v}] \text{ and } \mathsf{v} \in \mathcal{V}_2[\![\sigma]\!]\} \\
\mathcal{V}_2[\![\iota]\!] & \stackrel{\text{def}}{=} & \{\overline{n} \mid n \in \mathbb{N}\} \\
\mathcal{V}_2[\![\sigma_1 \rightarrow \sigma_2]\!] & \stackrel{\text{def}}{=} & \{\mathsf{v} \mid \forall \mathsf{v}' \in \mathcal{V}_2[\![\sigma_1]\!].(\mathsf{v}\ \mathsf{v}') \in \mathcal{C}_2[\![\sigma_2]\!]\}
\end{array}$$

By definition, any term in either of these sets terminates, so proof of termination amounts to showing that every well-typed term is in the logical relation at its type (often called the "fundamental theorem" of the logical relation). To show that, we need one essentially multi-language lemma that allows us to connect membership in one set with membership in the other:

LEMMA 19. *Both of the following propositions hold:*

(1) *If* $\Gamma \vdash_{M_1} ({}^{\tau}M_1M_2^{\sigma}\ \mathsf{e}) : \tau$, *$\gamma$ is a well-typed closing substitution for Γ such that all terms in γ are in their respective logical relations, and* $\gamma(\mathsf{e}) \in \mathcal{C}_2[\![\sigma]\!]$, *then* $\gamma({}^{\tau}M_1M_2^{\sigma}\ \mathsf{e}) \in \mathcal{C}_1[\![\tau]\!]$.

(2) *If* $\Gamma \vdash_{M_2} ({}^{\sigma}M_2M_1^{\tau}\ \boldsymbol{e}) : \sigma$, *$\gamma$ is a well-typed closing substitution for Γ such that all terms in gamma are in their respective logical relations, and* $\boldsymbol{e} \in \mathcal{C}_1[\![\tau]\!]$, *then* $\gamma({}^{\sigma}M_2M_1^{\tau}\ \boldsymbol{e}) \in \mathcal{C}_2[\![\sigma]\!]$.

PROOF. By simultaneous induction on τ and σ. For (1), if τ is ι, then by inversion $\sigma = \iota$. Since $\gamma(\mathsf{e}) \in \mathcal{C}_2[\![\iota]\!]$, it evaluates to a number. Then the boundary reduction rule reduces the entire term to a number, which is in $\mathcal{C}_1[\![\iota]\!]$.

If $\tau = \tau_1 \rightarrow \tau_2$, then $\sigma = \sigma_1 \rightarrow \sigma_2$ and

$$\begin{array}{ll}
& \mathcal{E}[({}^{\tau}M_1M_2^{\sigma}\ \gamma(\mathsf{v}))]_M \\
\rightarrow & \mathcal{E}[(\lambda \mathbf{x}' : \tau_1.({}^{\tau_2}M_1M_2^{\sigma_2}\ (\gamma(\mathsf{v})\ ({}^{\sigma_1}M_2M_1^{\tau_1}\ \mathbf{x}'))))]
\end{array}$$

This value is in $\mathcal{C}_1[\![\tau_1 \rightarrow \tau_2]\!]$. To see that, let v' be some value in $\mathcal{V}_1[\![\tau_1]\!]$. Then

$$\begin{array}{ll}
& \mathcal{E}[(\lambda \mathbf{x}' : \tau_1.({}^{\tau_2}M_1M_2^{\sigma_2}\ (\gamma(\mathsf{v})\ ({}^{\sigma_1}M_2M_1^{\tau_1}\ \mathbf{x}'))))\ \mathbf{v}']_M \\
\rightarrow & \mathcal{E}[({}^{\tau_2}M_1M_2^{\sigma_2}\ (\gamma(\mathsf{v})\ ({}^{\sigma_1}M_2M_1^{\tau_1}\ \mathbf{v}')))]
\end{array}$$

By induction hypothesis (2) the rightmost application yields a value in $\mathcal{V}_2[\![\sigma_1]\!]$; since by assumption $\gamma(\mathsf{v}) \in \mathcal{C}_2[\![\sigma_1 \rightarrow \sigma_2]\!]$ (and thus, since it is also a value, in $\mathcal{V}_2[\![\sigma_1 \rightarrow \sigma_2]\!]$), the resulting application is in $\mathcal{C}_2[\![\sigma_2]\!]$. Now by induction on (1), the boundary conversion is in $\mathcal{C}_1[\![\tau_2]\!]$ as required. □

With this lemma, it is straightforward to prove that all well-typed terms are in their respective relations.

LEMMA 20. *Both of the following propositions hold:*

(1) *If* $\Gamma \vdash_{M_1} \boldsymbol{e} : \tau$ *and γ is a well-typed closing substitution for Γ such that all terms in γ are in their respective logical relations, then* $\gamma(\boldsymbol{e}) \in \mathcal{C}_1[\![\tau]\!]$.

(2) *If* $\Gamma \vdash_{M_2} \mathsf{e} : \sigma$ *and γ is a well-typed closing substitution for Γ such that all terms in γ are in their respective logical relations, then* $\gamma(\mathsf{e}) \in \mathcal{C}_2[\![\sigma]\!]$.

PROOF. By simultaneous induction on the typing derivations for **e** and e. The cases are standard except for the boundary cases.

Case $\dfrac{\Gamma \vdash_{M_2} \mathsf{e} : \sigma \quad \tau = \sigma}{\Gamma \vdash_{M_1} ({}^{\tau}M_1 M_2^{\sigma}\ \mathsf{e}) : \tau}$:

By induction hypothesis 2, e reduces to a value $\mathsf{v} \in \mathscr{V}_2[\![\sigma]\!]$ in the evaluation context formed by composing the boundary expression with an arbitrary outer evaluation context. This is in the relation by case 1 of lemma 19 above, whose preconditions are satisfied by the induction hypothesis.

Case $\dfrac{\Gamma \vdash_{M_1} \mathbf{e} : \tau \quad \tau = \sigma}{\Gamma \vdash_{M_2} ({}^{\sigma}M_2 M_1^{\tau}\ \mathbf{e}) : \sigma}$:

Analogous to the above, using case 2 of lemma 19 instead of case 1. □

The main theorem is immediate from lemma 20. Thus for the ML-in-ML language, the natural embedding is strictly less powerful than the lump embedding.

We do not want to imply by this section that real multi-language systems ought to use a lump embedding strategy rather than a natural embedding strategy, for the same reason that we would not advocate that real programming languages use Church numerals rather than direct representations for natural numbers. Our intention was only to point out some surprising, counterintuitive facts about the relative power of the two embedding strategies we have presented so far.

5. EXCEPTIONS

The natural embedding can be extended straightforwardly to address effects such as the exception mechanisms found in many modern programming languages. Here we give two different ways to do so; one in which exceptions cannot cross boundaries, and one in which they can.

The two systems have the same syntax and type-checking rules. We add a new exception-raising form (`raise` τ *str*) to ML and extend Scheme's (`wrong` *str*) syntax form, both of which now raise an exception that can be handled. We also add a (`handle` e e) form to Scheme; in situations that do not involve language intermingling it evaluates its second subterm and returns its value unless that evaluation raises an exception (through (`wrong` *str*)), in which case it abandons that computation and returns the result of evaluating its exception handler subterm instead. ML has a `handle` expression with an identical syntax that works the same way. In both systems a completely unhandled exception causes the program to terminate with an error message.

5.1 System 1: Exceptions cannot propagate

One simple strategy for handling exceptions in a multilanguage system is to decide that if an exception ever reaches a boundary it aborts the entire program. This strategy can be found for instance in the current implementation of Moby [Fisher et al. 2001].

We present a model for this system in figure 9 as an extension to the natural embedding of figure 4. To make exceptions work, we need to give more structure to our evaluation contexts. To that end, we add several new kinds of contexts: **H** and H (for "no **h**andlers") contexts are roughly the same as the **E** and E evaluation contexts were in the languages before they were connected; they are full evaluation contexts except that they do not allow a nested context to appear inside an exception handler or a boundary. The **F** and F (for "no **f**oreign calls") contexts relax that restriction by allowing holes inside handlers but not

$$
\begin{array}{lcl}
\mathbf{e} & = & \cdots \mid (\texttt{handle}\ \mathbf{e}\ \mathbf{e}) \mid (\texttt{raise}\ \tau\ \mathit{str}) \\
\mathsf{e} & = & \cdots \mid (\texttt{handle}\ \mathsf{e}\ \mathsf{e}) \\
\\
\mathbf{H} & = & [\,]_M \mid (\mathbf{H}\ \mathbf{e}) \mid (\mathbf{v}\ \mathbf{H}) \mid (\mathit{op}\ \mathbf{H}\ \mathbf{e}) \mid (\mathit{op}\ \mathbf{v}\ \mathbf{H}) \mid (\texttt{if0}\ \mathbf{H}\ \mathbf{e}\ \mathbf{e}) \\
\mathbf{F} & = & \mathbf{H} \mid \mathbf{H}[(\texttt{handle}\ \mathbf{e}\ \mathbf{F})]_M \\
\mathbf{E} & = & \mathbf{F} \mid \mathbf{F}[(\mathit{MSG}^{\tau}\ \mathsf{E})]_M \\
\mathsf{H} & = & [\,]_S \mid (\mathsf{H}\ \mathsf{e}) \mid (\mathsf{v}\ \mathsf{H}) \mid (\mathit{op}\ \mathsf{H}\ \mathsf{e}) \mid (\mathit{op}\ \mathsf{v}\ \mathsf{H}) \mid (\texttt{if0}\ \mathsf{H}\ \mathsf{e}\ \mathsf{e}) \mid (\mathit{pr}\ \mathsf{H}) \\
\mathsf{F} & = & \mathsf{H} \mid \mathsf{H}[(\texttt{handle}\ \mathsf{e}\ \mathsf{F})]_S \\
\mathsf{E} & = & \mathsf{F} \mid \mathsf{F}[(\mathit{GSM}^{\tau}\ \mathbf{E})]_S \\
\\
\mathscr{E} & = & \mathbf{E}
\end{array}
$$

$$
\frac{}{\Gamma \vdash_M (\texttt{raise}\ \tau\ \mathit{str}) : \tau}
\qquad
\frac{\Gamma \vdash_M \mathbf{e}_1 : \tau \qquad \Gamma \vdash_M \mathbf{e}_2 : \tau}{\Gamma \vdash_M (\texttt{handle}\ \mathbf{e}_1\ \mathbf{e}_2) : \tau}
$$

$$
\frac{\Gamma \vdash_S \mathsf{e}_1 : \mathbf{TST} \qquad \Gamma \vdash_S \mathsf{e}_2 : \mathbf{TST}}{\Gamma \vdash_S (\texttt{handle}\ \mathsf{e}_1\ \mathsf{e}_2) : \mathbf{TST}}
$$

$$
\begin{array}{lcl}
\mathscr{E}[(\texttt{handle}\ \mathbf{e}\ \mathbf{v})]_M & \rightarrow & \mathscr{E}[\mathbf{v}] \\
\mathscr{E}[(\texttt{handle}\ \mathsf{e}\ \mathsf{v})]_S & \rightarrow & \mathscr{E}[\mathsf{v}] \\
\mathscr{E}[(\texttt{handle}\ \mathbf{e}\ \mathbf{H}[(\texttt{raise}\ \tau\ \mathit{str})]_M)]_M & \rightarrow & \mathscr{E}[\mathbf{e}] \\
\mathscr{E}[(\texttt{handle}\ \mathsf{e}\ \mathsf{H}[(\texttt{wrong}\ \mathit{str})]_S)]_S & \rightarrow & \mathscr{E}[\mathsf{e}] \\
\\
\mathbf{H}[(\texttt{raise}\ \tau\ \mathit{str})]_M & \rightarrow & \mathbf{Error}\text{: } \mathit{str} \\
\\
\mathscr{E}[(\mathit{GSM}^{\tau}\ \mathbf{H}[(\texttt{raise}\ \tau'\ \mathit{str})]_M)]_S & \rightarrow & \mathbf{Error}\text{: } \mathit{str} \\
\mathscr{E}[(\mathit{MSG}^{\tau}\ \mathsf{H}[(\texttt{wrong}\ \mathit{str})]_S)]_M & \rightarrow & \mathbf{Error}\text{: } \mathit{str}
\end{array}
$$

Fig. 9. Exceptions system 1 reduction rules

---

boundaries, and $\mathbf{E}$ and E relax it again by allowing holes in handlers or boundaries (notice that no ellipses precede these definitions; they are meant to replace rather than augment the prior definitions in figures 1, 2, and 4). These modifications are a minor variant on Wright and Felleisen's exceptions model [1994]; unlike their system, ours does not allow exceptions to carry values.

These new contexts allow us to give precise reduction rules for exceptions and handlers. First, we must implictly *remove* the rule for `wrong` that has been present in every system up to this point. That done, the first two reduction rules in figure 9 cover the least interesting cases, in which a handler expression's body evaluates to completion without raising an exception. The second two rules detect the cases where ML or Scheme, respectively, raises and catches an exception without crossing a language boundary. The fifth rule detects the situation where a complete program raises an unhandled exception without that exception crossing any boundaries.

From the interoperability standpoint, the last two rules are the most interesting. These two rules reuse the $\mathbf{H}$ and H contexts we used for exception handlers to make each boundary a kind of exception handler as well; the difference being that when boundaries catch an

$$
\begin{array}{lcl}
\mathbf{e} & = & \cdots \mid (\texttt{handle}\ \mathbf{e}\ \mathbf{e}) \mid (\texttt{raise}\ \tau\ \mathit{str}) \\
\mathsf{e} & = & \cdots \mid (\texttt{handle}\ \mathsf{e}\ \mathsf{e}) \\
\\
\mathbf{H} & = & [\,]_M \mid (\mathbf{H}\ \mathbf{e}) \mid (\mathbf{v}\ \mathbf{H}) \mid (\mathit{op}\ \mathbf{H}\ \mathbf{e}) \mid (\mathit{op}\ \mathbf{v}\ \mathbf{H}) \mid (\texttt{if0}\ \mathbf{H}\ \mathbf{e}\ \mathbf{e}) \\
\mathbf{F} & = & \mathbf{H} \mid \mathbf{H}[(\texttt{handle}\ \mathbf{e}\ \mathbf{F})]_M \\
\mathbf{E} & = & \mathbf{F} \mid \mathbf{F}[(\mathit{MSG}^\tau\ \mathsf{E})]_M \\
\mathsf{H} & = & [\,]_S \mid (\mathsf{H}\ \mathsf{e}) \mid (\mathsf{v}\ \mathsf{H}) \mid (\mathit{op}\ \mathsf{H}\ \mathsf{e}) \mid (\mathit{op}\ \mathsf{v}\ \mathsf{H}) \mid (\texttt{if0}\ \mathsf{H}\ \mathsf{e}\ \mathsf{e}) \mid (\mathit{pr}\ \mathsf{H}) \\
\mathsf{F} & = & \mathsf{H} \mid \mathsf{H}[(\texttt{handle}\ \mathsf{e}\ \mathsf{F})]_S \\
\mathsf{E} & = & \mathsf{F} \mid \mathsf{F}[(\mathit{GSM}^\tau\ \mathbf{E})]_S \\
\\
\mathscr{E} & = & \mathbf{E}
\end{array}
$$

$$
\frac{}{\Gamma \vdash_M (\texttt{raise}\ \tau\ \mathit{str}) : \tau}
\qquad
\frac{\Gamma \vdash_M \mathbf{e}_1 : \tau \qquad \Gamma \vdash_M \mathbf{e}_2 : \tau}{\Gamma \vdash_M (\texttt{handle}\ \mathbf{e}_1\ \mathbf{e}_2) : \tau}
$$

$$
\frac{\Gamma \vdash_S \mathsf{e}_1 : \mathbf{TST} \qquad \Gamma \vdash_S \mathsf{e}_2 : \mathbf{TST}}{\Gamma \vdash_S (\texttt{handle}\ \mathsf{e}_1\ \mathsf{e}_2) : \mathbf{TST}}
$$

$$
\begin{array}{lcl}
\mathscr{E}[(\texttt{handle}\ \mathbf{e}\ \mathbf{v})]_M & \to & \mathscr{E}[\mathbf{v}] \\
\mathscr{E}[(\texttt{handle}\ \mathsf{e}\ \mathsf{v})]_S & \to & \mathscr{E}[\mathsf{v}] \\
\mathscr{E}[(\texttt{handle}\ \mathbf{e}\ \mathbf{H}[(\texttt{raise}\ \tau\ \mathit{str})]_M)]_M & \to & \mathscr{E}[\mathbf{e}] \\
\mathscr{E}[(\texttt{handle}\ \mathsf{e}\ \mathsf{H}[(\texttt{wrong}\ \mathit{str})]_S)]_S & \to & \mathscr{E}[\mathsf{e}] \\
\\
\mathbf{H}[(\texttt{raise}\ \tau\ \mathit{str})]_M & \to & \mathbf{Error}\text{: } \mathit{str} \\
\\
\mathscr{E}[(\mathit{GSM}^\tau \mathbf{H}[(\texttt{raise}\ \tau'\ \mathit{str})]_M)]_S & \to & \mathscr{E}[(\texttt{wrong}\ \mathit{str})] \\
\mathscr{E}[(\mathit{MSG}^\tau\ \mathsf{H}[(\texttt{wrong}\ \mathit{str})]_S)]_M & \to & \mathscr{E}[(\texttt{raise}\ \tau\ \mathit{str})]
\end{array}
$$

Fig. 10. Exceptions system 2 reduction rules

exception, they abort the program instead of giving the programmer a chance to take less drastic corrective action.

THEOREM 21. *Exceptions system 1 is type-sound.*

PROOF. The proof of theorem 1, *mutatis mutandis*. □

5.2 System 2: Exceptions are translated

A more useful way to deal with exceptions encountering language boundaries, and the method preferred by most existing foreign function interfaces that connect high-level languages (*e.g.*, SML.NET [Benton et al. 2004], Scala [Odersky et al. 2005]), is to catch exceptions at a language boundary and reraise an equivalent exception on the other side of the boundary.

This behavior is easy to model; in fact, our model for it only differs from figure 9 in the last two rules. Figure 10 presents the rules as an extension to the natural embedding of figure 4: everything here is the same as it was in figure 9 until the rules that govern how a boundary treats an unhandled exception. In this system, rather than terminating the program we rewrite the boundary into a new exception-raising form. This allows the

$$\begin{aligned}\mathbf{e} &= \cdots \mid (\Lambda\alpha.\mathbf{e}) \mid \mathbf{e}\langle\tau\rangle \\ \mathbf{v} &= \cdots \mid (\mathit{MSG}^{\mathbf{L}}\ \mathsf{v}) \\ \tau &= \cdots \mid \forall\alpha.\tau \mid \alpha \mid \mathbf{L} \\ \mathbf{E} &= \cdots \mid \mathbf{E}\langle\tau\rangle\end{aligned}$$

$$\frac{\Gamma \vdash_M \mathbf{e} : \forall\alpha.\tau' \quad \Gamma \vdash \tau : \text{type}}{\Gamma \vdash_M \mathbf{e}\langle\tau\rangle : \tau'[\tau/\alpha]} \qquad \frac{\Gamma, \alpha\ \text{type} \vdash_M \mathbf{e} : \tau}{\Gamma \vdash_M (\Lambda\alpha.\mathbf{e}) : \forall\alpha.\tau}$$

$$\begin{aligned}\mathscr{E}[(\Lambda\alpha.\mathbf{e})\langle\tau\rangle]_M &\to \mathscr{E}[\mathbf{e}[\tau/\alpha]] \\ \mathscr{E}[(\mathit{MSG}^{\forall\alpha.\tau}\ \mathsf{v})]_M &\to \mathscr{E}[(\Lambda\alpha.(\mathit{MSG}^{\tau}\ \mathsf{v}))] \\ \mathscr{E}[(\mathit{GSM}^{\forall\alpha.\tau}\ \mathbf{v})]_S &\to \mathscr{E}[(\mathit{GSM}^{\tau[\mathbf{L}/\alpha]}\ \mathbf{v}\langle\mathbf{L}\rangle)] \\ \mathscr{E}[(\mathit{GSM}^{\mathbf{L}}\ (\mathit{MSG}^{\mathbf{L}}\ \mathsf{v}))]_S &\to \mathscr{E}[\mathsf{v}]\end{aligned}$$

Fig. 11. Extensions to figure 4 for non-parametric polymorphism

---

exception to continue propagating upwards, possibly being handled by a different language from the language than the one that raised it.

THEOREM 22. *Exceptions system 2 is type-sound.*

PROOF. The proof of theorem 1, *mutatis mutandis*. □

6. POLYMORPHISM

One of the deficiencies of the simply-typed lambda calculus as an ML stand-in is that it has no notion of type abstraction. To address that, we show here one method of adding the type abstraction features of Girard's System F [1989] to the simple natural embedding from figure 4. To ML we add type abstractions, written $(\Lambda\alpha.\mathbf{e})$, and type application, written $\mathbf{e}\langle\tau\rangle$, and types $\forall\alpha.\tau$ and α.

Our embedding converts Scheme functions that work polymorphically into polymorphic ML values, and converts ML type abstractions directly into plain Scheme functions that behave polymorphically. For example, ML might receive the Scheme function $(\lambda \mathsf{x}.\mathsf{x})$ from a boundary with type $\forall\alpha.\alpha \to \alpha$ and use it successfully as an identity function, and Scheme might receive the ML type abstraction $(\Lambda\alpha.\lambda\mathbf{x} : \alpha.\mathbf{x})$ as a regular function that behaves as the identity function for any value Scheme gives it.

To support this behavior, the model must create a type abstraction from a regular Scheme value when converting from Scheme to ML, and must drop a type abstraction when converting from ML to Scheme. The former is straightforward: we reduce a redex of the form $(\mathit{MSG}^{\forall\alpha.\tau}\mathsf{v})$ by dropping the $\forall$ quantifier on the type in the boundary and binding the now-free type variable in τ by wrapping the entire expression in a Λ form, yielding $(\Lambda\alpha.(\mathit{MSG}^{\tau}\mathsf{v}))$.

To convert a polymorphic ML value to a Scheme value, we need to remove its initial type-abstraction by applying it to some type; then we need to recursively convert the result. As for which type to apply it to, we need a type to which we can reliably convert any Scheme value, though it need not expose any of those values' properties. In section 2, we used the lump type to represent arbitrary, opaque Scheme values in ML; we reuse it here as the argument to the ML type abstraction. More specifically, we add $\mathbf{L}$ as a new base type in ML and we add the cancellation rule for lumps to the set of reductions: these changes, along with all the other additions required to support polymorphism, are summarized in figure 11. (We have left the relation $\Gamma \vdash \tau$: type undefined; it simply checks that all the free type variables in τ are bound in Γ. Technically, all of the existing type rules that involve type terms must also perform this check.)

THEOREM 23. *The polymorphic natural embedding is type-sound.*

PROOF. The proof of theorem 1, *mutatis mutandis*. □

6.1 Restoring parametricity

Although this embedding is type safe, the polymorphism is not parametric in the sense of Reynolds [1983]. We can see this with an example: it is well-known that in System F, for which parametricity holds, the only value with type $\forall\alpha.\alpha \to \alpha$ is the polymorphic identity function. In the system we have built so far, though, the term

$$(MSG^{\forall\alpha.\alpha\to\alpha}(\lambda \mathsf{x}.(\texttt{if0}\ (\texttt{nat?}\ \mathsf{x})\ (+\ \mathsf{x}\ \overline{1})\ \mathsf{x})))$$

has type $\forall\alpha.\alpha \to \alpha$ but when applied to the type ι evaluates to

$$(\lambda \mathbf{y}.(MSG^{\iota}((\lambda \mathsf{x}.(\texttt{if0}\ (\texttt{nat?}\ \mathsf{x})\ (+\ \mathsf{x}\ \overline{1})\ \mathsf{x})(GSM^{\iota}\mathbf{y})))))$$

Since the argument to this function is always a number, this is equivalent to

$$(\lambda \mathbf{y}.(MSG^{\iota}((\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))(GSM^{\iota}\mathbf{y}))))$$

which is well-typed but is not the identity function.

The problem with the misbehaving $\forall\alpha.\alpha \to \alpha$ function above is that while the type system rules out ML fragments that try to treat values of type α non-genercially, it still allows Scheme programs to observe the concrete choice made for α and act accordingly. To restore parametricity, we borrow the idea of dynamic sealing from Pierce and Sumii's work on encryption as information-hiding [Pierce and Sumii 2000; Sumii and Pierce 2004]: whenever we would pass Scheme a value whose type was originally a type variable, instead of converting that value according to the rules of the natural embedding we provide Scheme with an opaque sealed value about which Scheme cannot make any observations. When Scheme produces an α result, the conversion rules open the box and hand its results back to ML.

We turn this intuition into a system in figure 12. To do so we need to generalize our framework slightly: up to this point, we have annotated each boundary with the ML type expected of it, but for this system that strategy does not suffice. If ML expects a result of type ι from a boundary, it might be because the boundary had type ι in the original source program, or it might be because it had type α in the source program and during the program's run α was instantiated to ι; the term's ML type is identical in these two cases, but what happens at run time needs to vary. So, we need to generalize the annotation on each boundary so that rather than containing a *type*, it contains a *conversion scheme* from which we can derive a type when necessary using the $\lfloor\ \rfloor$ metafunction. We use the metavariable κ to represent conversion schemes, and in this case the conversion schemes are given by:

$$\kappa = \iota \mid \kappa_1 \to \kappa_2 \mid \forall\alpha.\kappa \mid \alpha \mid \mathbf{L} \mid < \beta;\tau >$$

All these are identical to their type counterparts except the last, $< \beta;\tau >$, which represents a type variable that has been bound to type τ (β is a set of type variables that we can

$$
\begin{aligned}
\mathbf{e} &= \cdots \mid (\Lambda\alpha.\mathbf{e}) \mid \mathbf{e}\langle\tau\rangle \mid (MSG^{\kappa}\ \mathsf{e})\\
\mathsf{e} &= \cdots \mid (GSM^{\kappa}\ \mathbf{e})\\
\mathbf{v} &= \cdots \mid (MSG^{\mathbf{L}}\ \mathsf{v})\\
\mathsf{v} &= \cdots \mid (GSM^{<\beta;\tau>}\ \mathbf{v})\\
\tau &= \cdots \mid \forall\alpha.\tau \mid \alpha \mid \mathbf{L}\\
\kappa &= \iota \mid \kappa_1 \to \kappa_2 \mid \forall\alpha.\kappa \mid \alpha \mid \mathbf{L} \mid <\beta;\tau>
\end{aligned}
$$

$$
\frac{\Gamma \vdash_M \mathbf{e} : \forall\alpha.\tau' \quad \Gamma \vdash \tau : \text{type}}{\Gamma \vdash_M \mathbf{e}\langle\tau\rangle : \tau'[\tau/\alpha]} \quad \frac{\Gamma, \alpha \text{ type} \vdash_M \mathbf{e} : \tau}{\Gamma \vdash_M (\Lambda\alpha.\mathbf{e}) : \forall\alpha.\tau}
$$

$$
\frac{\Gamma \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_M (MSG^{\kappa}\ \mathsf{e}) : \lfloor\kappa\rfloor} \quad \frac{\Gamma \vdash_M \mathbf{e} : \lfloor\kappa\rfloor}{\Gamma \vdash_S (GSM^{\kappa}\ \mathbf{e}) : \mathbf{TST}}
$$

$$
\begin{aligned}
\mathscr{E}[(GSM^{\forall\alpha.\tau}\ \mathbf{v})]_S &\to \mathscr{E}[(GSM^{\tau[\mathbf{L}/\alpha]}\ \mathbf{v}\langle\mathbf{L}\rangle)]\\
\mathscr{E}[(GSM^{\mathbf{L}}\ (MSG^{\mathbf{L}}\ \mathsf{v}))]_S &\to \mathscr{E}[\mathsf{v}]
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{E}[(\Lambda\alpha.\mathbf{e})\langle\tau\rangle]_M &\to \mathscr{E}[\mathbf{e}[<\beta;\tau>/\alpha]] \text{ (where } \beta \text{ is fresh)}\\
\mathscr{E}[(MSG^{\forall\alpha.\kappa}\ \mathsf{v})]_M &\to \mathscr{E}[(\Lambda\alpha.(MSG^{\kappa}\ \mathsf{v}))]\\
\mathscr{E}[(MSG^{<\beta;\tau>}\ (GSM^{<\beta;\tau>}\ \mathbf{v}))]_M &\to \mathscr{E}[\mathbf{v}]\\
\mathscr{E}[(MSG^{<\beta;\tau>}\ \mathsf{v})]_M &\to \mathbf{Error}\text{: bad value}\\
&\qquad \text{(where } \mathsf{v} \neq GSM^{<\beta;\tau>}\ \mathbf{v} \text{ for any } \mathbf{v})
\end{aligned}
$$

$$
\begin{aligned}
\lfloor\ \rfloor &: \quad \kappa \to \tau\\
\lfloor\iota\rfloor &= \iota\\
\lfloor\kappa_1 \to \kappa_2\rfloor &= \lfloor\kappa_1\rfloor \to \lfloor\kappa_2\rfloor\\
\lfloor\forall\alpha.\kappa\rfloor &= \forall\alpha.\lfloor\kappa\rfloor\\
\lfloor\alpha\rfloor &= \alpha\\
\lfloor\mathbf{L}\rfloor &= \mathbf{L}\\
\lfloor<\beta;\tau>\rfloor &= \tau
\end{aligned}
$$

Fig. 12. Modifications to figure 4 to support parametric polymorphism

---

imagine being drawn from the same set as α, though in principle the two sets need not be related in any particular way).

With these conversion schemes defined, we can give a more satisfactory model for polymorphism. Figure 12 has several changes: first, we extend the definition of Scheme values to includes suitably-wrapped ML values that were originally of polymorphic types. Second, the type-application reduction rule now makes up a fresh β with which to "brand" polymorphic values and substitutes in branded types rather than substituting types directly; to support substituting conversion schemes rather than types in this context, we must extend the notion of substitution so that when a conversion scheme κ would be substituted into a location that must syntactically be a type, we insert the type $\lfloor\kappa\rfloor$ instead. Finally, a new cancellation rule for ML accepts values of polymorphic type from Scheme only if they have the appropriate stamp. Another key is what we have *not* written: Scheme has no elimination rules for sealed values other than the boundary rule, which means Scheme has no way to get at the contents of abstract values other than through means provided explicitly by ML. In this way the system can enforce abstraction constraints directly at runtime.

THEOREM 24. *The embedding presented in figure 12 is type-sound.*

PROOF. The proof of theorem 1, *mutatis mutandis*. □

Under this scheme, the nonparametric term we presented above is not rejected, but becomes parametric. The term

$$(MSG^{\forall\alpha.\alpha\to\alpha}(\lambda\mathsf{x}.(\mathtt{if0}\ (\mathtt{nat?}\ \mathsf{x})\ (+\ \mathsf{x}\ \overline{1})\ \mathsf{x})))$$

applied to ι now reduces to

$$(MSG^{<\beta;\iota>\to<\beta;\iota>}(\lambda\mathsf{x}.(\mathtt{if0}\ (\mathtt{nat?}\ \mathsf{x})\ (+\ \mathsf{x}\ \overline{1})\ \mathsf{x})))$$

and then to

$$(\lambda\mathbf{y}:\iota.(MSG^{<\beta;\iota>}\ ((\lambda\mathsf{x}.(\mathtt{if0}\ (\mathtt{nat?}\ \mathsf{x})\ (+\ \mathsf{x}\ \overline{1})\ \mathsf{x})(GSM^{<\beta;\iota>}\ \mathbf{y})))))$$

Even though the argument to the ML function as a whole is always a number, so regardless of the fact that their contents are in fact numeric, Scheme's `nat?` function will not recognize them as numbers and thus this function is equivalent to

$$(\lambda\mathbf{y}:\iota.(MSG^{<\beta;\iota>}\ (GSM^{<\beta;\iota>}\ \mathbf{y})))$$

which is in turn equivalent to

$$(\lambda\mathbf{y}:\iota.\ \mathbf{y})$$

the identity function for numbers.

The identity function is a useful first example, since it is very simple and illustrates how our strategy works, but it is not entirely satisfactory for at least two reasons. First, there are only three possible values with that type — the function that always diverges, the function that always signals an error, and the identity function — so it does not illustrate parametric types that allow an infinite, but still constrained, set of values. Second, and relatedly, it does not include the interesting case where a polymorphic value receives from its context functions that allow it to receive or return polymorphic values at intermediate stages in its computation.

For these reasons, we think Church numerals are richer illustration of our strategy. The parametricity property is sufficient to show that all System F values that have type $\forall\alpha.(\alpha\to\alpha)\to\alpha\to\alpha$ represent Church numerals, *i.e.* functions that apply their first argument some constant number of times to their second argument and return the result, independent of those arguments' actual values (so, for instance, a value of that type cannot apply some functions four times and other functions five times). In the presence of effects, this property is a little messier, since for instance a term could take function argument $\mathbf{v}_f$ and value argument $\mathbf{v}_0$, evaluate $(\mathbf{v}_f^{12}\ \mathbf{v}_0)$ (that is, $\mathbf{v}_f$ applied to $\mathbf{v}_0$ 12 times successively) for effect and then return, say, $(\mathbf{v}_f^{3}\ \mathbf{v}_0)$. A term with this behavior could diverge when given $(\lambda\mathbf{x}:\iota.\ (\mathtt{if0}\ (-\ \mathbf{x}\ \overline{12})\ \Omega\ (+\ \mathbf{x}\ \overline{1})))$ and $\overline{0}$ (where Ω is some diverging term) but produce $\overline{3}$ when given $(\lambda\mathbf{x}:\iota.\ (+\ \mathbf{x}\ \overline{1}))$ and $\overline{0}$. Nonetheless, it is still parametric in the sense that it behaves completely generically with respect to the interpretation of α; the new behaviors come from the fact that the arguments themselves can behave in richer ways.

We sketch an argument that the same holds in the language of figure 12; or more precisely, that in the language of figure 12, for each closed term $\mathbf{c} = MSG^{\forall\alpha.(\alpha\to\alpha)\to\alpha\to\alpha}\ \mathsf{v}$, either

(1) For all τ and $\mathbf{v}_f : \tau \to \tau$, $(\mathbf{c}\langle\tau\rangle\ \mathbf{v}_f) \to^* \mathbf{Error}$: s for some s, or
(2) For all τ and $\mathbf{v}_f : \tau \to \tau$, $(\mathbf{c}\langle\tau\rangle\ \mathbf{v}_f)$ diverges, or
(3) There exists some term t such that $\mathbf{c} \simeq t$ where t has the form

$$\begin{aligned} t = \Lambda\alpha.\ \lambda\mathbf{f} : \alpha \to \alpha.\ \lambda\mathbf{x}_0 : \alpha.& \\ &\mathbf{let}\ \mathbf{x}_1 : \alpha = \mathbf{f}\ \mathbf{x}_0\ \mathbf{in} \\ &\ldots \\ &\mathbf{let}\ \mathbf{x}_m : \alpha = \mathbf{f}\ \mathbf{x}_{m-1}\ \mathbf{in} \\ &\mathit{result} \end{aligned}$$

where *result* is either some identifier $\mathbf{x}_n$ for $n \le m$, a closed expression that always raises **Error**: s, or a closed, nonterminating expression (and $\mathbf{let}\ \mathbf{x} : \tau = \mathbf{e}_1\ \mathbf{in}\ \mathbf{e}_2$ is syntactic sugar for $((\lambda\mathbf{x} : \tau.\mathbf{e}_2)\ \mathbf{e}_1)$ as usual).

We start by choosing an arbitrary evaluation context $\mathscr{E}[]_M$, embedded Scheme function $\mathbf{t}$ as above, type τ, and ML values $\mathbf{v}_f$ and $\mathbf{v}_0$. Our plan is to show that for any such choice, the reduction sequence formed by the application $\mathscr{E}[\mathbf{c}\langle\tau\rangle\ \mathbf{v}_f\ \mathbf{v}_0]_M$ will fall into one of the three enumerated cases. To begin:

$$\begin{aligned} \mathscr{E}[(\mathbf{c}\langle\tau\rangle\ \mathbf{v}_f\ \mathbf{v}_0)]_M = &\ \mathscr{E}[((MSG^{\forall\alpha.(\alpha\to\alpha)\to\alpha\to\alpha}\ \mathsf{v})\langle\tau\rangle\ \mathbf{v}_f\ \mathbf{v}_0)]_M \\ \to &\ \mathscr{E}[((MSG^{(<\beta;\tau>\to<\beta;\tau>)\to<\beta;\tau>\to<\beta;\tau>}\ \mathsf{v})\ \mathbf{v}_f\ \mathbf{v}_0)] \end{aligned}$$

If v is not a function value, this term reduces to an error and thus the first case of the enumeration holds. Otherwise it reduces again to

$$\begin{aligned} &\mathscr{E}[((\lambda\mathbf{f} : \tau \to \tau.\ (MSG^{<\beta;\tau>\to<\beta;\tau>}\ (\mathsf{v}\ (GSM^{<\beta;\tau>\to<\beta;\tau>}\ \mathbf{f}))))\ \mathbf{v}_f\ \mathbf{v}_0)] \\ \to\ &\mathscr{E}[((MSG^{<\beta;\tau>\to<\beta;\tau>}\ (\mathsf{v}\ (GSM^{<\beta;\tau>\to<\beta;\tau>}\ \mathbf{v}_f)))\ \mathbf{v}_0)] \\ \to\ &\mathscr{E}[((MSG^{<\beta;\tau>\to<\beta;\tau>}\ (\mathsf{v}\ (\lambda \mathsf{x}.GSM^{<\beta;\tau>}\ (\mathbf{v}_f\ (MSG^{<\beta;\tau>}\ \mathsf{x})))))\ \mathbf{v}_0)] \end{aligned}$$

Now either v applies the wrapped version of $\mathbf{v}_f$ that it receives to some value, or it does not. If it does, then the entire program will abort regardless of $\mathbf{v}_f$, because regardless of v's particulars it cannot forge new values with the shape $(GSM^{<\beta;\tau>}\mathbf{v})$: every type application creates a fresh β that does not appear anywhere else in the program, so in particular v can neither contain a pre-existing copy of β nor duplicate it during its run. In this case, then, the first case of the enumeration holds.

If, on the other hand, v does not ever apply its wrapped copy of $\mathbf{v}_f$ to an argument, then it must behave the same way (and if it terminates produce the same result, modulo embedded calls to $\mathbf{v}_f$) for any $\mathbf{v}_f$. The term may either always signal an error or always diverges independent of our choice for $\mathbf{v}_f$, in which case either the first or second item of the enumeration holds. The other possiblity is that it reduces to another function value v' that may contain embedded copies of $\mathbf{v}_f$ under the restriction that different choices of $\mathbf{v}_f$ lead to v' instances that differ only in their embedded copies of $\mathbf{v}_f$. In this case:

$$\begin{aligned} &\mathscr{E}[((MSG^{<\beta;\tau>\to<\beta;\tau>}\ (\mathsf{v}\ (\lambda \mathsf{x}.GSM^{<\beta;\tau>}\ (\mathbf{v}_f\ (MSG^{<\beta;\tau>}\ \mathsf{x})))))\ \mathbf{v}_0)] \\ \to^*\ &\mathscr{E}[((MSG^{<\beta;\tau>\to<\beta;\tau>}\ \mathsf{v}')\ \mathbf{v}_0)] \\ \to\ &\mathscr{E}[((\lambda\mathbf{x} : \tau.\ (MSG^{<\beta;\tau>}\ (\mathsf{v}'\ (GSM^{<\beta;\tau>}\ \mathbf{x}))))\ \mathbf{v}_0)] \\ \to\ &\mathscr{E}[(MSG^{<\beta;\tau>}\ (\mathsf{v}'\ (GSM^{<\beta;\tau>}\ \mathbf{v}_0)))] \end{aligned}$$

Now either the term eventually reduces to the form

$$\mathscr{E}[\mathbf{E}'[((\lambda \mathsf{x}.\ (GSM^{<\beta;\tau>}\ (\mathbf{v}_f\ (MSG^{<\beta;\tau>}\ \mathsf{x}))))\ (GSM^{<\beta;\tau>}\ \mathbf{v}_0))]]$$

for some nested ML evaluation context $\mathbf{E}'[\ ]_S$, or it does not. If it does not, then it falls under the last case of the enumeration with $m = 0$ either because it always diverges, always produces an error, or always returns $\mathbf{v}_0$. If it does, then

$$\begin{aligned} & \mathscr{E}[\mathbf{E}'[((\lambda \mathsf{x}.\ (GSM^{<\beta;\tau>}\ (\mathbf{v}_f\ (MSG^{<\beta;\tau>}\ \mathsf{x}))))\ (GSM^{<\beta;\tau>}\ \mathbf{v}_0))]] \\ \rightarrow\ & \mathscr{E}[\mathbf{E}'[(GSM^{<\beta;\tau>}\ (\mathbf{v}_f\ (MSG^{<\beta;\tau>}\ (GSM^{<\beta;\tau>}\ \mathbf{v}_0))))]] \\ \rightarrow\ & \mathscr{E}[\mathbf{E}'[(GSM^{<\beta;\tau>}\ (\mathbf{v}_f\ \mathbf{v}_0))]] \end{aligned}$$

This application may either diverge, produce an error, or produce a value $\mathbf{v}_1$. If $(\mathbf{v}_f\ \mathbf{v}_0)$ produces a value then the term as a whole continues to reduce; applying the same line of reasoning we know that the reduction sequence either always produces an error, always produces $\mathbf{v}_0$, always produces $\mathbf{v}_1$, or again reduces to an application of its wrapped version of $\mathbf{v}_f$ one of the wrapped results it has discovered so far ($(GSM^{<\beta;\tau>}\ \mathbf{v}_i)$ for $i = 0$ or 1), independent of the actual values of $\mathbf{v}_f$, $\mathbf{v}_0$, and $\mathbf{v}_1$. More generally we see that the only values that can ever sealed with β in terms in the reduction sequence are the values $\mathbf{v}_i$ where i is some natural number and $\mathbf{v}_i$ is the result of applying $\mathbf{v}_f$ to $\mathbf{v}_{i-1}$; that if $\mathbf{v}_f$ fails to terminate or raises an error when applied any of $\mathbf{v}_0$ through $\mathbf{v}_i$ then the entire term also fails to terminate or raises an error; and that the term must exhibit this behavior generically (i.e. regardless of the particular choices of $\mathbf{v}_f$ and $\mathbf{v}_0 \cdots \mathbf{v}_i$) because at no point in the reduction sequence does it have the opportunity to make a choice based on their particular values. After discovering some number i of sealed values, the term may return any of them, so long as it does so generically (or it may always diverge or always produce an error). Suppose it chooses to return $(GSM^{<\beta;\tau>}\ \mathbf{v}_j)$. Then the entire term we have been considering becomes

$$\begin{aligned} & \mathscr{E}[(MSG^{<\beta;\tau>}\ (GSM^{<\beta;\tau>}\ \mathbf{v}_j))] \\ \rightarrow\ & \mathscr{E}[\mathbf{v}_j] \end{aligned}$$

Now consider the term

$$\begin{aligned} t = (\Lambda\alpha.\ \lambda\mathbf{f} : \alpha \rightarrow \alpha.\ \lambda\mathbf{x}_0 : \alpha.\ & \\ & \mathbf{let}\ \mathbf{x}_1 : \alpha = (\mathbf{f}\ \mathbf{x}_0)\ \mathbf{in} \\ & \ldots \\ & \mathbf{let}\ \mathbf{x}_i : \alpha = (\mathbf{f}\ \mathbf{x}_{i-1})\ \mathbf{in} \\ & \mathbf{x}_j) \end{aligned}$$

It is easy to check that $\mathscr{E}[(t\langle\tau\rangle\ \mathbf{v}_f\ \mathbf{v}_0)]$ fails to terminate, signals an error, or produces $\mathscr{E}[\mathbf{v}_j]$ under the same circumstances. That exhausts all the possibilities; taken together, all of these cases imply that every term $\mathbf{c}$ behaves as some (generalized, potentially-effectful) Church numeral.

Examples like these suggest to us that this system is likely to preserve parametricity, but we have not been able to establish that claim formally. Instead we make a few observations. First, this problem is a more structured variant of the problem Sumii and Pierce have investigated in the context of enforcing polymorphism through information hiding [Pierce and Sumii 2000; Sumii and Pierce 2003], and establishing parametricity in this setting appears to correspond to an open problem raised by their work [Pierce and Sumii 2000, section 4]. Second, it appears to be very similar to the notion of information-hiding among principals [Zdancewic et al. 1999; Grossman et al. 2000], and the weaker information-hiding results established there should apply to our setting as well.

Finally, owing to the connection between multi-language systems and contracts discussed in section 3, we can use the same technique presented here to implement parametrically polymorphic contracts even in untyped languages such as Scheme. To do so, we implement a contract of the form $\forall\alpha.c$ as a term that generates a new unique name n every time it is evaluated, instances of α in negative positions in c become constructors that seal their arguments with n, and instances of α in positive positions in c become unboxing operations that check to make sure they receive packages labeled with n.

7. FROM TYPE-DIRECTED TO TYPE-MAPPED CONVERSION

In this section we generalize the framework we have built so far to allow boundaries to perform conversions that are not strictly type-directed. We can use the notion of conversion schemes for more than just establishing parametric polymorphism; in general, we can use it to support systems where conversion is not driven entirely by an expression's the static types. For instance, figure 13 shows a variation on the natural embedding in which Scheme and ML both have string values (and ML gives them type Σ) and Scheme also has a separate category of file-system path values, convertible to and from strings by built-in Scheme functions *string→path* and *path→string* (which we model in figure 13 by appealing to unspecified mappings *sp* for string to path conversion and *ps* for path to string conversion; the former is partial to capture the idea that not all strings correspond to valid paths). A reasonable foreign function interface might want ML programs to be able to provide strings to Scheme programs that expect paths, but that conversion would not be type-directed.

We allow for this more permissive kind of multilanguage system by generalizing boundaries to conversion strategies as in section 6. In this case, we can define the conversion strategy κ as in figure 13 and generalize boundaries to be of the form GSM^{κ} and MSG^{κ}, where ι and Σ mean to apply the straightforward base-value conversion to values at the boundary, and π means to convert strings to paths (for a GSM^{π} boundary) or paths to strings (for an MSG^{π} boundary). The reduction rules use these annotations to decide whether to introduce a call to *string→path* when passing an ML string into Scheme.

THEOREM 25. *The language of figure 13 is type-sound.*

PROOF. The proof of theorem 1, *mutatis mutandis*. □

While the language in figure 13 only affects value conversions, conversion strategies can impact control flow as well. For instance, since C does not have an exception mechanism, many C functions (e.g., `malloc`, `fopen`, `sqrt`) return a normal value on success and a sentinel value to indicate an error. A foreign function interface might automatically convert error indicators from such functions into exceptions, while converting non-errors as normal. We can model that choice by combining one of they exception-handling systems of section 5 with a conversion strategy that distinguishes "zero for error" functions from regular functions.

To make this more concrete, we give a system in which ML has no exception mechanism and some functions that return numbers use the zero-for-error convention, and Scheme has an exception mechanism. A Scheme exception cannot propagate through a boundary (i.e., it aborts the program, as in section 5.1) unless that boundary is a "zero for error" boundary, in which case it is represented as a zero in ML.

$$\begin{array}{rcl}
\mathbf{e} &=& \cdots \mid (\mathit{MSG}^{\kappa}\ \mathsf{e}) \mid \mathbf{s} \\
\mathsf{e} &=& \cdots \mid (\mathit{GSM}^{\kappa}\ \mathbf{e}) \mid \mathsf{s} \mid \mathsf{p} \mid \mathit{path{\rightarrow}string} \mid \mathit{string{\rightarrow}path} \\
\mathbf{v} &=& \cdots \mid \mathbf{s} \\
\mathsf{v} &=& \cdots \mid \mathsf{s} \mid \mathsf{p} \mid \mathit{path{\rightarrow}string} \mid \mathit{string{\rightarrow}path} \\
\mathbf{s}, \mathsf{s} &=& \text{character strings} \\
\mathsf{p} &=& \text{file system paths} \\
\\
\mathbf{E} &=& \cdots \mid (\mathit{MSG}^{\kappa}\ \mathsf{E}) \\
\mathsf{E} &=& \cdots \mid (\mathit{GSM}^{\kappa}\ \mathbf{E}) \\
\\
\tau &=& \cdots \mid \Sigma \\
\kappa &=& \iota \mid \Sigma \mid \pi \mid \kappa \rightarrow \kappa
\end{array}$$

$$\frac{}{\Gamma \vdash_M \mathbf{s} : \Sigma} \quad \frac{\Gamma \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_M (\mathit{MSG}^{\kappa}\ \mathsf{e}) : \lfloor \kappa \rfloor} \quad \frac{\Gamma \vdash_M \mathbf{e} : \lfloor \kappa \rfloor}{\Gamma \vdash_S (\mathit{GSM}^{\kappa}\ \mathbf{e}) : \mathbf{TST}}$$

$$\begin{array}{rcl}
\lfloor \iota \rfloor &=& \iota \\
\lfloor \Sigma \rfloor &=& \Sigma \\
\lfloor \pi \rfloor &=& \Sigma \\
\lfloor \kappa_1 \rightarrow \kappa_2 \rfloor &=& \lfloor \kappa_1 \rfloor \rightarrow \lfloor \kappa_2 \rfloor
\end{array}$$

$$\begin{array}{rcl}
\mathit{sp} &:& \text{unspecified partial map from } \mathsf{s} \text{ to } \mathsf{p} \\
\mathit{ps} &:& \text{unspecified total map from } \mathsf{p} \text{ to } \mathsf{s}
\end{array}$$

$$\begin{array}{rcll}
\mathscr{E}[(\mathit{path{\rightarrow}string}\ \mathsf{p})]_S &\rightarrow& \mathscr{E}[\mathit{ps}(\mathsf{p})] & \\
\mathscr{E}[(\mathit{string{\rightarrow}path}\ \mathsf{s})]_S &\rightarrow& \mathscr{E}[\mathit{sp}(\mathsf{s})] & (\text{where } \mathit{sp}(\mathsf{s}) \text{ is defined}) \\
\mathscr{E}[(\mathit{string{\rightarrow}path}\ \mathsf{s})]_S &\rightarrow& \mathscr{E}[(\mathtt{wrong}\ \text{“bad path”})] & (\text{where } \mathit{sp}(\mathsf{s}) \text{ is undefined}) \\
\mathscr{E}[(\mathit{GSM}^{\Sigma}\ \mathbf{s})]_S &\rightarrow& \mathscr{E}[\mathsf{s}] & (\text{where } \mathsf{s} = \mathbf{s}) \\
\mathscr{E}[(\mathit{GSM}^{\pi}\ \mathbf{s})]_S &\rightarrow& \mathscr{E}[(\mathit{path{\rightarrow}string}\ \mathsf{s})] & (\text{where } \mathsf{s} = \mathbf{s}) \\
\mathscr{E}[(\mathit{MSG}^{\Sigma}\ \mathsf{s})]_M &\rightarrow& \mathscr{E}[\mathbf{s}] & (\text{where } \mathbf{s} = \mathsf{s}) \\
\mathscr{E}[(\mathit{MSG}^{\pi}\ \mathsf{p})]_M &\rightarrow& \mathscr{E}[(\mathit{MSG}^{\Sigma}(\mathit{path{\rightarrow}string}\ \mathsf{p}))] &
\end{array}$$

Fig. 13. Extensions to figure 4 for mapped embedding 1

The core of the system, as before, is a conversion strategy and associated $\lfloor\ \rfloor$ metafunction; these are presented in figure 14 along with the other necessary extensions. The conversion strategy adds a single conversion, $\iota!$, indicating a number where $\overline{0}$ indicates an error. The evaluation contexts and reduction rules for ML are just like those of the natural embedding (since this version of ML cannot handle exceptions), and the Scheme evaluation contexts are as in section 5.1. ML's typing judgments are also just as in section 5.1, adapted to use the $\lfloor\kappa\rfloor$ conversion as necessary. Reducing a boundary is just as it was before, with additions corresponding to the ML-to-Scheme and Scheme-to-ML conversions for values at $\iota!$ boundaries.

THEOREM 26. *The language of figure 14 is type-sound.*

$$
\begin{aligned}
\mathbf{e} &= \cdots \mid (MSG^{\kappa}\ \mathsf{e}) \\
\mathsf{e} &= \cdots \mid (GSM^{\kappa}\ \mathbf{e}) \mid (\texttt{handle}\ \mathsf{e}\ \mathsf{e}) \\
\\
\mathbf{E} &= \cdots \mid (MSG^{\kappa}\ \mathsf{E}) \\
\mathsf{H}, \mathsf{F} &\ \text{as in figures 9 and 10} \\
\mathsf{E} &= \mathsf{F} \mid \mathsf{F}[GSM^{\kappa}\mathbf{E}]_S \\
\\
\kappa &= \iota \mid \iota! \mid \kappa \to \kappa
\end{aligned}
$$

$$
\frac{\Gamma \vdash_S \mathsf{e} : \mathbf{TST}}{\Gamma \vdash_M (MSG^{\kappa}\ \mathsf{e}) : \lfloor \kappa \rfloor}
\qquad
\frac{\Gamma \vdash_M \mathbf{e} : \lfloor \kappa \rfloor}{\Gamma \vdash_S (GSM^{\kappa}\ \mathbf{e}) : \mathbf{TST}}
$$

$$
\begin{aligned}
\lfloor \iota \rfloor &= \iota \\
\lfloor \iota! \rfloor &= \iota \\
\lfloor \kappa_1 \to \kappa_2 \rfloor &= \lfloor \kappa_1 \rfloor \to \lfloor \kappa_2 \rfloor
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{E}[(MSG^{\iota!}\ \overline{n})]_M &\to \mathscr{E}[\overline{n}] \\
\mathscr{E}[(MSG^{\iota!}\ \mathsf{H}[(\texttt{wrong}\ \text{str})]_S)]_M &\to \mathscr{E}[\overline{0}] \\
\mathscr{E}[(GSM^{\iota!}\ \overline{0})]_S &\to \mathscr{E}[(\texttt{wrong}\ \text{“zero”})] \\
\mathscr{E}[(GSM^{\iota!}\ \overline{n})]_S &\to \mathscr{E}[\overline{n}] \text{where } n \neq 0
\end{aligned}
$$

Fig. 14. Extensions to figure 4 for mapped embedding 2

PROOF. The proof of theorem 1, *mutatis mutandis*. □

These examples demonstrate a larger point: although we have used a boundary's type as its conversion strategy for most of the systems in this paper, they are separate ideas. Decoupling them has a number of pleasant effects: first, it allows us to use non-type-directed conversions, as we have shown. Second, the separation illustrates that type erasure still holds in all of the systems we have considered so long as we do not also erase conversion strategies — for all of the systems we have presented in this paper, one can make an easy argument by induction that as long as boundaries remain annotated, then we can erase other type annotations without effect on a program's evaluation. Finally, the separation makes it easier to understand the connection between these formal systems and tools like SWIG, in particular SWIG's type-map feature [Beazley 1997]: from this perspective, SWIG is a tool that automatically generates boundaries that pull C++ values into Python (or another high-level language), and type-maps allow the user to write a new conversion strategy and specify the circumstances under which it should be used.

8. RELATED WORK

This paper is an extension of a conference paper of the same title [Matthews and Findler 2007]. We have extended that work in two ways. First, we have shown how to use our technique to scale up standard proof techniques such as subject-reduction, contextual reasoning, and logical relations proofs to multi-language systems. Second, we have applied our technique to more language features, including polymorphism and an extended treatment of exceptions.

The work most directly related to ours is the type-indexed embedding and projection technique described by both Ramsey [2003] and Benton [2005], which can be thought of as an implementation of the natural embedding we describe in section 3. Ramsey and Benton only consider the asymmetric case where a typed host language embeds an untyped language, and focus on implementation rather than formal techniques. Still, readers will find that the flavor of their work is quite similar to this work. Zdancewic, Grossman, and Morrisett's work [1999] is also similar to ours in that it introduces two-agent calculi and boundaries; their work, however, focuses on information-hiding properties and does not allow different languages to interoperate.

Other related work at the semantic level tends to focus on the properties of multi-language runtime systems. This includes a pair of formalisms for COM [Pucella 2002; Ibrahim and Szyperski 1997] and also Gordon and Syme's formalization of a type-safe intermediate language designed for multi-language interoperation [2001]. Kennedy [2006] pointed out that in multi-language systems, observations in one language can break equations in the other and that this is a practical problem. Our system is a way to reason about these breaks precisely. Trifonov and Shao have developed an abstract intermediate language for multi-language programs that aids reasoning about interactions between effects in the two source languages [Trifonov and Shao 1999]. While the present work also addresses effects, we do not address their implementation; our work focuses their semantics as seen by the source languages, a topic Trifonov and Shao do not discuss. Finally, Furr and Foster have built a system for verifying certain safety properties of the OCaml foreign-function interface by analyzing C code for problematic uses of OCaml values [Furr and Foster 2005].

On the issue of combining typed and untyped code, Henglein and Rehof [Henglein and Rehof 1995; Henglein 1994] have done work on translating Scheme into ML, inserting ML equivalents of our guards to simulate Scheme's dynamic checks. Some languages have introduced ways of mixing typed and untyped code using type `dynamic` [Abadi et al. 1991], which are similar to our boundaries and lumps; for instance Cardelli's Amber [1986], Chambers *et al*'s Cecil [Chambers and The Cecil Group 2004] or Gray, Findler and Flatt's ProfessorJ [2005].

There has been far too much implementation work connecting high-level languages to list it all here. In addition to the projects we have already mentioned, there are dozens of compilers that target the JVM, the .NET CLR, or COM. There have also been more exotic embeddings; two somewhat recent examples are an embedding of Alice (an SML extension) into the Oz programming language [Kornstaedt 2001] and the LazyScheme educational embedding of a lazy variant of Scheme into strict-evaluation-order Scheme [Barzilay and Clements 2005]. There has been even more work on connecting high-level languages to low-level languages; in addition to the venerable SWIG [Beazley 1996] there are many more systems that try to sanitize the task of connecting C code to a high-level, functional programming system [Fisher et al. 2001; Blume 2001; Barzilay and Orlovsky 2004].

9. CONCLUSION

We have shown how to give operational semantics to multilanguage systems and still leverage the same formal techniques that apply to single languages. This work has focused on two points in the design space for interoperating languages: the lump and natural embeddings. We see aspects of these two points in many real foreign function interfaces: for instance, SML.NET translates flat values and .NET objects, but forces lump-like behavior for higher-order ML functions. The Java Native Interface provides all foreign values as lumps, but provides a large set of constant functions for manipulating them, similar to our **fa** ("foreign-apply") function from section 2. The lump embedding's ease of implementing and natural embedding's ease of use both pull on language design, and most real multi-language systems lie in between.

Furthermore, we have talked mostly about foreign function interfaces in this paper, but there are many other examples of multi-language systems. Any situation in which two logically different languages interact is a candidate for this treatment — *e.g.* embedded domain-specific languages, some uses of metaprogramming, and even contract systems, viewing each party to the contract as a separate language. By treating these uniformly as multi-language systems, we might be able to connect them in fruitful ways. Even with the limited scope considered in this paper, we have discovered connections between contracts, foreign function interfaces, and hybrid type systems (as we discussed in section 3.3).

We have implemented all the formal systems presented in this paper as PLT Redex programs [Matthews et al. 2004]. They are available online at `http://www.cs.uchicago.edu/~jacobm/papers/multilang/`.

REFERENCES

ABADI, M., CARDELLI, L., PIERCE, B., AND PLOTKIN, G. 1991. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems 13,* 2 (April), 237–268.

BARZILAY, E. AND CLEMENTS, J. 2005. Laziness without all the hard work. In *Workshop on Functional and Declarative Programming in Education (FDPE).*

BARZILAY, E. AND ORLOVSKY, D. 2004. Foreign interface for PLT Scheme. In *Workshop on Scheme and Functional Programming*.

BEAZLEY, D. 1996. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *4th Tcl/Tk Workshop*. 129–139. Available online: `http://www.swig.org/papers/Tcl96/tcl96.html`.

BEAZLEY, D. 1997. Pointers, constraints, and typemaps. In SWIG 1.1 Users Manual. Available online: `http://www.swig.org/Doc1.1/HTML/Typemaps.html`.

BENTON, N. 2005. Embedded interpreters. *Journal of Functional Programming 15*, 503–542.

BENTON, N. AND KENNEDY, A. 1999. Interlanguage working without tears: Blending SML with Java. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 126–137.

BENTON, N., KENNEDY, A., AND RUSSO, C. V. 2004. Adventures in interoperability: the SML.NET experience. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*. 215–226.

BLUME, M. 2001. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL).*

CARDELLI, L. 1986. Amber. In *Combinators and functional programming languages*, G. Cousineau, P.-L. Curien, and B. Robinet, Eds. Vol. 242. Springer-Verlag.

CHAKRAVARTY, M. M. T. 2002. The Haskell 98 foreign function interface 1.0. Available online: `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`.

CHAMBERS, C. AND THE CECIL GROUP. 2004. The Cecil language: Specification and rationale, version 3.2. Tech. rep., Department of Computer Science and Engineering, University of Washington. February. Available online: `http://www.cs.washington.edu/research/projects/cecil/pubs/cecil-spec.html`.

FELLEISEN, M. 1991. On the expressive power of programming languages. *Science of Computer Programming 17*, 35–75.

FELLEISEN, M., FRIEDMAN, D., KOHLBECKER, E., AND DUBA, B. 1987. A syntactic theory of sequential control. 205–237.

FELLEISEN, M. AND HIEB, R. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 102*, 235–271. Original version in: Technical Report 89-100, Rice University, June 1989.

FINDLER, R. B. AND BLUME, M. 2006. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming (FLOPS)*.

FINDLER, R. B. AND FELLEISEN, M. 2002. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

FINNE, S., LEIJEN, D., MEIJER, E., AND PEYTON JONES, S. 1999. Calling Hell from Heaven and Heaven from Hell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 114–125.

FISHER, K., PUCELLA, R., AND REPPY, J. 2001. A framework for interoperability. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*.

FLANAGAN, C. 2006. Hybrid type checking. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

FURR, M. AND FOSTER, J. S. 2005. Checking type safety of foreign function calls. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 62–72.

GIRARD, J.-Y., LAFONT, Y., AND TAYLOR, P. 1989. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge University Press.

GORDON, A. D. AND SYME, D. 2001. Typing a multi-language intermediate code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 248–260.

GRAY, K. E., FINDLER, R. B., AND FLATT, M. 2005. Fine grained interoperability through mirrors and contracts. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*.

GROSSMAN, D., MORRISETT, G., AND ZDANCEWIC, S. 2000. Syntactic type abstraction. *22*, 1037–1080.

HENGLEIN, F. 1994. Dynamic typing: Syntax and proof theory. *Science of Computer Programming 22,* 3, 197–230.

HENGLEIN, F. AND REHOF, J. 1995. Safe polymorphic type inference for a dynamically typed language: translating Scheme to ML. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA)*.

IBRAHIM, R. AND SZYPERSKI, C. 1997. The COMEL language. Tech. Rep. FIT-TR-97-06, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia.

KENNEDY, A. 2006. Securing the .NET programming model. *Theoretical Computer Science 364,* 3 (Nov.), 311–317. Available online: `http://research.microsoft.com/~akenn/sec/`.

KORNSTAEDT, L. 2001. Alice in the land of Oz - an interoperability-based implementation of a functional language on top of a relational language. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*.

MASON, I. AND TALCOTT, C. 1991. Equivalence in functional languages with effects. *Journal of Functional Programming 1*, 287–327.

MATTHEWS, J. AND FINDLER, R. B. 2005. An operational semantics for R5RS Scheme. In *Workshop on Scheme and Functional Programming*.

MATTHEWS, J. AND FINDLER, R. B. 2007. Operational semantics for multi-language programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

MATTHEWS, J., FINDLER, R. B., FLATT, M., AND FELLEISEN, M. 2004. A visual environment for developing context-sensitive term rewriting systems. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA)*.

MEIJER, E., PERRY, N., AND VAN YZENDOORN, A. 2001. Scripting .NET using Mondrian. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, London, UK, 150–164.

MEUNIER, P. AND SILVA, D. 2003. From Python to PLT Scheme. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*. 24–29.

ODERSKY, M., ALTHERR, P., CREMET, V., EMIR, B., MICHELOUD, S., MIHAYLOV, N., SCHINZ, M., STENMAN, E., AND ZENGER, M. 2005. An Introduction to Scala. `http://scala.epfl.ch/docu/files/ScalaIntro.pdf`.

OHORI, A. AND KATO, K. 1993. Semantics for communication primitives in an polymorphic language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 99–112.

PIERCE, B. AND SUMII, E. 2000. Relating cryptography and polymorphism. Available online: `http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/`.

PIERCE, B. C. 2002. *Types and Programming Languages*. The MIT Press.

PINTO, P. 2003. Dot-Scheme: A PLT Scheme FFI for the .NET framework. In *Workshop on Scheme and Functional Programming*.

PLOTKIN, G. D. 1977. LCF considered as a programming language. *Theoretical Computer Science*, 223–255.

PUCELLA, R. 2002. Towards a formalization for COM, part I: The primitive calculus. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*.

RAMSEY, N. 2003. Embedding an interpreted language using higher-order functions and types. In *Interpreters, Virtual Machines and Emulators (IVME '03)*. 6–14.

REYNOLDS, J. C. 1983. Types, abstraction and parametric polymorphism. In *IFIP Congress*. 513–523.

SABRY, A. AND FELLEISEN, M. 1993. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*.

STECKLER, P. 1999. MysterX: A Scheme toolkit for building interactive applications with COM. In *Technology of Object-Oriented Languages and Systems (TOOL)*. 364–373. Available online: `citeseer.ist.psu.edu/steckler99mysterx.html`.

SUMII, E. AND PIERCE, B. 2003. Logical relations for encryption. *Journal of Computer Security (JSC) 11,* 4, 521–554. Available online: `http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/`.

SUMII, E. AND PIERCE, B. 2004. A bisimulation for dynamic sealing. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

TAIT, W. 1967. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic 32,* 2 (June), 198–212.

TRIFONOV, V. AND SHAO, Z. 1999. Safe and principled language interoperation. In *European Symposium on Programming (ESOP)*. 128–146.

WRIGHT, A. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Information and Computation*, 38–94. First appeared as Technical Report TR160, Rice University, 1991.

ZDANCEWIC, S., GROSSMAN, D., AND MORRISETT, G. 1999. Principals in programming languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.