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.NET, F#, 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 pre-existing libraries written in 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 Lan-

guage 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 a 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 5, we argue that our technique can model more realistic languages by adding two different exception 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 [Felleisen and Hieb 1992].

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

$$
\frac{}{\Gamma, \mathbf{x} : \tau \vdash_M \mathbf{x} : \tau} \qquad \frac{\Gamma, \mathbf{x} : \tau_1 \vdash_M \mathbf{e} : \tau_2}{\Gamma \vdash_M (\lambda \mathbf{x} : \tau_1.\ \mathbf{e}) : \tau_1 \to \tau_2} \qquad \frac{}{\Gamma \vdash_M \overline{n} : \mathbf{Nat}}
$$

$$
\frac{\Gamma \vdash_M \mathbf{e}_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_M \mathbf{e}_2 : \tau_1}{\Gamma \vdash_M (\mathbf{e}_1\ \mathbf{e}_2) : \tau_2} \qquad \frac{\Gamma \vdash_M \mathbf{e}_1 : \mathbf{Nat} \quad \Gamma \vdash_M \mathbf{e}_2 : \mathbf{Nat}}{\Gamma \vdash_M (op\ \mathbf{e}_1\ \mathbf{e}_2) : \mathbf{Nat}}
$$

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

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

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

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

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

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

$$
\frac{\Gamma \vdash_S \mathsf{e}_1 : \mathbf{TST} \qquad \Gamma \vdash_S \mathsf{e}_2 : \mathbf{TST} \qquad \Gamma \vdash_S \mathsf{e}_3 : \mathbf{TST}}{\Gamma \vdash_S (\texttt{if0}\ \mathsf{e}_1\ \mathsf{e}_2\ \mathsf{e}_3) : \mathbf{TST}}
$$

$$
\frac{\Gamma \vdash_S \mathsf{e}_1 : \mathbf{TST}}{\Gamma \vdash_S (\mathit{pr}\ \mathsf{e}_1) : \mathbf{TST}} \qquad \frac{}{\Gamma \vdash_S (\texttt{wrong}\ \mathit{str}) : \mathbf{TST}}
$$

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

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.

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

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

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

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

---

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, which are 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 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 **Nat** 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").

[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.

In our lump embedding extension, we add a new type $\mathbf{L}$ (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 $\mathbf{L}$ as one might expect. If $\tau = \mathbf{L}$ we are sending a native Scheme value 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).[2] We also use a third, unspecified evaluation context ($\mathscr{E}$) for the definitions of the rewriting rules (whose precise role with respect to those rules is explained below). Intuitively, $\mathscr{E}$ corresponds to "whole program" evaluation contexts. A "whole program" is necessarily either a term in ML that might contain calls into Scheme or a term in Scheme that might contain calls into ML; in either case, even though a reduction step compute some intermediate value using the foreign language's rules, it cannot change whether the overall program in which that computation takes place is written in ML or Scheme. The formal reflection of that fact is that while a single program's reduction sequence might involve reduction rules drawn from both ML and Scheme, the overall evaluation context in which those rules apply will never change; it will either be an ML context for the entire sequence, or a Scheme context for the entire sequence. Since we have arbitrarily chosen to model ML programs that can call into Scheme (rather than Scheme programs that call into ML), our programs begin and end with ML and we take $\mathscr{E} = \mathbf{E}$ 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:

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

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. To keep things clear, we also introduce two different notations for holes: $[\,]_M$ for holes in which an ML term is expected, and $[\,]_S$ for holes in which a Scheme term is expected.

[2] We have chosen, for both of the languages we are combining, to follow a call-by-value evaluation order; if we wanted, say, our ML language to follow call-by-name evaluation order we could simply change the definition of its evaluation contexts (and its corresponding function application rule) to yield a working system. That choice is orthogonal to the rest of the development of this paper.

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. The primary problem is that the "interesting parts" of the rules have different domains and ranges: ML rules rewrite ML to ML, and Scheme rules rewrite Scheme to Scheme or a distinguished error state. To define single reduction rule for a multi-language system we must standardize on set of terms for the reduction relation to rewrite. In particular, since $\mapsto$ rewrites whole programs to whole programs, and we have decided that whole programs are written in ML, all of the relation's cases must rewrite ML terms to ML terms. Furthermore since Scheme reductions may abort the entire program with an error, the range of the final type must include the distinguished error marker **Error**: str. Concretely, to make Scheme rules operate on ML programs, we write Scheme's rules to apply to Scheme terms in arbitrary *top-level* ($\mathscr{E}$) contexts. For symmetry, we give the same treatment to ML terms.

A few examples may help further clarify why the top-level context is necessary for Scheme and ML rules. Consider the ML context

$$(\texttt{if0}\ [\,]_M\ \mathbf{e}_1\ \mathbf{e}_2)$$

This is a legal ML evaluation context to which we might expect ML reduction rules to apply. But if we plug $(^{\mathbf{Nat}}MS\,[\,]_S)$ into the hole, we get the new context

$$(\texttt{if0}\ (^{\mathbf{Nat}}MS\,[\,]_S)\ \mathbf{e}_1\ \mathbf{e}_2)$$

This is still an ML evaluation context, not a Scheme evaluation context; it just happens that its hole is a Scheme hole instead of an ML hole. Since we want Scheme reduction rules to apply to terms in this hole, we must therefore allow Scheme reduction rules to apply to ML contexts with Scheme holes.

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 $(SM^{\mathbf{L}}\ \mathbf{v})$ expressions will in fact be lump values, so the $SM^{\mathbf{L}}$ 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 $^{\tau}MS$ 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, imagine

how a very simple foreign interface between ML and Scheme might actually look. Suppose we have the following Scheme library:

```
(define add1 (lambda (x) (+ x 1)))
(define a-constant 3)
```

An ML program might interact with these definitions using a built-in `getSchemeValue` function and make foreign function calls using a `foreignApply` function, like this:

```
val schemeAdd1  : Scheme = getSchemeValue("add1")
val schemeConst : Scheme = getSchemeValue("a-constant")
val resl        : Scheme = foreignApply(schemeAdd1, schemeConst)
```

We can model this situation as follows, using boundaries to allow ourselves to write down the Scheme computations and using $\mathbf{L}$ for the `Scheme` type:[3]

$$
\begin{array}{rl}
 & (\mathbf{let}\ ((\mathbf{fa} : \mathbf{L} \to \mathbf{L} \to \mathbf{L}.\ (\lambda \mathbf{f} : \mathbf{L}.\ \lambda \mathbf{x} : \mathbf{L}.\ ({}^{\mathbf{L}}MS\ ((SM^{\mathbf{L}}\ \mathbf{f})\ (SM^{\mathbf{L}}\ \mathbf{x}))))) \\
 & \quad (\mathbf{fa}\ ({}^{\mathbf{L}}MS\ (\lambda \mathsf{x}.\ (+\ \mathsf{x}\ \overline{1})))) \\
 & \qquad ({}^{\mathbf{L}}MS\ \overline{3})) \\
\mapsto & ((\lambda \mathbf{f} : \mathbf{L}.\ \lambda \mathbf{x} : \mathbf{L}.\ ({}^{\mathbf{L}}MS\ ((SM^{\mathbf{L}}\ \mathbf{f})\ (SM^{\mathbf{L}}\ \mathbf{x}))))\ \\
 & \quad ({}^{\mathbf{L}}MS\ (\lambda \mathsf{x}.\ (+\ \mathsf{x}\ \overline{1})))\ \\
 & \quad ({}^{\mathbf{L}}MS\ \overline{3})) \\
\mapsto^2 & {}^{\mathbf{L}}MS\ ((SM^{\mathbf{L}}({}^{\mathbf{L}}MS(\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1})))(SM^{\mathbf{L}}\ ({}^{\mathbf{L}}MS\ \overline{3})))) \\
\mapsto & {}^{\mathbf{L}}MS\ ((\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))\ (SM^{\mathbf{L}}\ ({}^{\mathbf{L}}MS\ \overline{3})))) \\
\mapsto & {}^{\mathbf{L}}MS\ ((\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))\ \overline{3}) \\
\mapsto & {}^{\mathbf{L}}MS\ (+\ \overline{3}\ \overline{1}) \\
\mapsto & {}^{\mathbf{L}}MS\ \overline{4}
\end{array}
$$

In the initial term of this reduction sequence, we define **fa** (for "foreign-apply") using just the built-in boundaries of our model as an ML function that takes two foreign values, applies the first to the second in Scheme, and returns the result as a foreign value. We model the two uses of `getSchemeValue` as direct uses of ML-to-Scheme boundaries, the first containing a Scheme add-one function and the second 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 $(SM^{\mathbf{L}}\ ({}^{\mathbf{L}}MS\ \mathsf{v}))$ 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 $\overline{4}$.

If we try to apply an add-one function written directly in ML 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:

$$
\begin{array}{rl}
 & ({}^{\mathbf{Nat}}MS\ ((SM^{\mathbf{Nat}\to\mathbf{Nat}}(\lambda \mathbf{x} : \mathbf{Nat}.(+\ \mathbf{x}\ \overline{1})))\ \overline{3})) \\
\mapsto & ({}^{\mathbf{Nat}}MS\ (\mathtt{wrong}\ \text{“non-procedure”})) \\
\mapsto & \mathbf{Error}\text{: non-procedure}
\end{array}
$$

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 type system cannot protect us from

[3] In this example, we use **let** as a binding form though it is not in our language. We can expand it to the core language we have defined so far using the "left-left-lambda" encoding $(\mathbf{let}\ ((f : \tau.\ \mathbf{e}_1))\ \mathbf{e}_2) = ((\lambda f : \tau.\ \mathbf{e}_2)\ \mathbf{e}_1)$.

mistakes such as $(^{\mathbf{Nat}\to\mathbf{Nat}}MS\ (\lambda\times.(+\ \times\ \overline{1})))\ \overline{3}$, though. In this case ML expects Scheme to produce a lump containing an ML value of type **Nat** → **Nat**; since Scheme produces a native Scheme value instead, the term immediately steps to a runtime error.

The formulation of the lump embedding in Figure 3 allows us to prove type soundness using the standard technique of preservation and progress. In this statement, notice we permit the ML program **e** to reduce to an error even though a pure-ML program can never do so; this is because a Scheme subterm within the ML term might cause an error itself. The fact that Scheme is the only source of errors is implicit in this formulation, though, because the only rules that actually rewrite a term into the explicit **Error**: str form we use are Scheme reductions.

THEOREM 1. *If* $\Gamma \vdash_M \boldsymbol{e} : \tau$*, then either* $\boldsymbol{e} \mapsto^* \boldsymbol{v}$*,* $\boldsymbol{e} \mapsto^* \mathbf{Error}$*: 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* $\Gamma' \vdash_S e' : \boldsymbol{TST}$*, then for all terms* e'' *where* $\Gamma' \vdash_S e'' : \boldsymbol{TST}$*,* $\Gamma \vdash_M \boldsymbol{e}[e'/e''] : \tau$.

Given these we can show preservation:

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

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

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

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 $\mathscr{E}[(^{\tau'}MS\ \mathsf{v})]_M \mapsto \mathscr{E}[(^{\tau'}MS\ (\texttt{wrong}\ \text{“Bad value”}))]$ 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 $\mathscr{E}[(SM^{\mathbf{L}}\ (^{\mathbf{L}}MS\ \mathsf{v}))]_S \mapsto \mathscr{E}[\mathsf{v}]$

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* $\mathscr{E}[\boldsymbol{e}] \mapsto \boldsymbol{e}'$ *or* $\mathscr{E}[\boldsymbol{e}] \mapsto$ **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* $\mathscr{E}[\mathsf{e}] \mapsto \boldsymbol{e}'$ *or* $\mathscr{E}[\mathsf{e}] \mapsto$ **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 $\dfrac{\vdash_S \mathsf{e} : \mathbf{TST}}{\vdash_M (\tau \boldsymbol{MS}\ \mathsf{e}) : \tau}$:

We must show that either $(\tau \boldsymbol{MS}\ \mathsf{e})$ is a value or that for an arbitrary top-level evaluation context $\mathscr{E}[\,]_M$, $\mathscr{E}[(\tau \boldsymbol{MS}\ \mathsf{e})]$ 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 $\tau \boldsymbol{MS}$ 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 $\mathscr{E}[(\boldsymbol{SM}^{\tau}\ [\,]_S)]$ is an ML evaluation context with a Scheme hole; thus $\mathscr{E}[(\boldsymbol{SM}^{\tau}\ \mathsf{e})]$ 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:

$$\begin{array}{rcl} \mathscr{E}[\![(SM_N^{\mathbf{Nat}}\,\overline{n})]\!]_S & \mapsto & \mathscr{E}[\overline{n}] \\ \mathscr{E}[\![({}^{\mathbf{Nat}}MS_N\,\overline{n})]\!]_M & \mapsto & \mathscr{E}[\overline{n}] \end{array}$$

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 — for a concrete example involving paths and strings see Section 7.

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 $\tau_1 \mapsto \tau_2$ 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, $({}^{\tau_1\mapsto\tau_2}MS_N\,\lambda\mathsf{x}.\mathsf{e})$ 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

$$\begin{array}{rcl} \mathbf{e} & = & \cdots \mid ({}^{\tau}MSG\ \mathsf{e}) \\ \mathsf{e} & = & \cdots \mid (GSM^{\tau}\ \mathbf{e}) \end{array}$$

$$\begin{array}{rcl} \mathbf{E} & = & \cdots \mid ({}^{\tau}MSG\ \mathsf{E}) \\ \mathsf{E} & = & \cdots \mid (GSM^{\tau}\ \mathbf{E}) \end{array}$$

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

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

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^{τ}_N and ${}^{\tau}MS_N$ constructs to our core languages from Figures 1 and 2, we instead add "guarded" versions GSM^{τ} and ${}^{\tau}MSG$ (for "guard, then SM boundary" and "MS boundary, then guard") shown in Figure 4. These rules translate values in the same way that SM^{τ}_N 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:

$$\begin{array}{rl} & ((^{\mathbf{Nat}\to\mathbf{Nat}}MSG\ (\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1})))\ \overline{3}) \\ \mapsto & ((\lambda \mathbf{x}' : \mathbf{Nat}.{}^{\mathbf{Nat}}MSG\ ((\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))\ (GSM^{\mathbf{Nat}}\ \mathbf{x}')))\ \overline{3}) \\ \mapsto & ({}^{\mathbf{Nat}}MSG\ ((\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))\ (GSM^{\mathbf{Nat}}\ \overline{3}))) \\ \mapsto & ({}^{\mathbf{Nat}}MSG\ ((\lambda \mathsf{x}.(+\ \mathsf{x}\ \overline{1}))\ \overline{3})) \\ \mapsto & ({}^{\mathbf{Nat}}MSG\ (+\ \overline{3}\ \overline{1})) \\ \mapsto & ({}^{\mathbf{Nat}}MSG\ \overline{4}) \\ \mapsto & \overline{4} \end{array}$$

ML converts the Scheme add-one function to an ML function with type $\mathbf{Nat} \to \mathbf{Nat}$ 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 num-

ber $\overline{3}$, 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:

$$(^{(\mathbf{Nat}\to\mathbf{Nat})\to\mathbf{Nat}}MS_N\ (\lambda \mathtt{f}.\,(\mathtt{if0}\ (\mathtt{f}\ \overline{1})\ \overline{2}\ \mathtt{f})))$$

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 $(\mathbf{Nat} \to \mathbf{Nat}) \to \mathbf{Nat}$, and wraps it to the ML value

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

Whenever this value is applied to a function, that function is converted to a Scheme value at type $\mathbf{Nat} \to \mathbf{Nat}$ and the result is converted from Scheme to ML at type **Nat**. Thus conversion in either direction works at a given type if it works in *both* directions at all smaller types. In this case, if f returns 0, then the surrounding `if0` will return $\overline{2}$, which can be converted to the ML number $\overline{2}$ and the program can continue. If f returns a nonzero value, then the surrounding `if0` will return f itself which cannot be converted to a number. Thus the boundary will signal an error, halting the program.

THEOREM 5. *If* $\vdash_M \boldsymbol{e} : \tau$*, then either* $\boldsymbol{e} \mapsto^* \boldsymbol{v}$*,* $\boldsymbol{e} \mapsto^*$ **Error***: str, or* $\boldsymbol{e}$ *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 it couples the conceptually-unrelated tasks of converting values from one language to another and signalling errors. In this subsection and the next, we pull these two tasks apart and show that the boundaries we have defined implicitly contain Findler-Felleisen style higher-order contracts.

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 $(\mathscr{G}^{\tau}\ \mathtt{e})$ 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: $({}^{\tau}MS_N\ (\mathscr{G}^{\tau}\ \mathtt{e}))$ instead of $({}^{\tau}MSG\ \mathtt{e})$ and $(\mathscr{G}^{\tau}\ (SM_N^{\tau}\ \mathbf{e}))$ instead of $(GSM^{\tau}\ \mathbf{e})$.

Figure 5 shows the rules for guards. An **Nat** guard applied to a number reduces to that number, and the same guard applied to a procedure aborts the program. A $\tau_1 \to \tau_2$ 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 ${}^{\tau}MSG$ and unguarded boundaries with separated guards ${}^{\tau}MS_N$, SM_N^{τ}, and

$$\begin{array}{rcl}\mathbf{e} & = & \cdots \mid ({}^{\tau}MS_N\ \mathsf{e}) \\ \mathsf{e} & = & \cdots \mid (\mathscr{G}^{\tau}\ \mathsf{e}) \mid (SM_N^{\tau}\ \mathbf{e})\end{array}$$

$$\begin{array}{rcl}\mathbf{E} & = & \cdots \mid ({}^{\tau}MS_N\ \mathsf{E}) \\ \mathsf{E} & = & \cdots \mid (\mathscr{G}^{\tau}\ \mathsf{E}) \mid (SM_N^{\tau}\ \mathbf{E})\end{array}$$

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

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

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

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

---

$\mathscr{G}^{\tau}$; *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 observationally 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 $\vdash_M \mathbf{e} : \tau$,

$$\begin{array}{rcll}eval_N(\mathbf{e}) & = & \mathtt{proc} & \text{if } \mathbf{e} \mapsto^* \lambda \mathbf{x} : \tau.\mathbf{e} \\ eval_N(\mathbf{e}) & = & n & \text{if } \mathbf{e} \mapsto^* \overline{n} \\ eval_N(\mathbf{e}) & = & \mathbf{Error}\text{: } s & \text{if } \mathbf{e} \mapsto^* \mathbf{Error}\text{: } s\end{array}$$

THEOREM 6. *$eval_N()$ is a partial function.*

PROOF. That $eval_N()$ maps an input to at most one output follows from unique decomposition (Lemma 7 below). That it is not total follows from the existence of infinite reduction chains in Scheme. □

DEFINITION 1. *An* instruction *is a term that appears syntactically on the left-hand side of the $\mapsto$ reduction relation's definition in Figures 1 and 2.*

LEMMA 7.

—For any term $\mathbf{e}$, there exists at most one context $\mathbf{E}$ and instruction i such that $\mathbf{e} = \mathbf{E}[i]$.

—For any term e, there exists at most one context E and instruction i such that $\mathsf{e} = \mathsf{E}[i]$.

PROOF. By simultaneous induction on the structure of $\mathbf{e}$ and $\mathbf{e}$. □

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:

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

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. Notice that this formulation of contextual equivalence allows us to distinguish between different kinds of errors, and thus we believe is finer-grained than the more typical formulation which only allows observation of termination versus non-termination. The reason is that we have given our language no programmatic way to manipulate errors, so there is no way to write a program that intercepts errors and loops on receiving error message "a" but halts on receiving error message "b". Nonetheless we believe our notion of observational equivalence is the right one to use for this system, since even though a *program* cannot distinguish between two error messages we intend for a *user* to be able to do so (as the error message is printed onscreen, for example).

THEOREM 8. *For all Scheme expressions* e *and ML expressions* $\mathbf{e}$*, the following propositions hold:*

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

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 9. *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 [Mason and Talcott 1991].

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

LEMMA 10. *If for all evaluation contexts* $\mathscr{E}[\,]$, $\mathscr{E}[e_1] \mapsto^* \mathscr{E}[e_1']$ *(uniquely) and* $\mathscr{E}[e_2] \mapsto^* \mathscr{E}[e_2']$ *(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, $\mathscr{E}'[e_1] \mapsto^* \mathscr{E}'[e_1']$ and $\mathscr{E}'[e_2] \mapsto^* \mathscr{E}'[e_2']$, 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 9, 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 $\tau = \mathbf{Nat}$: For (1): if v is the number $\overline{n}$:

$$\mathscr{E}[(^{\mathbf{Nat}}MSG\,\overline{n})] \mapsto \mathscr{E}[\overline{n}]$$
$$\mathscr{E}[(^{\mathbf{Nat}}MS_N\ (\mathscr{G}^{\mathbf{Nat}}\ \overline{n}))] \mapsto \mathscr{E}[(^{\mathbf{Nat}}MS_N\ \overline{n})] \mapsto \mathscr{E}[\overline{n}]$$

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

$$\mathscr{E}[(^{\mathbf{Nat}}MSG\,(\lambda \mathsf{x}.\mathsf{e}'))] \mapsto \mathscr{E}[(^{\mathbf{Nat}}MSG\,(\texttt{wrong}\ \text{“Non-number”}))] \mapsto \textbf{Error}\text{: Non-number}$$
$$\mathscr{E}[(^{\mathbf{Nat}}MS_N\,(\mathscr{G}^{\mathbf{Nat}}\,(\lambda \mathsf{x}.\mathsf{e}')))] \mapsto \mathscr{E}[(^{\mathbf{Nat}}MS_N\,(\texttt{wrong}\ \text{“Non-number”}))] \mapsto \textbf{Error}\text{: Non-number}$$

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

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

For the right-hand side:

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

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

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

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

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 8, 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 8.

COROLLARY 11. *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* $elab_M(\boldsymbol{e}) \mapsto^* \boldsymbol{v}$, $elab_M(\boldsymbol{e}) \mapsto^*$ **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 8, $elab_M(\mathbf{e})$ 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 $\mathbf{e}$ have no guards in corollary 11 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 8. 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

$$(^{\mathbf{Nat}\to\mathbf{Nat}}MS_N\ (\mathscr{G}^{\mathbf{Nat}\to\mathbf{Nat}}\ (\lambda \mathsf{x}.\mathsf{x})))$$

reduces to

$$(\lambda \mathbf{x} : \mathbf{Nat}.(^{\mathbf{Nat}}MS_N(\mathscr{G}^{\mathbf{Nat}}\ ((\lambda \mathsf{x}.\mathsf{x})\ (\mathscr{G}^{\mathbf{Nat}}\ (SM_N^{\mathbf{Nat}}\ \mathbf{x}))))))$$

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: Scheme-to-ML guards, written $\mathscr{G}_{MS}^{\tau}$, and ML-to-Scheme guards, written $\mathscr{G}_{SM}^{\tau}$. Their reduction rules are:

$$\begin{array}{lcl} \mathscr{E}[(\mathscr{G}_{MS}^{\mathbf{Nat}}\ \overline{n})]_S & \mapsto & \mathscr{E}[\overline{n}] \\ \mathscr{E}[(\mathscr{G}_{MS}^{\mathbf{Nat}}\mathsf{v})]_S & \mapsto & \mathscr{E}[(\texttt{wrong}\ \text{“Non-number”})]\ (\mathsf{v} \neq \overline{n} \text{ for any } n) \\ \mathscr{E}[(\mathscr{G}_{MS}^{\tau_1\to\tau_2}\ (\lambda \mathsf{x}.\mathsf{e}))]_S & \mapsto & \mathscr{E}[\mathscr{G}_{MS}^{\tau_2}\ ((\lambda \mathsf{x}.\mathsf{e})\ (\mathscr{G}_{SM}^{\tau_1}\mathsf{x}))] \\ \mathscr{E}[(\mathscr{G}_{MS}^{\tau_1\to\tau_2}\mathsf{v})]_S & \mapsto & \mathscr{E}[(\texttt{wrong}\ \text{“Non-function”})]\ (\mathsf{v} \neq (\lambda \mathsf{x}.\mathsf{e}) \text{ for any } \mathsf{x}, \mathsf{e}) \\ \\ \mathscr{E}[(\mathscr{G}_{SM}^{\mathbf{Nat}}\mathsf{v})]_S & \mapsto & \mathscr{E}[\mathsf{v}] \\ \mathscr{E}[(\mathscr{G}_{SM}^{\tau_1\to\tau_2}\mathsf{v})]_S & \mapsto & \mathscr{E}[(\lambda \mathsf{x}.(\mathscr{G}_{SM}^{\tau_2}(\mathsf{v}(\mathscr{G}_{MS}^{\tau_1}\mathsf{x}))))] \end{array}$$

The function cases are the interesting rules here. Since functions that result from Scheme-to-ML guards are bound for ML, we check the inputs that ML will supply them using an ML-to-Scheme guard; since the result of those functions will be Scheme values going to ML, they must be guarded with Scheme-to-ML guards. ML-to-Scheme guards never directly signal an error; they exist only to protect ML functions from erroneous Scheme inputs. They put Scheme-to-ML guards on the arguments to ML functions but use ML-to-Scheme guards on their results because those values will have come from ML. Because Scheme-to-ML guards perform a first-order check that may signal an error, we also call them "positive" guards in reference to their potential for positive blame in the sense of higher-order contracts. Since ML-to-Scheme guards do not have such potential, but could give rise to negative blame, we also call them negative guards.

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

$$(1)\ \ ({}^{\tau}MSG\,\mathsf{e}) \simeq ({}^{\tau}MS_N\,(\mathscr{G}^{\tau}_{MS}\,\mathsf{e}))$$
$$(2)\ \ (GSM^{\tau}\,\boldsymbol{e}) \simeq (\mathscr{G}^{\tau}_{SM}\,(SM^{\tau}_N\,\boldsymbol{e}))$$

PROOF. As the proof of Theorem 8, *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 13. *If* $\vdash_M \boldsymbol{e} : \tau$ *where* $\boldsymbol{e}$ *is a term in the natural, separated guard language, then either* $elab_{C,M}(\boldsymbol{e}) \mapsto^* \boldsymbol{v}$*,* $elab_{C,M}(\boldsymbol{e}) \mapsto^*$ **Error***: str, or* $elab_{C,M}(\boldsymbol{e})$ *diverges.*

PROOF. Combine Theorem 12 and Theorem 5 as in the proof of corollary 11. □

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

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

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

THEOREM 14. *For all* e *and* τ*, both of the following hold:*

$$(1)\ \ \mathsf{G}^{\tau}_{MS}\,\mathsf{e} \simeq \mathscr{G}^{\tau}_{MS}\,\mathsf{e}$$
$$(2)\ \ \mathsf{G}^{\tau}_{SM}\,\mathsf{e} \simeq \mathscr{G}^{\tau}_{SM}\,\mathsf{e}$$

PROOF. As in the proof of Theorem 8, we show equivalence by induction on τ where e is a value and the term appears in an evaluation context; this suffices to prove the theorem as stated.

Case $\tau = \mathbf{Nat}$:

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

$$\begin{array}{l}\mathscr{E}[(\mathtt{G}^{\mathbf{Nat}}_{MS}\,\mathsf{v})]\\ \mathscr{E}[((\lambda \mathsf{x}.\\ \quad (\mathtt{if0}\ (\mathtt{nat?}\ \mathsf{x})\\ \quad\quad \mathsf{x}\\ \quad\quad (\mathtt{wrong}\ \text{“non-number”})))\\ \quad \mathsf{v})]\\ \mapsto\ \mathscr{E}[(\mathtt{if0}\ (\mathtt{nat?}\ \mathsf{v})\ \mathsf{v}\ (\mathtt{wrong}\ \text{“Non-number”}))]\end{array}$$

If v is a natural number, then:

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

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

$$\begin{array}{l}\mathscr{E}[(\mathtt{if0}\ (\mathtt{nat?}\ \mathsf{v})\\ \quad\quad \mathsf{v}\\ \quad\quad (\mathtt{wrong}\ \text{“Non-number”}))]\\ \mapsto\ \mathscr{E}[\mathtt{wrong}\ \text{“Non-number”}]\end{array}$$

and $\mathscr{E}[(\mathscr{G}^{\mathbf{Nat}}_{MS}\mathsf{v})] \mapsto \mathscr{E}[\mathtt{wrong}\ \text{“Non-number”}]$. 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 $\tau = \tau_1 \to \tau_2$:

For (1): Assume $\mathscr{E}$ is some evaluation context where $\mathscr{E}[\mathsf{e}] \mapsto^* \mathscr{E}[\mathsf{v}]$. Then

$$\begin{array}{lll}\mathscr{E}[(\mathtt{G}^{\tau_1\to\tau_2}_{MS}\ \mathsf{e})] & \mapsto^* & \mathscr{E}[(\mathtt{G}^{\tau_1\to\tau_2}_{MS}\ \mathsf{v})]\\ \mathscr{E}[(\mathscr{G}^{\tau_1\to\tau_2}_{MS}\ \mathsf{e})] & \mapsto^* & \mathscr{E}[(\mathscr{G}^{\tau_1\to\tau_2}_{MS}\ \mathsf{v})]\end{array}$$

The former then reduces as follows:

$$\begin{array}{l}\mathscr{E}[\mathtt{G}^{\tau_1\to\tau_2}_{MS}\ \mathsf{v}]\\ \mathscr{E}[((\lambda \mathsf{x}.\\ \quad (\mathtt{if0}\ (\mathtt{proc?}\ \mathsf{x})\\ \quad\quad (\lambda \mathsf{y}.\ (\mathtt{G}^{\tau_2}_{MS}\ (\mathsf{x}\ (\mathtt{G}^{\tau_1}_{SM}\ \mathsf{y}))))\\ \quad\quad (\mathtt{wrong}\ \text{“Non-procedure”})))\\ \quad\ \mathsf{v})]\\ \mapsto\ \mathscr{E}[(\mathtt{if0}\ (\mathtt{proc?}\ \mathsf{v})\\ \quad\quad (\lambda \mathsf{y}.\ (\mathtt{G}^{\tau_2}_{MS}\ (\mathsf{v}\ (\mathtt{G}^{\tau_1}_{SM}\ \mathsf{y}))))\\ \quad\quad (\mathtt{wrong}\ \text{“Non-procedure”}))]\end{array}$$

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

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

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

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

Fig. 6. Translation functions for lump values

(where y is not free in v), and

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

(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 $\mathtt{G}^{\tau}_{MS}$ and $\mathtt{G}^{\tau}_{SM}$ are a pair of projections in the sense of Findler and Blume [2006] where the two parties are ML and Scheme and we have prior knowledge from ML's type system that it always upholds its end of the contract, and thus we do not need to enforce the checks associated with that end at run-time. 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; the lump embedding by contrast seems much more limited. Thus we might suspect that the natural embedding allow us to write more programs. In fact the exact opposite is true: the natural-embedding ${}^{\tau}MS_N$ and SM^{τ}_N 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, $\mathscr{T}^{\tau}_{SM}$ and $\mathscr{T}^{\tau}_{MS}$, that can be written in the lump embedding. These functions translate values whose ML type is τ from ML to Scheme and from Scheme to ML, respectively; they are shown in Figure 6. (In that figure and below, for clarity we use the notation ${}^{\tau}MS_L$ and SM^{τ}_L 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, numbers in our case.

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:

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

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 15. *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:*

$$\begin{array}{rcl}(GSM^{\tau}\boldsymbol{e}) & \simeq & \mathscr{T}^{\tau}_{SM}\,(\mathtt{G}^{\tau}_{SM}\,(SM^{\tau}_{L}\,\boldsymbol{e})) \\ ({}^{\tau}MSG\mathsf{e}) & \simeq & \mathscr{T}^{\tau}_{MS}\,({}^{\tau}MS_{L}\,(\mathtt{G}^{\tau}_{MS}\,\mathsf{e}))\end{array}$$

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 $\tau = \mathbf{Nat}$: Lemma 16 (below) establishes that the translated form of a **Nat** 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 8. □

LEMMA 16. *Both of the following propositions hold:*

—*For any Scheme value* $(SM^{\mathbf{Nat}}_{L}\ \overline{n})$ *and for any top-level evaluation context* $\mathscr{E}[\ ]_S$*:*
—$\mathscr{E}[\mathscr{T}^{\mathbf{Nat}}_{SM}\,(\mathtt{G}^{\mathbf{Nat}}_{SM}\,(SM^{\mathbf{Nat}}_{L}\,\overline{n}))]_S \mapsto^{*} \mathscr{E}[\overline{n}]$.

—For any ML value (${}^{\mathbf{L}}MS_L$ v) and for any top-level evaluation context $\mathscr{E}[\ ]_M$:
—If $v = \overline{n}$ for some n, then $\mathscr{E}[\mathscr{T}^{\mathbf{Nat}}_{MS}\ ({}^{\mathbf{L}}MS_L\ (\mathtt{G}^{\mathbf{Nat}}_{MS}\ v))]_M \mapsto^* \mathscr{E}[\overline{n}]$.
*—If $v \neq \overline{n}$ for any n, then $\mathscr{E}[\mathscr{T}^{\mathbf{Nat}}_{MS}\ ({}^{\mathbf{L}}MS_L\ (\mathtt{G}^{\mathbf{Nat}}_{MS}\ v))]_M \mapsto^*$ **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 15, we can run natural-embedding programs using the lump-embedding evaluator by preprocessing them with these program conversion functions:

$$\begin{array}{rcl}
trans_M() & : & \mathbf{e} \to \mathbf{e} \\
trans_M((\mathbf{e}_1\ \mathbf{e}_2)) & = & (trans_M(\mathbf{e}_1)\ trans_M(\mathbf{e}_2)) \\
 & \vdots & \\
trans_M((^{\tau}MSG\mathtt{e})) & = & (\mathscr{T}^{\tau}_{MS}\ (^{\tau}MS_L\ (\mathtt{G}^{\tau}_{MS}\ trans_S(\mathtt{e}))))
\end{array}$$

$$\begin{array}{rcl}
trans_S() & : & \mathtt{e} \to \mathtt{e} \\
trans_S((\mathtt{e}_1\ \mathtt{e}_2)) & = & (trans_S(\mathtt{e}_1)\ trans_S(\mathtt{e}_2)) \\
 & \vdots & \\
trans_S((GSM^{\tau}\mathbf{e})) & = & (\mathscr{T}^{\tau}_{SM}(\mathtt{G}^{\tau}_{SM}\ (SM^{\tau}_L\ trans_M(\mathbf{e}))))
\end{array}$$

COROLLARY 17. *If $\boldsymbol{e}$ 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 12 and 14 and Lemma 15. □

COROLLARY 18. *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.

More generally, variants of this technique can be used to transmit an arbitrary element from an enumeration (*e.g.* symbols, strings) as long as a few basic conditions hold: first, the transmitting language must be able to identify an arbitrary element's position in the enumeration and be able to iterate a number of times corresponding to that position. Second, the receiving language must be able to provide a canonical zero'th element. Third, given an arbitrary element in the enumeration, the receiving language must be able to compute the next element. Admittedly this is probably never a good way to go about transmitting values in a real program, but nonetheless it is possible.

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

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

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

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

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

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

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

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 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 $\mathbf{L}_1 \to \mathbf{L}_1$ into a value of type $\mathbf{L}_1$ and later "unpack" that value back into a value of type $\mathbf{L}_1 \to \mathbf{L}_1$. This is the fundamental property identified by Scott for giving semantics to the untyped lambda calculus,

and more concretely it allows us to build something akin to the type `dynamic` [Abadi et al. 1991], and enough power to build a nonterminating term.

THEOREM 19. *The language of Figure 7 is non-terminating.*

PROOF. By construction. Given the following definitions:
$\mathrm{P} \stackrel{\text{def}}{=} (\lambda \mathbf{x} : \mathbf{L}_1 \to \mathbf{L}_1.\ ({}^{\mathbf{L}_1} M_1 M_2^{\mathbf{Nat}\to\mathbf{L}_2}\ (\lambda \mathbf{y} : \mathbf{Nat}.\ ({}^{\mathbf{L}_2} M_2 M_1^{\mathbf{L}_1\to\mathbf{L}_1}\ \mathbf{x}))))$
$\mathrm{U} \stackrel{\text{def}}{=} (\lambda \mathbf{x} : \mathbf{L}_1.\ ({}^{\mathbf{L}_1\to\mathbf{L}_1} M_1 M_2^{\mathbf{L}_2}\ (({}^{\mathbf{Nat}\to\mathbf{L}_2} M_2 M_1^{\mathbf{L}_1}\ \mathbf{x}))\ \overline{0}))$
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 $M_1 M_2$ program, and that its reduction graph contains a cycle. □

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. We can also write Y, for instance here for functions of type $\mathbf{Nat} \to \mathbf{Nat}$, by suitably modifying the types of our packing and unpacking functions:

$\mathrm{P} \stackrel{\text{def}}{=} (\lambda \mathbf{x} : \mathbf{L}_1 \to \mathbf{Nat} \to \mathbf{Nat}.\ ({}^{\mathbf{L}_1} M_1 M_2^{\mathbf{Nat}\to\mathbf{L}_2}\ (\lambda \mathbf{y} : \mathbf{Nat}.\ ({}^{\mathbf{L}_2} M_2 M_1^{\mathbf{L}_1\to\mathbf{Nat}\to\mathbf{Nat}}\ \mathbf{x}))))$
$\mathrm{U} \stackrel{\text{def}}{=} (\lambda \mathbf{x} : \mathbf{L}_1.\ ({}^{\mathbf{L}_1\to\mathbf{Nat}\to\mathbf{Nat}} M_1 M_2^{\mathbf{L}_2}\ (({}^{\mathbf{Nat}\to\mathbf{L}_2} M_2 M_1^{\mathbf{L}_1}\ \mathbf{x}))\ \overline{0}))$
$\mathsf{Y} \stackrel{\text{def}}{=} \lambda \mathbf{f} : (\mathbf{Nat} \to \mathbf{Nat}) \to \mathbf{Nat} \to \mathbf{Nat}.$
$\quad ((\lambda \mathbf{x} : \mathbf{L}_1.\ (f\ (\lambda \mathbf{m} : \mathbf{Nat}.\ (((\mathrm{U}\ x)\ x)\ m))))$
$\quad\ (\mathrm{P}\ (\lambda \mathbf{x} : \mathbf{L}_1.\ (f\ (\lambda \mathbf{m} : \mathbf{Nat}.\ (((\mathrm{U}\ x)\ x)\ m))))))$

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 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 20. *If $\boldsymbol{e}$ is a program in the language of Figure 8 and $\vdash_M \boldsymbol{e} : \tau$, then $\boldsymbol{e} \mapsto^* \boldsymbol{v}$.*

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

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

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

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

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

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

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

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

(though since these "relations" are unary, they are perhaps better called logical predicates):

$$
\begin{array}{lcl}
\mathscr{C}_1[\![\tau]\!] & \stackrel{\text{def}}{=} & \{\mathbf{e} \mid \forall \mathscr{E}[\,]_M.\ \mathscr{E}[\mathbf{e}]_M \mapsto^* \mathscr{E}[\mathbf{v}] \text{ and } \mathbf{v} \in \mathscr{V}_1[\![\tau]\!]\} \\
\mathscr{V}_1[\![\mathbf{Nat}]\!] & \stackrel{\text{def}}{=} & \{\overline{n} \mid n \in \mathbb{N}\} \\
\mathscr{V}_1[\![\tau_1 \to \tau_2]\!] & \stackrel{\text{def}}{=} & \{\mathbf{v} \mid \forall \mathbf{v}' \in \mathscr{V}_1[\![\tau_1]\!].(\mathbf{v}\ \mathbf{v}') \in \mathscr{C}_1[\![\tau_2]\!]\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathscr{C}_2[\![\sigma]\!] & \stackrel{\text{def}}{=} & \{\mathsf{e} \mid \forall \mathscr{E}[\,]_S.\ \mathscr{E}[\mathsf{e}]_S \mapsto^* \mathscr{E}[\mathsf{v}] \text{ and } \mathsf{v} \in \mathscr{V}_2[\![\sigma]\!]\} \\
\mathscr{V}_2[\![\mathbf{Nat}]\!] & \stackrel{\text{def}}{=} & \{\overline{n} \mid n \in \mathbb{N}\} \\
\mathscr{V}_2[\![\sigma_1 \to \sigma_2]\!] & \stackrel{\text{def}}{=} & \{\mathsf{v} \mid \forall \mathsf{v}' \in \mathscr{V}_2[\![\sigma_1]\!].(\mathsf{v}\ \mathsf{v}') \in \mathscr{C}_2[\![\sigma_2]\!]\}
\end{array}
$$

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 21. *Both of the following propositions hold:*

(1) *If* $\Gamma \vdash_{M_1} ({}^{\tau}M_1 M_2^{\sigma}\ e) : \tau$, γ *is a well-typed closing substitution for* Γ *such that all terms in* γ *are in their respective logical relations, and* $\gamma(e) \in \mathscr{C}_2[\![\sigma]\!]$, *then* $\gamma({}^{\tau}M_1 M_2^{\sigma}\ e) \in \mathscr{C}_1[\![\tau]\!]$.

(2) *If* $\Gamma \vdash_{M_2} ({}^{\sigma}M_2 M_1^{\tau}\ \boldsymbol{e}) : \sigma$, γ *is a well-typed closing substitution for* Γ *such that all terms in gamma are in their respective logical relations, and* $\boldsymbol{e} \in \mathscr{C}_1[\![\tau]\!]$, *then* $\gamma({}^{\sigma}M_2 M_1^{\tau}\ \boldsymbol{e}) \in \mathscr{C}_2[\![\sigma]\!]$.

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

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

$$\begin{array}{l} \quad \mathscr{E}[({}^{\tau}M_1 M_2^{\sigma}\ \gamma(\mathsf{v}))]_M \\ \mapsto\ \mathscr{E}[(\lambda \mathbf{x}' : \tau_1.({}^{\tau_2}M_1 M_2^{\sigma_2}\ (\gamma(\mathsf{v})\ ({}^{\sigma_1}M_2 M_1^{\tau_1}\ \mathbf{x}'))))] \end{array}$$

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

$$\begin{array}{l} \quad \mathscr{E}[(\lambda \mathbf{x}' : \tau_1.({}^{\tau_2}M_1 M_2^{\sigma_2}\ (\gamma(\mathsf{v})\ ({}^{\sigma_1}M_2 M_1^{\tau_1}\ \mathbf{x}'))))\ \mathbf{v}']_M \\ \mapsto\ \mathscr{E}[({}^{\tau_2}M_1 M_2^{\sigma_2}\ (\gamma(\mathsf{v})\ ({}^{\sigma_1}M_2 M_1^{\tau_1}\ \mathbf{v}')))] \end{array}$$

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

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

LEMMA 22. *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* $\gamma(\boldsymbol{e}) \in \mathscr{C}_1[\![\tau]\!]$.

(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* $\gamma(\mathsf{e}) \in \mathscr{C}_2[\![\sigma]\!]$.

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 21 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 21 instead of case 1. □

The main theorem is immediate from Lemma 22. 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

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

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

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

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

Fig. 9. Exceptions system 1 reduction rules

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.

Another caveat is also in order. As we said in Section 2, we intend the lump embedding to resemble totally-opaque "void-pointer" style foreign interfaces in which a program cannot observe anything about a foreign value directly. We think of this kind of embedding as a legitimate object of study, since many foreign function interfaces between higher-level languages and C really do present foreign language objects as void pointers. However, it is also worth saying that we could design a typed lump embedding that did not break termination guarantees, for instance by making the lump type take an extra foreign-type parameter that represents the foreign-language type of the lump's contents. (An implementation could use phantom types to track this information, reminiscent of the way phantom types are used in Blume's NLFFI [Blume 2001].)

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

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

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

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

Fig. 10. Exceptions system 2 reduction rules

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` *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 boundaries, and **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 [Wright and Felleisen 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 **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 exception, they abort the program instead of giving the programmer a chance to take less drastic corrective action.

THEOREM 23. *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 exception to continue propagating upwards, possibly being handled by a different language from the language than the one that raised it.

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

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

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

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

---

THEOREM 24. *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 [Girard et al. 1989] to the simple natural embedding from Figure 4. To ML we add type abstractions, written $(\Lambda\alpha.\mathbf{e})$, 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 $({}^{\forall\alpha.\tau}MSG\mathsf{v})$ 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 $(\Lambda\alpha.({}^{\tau}MSG\mathsf{v}))$.

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 25. *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

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

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

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

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

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

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-generically, 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 **Nat** from a boundary, it might be because the boundary had type **Nat** 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 **Nat**; 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:

$$\kappa = \mathbf{Nat} \mid \kappa_1 \to \kappa_2 \mid \forall\alpha.\kappa \mid \alpha \mid \mathbf{L} \mid \langle\beta; \tau\rangle$$

All these are identical to their type counterparts except the last, $\langle\beta; \tau\rangle$, which represents a type variable that has been bound to type τ. (Note β is a set of type variables that we can imagine being drawn from the same set as α, though in principle the two sets need not be related in any particular way.)

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

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

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

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

$$
\begin{array}{rcl}
\lfloor\ \rfloor : & & \kappa \to \tau \\
\lfloor\mathbf{Nat}\rfloor & = & \mathbf{Nat} \\
\lfloor\kappa_1 \to \kappa_2\rfloor & = & \lfloor\kappa_1\rfloor \to \lfloor\kappa_2\rfloor \\
\lfloor\forall\alpha.\kappa\rfloor & = & \forall\alpha.\lfloor\kappa\rfloor \\
\lfloor\alpha\rfloor & = & \alpha \\
\lfloor\mathbf{L}\rfloor & = & \mathbf{L} \\
\lfloor\langle\beta;\tau\rangle\rfloor & = & \tau
\end{array}
$$

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

---

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 include 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.

More formally, we must define three related kinds of type substitution for terms (both ML and Scheme), conversion schemes, and types. For clarity in these definitions we will refer to these kinds of substitutions respectively as $[\cdot/\cdot]_{\mathbf{e}}$, $[\cdot/\cdot]_{\mathsf{e}}$, $[\cdot/\cdot]_{\kappa}$, and $[\cdot/\cdot]_{\tau}$. The last of these is only defined when substituting concrete types into other concrete types with

free variables, and does not need redefinition. The others are defined as follows:

$$\begin{array}{rcl}
(\lambda \mathbf{x} : \tau.\, \mathbf{e})[\kappa/\alpha]_{\mathbf{e}} & = & (\lambda \mathbf{x} : \tau[\lfloor \kappa \rfloor/\alpha]_{\tau}.\, \mathbf{e}[\kappa/\alpha]_{\mathbf{e}}) \\
({}^{\kappa'}MSG\ \mathsf{e})[\kappa/\alpha]_{\mathbf{e}} & = & ({}^{\kappa'[\kappa/\alpha]}MSG\ \mathsf{e}[\kappa/\alpha]_{\mathsf{e}}) \\
(\mathbf{e}_1\langle\tau\rangle)[\kappa/\alpha]_{\mathbf{e}} & = & (\mathbf{e}_1[\kappa/\alpha]_{\mathbf{e}})\langle\tau[\lfloor\kappa\rfloor/\alpha]_{\tau}\rangle \\
(\mathbf{e}_1\ \mathbf{e}_2)[\kappa/\alpha]_{\mathbf{e}} & = & (\mathbf{e}_1[\kappa/\alpha]_{\mathbf{e}}\ \mathbf{e}_2[\kappa/\alpha]_{\mathbf{e}}) \\
 & \vdots & \\
(\lambda \mathsf{x}.\, \mathsf{e})[\kappa/\alpha]_{\mathsf{e}} & = & (\lambda \mathsf{x}.\, \mathsf{e}[\kappa/\alpha]_{\mathsf{e}}) \\
(GSM^{\kappa'}\ \mathbf{e})[\kappa/\alpha]_{\mathsf{e}} & = & (GSM^{\kappa'[\kappa/\alpha]}\ \mathbf{e}[\kappa/\alpha]_{\mathbf{e}}) \\
(\mathsf{e}_1\ \mathsf{e}_2)[\kappa/\alpha]_{\mathsf{e}} & = & (\mathsf{e}_1[\kappa/\alpha]_{\mathsf{e}}\ \mathsf{e}_2[\kappa/\alpha]_{\mathsf{e}}) \\
 & \vdots & \\
\alpha[\kappa/\alpha]_{\kappa} & = & \kappa \\
(\kappa_1 \to \kappa_2)[\kappa/\alpha]_{\kappa} & = & \kappa_1[\kappa/\alpha]_{\kappa} \to \kappa_2[\kappa/\alpha]_{\kappa} \\
 & \vdots &
\end{array}$$

Essentially we perform conversion scheme substitution just as we would perform type substitution except that we convert conversion schemes to types when we must substitute a type scheme into literal type annotation on a function argument or type application, retaining the conversion scheme intact for substituting into boundary annotations.

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 26. *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

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

applied to **Nat** now reduces to

$$({}^{\langle\beta;\mathbf{Nat}\rangle\to\langle\beta;\mathbf{Nat}\rangle}MSG(\lambda \mathsf{x}.(\mathtt{if0}\ (\mathtt{nat?}\ \mathsf{x})\ (+\ \mathsf{x}\ \overline{1})\ \mathsf{x})))$$

and then to

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

Even though the argument to the ML function as a whole is always a number, Scheme's `nat?` function will not recognize it as a number and thus this function is equivalent to

$$(\lambda \mathbf{y} : \mathbf{Nat}.({}^{\langle\beta;\mathbf{Nat}\rangle}MSG\ (GSM^{\langle\beta;\mathbf{Nat}\rangle}\ \mathbf{y})))$$

which is in turn equivalent to

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

the identity function for numbers.

Examples like that one suggest to us that this system preserves parametricity, but it would be much more satisfactory to establish that fact formally. This can be done, but the proof is quite involved and so is out of the scope of this paper. We refer the interested reader to Matthews and Ahmed [2008].

Here, we limit ourselves to a few observations. First, we should point out the caveat that while we have considered nontermination and errors as effects, more general kinds of effects such as mutable state would seriously enlarge the set of behaviors that functions of a given type could exhibit (as is generally true of languages that exhibit parametric polymorphism).

Second, 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 be related to an open problem raised by their work [Pierce and Sumii 2000, Section 4]. Third, it appears to be very similar to the notion of information-hiding among principals [Zdancewic et al. 1999; Grossman et al. 2000]. In particular, the syntactic abstraction properties established there also apply to our polymorphic embedding; from that vantage point the development in this section can be seen as a generalization of principals.

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 the static types on either side of a boundary. 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 multi-language system by generalizing the annotations on boundaries so that they are conversion strategies as in Section 6 rather than being restricted to types. 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 ${}^{\kappa}MSG$, where **Nat** 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 ${}^{\pi}MSG$

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

$$
\begin{array}{rcl}
\mathbf{E} & = & \cdots \mid (^{\kappa}\mathit{MSG}\ \mathsf{E}) \\
\mathsf{E} & = & \cdots \mid (\mathit{GSM}^{\kappa}\ \mathbf{E})
\end{array}
$$

$$
\begin{array}{rcl}
\tau & = & \cdots \mid \Sigma \\
\kappa & = & \mathbf{Nat} \mid \Sigma \mid \pi \mid \kappa \rightarrow \kappa
\end{array}
$$

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

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

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

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

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

---

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 27. *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

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

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

$$
\begin{array}{rcl}
\lfloor \mathbf{Nat} \rfloor & = & \mathbf{Nat} \\
\lfloor \mathbf{Nat!} \rfloor & = & \mathbf{Nat} \\
\lfloor \kappa_1 \to \kappa_2 \rfloor & = & \lfloor \kappa_1 \rfloor \to \lfloor \kappa_2 \rfloor
\end{array}
$$

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

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

---

as normal. We can model that choice by combining one of the 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.

The core of the system, as before, is a conversion strategy κ and an associated $\lfloor\ \rfloor$ meta-function that maps elements of κ to types; these are presented in Figure 14 along with the other necessary extensions. The conversion strategy adds a single conversion, **Nat**!, 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 **Nat**! boundaries.

THEOREM 28. *The language of Figure 14 is type-sound.*

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 [Zdancewic et al. 1999; Grossman et al. 2000] is also similar to ours in that it introduces two-agent calculi and boundaries, and as we discuss in Section 6 our investigation of type abstraction can be thought of as a generalization of theirs.

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; 2008].

On the issue of combining typed and untyped code, it is possible to see our lump embedding as a variation on the `dynamic` type and associated mechanisms introduced by Abadi *et al.* [1991] in which there is an entire second language devoted to computing with values of type `dynamic`. From this perspective the lump type is not quite the same thing as the

dynamic type. It is possible for a program to use the lump type to achieve a dynamic type (or at least a variation in which `typecase` is limited to a single branch), as we show in Section 4. But the lump type is inhabited by arbitrary Scheme values wrapped in boundaries, not just those that hide an ML value from ML's type system; these extra values do not have a clear analogue in Abadi *et al.* [1991].

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.

Our running examples have focussed on two languages with similar operational semantics and values but different type systems. However, we believe our technique scales to other languages whose features fit into a Felleisen-and-Hieb-style rewriting framework. Our evidence for this belief is that in our own investigations, we have not yet found a situation in which we had a clear intuition about how a multi-language system ought to work that we could not express easily using our technique. For instance, we had no trouble combining languages with different base values (as in Section 7) or adding new control behavior to one language without modifying the other (as in Section 5.1). Though we have only mentioned it briefly here (in Section 2.3), the framework we have developed also scales straightforwardly to a call-by-name/call-by-value combination. In all of these cases we have found that our models sharpened our understanding of underlying problems we were trying to solve, and the problems we encountered in building good models corresponded to real multi-language design issues rather than artificial problems imposed by the framework.

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/`.

Acknowledgements. The authors would like to thank Matthias Felleisen, Cormac Flanagan, Dave MacQueen, Phil Wadler, David Walker, Jack Wileden, the Northeastern University Programming Research Laboratory, and three sets of anonymous reviewers for their many contributions at various stages of the development of this paper. This work was funded in part by the National Science Foundation.

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.

BEAZLEY, D. 1997. Pointers, constraints, and typemaps. In SWIG 1.1 Users Manual.

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.

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.

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. *Theoretical Computer Science*, 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.

FURR, M. AND FOSTER, J. S. 2008. Checking type safety of foreign function calls. *ACM Transactions on Programming Languages and Systems*. To appear.

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. *ACM Transactions on Programming Languages and Systems 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.

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 AHMED, A. 2008. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *European Symposium on Programming (ESOP)*.

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

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. Unpublished manuscript.

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.

SUMII, E. AND PIERCE, B. 2003. Logical relations for encryption. *Journal of Computer Security (JSC) 11,* 4, 521–554.

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)*.