THE UNIVERSITY OF CHICAGO

THE MEANING OF MULTILANGUAGE PROGRAMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

JACOB BURTON MATTHEWS

CHICAGO, ILLINOIS

DECEMBER 2007

ABSTRACT

In this dissertation we develop a formal framework for examining multi-language systems. We introduce the framework with a simple language that connects two call-by-value languages with different type systems, then extend the language features we consider. This analysis allows us to reveal essential similarities in many different problems in programming language design, such as the surprising fact that safe foreign interfaces share a strong connection with contracts. We also propose that a good multi-language system ought to preserve the equational properties of its constituents and illustrate how to use this criterion as a design principle. Finally, we extend our methodology to domain-specific languages by presenting a formal design for Topsl, a domain-specific language for writing web-based surveys embedded into PLT Scheme.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 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; Wierzbicki et al. 2007; Odersky et al. 2005), and most recently Microsoft's Common Language Runtime (Benton et al. 2004; Meijer et al. 2001; Pinto 2003). Furthermore, where foreign function interfaces have historically been used in practice to allow high-level safe languages to call libraries written in low-level unsafe languages like C (as was the motivation for the popular wrapper generator SWIG (Beazley 1996)), these new foreign function interfaces are built to allow high-level, safe languages to interoperate with other high-level, safe languages, such as Python with Scheme (Meunier and Silva 2003) and Lua with OCaml (Ramsey 2003).

Since these embeddings are driven by practical concerns, the research that accompanies them rightly focuses on the bits and bytes of interoperability — how to represent data in memory, how to call a foreign function efficiently, and so on. But an important theoretical problem arises, independent of these implementation-level concerns: how can we reason formally about multi-language programs? This is a particularly important question for systems that involve typed languages, because we have to show that the embeddings respect their constituents' type systems, but important and interesting questions abound in other areas too: for instance, how can we mesh two languages with different type systems, or different orders of evaluation? How does the expressive power of a multi-language system relate to the expressive power of the individual languages it comprises? How can we design

a multi-language system so that the reasoning we perform on its constituents is still valid when we combine them?

Currently there is no way even to pose these questions precisely enough that they can be addressed formally. In this dissertation, we address that problem by developing a formal framework that researchers can use to give operational semantics to multi-language systems. The framework hinges on a simple syntactic extension we call *boundaries* that represent a switch from one language to another, and which come with an associated proof technique we call *bridging* that lets us show that properties such as type-soundness or termination that hold in both languages separately also hold in the combined language.

We introduce the technique by forming embeddings out of several different programming language models, each pair representing a different kind of interoperating system. We begin in chapter 3 by introducing the basic concept of boundaries and showing a few proofs that involve bridges. The languages we use are as simple as possible — we combine call-by-value versions of the simply-typed lambda calculus and the untyped lambda calculus.

In chapter 4, we enrich the type system of our simply-typed model with type variables, making it into System F. While making a type-sound embedding of System F and an untyped lambda calculus is simple, making one that preserves Reynolds' *abstraction* (better known as *parametricity*) guarantee (Reynolds 1983; Wadler 1989), is more challenging. We show how to devise such an embedding and prove that it preserves parametricity. As a consequence, we can also show that by using dynamic sealing we can add a dynamic type to System F that preserves parametricity, and that polymorphic Scheme contracts (or at least a limited notion thereof) that use dynamic seals to hide universally-quantified arguments also obey a parametricity guarantee.

In chapter 5, we turn our attention from differing type systems to differing evaluation strategies by combining a simply-typed, call-by-name language with a simply-typed call-by-value language. We do this in two ways. The first way is a very straightforward adaptation of the natural embedding introduced in chapter 3; we introduce the embedding, show it to be type sound, and show how to use Plotkin's CPS translations (Plotkin 1975) to eliminate these boundaries entirely. Then we show that the embedding has the unfortunate property that terms that would be contextually equivalent in a purely call-by-name setting

are no longer so when considering contexts that might contain call-by-value pieces. We investigate why this happens and then propose another embedding that does not have the same problem. We propose what we call the "gold standard" for multilanguage embeddings, namely that operational equivalence in either constituent of a multilanguage system should imply operational equivalence in the system as a whole, and prove that the gold standard holds for our second embedding. We conclude the chapter by presenting a plan for future work by sketching several situations in which the gold standard could be applied to gain insight on real-world foreign interfaces.

In chapter 6, we apply our multi-language methodology to the design of domain-specific embedded languages. Specifically, we develop a formal model for an embedding of Topsl, a domain-specific embedded language for constructing online surveys, into Scheme, and prove that it satisfies important domain-specific safety guarantees.

Chapter 3 is derived from Matthews and Findler (2007). Chapter 4 is derived from Matthews and Ahmed (2007). Chapter 5 is derived from Matthews (2007). Chapter 6 is derived from Matthews (2008).

CHAPTER 2
BACKGROUND

Multi-language systems are essentially as old as programming languages themselves. The earliest reference I have found to an explicit interoperability feature is for the FORTRAN II compiler for the IBM 704 Data Processing System released in 1958, which introduced the idea that a FORTRAN "subprogram" could be compiled in such a way that other subprograms written in assembly language could interface with it (Backus 1978; Int 1958). The Burroughs ALGOL 60 compiler, released in January 1961, supported an `external` keyword that brought a machine-language function into the scope of ALGOL code (Bur 1961). By 1967, the IBM OS/360 ALGOL 60 compiler had its own separate compilation feature that was designed in part to let programmers write programs in a mixture of ALGOL 60 and FORTRAN (Hoffman 1998). The preliminary Ada reference manual (Ichbiah 1979) made foreign function interoperation a part of the language standard. It defined an interface for Ada programs to import functions from other programming languages by declaring their names, argument and return types, and a pragma indicating the foreign language involved. The notion of an interface description language (IDL) came as an outgrowth of work on Ada (Lamb 1983, 1987).

These early systems showed, in raw form, the guiding principle that has driven multi-language systems implementations since: to get two languages to work together, both languages establish some common base for data exchange and control-flow (*e.g.* a particular assembly language calling convention), and then each language independently maps that common base into more appropriate language-specific constructs. These tasks are complicated by languages with sophisticated notions of control, and particularly by higher-order phenomena such as first-class functions and objects. Implementors have found several different ways to make that principle concrete, each with their own strengths and weaknesses.

2.1 The protocol-based approach

The method that involves the least amount of resource sharing is what I will call protocol-based interoperability, in which two language implementations agree on a network protocol for exchanging data and flow control, and the pieces of a program written in different languages run as separate tasks, perhaps on separate computers, that communicate using the shared protocol.[1] The protocol-based method is an outgrowth of distributed systems research, where multi-language interoperability is part of the larger problem of multi-*system* interoperability.

Early remote-procedure call (RPC) literature from the mid-1980s mentions the idea that language interoperation is a potentially interesting part of multi-system interoperation but does not develop it (Birrell and Nelson 1984; Bershad et al. 1987). In the mid-1990s, though, the rise of distributed component technologies (Szyperski 1998) made the problem of interlanguage interoperability more pressing. Technologies such as CORBA (The Object Management Group 2004; Henning 2006) and Microsoft's COM and DCOM systems (Box 1998) solved the problem by defining their own notions of types and values and providing IDL-based mappings from these "language-agnostic" notions into different concrete representations in different languages. Higher-order values (objects in the COM and CORBA models) are not passed in their entirety; instead the sender gives out a reference to the object that the receiver can use to identify it; that reference in conjunction with interface information are enough for the receiver to build a proxy object that will communicate any method invocations back to the object's owner. COM (and to a lesser extent CORBA) touched off a wave of interoperability research as language implementors made their systems interoperate with these foreign objects (Peyton Jones et al. 1997; Finne et al. 1999; Leijen 1998, 2003; Pucella 2002; Steckler 1999, 2000; Ibrahim and Szyperski 1997; Gray et al. 1998; Jeffery et al. 1999; Pucella 2000; Leroy 2001).

While the COM and CORBA approaches used IDLs to represent interface information for this purpose, many researchers have tried to move away from that approach on the grounds that it adds an extra hurdle for programmers to clear when they try to connect

1. I have chosen the phrase "protocol-based" to describe this approach in an attempt to avoid names that are too close to the names of particular technologies that implement the idea.

two already-existing programs. For example the PolySPINner tool (Barrett 1998; Barrett et al. 1996; Kaplan et al. 1988; Kaplan and Wileden 1996) and Mockingbird (Auerbach and Chu-Carroll 1997; Auerbach et al. 1998) both try to make interoperability more seamless by pairing data definitions in different languages and attempting to automatically generate appropriate converters and IDL descriptions. When successful, this approach eliminates the need for programmers to write a separate IDL file that in principle only contains redundant information. Grechanik, Batory and Perry take an alternative approach by introducing a generalized form of reflection, which they call "reification," that implements interoperability in what they consider a more scalable way by presenting a foreign object as a reflected object (Grechanik et al. 2004).

The main distinguishing feature of the protocol-based approach to interoperability, apart from the fact that it can connect pieces of programs running on different computers, is that each party to a communication maintains its own internal data structures. That can be a big advantage, since languages with widely different implementation strategies can interoperate just as easily as language implementations with the same implementation strategy (for instance it poses no special problem for a language implementation that represents program control on the heap using continuations to communicate with an implementation that represents control as a stack). It can also be a disadvantage for several reasons. First, it makes it harder for an implementation to monitor global invariants about a program; for instance, an implementation cannot free the memory associated with a particular object unless it knows that no one owns any reference to that object, which means that if it wants to use a garbage-collection strategy then it must use distributed garbage-collection techniques (Plainfossé and Shapiro 1995), and even implementations that plan to give up on automatic garbage-collection for foreign objects may need to adjust their internal memory management systems (see, *e.g.*, Peyton Jones, Marlow, and Elliot (Peyton Jones et al. 1999)). Second, a foreign function call always involves marshalling, communication, and unmarshalling, which is generally quite expensive compared to a native call.

2.2 The foreign interface approach

Another approach to interoperability is what I will call the foreign interface, or FI, approach[2]. This approach descends from the FORTRAN II approach mentioned at the beginning of this section: one implementation (typically of a "higher-level" language) arranges to interoperate with another implementation (typically of a "lower-level" language, and always a compiler) by mimicking its calling conventions in compiled code, which allows a linker to combine separately-compiled high- and low-level modules together. Calls from the higher-level language into the lower-level language are often called *call-outs*, and if the lower-level language can respond by calling functions from the higher-level language then such function applications are referred to as *call-backs*.

The earliest examples of the FI approach were built as a form of backwards compatibility, so new programs could reuse old libraries with a minimum of hassle. That motivation persists; designers of new languages sometimes go to great lengths to ensure that their new language implementations retain as much "data-level interoperability" as possible (Ohori et al.; Fisher et al. 2001). Choosing good representations for values can make interacting with a foreign language easier — for instance, in SML# a value of type `real array` is represented with exactly the same bits as a C value of type `double[]`, so SML# and C can freely exchange values with these types without any conversion. It may also be necessary to mediate between incompatible assumptions made by the two language implementations — for instance, functional languages almost always use automatic memory management, but memory addresses that escape into C code may also escape the garbage collector's ability to track them (see, for instance, Huelsbergen's discussion of the problem in his description a C interface for SML/NJ (Huelsbergen 1995)). This second problem is a variant on the problem that arises in the protocol based approach; however, it is simpler because at least in the foreign approach both programs are executing on the same machine and in the same address space, which enables lower-level, language-spanning algorithms like conservative garbage collection. (Generally, shared address space may help with implementation-related

2. Programmers use the terms "foreign interface" and "foreign function interface" interchangeably to describe this approach; I choose "foreign interface" because it does not misleadingly suggest that the mapping is limited to functions.

incompatibilities, but not with semantic-level incompatiblities. For instance, due to its insistence that all functions be pure, Haskell must control the effects of foreign calls through the IO monad (Peyton Jones and Wadler 1994); this constraint is independent of the way the implementation chooses to call foreign languages.)

Addressing these issues allow an implementation to incorporate a working, low-level foreign function interface, but as with the protocol-based approach programmers quickly found that writing low-level conversions by hand is frustratingly mechanical, and as with the protocol-based approach many tools have been implemented to ease the task. However, the IDL-based approach that features heavily in the protocol-based approach is not nearly as heavily used in the FI setting, probably due to different usage patterns. While the protocol-based approach is often used in situations where all involved components are designed to interoperate, the FI approach is more often used so a high-level language programmer can reuse a C library that was written without interoperability in mind. In the former situation it is reasonable to expect well-documented language-agnostic interfaces from which a tool can mechanically generate glue code for a variety of languages; in the latter it is not. (There certainly are IDL-based approaches to glue-code generation, though, notably H/Direct (Finne et al. 1998), which uses IDL to generate both C and COM interfaces.)

There are two major ways researchers have tried to tackle this problem: make glue code easier to write, and make tools that can treat C or C++ header files as a kind of IDL and automatically generate glue code from them. The former approach, typified by Barzilay's PLT Scheme interface (Barzilay and Orlovsky 2004) and Blume's SML/NJ interface (Blume 2001), introduces new kinds of values in the higher-level language that correspond directly to low-level values. This makes data-level interoperability between the two languages extremely easy and thus makes glue code easier to write, since it can be written in a high-level language. The cost is that this approach may introduce more complexity elsewhere in the system — for instance, the NLFFI maps C values directly into SML, but at the cost of having to build an entire C-like type system out of ML components. Another approach to making glue code easier to write is typified by Furr and Foster, who have built systems for verifying certain safety properties of the OCaml (Furr and Foster 2005) and the JNI (Furr and Foster 2006) by analyzing C code for problematic uses of foreign values. Under this

approach, programmers continue to write glue code in C but can automatically detect bugs that would not be detected by C's type system.

The other major approach to simplifying glue code is to generate it automatically from C or C++ header files as though they were a kind of IDL. Unfortunately, C header files are not as informative as IDL descriptions, and in particular often use the same types for different purposes in different parts of the interface. To deal with this, SWIG (Beazley 1996) introduced the concept of a *typemap*, which allows a programmer to fine-tune glue code generation for specific parts of a program; Reppy and Song's Fig (Reppy and Song 2006) extends this idea to allow interface writers to select translations anywhere on a continuum between efficient but cumbersome data-level interoperability and slower but easier-to-work-with high-level data mappings.

2.3 The common runtime approach

The idea that multiple languages should share a common, higher-level core goes back in principle at least as far as 1966 with Landin's "The Next 700 Programming Languages" (Landin 1966), and Weiser, Demers, and Hauser suggested exploiting it as an approach to interoperability in 1989 (Weiser et al. 1989). Their proposal, which I call the common runtime approach, is to arrange implementations so that each interacting language's runtime sits atop the same shared runtime, which provides higher-level, operating-system-like features that the language implementations share. For instance, Weiser, Demers, and Hauser's common runtime supported automatic garbage collection, networking, and threads and their associated locking mechanisms.

While their approach purposefully avoided requiring special compilation support, in the time since programmers have been much more receptive to the idea of using a special compiler to compile for a common runtime system, and in fact that approach has grown to dominate the field. For that reason, researchers have become interested in making intermediate representations that support interoperation effectively. Research on this front includes Gordon and Syme's formalization of a type-safe intermediate language designed specifically for multi-language interoperation (Gordon and Syme 2001), Shao's FLINT project (Shao 1997), Trifonov and Shao's work (Trifonov and Shao 1999) on the design

of intermediate languages that can handle the effects of source languages with different features, and Reig's (Reig 2001) work on establishing a good set of annotations that different high-level language compilers can use to communicate important facts for program optimization to a common back-end.

While that line of research focuses on intermediate languages in the abstract, there are two platforms in particular that have received special attention: Microsoft's Common Language Infrastructure (ECMA International) and Sun's Java Virtual Machine (Lindholm and Yellin 1999). (An open-source alternative, called the Parrot Virtual Machine (parrot), aims to be a common virtual machine for dynamic languages such as Perl and Python.) Because of the prominence of these platforms, researchers have paid special attention to them. Syme wrote about facilitating functional languages in .NET (Syme 2001), and Schinz and Odersky wrote about facilitating tail-call elimination on the JVM (Schinz and Odersky 2001). Housel *et al* explored adapting the JVM's transport mechanisms to code written in languages that are not similar to Java (Housel et al. 2001). Dowd, Henderson, and Ross (Dowd et al. 2001) wrote on how to compile the logic language Mercury for .NET, and Gutknecht (Gutknecht 2001) wrote on how to compile Oberon for .NET.

This style of interoperability's major advantage is that it standardizes system-level services, giving the advantages of shared address spaces without the disadvantage that each language implementation has to be conscious of the low-level details of making a shared address space work. The main disadvantage is that a common core must make assumptions about the services its clients will want, and languages can only work well on the common core to the extent that they make those assumptions true. For instance, Haskell, ML, and C# make very different use of memory each would probably perform best with a different garbage-collector tuned specifically for it; but if the three were running on the .NET CLR, they would have to share the same collector.

2.4 The embedded language approach

The embedded language approach is a special case of the common runtime approach, but it is distinct enough in flavor and has had enough effort put into it that I have given it its own section. In the embedded language approach, one language is compiled directly

into another language with the intention that programs written in the two languages will commingle types, values and control. Early examples of this approach were often compilers that compiled a high-level language to some popular lower-level languages, especially C, which makes it easy to implement calls into the lower-level language and, if one is careful, out of that language as well. Two examples of this approach are Bartlett's Scheme-to-C compiler (Bartlett 1989) and the SPiCE Smalltalk-to-C compiler (Yasumatsu and Doi 1995).

More recently, language implementors have used this technique to allow higher-level languages to interoperate with each other (often in this situation the technique is called "source-to-source translation"). For instance, Koser, Larsen and Vaughan have implemented an SML-to-Java compiler (Koser et al. 2003). Gray, Findler, and Flatt have implemented a compiler from Java to Scheme (Gray et al. 2005), Henglein and Rehof implemented a Scheme-to-ML compiler that aimed to perform as few dynamic type-tagging checks as possible (Henglein and Rehof 1995), Tolmach and Oliva implemented a compiler from ML to Ada (Tolmach and Oliva 1998), Kornstaedt has written a compiler for Alice (an SML extension) into the Oz programming language (Kornstaedt 2001), Bothner has written a Scheme-to-Java compiler (Bothner 1998), and Meunier and Silva have implemented a Python-to-Scheme compiler (Meunier and Silva 2004). The implementors of all these compilers listed interoperability as a goal of compilation.

The advantage of this approach is that since the embedded language is implemented in the host language, the embedded language's values are usually easy to expose to the host and vice versa — for instance, since Gray, Findler, and Flatt's ProfessorJ represents Java objects as objects in MzScheme's underlying object system, Scheme programs can easily invoke methods on Java objects. There are several downsides: first, the strategy doesn't scale very well, since it necessarily requires one to write a new compiler for the purpose of interoperability. Second, it is fundamentally asymmetric: language features of the host language are supported well, but language features of the embedded language can only be supported to the extent that they are implementable in terms of the host language. This is why, for instance, the Kawa Scheme-to-Java-bytecode compiler does not support full continuations.

2.5 Domain-specific embedded languages

An offshoot of the embedded language approach is the *domain-specific* embedded language approach, which uses the embedded language technique with to embed a domain-specific language into a general-purpose language, gaining the ease-of-use of DSLs while inheriting general-purpose facilities from the host language. Though the term DSEL is due to Hudak (Hudak 1996, 1998), the idea had been folklore in the Lisp and Scheme community for many years. Lisp implementations have had sophisticated macro systems since 1964 (Steele and Gabriel 1996). though it is not clear exactly when the idea that macros represented embedded-language compilers gained currency, Abelson and Sussman suggested that Lisp's features made it suitable for embedded language design in 1987 (Abelson and Sussman 1988), Baker presented a macro that served as a compiler for an embedded parser generator in 1991 (Baker 1991), Graham presented macros as tools for defining embedded languages in 1993 (Graham 1993), and Shivers made an explicit argument for the idea in 1996 (Shivers 1996). In 2001, Krishnamurthi's dissertation (Krishnamurthi 2001) went into great detail on the idea, giving a model for interoperating languages in which programming languages can communicate with each other transparently by compiling programs in all languages ultimately back to a single host (Scheme in his setting). He returned to the theme in 2006 (Krishnamurthi 2006).

The list of embedded domain-specific languages generated by the Lisp and Scheme community is nearly unending: for example, the last few years have seen parser generators (Owens et al. 2004), a mathematical plotter (Friedman and Raymond 2003), logic programming facilities (Sitaram 2003; Friedman et al. 2005; Byrd and Friedman 2006), a unit testing language (Welsh et al. 2002), a database-interaction language (Welsh et al. 2002), a term-rewriting engine (Matthews et al. 2004), a sophisticated pattern matching facility (Bagrak and Shivers 2004), a pedagogical lazy language (Barzilay and Clements 2005), a functional loop language (Shivers 2005), and two separate online survey languages (MacHenry and Matthews 2004; Queinnec 2002) all written for Scheme dialects.

The Lisp community has not completely cornered this market, of course. Examples of domain-specific languages embedded into other languages include Bjesse, Claessen, and Sheeran's Lava hardware design DSEL (Bjesse et al. 1998), Hudak, Makucevich,

Gadde, and Whong's Haskore music language (Hudak et al. 1996), Elliott and Hudak's functional reactive graphics language (Elliott and Hudak 1997), Leijen and Meijer's PARSEC parser generator (Leijen and Meijer 2001), the FunWorlds functional-reactive VRML language (Reinke 2001), all embedded in Haskell; and the MetaBorg project, which uses term rewriting to embed user-interface, regular-expression, XML and XPath DSLs into Java (Bravenboer and Visser 2004; Bravenboer et al. 2005). Finally, Benton describes a method for using type-indexed embedding and projection pairs to allow an untyped interpreted language to share values with its typed host in the process of describing a DSEL for tactical theorem proving embedded into MLj (Benton 2005) (the same method was also discovered independently by Ramsey (Ramsey 2003), though Ramsey was pursuing the FI approach).

A final twist on the embedded language theme was introduced and explored by Kamin (Kamin 1998) and later by Elliott, Finne, and de Moor (Elliott et al. 2000). In their approach, they embedded not interpreters but *compilers* (into some third language, often C or C++), effectively turning the general-purpose language into a macro or staged-computation language used for scripting the compilation process.

A key point is that these domain-specific languages are not just implemented in terms of some other host language, they are *embedded* in the host language. Using some approprate mechanism, a programmer actually inserts the code of the embedded language into a program written in the host language, and the two languages interoperate directly, perhaps interleaving in very fine-grained ways. In general this implementation strategy has some compelling advantages: since a DSEL uses the host's underlying types and values in its implementation, there is often a trivial mapping between the two languages, and in the case where a DSEL is built out of standard in-language tools, one can arrange for existing program analysis tools to analyze embedded programs as well "for free" (Herman and Meunier 2004). The main drawback of DSEL approach is its limited realm of application, since DSELs are only really feasible for small embedded languages and for hosts that have appropriate features.

2.6 Some formal frameworks

In the context of the .NET world where applications are often the product of multiple languages interoperating, Kennedy observed a problem (Kennedy 2006):

> A translation is fully abstract if it preserves and reflects observational equivalence. So if source-language compilation [into a common virtual machine instruction set] is not fully abstract, then there exist contexts (think 'attackers') in the target language that can observably distinguish two program fragments not distinguishable by source contexts. Such abstraction holes can sometimes be turned into security holes: if the author of a library has reasoned about the behaviour of his code by considering only source-level contexts (i.e. other components written in the same source language), then it may be possible to construct a component in the target language which provokes unexpected and damaging behaviour.

Though Kennedy is talking about translation, the problem he raises can be put more generally: programmers expect the languages they use to obey certain laws, but laws that hold of a programming language in isolation might break in a multi-language context. Kennedy expresses this as a potential security problem, but we can also think of it as an opportunity: just as formal models of programming languages have helped shaped language design and programming tools, understanding multi-language systems better could lead to better multi-language facilities for programmers.

Though there have been many multi-language system implementations, there has only been a tiny amount of formal underpinning. One possible starting point is Plotkin's "Call-by-name, call-by-value and the λ-calculus" (Plotkin 1975), which among other things demonstrates how call-by-name and call-by-value variants of the lambda calculus can be made to interoperate by using continuation-passing style to compile either language into the other. In the context of establishing an expressiveness hierachy for programming languages, Felleisen (1991) discussed the possibility of translating between various languages; Riecke did so in a more explicitly multi-language context by introducing the notion of fully-abstract translations from one language to another (Riecke 1991). Trifonov and Shao (Trifonov and Shao 1999) further explored the method of giving multi-language semantics by

translating each language into a shared core by characterizing what an intermediate language needs to support multiple, differently-featured interoperating languages efficiently. Henglein and Rehof (Henglein and Rehof 1995; Henglein 1994) take a semantic approach to translating Scheme into ML, putting special attention on where to best place dynamic checks, an approach that recalls Abadi, Cardelli, Pierce, and Plotkin's notion of a **dynamic** type (Abadi et al. 1991b).

One approach to multi-language semantics that does not use the translational approach is the theoretical development behind Furr and Foster's foreign-function safety checker Saffire, which checks for errors in C-to-OCaml glue code by imposing a new type system on C code. The proof of soundness for Saffire's C type system does not use any kind of translation, but neither does it treat the constituent languages as peers. The type system is imposed on C code only and the operational semantics do not allow OCaml code to run at all.

Beyond that work, there has been little that explicitly addresses multi-language interoperability. There have been relevant related developments, though: Zdancewic et al. (1999) develop calculi in which different pieces of a program (*e.g.*, modules), called "principals," are represented as syntactically distinguishable code fragments, allowing syntactic proofs of abstraction properties that would otherwise require more involved machinery such as logical relations. While their framework does not allow agents to be written in different languages, it contains ideas that are very suggestive of a multi-language framework. Indeed, in a future work section, they write:

> We can use essentially the same mechanism to formalize foreign function calls, where each agent uses a different set of operational rules. For example, we could give some agents a call-by-name semantics, allowing a mixture of eager and lazy evaluation. For less similar languages, the embeddings express exactly where foreign data conversions and calling conventions need to occur.

A final area of research that deserves mention comes from logic. Giunchiglia introduced multilanguage logics and later multilanguage hierarchical logics as an alternative to modal logic (Giunchiglia 1991; Giunchiglia and Serafini 1994) for intelligent reasoning systems. While despite the name, multilanguage logics were not at all designed for use as

the underlying logic of multilanguage programming systems, it appears that they may be useful for that purpose.

CHAPTER 3
MULTILANGUAGE SYSTEMS

3.1 Introduction

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

In section 3.2, 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 3.3, 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 3.4 shows two surprising relationships between the expressive power of these embeddings. In section 3.5, 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 3.6 we show how to extend the technique to non-type-directed conversions.

3.2 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

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

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

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

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

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

Figure 3.1: Core calculus for ML, primed for interoperability

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

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

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

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

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

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

Figure 3.2: Core calculus for Scheme, primed for interoperability

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 3.1 and 3.2 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).

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). Fourth, the language we develop is strongly reminiscent of Leroy's wrapped representations of polymorphic values (Leroy 1990) in that we develop opaque values with explicit conversions into and out of normal types; in fact, we exploit this ability in Section 3.4 to show that the lump embedding can add expressive power to a pair of programming languages.

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 nontermi-

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

Figure 3.3 summarizes the extensions to Figures 3.1 and 3.2 to support the lump embedding, and the next four subsections describe its syntax, type system, and operational semantics.

3.2.1 Syntax

The syntaxes of the two languages we use as our starting point are shown in Figures 3.1 and 3.2. On the ML side, we have taken the explicitly-typed lambda calculus and added numbers (where $\overline{n}$ indicates the syntactic term representing the number n) and a few built-in primitives, including an `if0` form. On the Scheme side, we have taken an untyped lambda calculus and added the same extensions plus some useful predicates and a `wrong` form that takes a literal error message string.

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

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

$$
\begin{array}{rcl}
\mathbf{e} & = & \cdots \mid (\tau MS\ \mathsf{e}) \\
\mathbf{v} & = & \cdots \mid (\mathbf{L} MS\ \mathsf{v}) \\
\tau & = & \cdots \mid \mathbf{L} \\
\mathbf{E} & = & \cdots \mid (\tau MS\ \mathsf{E})
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{e} & = & \cdots \mid (SM^{\tau}\ \mathbf{e}) \\
\mathsf{v} & = & \cdots \mid (SM^{\tau}\ \mathbf{v}) \text{ where } \tau \neq \mathbf{L} \\
\mathsf{E} & = & \cdots \mid (SM^{\tau}\ \mathbf{E}) \\
\multicolumn{3}{c}{\mathcal{E} = \mathbf{E}}
\end{array}
$$

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

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

Figure 3.3: Extensions to Figures 3.1 and 3.2 to form the lump embedding

---

(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 3.3.[1]

3.2.2 Typing rules

In Figure 3.1, 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 3.2, Scheme has a trivial type system with the judgment $\vdash_S$ that gives all closed terms the type **TST** (“the Scheme type”).

In our lump embedding extension, we add a new type **L** (for “lump”) to ML and we add a new rule to each typing judgment corresponding to the new syntactic forms. The new

---

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.

Scheme judgment says that an (SM^{τ} **e**) boundary is well-typed if ML's type system proves **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 (${}^{\tau}MS$ e) has type τ if e type-checks under Scheme's typing system. In both cases, τ can be any type, not just **L** as one might expect. If $\tau = \mathbf{L}$ we are sending a native Scheme value 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 3.2.4). We assume that Scheme and ML variables are drawn from disjoint sets.

3.2.3 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 (**E**) and one for Scheme (E)[2]. We also use a third, unspecified evaluation context ($\mathcal{E}$) for the definitions of the rewriting rules (whose precise role with respect to those rules is explained below). Intuitively, $\mathcal{E}$ corresponds to "whole program" evaluation contexts. A "whole program" is necessarily either a term in ML or a term in Scheme, so a whole-program evaluation context must be either an ML or a Scheme evaluation context. 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 have $\mathcal{E} = \mathbf{E}$ in Figure 3.3.

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

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.

versa. We do that by adding new context productions for boundaries:

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

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

3.2.4 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* ($\mathcal{E}$) 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 3.3'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:

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

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[3]. 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

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 $(\textbf{let}\ ((f : \tau.\ \mathbf{e}_1))\ \mathbf{e}_2) = ((\lambda f : \tau.\ \mathbf{e}_2)\ \mathbf{e}_1)$.

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:

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

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

The formulation of the lump embedding in Figure 3.3 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 3.2.1. *If* $\Gamma \vdash_M \mathbf{e} : \tau$*, then either* $\mathbf{e} \mapsto^* \mathbf{v}$*,* $\mathbf{e} \mapsto^*$ **Error***: str, or* $\mathbf{e}$ *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 (Wright and Felleisen 1994).

Lemma 3.2.2. *If* $\Gamma \vdash_M \mathbf{e} : \tau$*, then:*

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

Given these we can show preservation:

Lemma 3.2.3. *If* $\vdash_M \mathbf{e} : \tau$ *and* $\mathbf{e} \mapsto \mathbf{e}'$, *then* $\vdash_M \mathbf{e}' : \tau$.

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

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, $\mathcal{E}[\mathbf{v}]$ has type τ.

Case $\mathcal{E}[({}^{\tau'}MS\ \mathsf{v})]_M \mapsto \mathcal{E}[({}^{\tau'}MS\ (\texttt{wrong}\ \text{“Bad value”}))]$ 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

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

has type τ.

Case $\mathcal{E}[(SM^{\mathbf{L}}\ ({}^{\mathbf{L}}MS\ \mathsf{v}))]_S \mapsto \mathcal{E}[\mathsf{v}]$

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, $\mathcal{E}[\mathsf{v}]$ 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 3.2.4. *For all ML expressions* $\mathbf{e}$ *and Scheme expressions* e, *both of the following hold:*

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

(2) If $\vdash_S e$: ***TST****, then either* e *is a Scheme value or for all top-level evaluation contexts* $\mathcal{E}[\,]_S$ *either there exists an* e' *such that* $\mathcal{E}[e] \mapsto e'$ *or* $\mathcal{E}[e] \mapsto$ **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 $\dfrac{\vdash_S e : \mathbf{TST}}{\vdash_M ({}^{\tau}MS\ e) : \tau}$:

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

With these two lemmas established, theorem 3.2.1 is nearly immediate:

Proof. Combination of lemmas 3.2.3 and 3.2.4. □

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 3.2.1 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.

3.3 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 3.1 and 3.2, just as we did before to form the lump embedding. Again, we add new syntax reduction rules to figures 3.1 and 3.2 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 3.2.

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:

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

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 3.6.

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, $({}^{\tau_1 \mapsto \tau_2}MS_N\ \lambda x.e)$ becomes $(\lambda \mathbf{x} : \tau_1.{}^{\tau_2}MS_N\ ((\lambda x.e)\ (SM_N^{\tau_1}\ \mathbf{x})))$ 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.

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

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

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

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

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

Figure 3.4: Extensions to Figure 3.1 and 3.2 to form the simple natural embedding

To model this method, rather than adding the SM_N^τ and ${}^\tau MS_N$ constructs to our core languages from Figures 3.1 and 3.2, we instead add "guarded" versions GSM^τ and MSG^τ shown in Figure 3.4. These rules translate values in the same way that SM_N^τ and ${}^\tau MS_N$ did before, but also detect concrete, first-order witnesses to an invalid type ascription (*i.e.*, numbers for procedures or procedures for numbers) and abort the program if one is found. We call the language formed by these rules the simple natural embedding. We give its rules in Figure 3.4, but it may be easier to understand how it works by reconsidering the examples we gave at the end of Section 3.2. 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:

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

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 number $\overline{3}$, 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:

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

Depending on the behavior of its arguments, the Scheme procedure may or may not produce

a number. ML treats it as though it definitely had type $(\mathbf{Nat} \to \mathbf{Nat}) \to \mathbf{Nat}$, and wraps it to the ML value

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

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 $\mathbf{Nat}$. 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 3.3.1. *If* $\vdash_M \boldsymbol{e} : \tau$*, then either* $\boldsymbol{e} \mapsto^* \boldsymbol{v}$*,* $\boldsymbol{e} \mapsto^* \mathbf{Error}$*: str, or* $\boldsymbol{e}$ *diverges.*

Proof. By a standard argument along the lines of theorem 3.2.1. □

3.3.2 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 $(\mathcal{G}^{\tau}\ \mathsf{e})$ that perform all dynamic checks necessary to ensure that their arguments behave as τ in the sense of the previous subsection. In all initial terms, we will wrap every boundary with an appropriate guard: $(^{\tau}MS_N\ (\mathcal{G}^{\tau}\ \mathsf{e}))$ instead of $(MSG^{\tau}\ \mathsf{e})$ and $(\mathcal{G}^{\tau}\ (SM^{\tau}\ \mathbf{e}))$ instead of $(GSM^{\tau}\ \mathbf{e})$.

Figure 3.5 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 $\tau_1 \to \tau_2$

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

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

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

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

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

Figure 3.5: Extensions to Figures 3.1 and 3.2 to form the separated-guards natural embedding

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

The guard distribution rules for functions can move a guard arbitrarily far away from the boundary it protects; if this motion ever gave a value the opportunity to get to a boundary without first being blessed by the appropriate guard, the system would get stuck. We can prove this never happens by defining a combined language that has *both* guarded boundaries GSM^{τ} and MSG^{τ} (for "guard, then SM boundary" and "MS boundary, then guard") and unguarded boundaries with separated guards ${}^{\tau}MS_N$, SM_N^{τ}, and $\mathcal{G}^{\tau}$; *i.e.* the language formed by combining Figures 3.1 and 3.2 with Figures 3.4 and 3.5. In this combined language, an argument by induction shows that guarded boundaries are observably equivalent to guards combined with unguarded boundaries.

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

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

Theorem 3.3.2. *$eval_N()$ is a partial function.*

Proof. That $eval_N()$ maps an input to at most one output follows from unique decomposition (lemma 3.3.4 below). That it is not total follows from the existence of infinite reduction chains in Scheme. □

Definition 3.3.3. An *instruction* is a term that appears syntactically on the left-hand side of the $\mapsto$ reduction relation's definition in Figures 3.1 and 3.2.

Lemma 3.3.4.

- *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 $e = E[i]$.*

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 3.2. 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 3.3.5. *For all Scheme expressions e and ML expressions $\mathbf{e}$, the following propositions hold:*

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

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

Proof. As in Felleisen *et al* (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 states that if two terms uniquely reduce to terms that are observationally equivalent, then they are equivalent themselves:

Lemma 3.3.7. *If for all evaluation contexts* $\mathcal{E}[\,]$, $\mathcal{E}[e_1] \mapsto^* \mathcal{E}[e_1']$ *(uniquely) and* $\mathcal{E}[e_2] \mapsto^* \mathcal{E}[e_2']$ *(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 $\mathcal{E}'[\,]$ such that $eval_N(\mathcal{E}'[e_1])$ produces a different result from $eval_N(\mathcal{E}'[e_2])$. But by premises, $\mathcal{E}'[e_1] \mapsto^* \mathcal{E}'[e_1']$ and $\mathcal{E}'[e_2] \mapsto^* \mathcal{E}'[e_2']$, 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 3.3.6, for each proposition it suffices to show that the two forms are equivalent in any evaluation context $\mathcal{E}[\,]$. 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 $\overline{n}$:

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

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

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

For (2): the analogous argument applies, but by inversion we know that $\mathbf{v}$ must be a natural number so the error case is impossible.

Case $\tau = \tau_1 \to \tau_2$: For (1): if v is $(\lambda x.e')$, then the left-hand side expression reduces as follows:

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

For the right-hand side:

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

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

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

and

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

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

From theorem 3.3.5, 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:

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

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

(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 3.3.5.

Corollary 3.3.8. *If* $\vdash_M \mathbf{e} : \tau$ *in the language formed by combining Figures 3.1, 3.2, and 3.5 (i.e., the natural, separated guard language) and* $\mathbf{e}$ *contains no guards, then either* $elab_M(\mathbf{e}) \mapsto^* \mathbf{v}$, $elab_M(\mathbf{e}) \mapsto^* \mathbf{Error}$*: str, or* $elab_M(\mathbf{e})$ *diverges.*

Proof. If $\vdash_M \mathbf{e} : \tau$, then $\vdash_M elab_M(\mathbf{e}) : \tau$ as well. By theorem 3.3.5, $elab_M(\mathbf{e})$ is equivalent to a program in the simple natural embedding, and by theorem 3.3.1 that program is well-behaved. Thus, so is $elab_M(\mathbf{e})$. □

The restriction that $\mathbf{e}$ have no guards in corollary 3.3.8 is only necessary because theorem 3.3.1 does not account for guards, so we need to be able to guarantee that we can eliminate every guard using the equivalence of theorem 3.3.5. This is purely technical — if we wanted to allow programmers to use guards directly, we could reprove theorem 3.3.1 with that extension.

3.3.3 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 $(^{\mathbf{Nat}\to\mathbf{Nat}}MS_N\ (\mathcal{G}^{\mathbf{Nat}\to\mathbf{Nat}}\ (\lambda \mathsf{x}.\mathsf{x})))$ reduces to $(\lambda \mathbf{x} : \mathbf{Nat}.(^{\mathbf{Nat}}MS_N\ (\mathcal{G}^{\mathbf{Nat}}\ ((\lambda \mathsf{x}.\mathsf{x})\ (\mathcal{G}^{\mathbf{Nat}}\ (SM_N^{\mathbf{Nat}}\ \mathbf{x}))))))$.

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

$$\begin{array}{lcl}
\mathcal{E}[(\mathcal{G}_+^{\mathbf{Nat}}\,\overline{n})]_S & \mapsto & \mathcal{E}[\overline{n}] \\
\mathcal{E}[(\mathcal{G}_+^{\mathbf{Nat}}\mathsf{v})]_S & \mapsto & \mathcal{E}[(\texttt{wrong}\text{ ``Non-number''})]\ (\mathsf{v} \neq \overline{n} \text{ for any } n) \\
\mathcal{E}[(\mathcal{G}_+^{\tau 1\to\tau 2}\,(\lambda \mathsf{x}.\mathsf{e}))]_S & \mapsto & \mathcal{E}[\mathcal{G}_+^{\tau 2}\,((\lambda \mathsf{x}.\mathsf{e})\,(\mathcal{G}_-^{\tau 1}\,\mathsf{x}))] \\
\mathcal{E}[(\mathcal{G}_+^{\tau 1\to\tau 2}\mathsf{v})]_S & \mapsto & \mathcal{E}[(\texttt{wrong}\text{ ``Non-function''})]\ (\mathsf{v} \neq (\lambda \mathsf{x}.\mathsf{e}) \text{ for any } \mathsf{x}, \mathsf{e})
\end{array}$$

$$\begin{array}{lcl}
\mathcal{E}[(\mathcal{G}_-^{\mathbf{Nat}}\mathsf{v})]_S & \mapsto & \mathcal{E}[\mathsf{v}] \\
\mathcal{E}[(\mathcal{G}_-^{\tau 1\to\tau 2}\mathsf{v})]_S & \mapsto & \mathcal{E}[(\lambda \mathsf{x}.(\mathcal{G}_-^{\tau 2}(\mathsf{v}(\mathcal{G}_+^{\tau 1}\,\mathsf{x}))))]
\end{array}$$

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

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

Theorem 3.3.9. *For all ML expressions* $\boldsymbol{e}$ *and Scheme expressions* e*, both of the following propositions hold:*

$$\begin{array}{lcl}
(1)\quad (MSG^\tau\,\boldsymbol{e}) & \simeq & ({}^\tau MS_N\,(\mathcal{G}_+^\tau\,\mathsf{e})) \\
(2)\quad (GSM^\tau\,\boldsymbol{e}) & \simeq & (\mathcal{G}_-^\tau\,(SM_N^\tau\,\boldsymbol{e}))
\end{array}$$

Proof. As the proof of theorem 3.3.5, *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_N^{τ} boundaries, we can obtain the same type-soundness result for this system.

Corollary 3.3.10. *If* $\vdash_M e : \tau$ *where* e *is a term in the natural, separated guard language, then either* $elab_{C,M}(e) \mapsto^* v$, $elab_{C,M}(e) \mapsto^*$ **Error***: str, or* $elab_{C,M}(e)$ *diverges.*

Proof. Combine theorem 3.3.9 and theorem 3.3.1 as in the proof of corollary 3.3.8. □

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

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

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

Theorem 3.3.11. *For all* $\mathtt{e}$ *and* τ*, both of the following hold:*

$$
\begin{array}{rrcl}
(1) & \mathtt{G}_+^{\tau}\ e & \simeq & \mathcal{G}_+^{\tau}\ e \\
(2) & \mathtt{G}_-^{\tau}\ e & \simeq & \mathcal{G}_-^{\tau}\ e
\end{array}
$$

Proof. As in the proof of theorem 3.3.5, we show equivalence by induction on τ where $\mathtt{e}$ is a value and the term appears in an evaluation context; this suffices to prove theorem as stated.

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

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

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

If v is a natural number, then:

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

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

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

and $\mathcal{E}[(\mathcal{G}_+^{\mathbf{Nat}}\mathsf{v})] \mapsto \mathcal{E}[\mathtt{wrong}\ \text{“Non-number”}]$. 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 $\mathcal{E}$ is some evaluation context where $\mathcal{E}[\mathsf{e}] \mapsto^* \mathcal{E}[\mathsf{v}]$. Then $\mathcal{E}[(\mathtt{G}_+^{\tau_1\to\tau_2}\ \mathsf{e})] \mapsto^* \mathcal{E}[(\mathtt{G}_+^{\tau_1\to\tau_2}\ \mathsf{v})]$ and $\mathcal{E}[(\mathcal{G}_+^{\tau_1\to\tau_2}\ \mathsf{e})] \mapsto^* \mathcal{E}[(\mathcal{G}_+^{\tau_1\to\tau_2}\ \mathsf{v})]$. The former then reduces as fol-

lows:

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

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

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

(where y is not free in v), and

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

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

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

Given these implementations, we can observe that $\mathtt{G}_{+}^{\tau}$ and $\mathtt{G}_{-}^{\tau}$ are a pair of projections in the sense of Findler and Blume (Findler and Blume 2006) where the two parties are + and – and we have prior knowledge from ML's type system that – never violates its contract. From a practical perspective, this means that rather than implementing separate, special-purpose error-containment code for our multi-language systems, we can use an off-the-shelf mechanism with confidence instead. This adds to our confidence in the fine-grained interoperability scheme in Gray *et al.* Gray et al. (2005). More theoretically, it means we can use the simple system of Section 3.3.1 for our models and be more confident that our soundness results apply to actual multi-language embeddings that we write with contracts.

$$
\begin{array}{lll}
\mathcal{T}_M^{\mathbf{Nat}} & \stackrel{\text{def}}{=} & (\lambda \mathbf{x} : \mathbf{L}. \\
& & \quad ({}^{\mathbf{Nat}}MS_L\ ((\mathsf{Y}\ (\lambda \mathsf{f}.\ \lambda \mathsf{n}. \\
& & \qquad (\mathtt{if0}\ \mathsf{n} \\
& & \qquad\quad (SM_L^{\mathbf{Nat}}\ \overline{0}) \\
& & \qquad\quad (SM_L^{\mathbf{Nat}}\ (+\ \overline{1}\ ({}^{\mathbf{Nat}}MS_L\ (\mathsf{f}\ (-\ \mathsf{n}\ \overline{1})))))))) \\
& & \qquad (SM_L^{\mathbf{L}}\ \mathbf{x})))) \\
\mathcal{T}_S^{\mathbf{Nat}} & \stackrel{\text{def}}{=} & (\lambda \mathsf{x}.\ (\mathsf{Y}\ (\lambda \mathsf{f}.\ \lambda \mathsf{n}. \\
& & \qquad (SM_L^{\mathbf{L}}\ (\mathtt{if0}\ ({}^{\mathbf{Nat}}MS_L\ \mathsf{n}) \\
& & \qquad\quad ({}^{\mathbf{L}}MS_L\ \overline{0}) \\
& & \qquad\quad ({}^{\mathbf{L}}MS_L\ (+\ \overline{1}\ (\mathsf{f}\ (SM_L^{\mathbf{Nat}}\ (-\ ({}^{\mathbf{Nat}}MS_L\ \mathsf{n})\ \overline{1}))))))) \\
& & \quad \mathsf{x}))
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{T}_M^{\tau_1 \to \tau_2} & \stackrel{\text{def}}{=} & (\lambda \mathbf{x} : \mathbf{L}.\lambda \mathbf{y} : \tau_1.(\mathcal{T}_M^{\tau_2}({}^{\mathbf{L}}MS_L((SM_L^{\mathbf{L}}\mathbf{x})\mathcal{T}_S^{\tau_1}(SM_L^{\tau_1}\mathbf{y}))))) \\
\mathcal{T}_S^{\tau_1 \to \tau_2} & \stackrel{\text{def}}{=} & (\lambda \mathsf{x}.\lambda \mathsf{y}.(\mathcal{T}_S^{\tau_2}(SM_L^{\tau_2}(({}^{\tau_1 \to \tau_2}MS_L\mathsf{x})(\mathcal{T}_M^{\tau_1}({}^{\mathbf{L}}MS_L\mathsf{y}))))))
\end{array}
$$

Figure 3.6: Translation functions for lump values

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).

3.4 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_N^{τ} boundaries are macro-expressible in the lump embedding (in the sense of Felleisen 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.

3.4.1 The lump embedding simulates the natural embedding

As it turns out, two sufficiently determined parties using the lump embedding of Section 3.2 can simulate the natural embedding of Section 3.3. To show that, we define two type-indexed functions, $\mathcal{T}_S^\tau$ and $\mathcal{T}_M^\tau$, that can be written in the lump embedding. These functions translate values whose ML type is τ from Scheme to ML and from ML to Scheme, respectively; they are shown in Figure 3.6. (In that Figure and below, for clarity we use the notation ${}^\tau MS_L$ and SM_L^τ rather than ${}^\tau MS$ and SM^τ to refer to the lump embedding's boundaries.) The translation functions for higher-order types use essentially the same method we presented in Section 3.3 for converting a procedure value — they translate a procedure by left- and right-composing appropriate translators for its input and output types. That leaves us with nothing but the base types, in our case just numbers.

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

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

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 3.6 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 3.4.1. *In the language formed by extending the language of Figures 3.1 and 3.2 with both Figure 3.3 and Figure 3.4, both of the following propositions hold:*

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

Proof. As before, it is sufficient to consider the case where $\mathbf{e} = \mathbf{v}$ (for the first proposition) or $\mathsf{e} = \mathsf{v}$ (for the second) and the overall expression is in an evaluation context.

Case $\tau = \mathbf{Nat}$: Lemma 3.4.2 (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 3.3.5. □

Lemma 3.4.2. *Both of the following propositions hold:*

- *For any Scheme value* $(SM_L^{\mathbf{Nat}}\ \overline{n})$ *and for any top-level evaluation context* $\mathcal{E}[\ ]_S$, $\mathcal{E}[\mathcal{T}_S^{\mathbf{Nat}}\,(\mathtt{G}_-^{\mathbf{Nat}}\,(SM_L^{\mathbf{Nat}}\,\overline{n}))]_S \mapsto^* \mathcal{E}[\overline{n}]$.
- *For any ML value* $({}^{\mathbf{L}}MS_L\ \mathsf{v})$ *and for any top-level evaluation context* $\mathcal{E}[\ ]_M$*:*
  - *If* $\mathsf{v} = \overline{n}$ *for some n, then* $\mathcal{E}[\mathcal{T}_M^{\mathbf{Nat}}\,({}^{\mathbf{L}}MS_L\,(\mathtt{G}_+^{\mathbf{Nat}}\,\mathsf{v}))]_M \mapsto^* \mathcal{E}[\overline{n}]$.
  - *If* $\mathsf{v} \neq \overline{n}$ *for any n, then* $\mathcal{E}[\mathcal{T}_M^{\mathbf{Nat}}\,({}^{\mathbf{L}}MS_L\,(\mathtt{G}_+^{\mathbf{Nat}}\,\mathsf{v}))]_M \mapsto^*$ **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 3.4.1, we can run natural-embedding programs using the lump-embedding

evaluator by preprocessing them with these program conversion functions:

$$
\begin{aligned}
trans_M() &: \mathbf{e} \to \mathbf{e} \\
trans_M((\mathbf{e}_1\ \mathbf{e}_2)) &= (trans_M(\mathbf{e}_1)\ trans_M(\mathbf{e}_2)) \\
&\vdots \\
trans_M((MSG^{\tau}\mathbf{e})) &= (\mathcal{T}_M^{\tau}\ ({}^{\tau}MS_L\ (\mathtt{G}_+^{\tau}\ trans_S(\mathbf{e}))))
\end{aligned}
$$

$$
\begin{aligned}
trans_S() &: \mathsf{e} \to \mathsf{e} \\
trans_S((\mathsf{e}_1\ \mathsf{e}_2)) &= (trans_S(\mathsf{e}_1)\ trans_S(\mathsf{e}_2)) \\
&\vdots \\
trans_S((GSM^{\tau}\mathsf{e})) &= (\mathcal{T}_S^{\tau}(\mathtt{G}_-^{\tau}\ (SM_L^{\tau}\ trans_M(\mathbf{e}))))
\end{aligned}
$$

Corollary 3.4.3. *If* $\boldsymbol{e}$ *is a well-typed closed program in the natural embedding of Section 3.3, then* $eval_N(\boldsymbol{e}) = eval_L(trans_M(\boldsymbol{e}))$.

Proof. Combine theorems 3.3.9 and 3.3.11 and lemma 3.4.1. □

Corollary 3.4.4. *Both the* SM_N^{τ} *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.

3.4.2 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 3.7, which shows an embedding analogous to the ML-to-Scheme lump embedding from Section 3.2 where both sides are copies of our ML stand-in (Figure 3.7 elides the standard rules for abstractions, application, numeric operators and so on, which are as in Figure 3.1). 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. 1991a), and enough power to build a nonterminating term.

Theorem 3.4.5. *The language of Figure 3.7 is non-terminating.*

Proof. By construction. Given the following definitions:
$\mathrm{P} \stackrel{\text{def}}{=} (\lambda \mathbf{x} : \mathbf{L}_1 \to \mathbf{L}_1 . (^{\mathbf{L}_1} M_1 M_2^{\mathbf{Nat} \to \mathbf{L}_2} (\lambda \mathbf{y} : \mathbf{Nat} . (^{\mathbf{L}_2} M_2 M_1^{\mathbf{L}_1 \to \mathbf{L}_1} \mathbf{x}))))$
$\mathrm{U} \stackrel{\text{def}}{=} (\lambda \mathbf{x} : \mathbf{L}_1 . (^{\mathbf{L}_1 \to \mathbf{L}_1} M_1 M_2^{\mathbf{L}_2} ((^{\mathbf{Nat} \to \mathbf{L}_2} M_2 M_1^{\mathbf{L}_1} \mathbf{x})) \overline{0}))$

$$
\begin{array}{lcl}
\mathbf{e} & = & \mathbf{x} \mid \mathbf{v} \mid (\mathbf{e}\ \mathbf{e}) \mid (op\ \mathbf{e}\ \mathbf{e}) \mid (\texttt{if0}\ \mathbf{e}\ \mathbf{e}\ \mathbf{e}) \mid ({}^{\tau}M_1 M_2^{\sigma}\ \mathsf{e}) \\
\mathbf{v} & = & (\lambda \mathbf{x} : \tau.\ \mathbf{e}) \mid \overline{n} \mid ({}^{\mathbf{L}_1}M_1 M_2^{\sigma}\ \mathsf{v}) \\
\tau & = & \mathbf{Nat} \mid \tau \to \tau \mid \mathbf{L}_1 \\
\mathbf{E} & = & [\,]_{M_1} \mid (\mathbf{E}\ \mathbf{e}) \mid (\mathbf{v}\ \mathbf{E}) \mid (op\ \mathbf{E}\ \mathbf{e}) \mid (op\ \mathbf{v}\ \mathbf{E}) \mid \\
 & & (\texttt{if0}\ \mathbf{E}\ \mathbf{e}\ \mathbf{e}) \mid ({}^{\tau}M_1 M_2^{\sigma}\ \mathsf{E})
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathcal{E}[({}^{\tau}M_1 M_2^{\mathbf{L}_2}\ ({}^{\mathbf{L}_2}M_2 M_1^{\tau}\ \mathbf{v}))]_{M_1} & \mapsto & \mathcal{E}[\mathbf{v}] \\
\mathcal{E}[({}^{\tau}M_1 M_2^{\mathbf{L}_2}\ ({}^{\mathbf{L}_2}M_2 M_1^{\tau'}\ \mathbf{v}))]_{M_1} & \mapsto & \textbf{Error: Bad conversion} \\
 & & (\text{where } \tau \neq \tau')
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{e} & = & \mathsf{x} \mid \mathsf{v} \mid (\mathsf{e}\ \mathsf{e}) \mid (op\ \mathsf{e}\ \mathsf{e}) \mid (\texttt{if0}\ \mathsf{e}\ \mathsf{e}\ \mathsf{e}) \mid ({}^{\sigma}M_2 M_1^{\tau}\ \mathbf{e}) \\
\mathsf{v} & = & (\lambda \mathsf{x} : \sigma.\ \mathsf{e}) \mid \overline{n} \mid ({}^{\mathbf{L}_2}M_2 M_1^{\tau}\ \mathsf{v}) \\
\sigma & = & \mathbf{Nat} \mid \sigma \to \sigma \mid \mathbf{L}_2 \\
\mathsf{E} & = & [\,]_{M_2} \mid (\mathsf{E}\ \mathsf{e}) \mid (\mathsf{v}\ \mathsf{E}) \mid (op\ \mathsf{E}\ \mathsf{e}) \mid (op\ \mathsf{v}\ \mathsf{E}) \mid \\
 & & (\texttt{if0}\ \mathsf{E}\ \mathsf{e}\ \mathsf{e}) \mid ({}^{\sigma}M_2 M_1^{\tau}\ \mathsf{E})
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathcal{E}[({}^{\sigma}M_2 M_1^{\mathbf{L}_1}\ ({}^{\mathbf{L}_1}M_1 M_2^{\sigma}\ \mathsf{v}))]_{M_2} & \mapsto & \mathcal{E}[\mathsf{v}] \\
\mathcal{E}[({}^{\sigma}M_2 M_1^{\mathbf{L}_1}\ ({}^{\mathbf{L}_1}M_1 M_2^{\sigma'}\ \mathsf{v}))]_{M_2} & \mapsto & \textbf{Error: Bad conversion} \\
 & & (\text{where } \sigma \neq \sigma')
\end{array}
$$

Figure 3.7: An ML-in-ML lump embedding

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 3.6, 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:

$\mathrm{P} \stackrel{\text{def}}{=} (\mathit{lambdam{:}x{:}l1\text{->}int\text{->}int.}\ ({}^{\mathbf{L}_1} M_1 M_2^{\mathbf{Nat}\to\mathbf{L}_2}\ (\lambda \mathbf{y} : \mathbf{Nat}.\ (\mathit{m2m1{:}l2l1\text{->}int\text{->}int}\ \mathbf{x}))))$
$\mathrm{U} \stackrel{\text{def}}{=} (\lambda \mathbf{x} : \mathbf{L}_1.\ (\mathit{m1m2{:}l1\text{->}int\text{->}intl2}\ (({}^{\mathbf{Nat}\to\mathbf{L}_2} M_2 M_1^{\mathbf{L}_1}\ \mathbf{x}))\ \overline{0}))$
$\mathrm{Y} \stackrel{\text{def}}{=} \mathit{lambdaf{:}int\text{->}int\text{->}int\text{->}int.}$
$\quad ((\mathit{lambdax{:}lumpo.}\ (f\ (\mathit{lambdam{:}int.}\ (((\mathrm{U}\ x)\ x)\ m))))$
$\quad\ (\mathrm{P}\ (\mathit{lambdax{:}lumpo.}\ (f\ (\mathit{lambdam{:}int.}\ (((\mathrm{U}\ x)\ x)\ m))))))$

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 3.8. It makes changes analogous the changes made our ML-and-Scheme system between Sections 3.2 and 3.3: 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 3.7, we have omitted several standard rules for language features that do not directly bear on interoperation.

Theorem 3.4.6. *If e is a program in the language of Figure 3.8 and $\vdash_M e : \tau$, then $e \mapsto^* v$.*

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

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

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

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

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

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

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

Figure 3.8: An ML-in-ML natural embedding

---

for M_2 (though since these "relations" are unary, they are perhaps better called logical predicates):

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

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

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

Lemma 3.4.7. *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* $\gamma(e) \in \mathcal{C}_2[\![\sigma]\!]$, *then* $\gamma({}^{\tau}M_1 M_2^{\sigma}\ e) \in \mathcal{C}_1[\![\tau]\!]$.

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

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

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

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

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

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

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

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

Lemma 3.4.8. *Both of the following propositions hold:*

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

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

Proof. By simultaneous induction on the typing derivations for $\mathbf{e}$ 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 $\mathsf{v} \in \mathcal{V}_2[\![\sigma]\!]$ 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 3.4.7 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 3.4.7 instead of case 1. □

The main theorem is immediate from lemma 3.4.8. Thus for the ML-in-ML language, the natural embedding is strictly less powerful than the lump embedding.

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

Another caveat is also in order. As we said in section 3.2, 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).)

3.5 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

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

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

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

Figure 3.9: Exceptions system 1 reduction rules

$$
\begin{array}{lcl}
\mathbf{e} & = & \cdots \mid (\texttt{handle}\ \mathbf{e}\ \mathbf{e}) \mid (\texttt{raise}\ str) \\
\mathsf{e} & = & \cdots \mid (\texttt{handle}\ \mathsf{e}\ \mathsf{e})
\end{array}
$$

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

$$
\mathcal{E} \;=\; \mathbf{E}
$$

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

$$
\begin{array}{lcl}
\mathcal{E}[(\texttt{handle}\ \mathbf{e}\ \mathbf{v})]_M & \mapsto & \mathcal{E}[\mathbf{v}] \\
\mathcal{E}[(\texttt{handle}\ \mathsf{e}\ \mathsf{v})]_S & \mapsto & \mathcal{E}[\mathsf{v}] \\
\mathcal{E}[(\texttt{handle}\ \mathbf{e}\ \mathbf{H}[(\texttt{raise}\ str)]_M)]_M & \mapsto & \mathcal{E}[\mathbf{e}] \\
\mathcal{E}[(\texttt{handle}\ \mathsf{e}\ \mathsf{H}[(\texttt{wrong}\ str)]_S)]_S & \mapsto & \mathcal{E}[\mathsf{e}]
\end{array}
$$

$$
\mathbf{H}[(\texttt{raise}\ str)]_M \;\mapsto\; \textbf{Error}\text{: } str
$$

$$
\begin{array}{lcl}
\mathcal{E}[(GSM^{\tau}\mathbf{H}[(\texttt{raise}\ str)]_M)]_S & \mapsto & \mathcal{E}[(\texttt{wrong}\ str)] \\
\mathcal{E}[(MSG^{\tau}\mathsf{H}[(\texttt{wrong}\ str)]_S)]_M & \mapsto & \mathcal{E}[(\texttt{raise}\ str)]
\end{array}
$$

Figure 3.10: Exceptions system 2 reduction rules

that works the same way. In both systems a completely unhandled exception causes the program to terminate with an error message.

3.5.1 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 3.9 as an extension to the natural embedding of Figure 3.4. 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 3.1, 3.2, and 3.4). 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 3.9 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 3.5.1. *Exceptions system 1 is type-sound.*

Proof. The proof of theorem 3.2.1, *mutatis mutandis*. □

3.5.2 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 3.9 in the last two rules. Figure 3.10 presents the rules as an extension to the natural embedding of Figure 3.4: everything here is the same as it was in Figure 3.9 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.

Theorem 3.5.2. *Exceptions system 2 is type-sound.*

Proof. The proof of theorem 3.2.1, *mutatis mutandis*. □

3.6 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 to support systems where conversion is not driven entirely by the static types on either side of a boundary. For instance, Figure 3.11 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

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

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

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

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

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

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

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

Figure 3.11: Extensions to Figure 3.4 for mapped embedding 1

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

$$
\begin{array}{rcl}
\mathbf{E} & = & \cdots \mid (MSG^{\kappa}\ \mathsf{E}) \\
\mathsf{H},\ \mathsf{F} & & \text{as in Figures 3.9 and 3.10} \\
\mathsf{E} & = & \mathsf{F} \mid \mathsf{F}[GSM^{\kappa}\mathbf{E}]_S
\end{array}
$$

$$
\kappa \ = \ \mathbf{Nat} \mid \mathbf{Nat!} \mid \kappa \to \kappa
$$

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

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

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

Figure 3.12: Extensions to Figure 3.4 for mapped embedding 2

strings by built-in Scheme functions *string→path* and *path→string* (which we model in Figure 3.11 by appealing to unspecified mappings *sp* for string to path conversion and *ps* for path to string conversion; the former is partial to capture the idea that not all strings correspond to valid paths). A reasonable foreign function interface might want ML programs to be able to provide strings to Scheme programs that expect paths, but that conversion would not be type-directed.

We allow for this more permissive kind of multilanguage system by generalizing the annotations on boundaries so that they are conversion strategies rather than being restricted to types. In this case, we can define the conversion strategy κ as in Figure 3.11 and generalize boundaries to be of the form GSM^{κ} and MSG^{κ}, 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 MSG^{π} boundary). The reduction rules use these annotations to decide whether to introduce a call to *string→path* when passing an ML string into Scheme.

Theorem 3.6.1. *The language of Figure 3.11 is type-sound.*

Proof. The proof of theorem 3.2.1, *mutatis mutandis*. □

While the language in Figure 3.11 only affects value conversions, conversion strategies can impact control flow as well. For instance, since C does not have an exception mechanism, many C functions (e.g., `malloc`, `fopen`, `sqrt`) return a normal value on success and a sentinel value to indicate an error. A foreign function interface might automatically convert error indicators from such functions into exceptions, while converting non-errors as normal. We can model that choice by combining one of they exception-handling systems of Section 3.5 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 3.5.1) 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 is a conversion strategy κ and an associated $\lfloor\ \rfloor$ metafunction that maps elements of κ to types; these are presented in Figure 3.12 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 3.5.1. ML's typing judgments are also just as in Section 3.5.1, 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 3.6.2. *The language of Figure 3.12 is type-sound.*

Proof. The proof of theorem 3.2.1, *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.

CHAPTER 4
PARAMETRIC POLYMORPHISM

4.1 Introduction

There have been two major strategies for hiding the implementation details of one part of a program from its other parts: the static approach and the dynamic approach.

The static approach can be summarized by the slogan "information hiding = parametric polymorphism." In it, the language's type system is equipped with a facility such as existential types so that it can reject programs in which one module makes unwarranted assumptions about the internal details of another, even if those assumptions happen to be true. This approach rests on Reynolds' notion of abstraction (Reynolds 1983), later redubbed the "parametricity" theorem by Wadler (Wadler 1989).

The dynamic approach, which goes back to Morris (Morris, Jr. 1973), can be summarized by the alternate slogan "information hiding = local scope + generativity." Rather than statically rejecting programs that make unwarranted assumptions, the dynamic approach simply takes away programs' ability to see if those assumptions are correct. It allows a programmer to *dynamically seal* values by creating unique keys (*create-seal* $: \rightarrow key$) and using those keys with locking and unlocking operations (*seal* $: v \times key \rightarrow opaque$ and *unseal* $: opaque \times key \rightarrow v$ respectively). A value locked with a particular key is opaque to third parties: nothing can be done but unlock it with the same key. Here is a simple implementation written in Scheme, where **gensym** is a function that generates a new, completely unique symbol every time it is called:

```
(define (create-seal) (gensym))
(define (seal v s1) (λ (s2) (if (eq? s1 s2) v (error))))
(define (unseal sealed-v s) (sealed-v s))
```

Using this facility a module can hand out a particular value while hiding its representation by creating a fresh seal in its private lexical scope, sealing the value and hand the result to clients, and then unsealing it again whenever it returns. This is the primary information-hiding mechanism in many untyped languages. For instance PLT Scheme (Flatt 1997) uses generative `structs`, essentially a (much) more sophisticated version of seals, to build abstractions for a great variety of programming constructs such as an object system. Furthermore, the idea has seen some use recently even in languages whose primary information-hiding mechanism is static, as recounted by Sumii and Pierce (Sumii and Pierce 2004).

Both of these strategies seem to match an intuitive understanding of what information-hiding ought to entail. So it is surprising that a fundamental question — what is the relationship between the guarantee provided by the static approach and the dynamic approach? — has not been answered in the literature.

In this chapter we take a new perspective on the problem, posing it as a question of parametricity in a multi-language system using the techniques of the previous chapter. Specifically, we introduce a new embedding that embeds System F (which will be our new "ML" for this chapter) and Scheme using dynamic seals. We then show two results. First, in section 4.4 we show that dynamic sealing preserves ML's parametricity guarantee even when interoperating with Scheme. For the proof, we define two step-indexed logical relations (Ahmed 2006), one for ML (indexed by both types as well as, intuitively, the number of steps available for future evaluation) and one for Scheme (indexed only by steps since Scheme is untyped). The stratification provided by step-indexing is essential for modeling the recursive functions present in both languages (available in Scheme via encoding `fix`, and available in ML via interactions with Scheme). Then we show the fundamental theorems of each relation. The novelty of this proof is its use of what we call the "bridge lemma," which states that if two terms are related in one language, then wrapping those terms in boundaries results in terms that are related in the other. The proof is otherwise essentially standard. Second, in section 4.5 we restrict our attention to Scheme programs that use boundaries with ML only to implement a contract system (Findler and Felleisen 2002). Appealing to the first parametricity result, we give a more useful, contract-indexed relation for dealing with these terms and prove that it relates contracted terms to themselves. In section 4.5.1 we show that our notion of contracts corresponds to Findler and Felleisen's,

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

$$\frac{\Delta, \alpha; \Gamma \vdash_M \mathbf{e} : \tau}{\Delta; \Gamma \vdash_M (\Lambda\alpha.\mathbf{e}) : \forall\alpha.\tau}$$

$$\frac{\Delta; \Gamma \vdash_M \mathbf{e} : \forall\alpha.\tau' \quad \Delta \vdash \tau}{\Delta; \Gamma \vdash_M \mathbf{e}\langle\tau\rangle : \tau'[\tau/\alpha]}$$

$$\mathcal{E}[(\Lambda\alpha.\mathbf{e})\langle\tau\rangle]_M \mapsto \mathcal{E}[\mathbf{e}[\tau/\alpha]]$$

$$\mathcal{E}[({}^{\forall\alpha.\tau}MS\ \mathsf{v})]_M \mapsto \mathcal{E}[(\Lambda\alpha.({}^{\tau}MS\ \mathsf{v}))]$$

$$\mathcal{E}[(SM^{\forall\alpha.\tau}\ \mathbf{v})]_S \mapsto \mathcal{E}[(SM^{\tau[\mathbf{L}/\alpha]}\ \mathbf{v}\langle\mathbf{L}\rangle)]$$

$$\mathcal{E}[(SM^{\mathbf{L}}\ ({}^{\mathbf{L}}MS\ \mathsf{v}))]_S \mapsto \mathcal{E}[\mathsf{v}]$$

Figure 4.1: Extensions to figure 3.4 for non-parametric polymorphism

and to a translation given by Sumii and Pierce (Sumii and Pierce 2004, section 8). In section 4.6 we re-work examples from Wadler's "Theorems for Free!" in the untyped setting; we call these "theorems for low, low prices" because the theorems are not quite "free" consequences of terms' type-soundness; they only apply to terms that have been dynamically forced to behave properly by a contract.

4.2 Polymorphism, attempt one

An omission from the "ML" side of the natural embedding to this point is that it contains no polymorphism. We now extend it to support polymorphism by replacing the simply-typed lambda calculus with System F. When we do so, we immediately hit the question of how to properly handle boundaries. In this subsection we make what we consider the most straightforward decision of how to handle boundaries and show that it results in a system that does not preserve System F's parametricity property; in the next subsection we refine our strategy using dynamic sealing techniques.

Figure 4.1 shows the extensions we need to make to figure 3.4 to support non-parametric polymorphism. To ML's syntax we add type abstractions ($\Lambda\alpha.\ \mathbf{e}$) and type application ($\mathbf{e}\langle\tau\rangle$); to its types we add $\forall\alpha.\ \tau$ 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 $(\lambda x.x)$ 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 $({}^{\forall\alpha.\tau}MS\ \mathsf{v})$ 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 $(\Lambda\alpha.\ ({}^{\tau}MS\ \mathsf{v}))$.

This works for ML, but making a dual of it in Scheme would be somewhat silly, since every Scheme value inhabits the same type so type abstraction and application forms would be useless. Instead, we would like to allow Scheme to use an ML value of type, say, $\forall\alpha.\alpha \to \alpha$ directly as a function. To make boundaries with universally-quantified types behave that way, when we convert a polymorphic ML value to a Scheme value we need to remove its initial type-abstraction by applying it to some type and then convert the resulting value according to the resulting type. As for which type to apply it to, we need a type to which we can reliably convert any Scheme value, though it must not expose any of those values' properties. In prior work, 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 **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 4.1.

Theorem 4.2.1. *The polymorphic natural embedding is type-sound.*

4.3 Polymorphism, attempt two

Although this embedding is type safe, the polymorphism is not parametric in the sense of Reynolds (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

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

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

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

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

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

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

$$\frac{\Delta;\Gamma \vdash_S \mathsf{e} : \mathbf{TST} \quad \Delta \vdash \lfloor\kappa\rfloor}{\Delta;\Gamma \vdash_M ({}^{\kappa}MS\ \mathsf{e}) : \lfloor\kappa\rfloor} \qquad \frac{\Delta;\Gamma \vdash_M \mathbf{e} : \lfloor\kappa\rfloor \quad \Delta \vdash \lfloor\kappa\rfloor}{\Delta;\Gamma \vdash_S (SM^{\kappa}\ \mathbf{e}) : \mathbf{TST}}$$

$$
\begin{array}{rl}
\mathcal{E}[(SM^{\forall\alpha.\tau}\ \mathbf{v})]_S & \mapsto \mathcal{E}[(SM^{\tau[\mathbf{L}/\alpha]}\ \mathbf{v}\langle\mathbf{L}\rangle)] \\
\mathcal{E}[(SM^{\mathbf{L}}\ ({}^{\mathbf{L}}MS\ \mathsf{v}))]_S & \mapsto \mathcal{E}[\mathsf{v}] \\
\\
\mathcal{E}[(\Lambda\alpha.\mathbf{e})\langle\tau\rangle]_M & \mapsto \mathcal{E}[\mathbf{e}[\tau/\alpha]] \\
\mathcal{E}[({}^{\forall\alpha.\kappa}MS\ \mathsf{v})]_M & \mapsto \mathcal{E}[(\Lambda\alpha.({}^{\kappa}MS\ \mathsf{v}))] \\
\mathcal{E}[({}^{\langle\alpha;\tau\rangle}MS\ (SM^{\langle\alpha;\tau\rangle}\ \mathbf{v}))]_M & \mapsto \mathcal{E}[\mathbf{v}] \\
\mathcal{E}[({}^{\langle\alpha;\tau\rangle}MS\ \mathsf{v})]_M & \mapsto \mathbf{Error}\text{: bad value} \\
& \quad (\text{where } \mathsf{v} \neq SM^{\langle\alpha;\tau\rangle}\ \mathbf{v} \text{ for any } \mathbf{v})
\end{array}
$$

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

Figure 4.2: Extensions to figure 3.4 to support parametric polymorphism

$$(\lambda\mathbf{y}.({}^{\mathbf{Nat}}MS((\lambda\mathsf{x}.(+\ \mathsf{x}\ \overline{1}))(SM^{\mathbf{Nat}}\mathbf{y}))))$$

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 use dynamic seals to protect ML values whose implementation should not be observed. When ML provides Scheme with a value whose original type was α, Scheme gets a sealed value; when Scheme returns a value to ML at a type that was originally α, ML unseals it or signals an error if it is not a sealed value with the appropriate key.

This means that we can no longer directly substitute types for free type variables on boundary annotations. Instead we introduce *seals* as type-like annotations of the form $\langle\alpha;\tau\rangle$ that indicate on a boundary's type annotation that a particular type is the instantiation of what was originally a type variable. We call types that can contain such seals *conversion schemes* and use κ to range over them; we also use the *seal erasure* metafunction $\lfloor\ \rfloor$ to project conversion schemes to types. Figure 4.2 defines these changes precisely. One final

subtlety not written in figure 4.2 is that we treat a seal $\langle\alpha;\tau\rangle$ as a free occurrence of α for the purposes of capture-avoiding substitution, and we treat boundaries that include $\forall\alpha.\tau$ types as though they were binding instances of α. In fact, the production of fresh names by capture-avoiding substitution corresponds exactly to the production of fresh seals for information hiding, and the system would be neither parametric nor even type-sound were we to omit this detail.

Theorem 4.3.1. *The embedding presented in figure 4.2 is type-sound.*

4.4 Parametricity

In this section we establish that the language of figure 4.2 is parametric, in the sense that all terms in the language map related environments to related results, using a syntactic logical relation. Our parametricity property does not establish the exact same equivalences that would hold for terms in plain System F, but only because the embedding we are considering gives terms the power to diverge and to signal errors. So, for example, we cannot show that any ML value of type $\forall\alpha.\alpha \to \alpha$ must be the identity function, but we *can* show that it must be either the identity function, the function that always diverges, or the function that always signals an error.

Our proof amounts to defining two logical relations, one for ML and one for Scheme (see figure 4.3) and proving that the ML (Scheme) relation relates each ML (Scheme) term to itself regardless of the interpretation of free type variables. Though logical relations in the literature are usually defined by induction on types, we cannot use a type-indexed relation for Scheme since Scheme has only one type. This means in particular that the arguments to function values have types that are as large as the type of the function values themselves; thus any relation that defines two functions to be related if the results are related for any pair of related arguments would not be well-founded. Instead we use a minor adaptation of the step-indexed logical relation for recursive types given by Ahmed (Ahmed 2006): our Scheme logical relation is indexed by the number of steps k available for computation. Intuitively, two any values are related for k steps if they cannot be distinguished by any computation running for no more than k steps.

Since we are interested in proving properties of ML terms that may contain Scheme subterms, the ML relation must also be step-indexed — if the Scheme subterms are only related for (say) 50 steps, then the ML terms cannot always be related for arbitrarily many steps. Thus, the ML relation is indexed by both types and steps (as in Ahmed (Ahmed 2006)).

The definitions are largely independent (though we do make a few concessions on this front, in particular at the definition of the ML relation at type $\mathbf{L}$), but the proofs cannot be, since an ML term can have an embedded Scheme subexpression and vice versa. Instead, we prove the two claims by simultaneous induction and rely on a critical "bridge lemma" (lemma 4.4.2, see below) that lets us carry relatedness between languages.

Preliminaries. A *type substitution* η is a partial function from type variables to closed types. We extend type substitutions to apply to types, conversion schemes, and terms as follows (we show the interesting cases, the rest are merely structural recursion):

$$\begin{array}{lcl}
\eta(\alpha) & \stackrel{\text{def}}{=} & \begin{cases} \tau & \text{if } \exists \eta'.\ \eta = \eta', \alpha : \tau \\ \alpha & \text{otherwise} \end{cases} \\
\eta({}^{\kappa}MS\ \mathsf{e}) & \stackrel{\text{def}}{=} & \mathbf{sl}(\eta,\kappa)MS\ \eta(\mathsf{e}) \\
\eta(SM^{\kappa}\ \mathbf{e}) & \stackrel{\text{def}}{=} & SM^{\mathbf{sl}(\eta,\kappa)}\ \eta(\mathbf{e})
\end{array}$$

The boundary cases (which use the seal metafunction $\mathbf{sl}(\cdot,\cdot)$ defined below) are different from the regular type cases. When we close a type with respect to a type substitution η, we simply replace all occurrences of free variables with their mappings in η, but when we close a conversion scheme with respect to a type substitution we replace free variables with "sealed" instances of the types in η. The effect of this is that even when we have performed a type substitution, we can distinguish between a type that was concrete in the original program and a type that was abstract in the original program but has been substituted with a concrete type. The $\mathbf{sl}(\cdot,\cdot)$ metafunction maps a type τ (or more generally a conversion scheme κ) to an isomorphic conversion scheme κ where each instance of each type variable that occurs free in τ is replaced by an appropriate sealing declaration, if the type variable is in the domain of η.

Definition 4.4.1 (sealing)**.** The metafunction $\mathbf{sl}(\eta,\kappa)$ is defined as follows:

$$\mathbf{sl}(\cdot,\cdot) : \eta \times \kappa \to \kappa$$

$$\mathbf{sl}(\eta,\alpha) \stackrel{\text{def}}{=} \begin{cases} \langle \alpha;\eta(\alpha)\rangle & \text{if } \eta(\alpha) \text{ is defined} \\ \alpha & \text{otherwise} \end{cases}$$

$$\mathbf{sl}(\eta,\langle \alpha;\tau\rangle) \stackrel{\text{def}}{=} \langle \alpha;\tau\rangle$$

$$\mathbf{sl}(\eta,\mathbf{Nat}) \stackrel{\text{def}}{=} \mathbf{Nat}$$

$$\mathbf{sl}(\eta,\kappa_1 \to \kappa_2) \stackrel{\text{def}}{=} \mathbf{sl}(\eta,\kappa_1) \to \mathbf{sl}(\eta,\kappa_2)$$

$$\mathbf{sl}(\eta,\forall\alpha.\kappa_1) \stackrel{\text{def}}{=} \forall\alpha.\mathbf{sl}(\eta,\kappa_1)$$

A *type relation* δ is a partial function from type variables to triples $(\tau_1,\tau_2,\mathbf{R})$, where τ_1 and τ_2 are closed types and R is a set of triples of the form $(k,\mathbf{v}_1,\mathbf{v}_2)$ (which intuitively means that $\mathbf{v}_1$ and $\mathbf{v}_2$ are related for k steps). We use the following notations: If $\delta(\alpha) = (\tau_1,\tau_2,\mathbf{R})$ then $\delta_1(\alpha) = \tau_1$, $\delta_2(\alpha) = \tau_2$, and $\delta_R(\alpha) = \mathbf{R}$. We also treat δ_1 and δ_2 as type substitutions. In the definition of the logical relation we only allow *downward closed* relations as choices for $\mathbf{R}$; *i.e.* relations that relate two values for k steps must also relate them for all $j < k$ steps. We make this restriction because downward closure is a critical property of the logical relation that would not otherwise hold.

Two final bits of notation: first, a Scheme (ML) substitution γ_S (γ_M) is a partial map from Scheme (ML) variables to closed Scheme (ML) values (and a substitution $\gamma = \gamma_S \cup \gamma_M$ for some γ_S, γ_M). Second, we say that $e \hookrightarrow v$ (or **Error**: s) if in all evaluation contexts $\mathcal{E}[e] \mapsto \mathcal{E}[v]$ (or **Error**: s).

Lemma 4.4.2 (bridge lemma)**.** *For all $k \geq 0$, type environments Δ, type relations δ such that $\Delta \vdash \delta$, types τ such that $\Delta \vdash \tau$, both of the following hold:*

1. *For all e_1 and e_2, if $\delta \vdash e_1 \lesssim_S^k e_2 : \textbf{\textit{TST}}$ then $\delta \vdash ({}^{\mathbf{sl}(\delta_1,\tau)}MS\ e_1) \lesssim_M^k ({}^{\mathbf{sl}(\delta_2,\tau)}MS\ e_2) : \tau$.*
2. *For all $\mathbf{e}_1$ and $\mathbf{e}_2$, if $\delta \vdash \mathbf{e}_1 \lesssim_M^k \mathbf{e}_2 : \tau$, then $\delta \vdash (SM^{\mathbf{sl}(\delta_1,\tau)}\ \mathbf{e}_1) \lesssim_S^k (SM^{\mathbf{sl}(\delta_2,\tau)}\ \mathbf{e}_2) : \textbf{\textit{TST}}$.*

With the bridge lemma established, the fundamental theorem is essentially standard. We must restrict the parametricity theorem to seal-free terms; otherwise we would have to show that any sealed value is related to itself at type α which is obviously false. (A conversion strategy is seal-free if it contains no instances of $\langle \alpha;\tau\rangle$ for any α. A term is seal-free if it contains no conversion strategies with seals.) A consequence of the fundamental theorem is that logical approximation implies contextual approximation.

Theorem 4.4.3 (parametricity / fundamental theorem)**.** *For all seal-free terms $\mathbf{e}$ and e:*

1. *If* $\Delta;\Gamma \vdash_M \boldsymbol{e} : \tau$, *then* $\Delta;\Gamma \vdash \boldsymbol{e} \lesssim_M \boldsymbol{e} : \tau$.

2. *If* $\Delta;\Gamma \vdash_S \boldsymbol{e} : \boldsymbol{TST}$, *then* $\Delta;\Gamma \vdash \boldsymbol{e} \lesssim_S \boldsymbol{e} : \boldsymbol{TST}$.

Proof. By simultaneous induction on the derivations $\Delta;\Gamma \vdash_M \mathbf{e} : \tau$ and $\Delta;\Gamma \vdash_S \mathsf{e} : \mathbf{TST}$. The boundary cases both follow from lemma 4.4.2. □

Theorem 4.4.3 establishes a very strong reasoning principle for our embedding. Here we given an example of how it can be used by proving that the only values of type $\forall\alpha.\alpha \to \alpha$ are now either the function that always diverges, or the function that always signals an error, or the identity function.

Theorem 4.4.4. *If* $; \vdash_M \boldsymbol{v} : \forall\alpha.\alpha \to \alpha$, *then one of the following holds:*

1. $\forall\tau$, $\boldsymbol{v}'$ *such that* $; \vdash_M \boldsymbol{v}' : \tau$, $(\boldsymbol{v}\langle\tau\rangle\ \boldsymbol{v}')$ *diverges;*

2. $\forall\tau$, $\boldsymbol{v}'$ *such that* $; \vdash_M \boldsymbol{v}' : \tau$, $(\boldsymbol{v}\langle\tau\rangle\ \boldsymbol{v}') \mapsto^*$ **Error***: s for some s; or*

3. $\forall\tau$, $\boldsymbol{v}'$ *such that* $; \vdash_M \boldsymbol{v}' : \tau$, $(\boldsymbol{v}\langle\tau\rangle\ \boldsymbol{v}') \mapsto^* \boldsymbol{v}'$.

Proof. By theorem 4.4.3, $\forall k. \vdash \mathbf{v} \lesssim^k_M \mathbf{v} : \forall\alpha.\alpha \to \alpha$, hence $\forall k, \mathbf{R} \in \mathbf{Rel}_{\tau_1,\tau_2}.\alpha : \mathbf{R} \vdash \mathbf{v}\langle\tau_1\rangle \lesssim^k_M \mathbf{v}\langle\tau_2\rangle : \alpha \to \alpha$. Choose $\tau_1, \tau_2 = \tau$ and choose $\mathbf{R} = \{(k, \mathbf{v}', \mathbf{v}') \mid k \in \mathbb{N}\}$. We have by definition of the logical relation that either the application diverges, signals an error, or produces $\mathbf{v}'$, each of which corresponds to one of the three cases in our claim.

To establish that the same case of the claim holds for *any* choice we make, assume the opposite, that there exist $\tau'_1, \mathbf{v}'_1$ and $\tau'_2, \mathbf{v}'_2$ such that $(\mathbf{v}\langle\tau'_1\rangle\ \mathbf{v}'_1)$ falls under one case of the claim and $(\mathbf{v}\langle\tau'_2\rangle\ \mathbf{v}'_2)$ falls under another. This is impossible: going back to the definition of the logical relation, we can also choose $\tau_1 = \tau'_1$, $\tau_2 = \tau'_2$, and $\mathbf{R} = \{(k, \mathbf{v}'_1, \mathbf{v}'_2) \mid k \in \mathbb{N}\}$, and the definition shows that the resulting type and value applications must behave the same way: either both diverge, both produce the same error, or both produce values. □

4.5 From multi-language to single-language sealing

Suppose that instead of reasoning about multi-language programs, we want to reason about Scheme terms but also to use a closed ML type τ as a behavioral specification for a Scheme term — **Nat** means the term must evaluate to a number, **Nat** $\to$ **Nat** means the term must evaluate to a function that returns a number under the promise that the context will always provide it a number, and so on. We can implement this using boundaries with the program fragment $\mathsf{e}^{\tau} = SM^{\tau}\ ({}^{\tau}MS\ \mathsf{e})$.

It is easy to check that such terms are always well-typed as long as e itself is well-typed. Therefore, since we have defined a contract as just a particular usage pattern for boundaries, we have by virtue of theorem 4.4.3 that every contracted term corresponds to itself, so intuitively every contracted term of polymorphic type should behave parametrically. However, the logical relation we defined in the previous section is not particularly convenient for proving facts about contracted Scheme terms, so instead we give another relation in figure 4.4 that we think of as the "contracted-Scheme-terms" relation, which gives criteria for two Scheme terms being related at an ML type (which we now interpret as a behavioral contract). Here σ is an *untyped* mapping from type variables α to downward-closed relations R on Scheme values: that is, $\sigma = (\alpha_1 \mapsto \mathsf{R}_1, \cdots, \alpha_n \mapsto \mathsf{R}_n)$ where each $\mathsf{R}_i \in \mathsf{Rel}$ (see figure 4.4).

Our goal is to prove that closed, contracted terms are related to themselves under this relation. Proving this directly is intractable, but we can prove it by showing that boundaries "reflect their relations"; *i.e.* that if $\delta \vdash \mathbf{e}_1 \lesssim_M^k \mathbf{e}_2 : \tau$ then for some appropriate σ we have that $\sigma \vdash (SM^{\tau}\ \mathbf{e}_1) \leq^k (SM^{\tau}\ \mathbf{e}_2) : \tau$ and vice versa; this is the claim we show in lemma 4.5.4 (bridge lemma 2) below, and the result we want is an easy corollary when combined with theorem 4.4.3. Before we can precisely state the claim, though, we need some machinery for specifying what relationship between δ and σ we want to hold.

Definition 4.5.1 (hybrid environments)**.** An hybrid environment ϕ is a partial map from type variables to tuples of the form (S, R) or $(\mathrm{M}, \tau_1, \tau_2, \mathbf{R})$.

The intuition is that a hybrid environment is a tagged union of a Scheme environment σ (each element of which is tagged with S) and an ML environment δ (each element of which is tagged with M). Given such a hybrid environment, one can mechanically derive both

a Scheme and an ML representation of it by keeping native elements as-is and wrapping foreign elements in the appropriate boundary:

Definition 4.5.2 (Scheme and ML projections of hybrid environments)**.** For a hybrid environment ϕ, if $\phi(\alpha) = (\mathrm{S}, \mathsf{R})$, then:

$$\begin{array}{lcl} \sigma_\phi(\alpha) & \stackrel{\text{def}}{=} & \mathsf{R} \\ \delta_\phi(\alpha) & \stackrel{\text{def}}{=} & (\mathbf{L},\ \mathbf{L},\ \{(k,\ (^{\mathbf{L}}MS\ \mathsf{v}_1),\ (^{\mathbf{L}}MS\ \mathsf{v}_2)) \mid (k, \mathsf{v}_1, \mathsf{v}_2) \in \mathsf{R}\}) \end{array}$$

If $\phi(\alpha) = (\mathrm{M}, \tau_1, \tau_2, \mathbf{R})$, then:

$$\begin{array}{lcl} \sigma_\phi(\alpha) & \stackrel{\text{def}}{=} & \{(k,\ (SM^{\langle\alpha;\tau_1\rangle}\ \mathbf{v}_1),\ (SM^{\langle\alpha;\tau_2\rangle}\ \mathbf{v}_2)) \mid (k, \mathbf{v}_1, \mathbf{v}_2) \in \mathbf{R}\} \\ \delta_\phi(\alpha) & \stackrel{\text{def}}{=} & (\tau_1, \tau_2, \mathbf{R}) \end{array}$$

We say that $\Delta \vdash \phi$ if for all $\alpha \in \Delta$, $\phi(\alpha)$ is defined, and if $\phi(\alpha) = (\mathrm{S}, \mathsf{R})$ then $\mathsf{R} \in \mathsf{Rel}$, and if $\phi(\alpha) = (\mathrm{M}, \tau_1, \tau_2, \mathbf{R})$ then $\mathbf{R} \in \mathbf{Rel}_{\tau_1,\tau_2}$. We also implicitly use the fact that if $\Delta \vdash \phi$, then $\Delta \vdash \sigma_\phi$ and $\Delta \vdash \delta_\phi$, and that if $\phi' = \phi, \alpha \mapsto t$ for some tuple, then there exist σ' and δ' such that $\sigma' = \sigma_\phi, \alpha \mapsto t'$ and $\delta' = \delta_\phi, \alpha \mapsto t''$ for some tuples t' and t''. Finally, we define operations $c_1(\cdot,\cdot)$ and $c_2(\cdot,\cdot)$ (analogous to $\mathbf{sl}(\cdot,\cdot)$ defined earlier) from hybrid environments ϕ and types τ to conversion schemes κ:

Definition 4.5.3 (closing with respect to a hybrid environment)**.** For $i \in \{1,2\}$:

$$c_i(\phi,\alpha) \stackrel{\text{def}}{=} \begin{cases} \mathbf{L} & \text{if } \phi(\alpha) = (\mathrm{S}, \mathsf{R}) \\ \langle\alpha;\tau_i\rangle & \text{if } \phi(\alpha) = (\mathrm{M}, \tau_1, \tau_2, \mathbf{R}) \\ \alpha & \text{otherwise} \end{cases} \qquad \begin{array}{lcl} c_i(\phi, \mathbf{Nat}) & \stackrel{\text{def}}{=} & \mathbf{Nat} \\ c_i(\phi, \tau_1 \to \tau_2) & \stackrel{\text{def}}{=} & c_i(\phi,\tau_1) \to c_i(\phi,\tau_2) \\ c_i(\phi, \forall\alpha.\tau') & \stackrel{\text{def}}{=} & \forall\alpha.c_i(\phi,\tau') \end{array}$$

The interesting part of the definition is its action on type variables. Variables that ϕ maps to Scheme relations are converted to type $\mathbf{L}$, since when Scheme uses a polymorphic value in ML its free type variables are instantiated as $\mathbf{L}$. Similarly, variables that ϕ maps to ML relations are instantiated as seals because when ML uses a Scheme value as though it were polymorphic it uses dynamic seals to protect parametricity.

Now we can show that contracts respect the relation in figure 4.4 via a bridge lemma.

Lemma 4.5.4 (bridge lemma 2)**.** *For all $k \geq 0$, type environments Δ, hybrid environments ϕ such that $\Delta \vdash \phi$, τ such that $\Delta \vdash \tau$, and for all terms e_1, e_2, $\boldsymbol{e}_1$, $\boldsymbol{e}_2$:*

1. *If $\delta_\phi \vdash \boldsymbol{e}_1 \lesssim^k_M \boldsymbol{e}_2 : \tau$ then $\sigma_\phi \vdash (SM^{c_1(\phi,\tau)}\ \boldsymbol{e}_1) \leq^k (SM^{c_2(\phi,\tau)}\ \boldsymbol{e}_2) : \tau$.*

2. *If* $\sigma_\phi \vdash e_1 \leq^k e_2 : \tau$ *then* $\delta_\phi \vdash {}^{c_1(\phi,\tau)}MS\ e_1 \lesssim_M^k ({}^{c_2(\phi,\tau)}MS\ e_2) : \tau$.

Theorem 4.5.5. *For any seal-free term* e *such that* $\vdash_S e : \mathbf{TST}$ *and for any closed type* τ*, we have that for all* $k \geq 0$, $\vdash e^\tau \leq^k e^\tau : \tau$.

Proof. By theorem 4.4.3, for all $k \geq 0$, $\vdash ({}^\tau MS\ e) \lesssim_M^k ({}^\tau MS\ e) : \tau$. Thus, by lemma 4.5.4, we have that for all $k \geq 0$, $\vdash (SM^\tau\ ({}^\tau MS\ e)) \leq^k (SM^\tau\ ({}^\tau MS\ e)) : \tau$. □

Definition 4.5.6 (relational equality)**.** We write $\sigma \vdash e_1 = e_2 : \tau$ if for all $k \geq 0$, $\sigma \vdash e_1 \leq^k e_2 : \tau$ and $\sigma \vdash e_2 \leq^k e_1 : \tau$.

Corollary 4.5.7. *For any seal-free term* e *such that* $\vdash_S e : \mathbf{TST}$ *and for any closed type* τ*, we have that* $\vdash e^\tau = e^\tau : \tau$.

4.5.1 Dynamic sealing replaces boundaries

The contract system of the previous section is a multi-language system, but just barely, since the only part of ML we make any use of is its boundary form to get back into Scheme. In this section we restrict our attention to Scheme plus boundaries used only for the purpose of implementing contracts, and we show an alternate implementation of contracts that uses dynamic sealing. Rather than the concrete implementation of dynamic seals we gave in the introduction, we opt to use (a slight restriction of) the more abstract constructs taken from Sumii and Pierce's $\lambda_{\texttt{seal}}$ language (Sumii and Pierce 2004). Specifically, we use the following extension to our Scheme model:

$$
\begin{array}{lcll}
e & = & \cdots \mid \nu \mathsf{sx}.\ e \mid \{e\}_{se} \mid (\textbf{let}\ \{x\}_{se} = e\ \textbf{in}\ e) & \mathcal{E}[\nu \mathsf{sx}.\ e]_S \mapsto \mathcal{E}[e[\mathsf{sv}/\mathsf{sx}]] \\
v & = & \cdots \mid \{v\}_{sv} & \text{where sv fresh} \\
se & = & \mathsf{sx} \mid \mathsf{sv} & \mathcal{E}[(\textbf{let}\ \{x\}_{sv_1} = \{v\}_{sv_1}\ \textbf{in}\ e)]_S \mapsto \mathcal{E}[e_1[v/x]] \\
\mathsf{sx} & = & \text{[variables distinct from x]} & \mathcal{E}[(\textbf{let}\ \{x\}_{sv_1} = v\ \textbf{in}\ e)]_S \mapsto \textbf{Error}\text{: bad value} \\
\mathsf{sv} & = & \text{[unspecified, unique brands]} & \text{where } v \neq \{v'\}_{sv_1} \text{ for any } v' \\
E & = & \cdots \mid \{E\}_{sv} \mid (\textbf{let}\ \{x\}_{sv} = E\ \textbf{in}\ e) &
\end{array}
$$

We introduce a new set of seal variables sx that stand for seals (elements of sv) that will be computed at runtime. They are bound by νsx. e, which evaluates its body (e) with sx bound to a freshly-generated sv. Two operations make use of these seals. The first,

$\{\mathsf{e}\}_{\mathsf{se}}$, evaluates e to a value and then itself becomes an opaque value sealed with the key to which se evaluates. The second, $(\textbf{let } \{\mathsf{x}\}_{\mathsf{se}} = \mathsf{e}_1 \textbf{ in } \mathsf{e}_2)$, evaluates e_1 to a value; if that value is an opaque value sealed with the seal to which se evaluates, then the entire unsealing expression evaluates to e_2 with x bound to the value that was sealed, otherwise the expression signals an error.[1]

Using these additional constructs we can demonstrate that a translation essentially the same as the one given by Sumii and Pierce (Sumii and Pierce 2004, figure 4) does in fact generate parametrically polymorphic type abstractions. Their translation essentially attaches a higher-order contract (Findler and Felleisen 2002) τ to an expression of type τ (though they do not point this out). It extends Findler and Felleisen's notion of contracts, which does not include polymorphic types, by adding an environment ρ that maps a type variable to a tuple consisting of a seal and a symbol indicating the party (either $+$ or $-$ in Sumii and Pierce) that has the power to instantiate that type variable, and translating uses of type variable α in a contract to an appropriate seal or unseal based on the value of $\rho(\alpha)$. We define it as follows: when p and q are each parties ($+$ or $-$) and $p \neq q$,

$$
\begin{array}{lcl}
E^{p,q}_{\mathbf{Nat}}(\rho,\mathsf{e}) & = & (+\ \mathsf{e}\ 0) \\
E^{p,q}_{\tau*}(\rho,\mathsf{e}) & = & (\textbf{let } ((v\ e))\ (\mathit{if0}\ (\mathit{nil?}\ v) \\
 & & \qquad \mathit{nil} \\
 & & \qquad (\mathit{if0}\ (\mathit{pair?}\ v) \\
 & & \qquad\quad (\mathit{cons}\ E^{p,q}_{\tau}(\rho,(\mathsf{fst}\ \mathsf{v}))\ E^{p,q}_{\tau*}(\rho,(\mathsf{rst}\ \mathsf{v}))) \\
 & & \qquad\quad (\texttt{wrong "Non-list"})))) \\
E^{p,q}_{\tau_1\to\tau_2}(\rho,\mathsf{e}) & = & (\textbf{let } ((v\ e))\ (\mathit{if0}\ (\texttt{proc?}\ v) \\
 & & \qquad (\lambda\, x.\ E^{p,q}_{\tau_2}(\rho,(\mathsf{v}\ E^{q,p}_{\tau_1}(\rho,\mathsf{x})))) \\
 & & \qquad (\texttt{wrong "Non-procedure"}))) \\
E^{p,q}_{\forall\alpha.\tau'}(\rho,\mathsf{e}) & = & \nu\mathsf{sx}.\ E^{p,q}_{\mathsf{e}}(\rho,\alpha\mapsto(\mathsf{sx},q),\mathsf{e}) \\
E^{p,q}_{\alpha}(\rho,\alpha\mapsto(\mathsf{sx},p),\mathsf{e}) & = & \{\mathsf{e}\}_{\mathsf{sx}} \\
E^{p,q}_{\alpha}(\rho,\alpha\mapsto(\mathsf{sx},q),\mathsf{e}) & = & (\textbf{let } \{\mathsf{x}\}_{\mathsf{sx}} = \mathsf{e}\ \textbf{in}\ \mathsf{x})
\end{array}
$$

1. This presentation is a simplification of $\lambda_{\texttt{seal}}$ in two ways. First, in $\lambda_{\texttt{seal}}$ the key position for a sealed value or for an unseal statement may be an arbitrary expression, whereas here we syntactically restrict expressions that appear in those positions to be either seal variables or seal values. Second, in $\lambda_{\texttt{seal}}$ an unseal expression has an "else" clause that allows the program to continue even if an unsealing operation fails; we do not allow those clauses.

The differences between our translation and Sumii and Pierce's are as follows. First, we have mapped everything into our notation and adapted to our types (we omit booleans, tuples, and existential types and add numbers and lists). Second, our translations apply to arbitrary expressions rather than just variables. Third, because we are concerned with the expression violating parametricity as well as the context, we have to seal values provided by the context as well as by the expression, and our decision of whether to seal or unseal at a type variable is based on whether the party that instantiated the type variable is providing a value with that contract or expecting one. Fourth, we modify the result of $\forall\alpha.\tau$ so that it does not require application to a dummy value. (The reason we do this bears explanation. There are two components to a type abstraction in System F — abstracting over an interpretation of a variable and suspending a computation. Sumii and Pierce's system achieves the former by generating a fresh seal, and the latter by wrapping the computation in a lambda abstraction. In our variant, $\forall\alpha.\tau$ contracts still abstract over a free contract variable's interpretation, but they do not suspend computation; for that reason we retain fresh seal generation but eliminate the wrapper function.)

Definition 4.5.8 (boundary replacement)**.** $\mathcal{R}[\mathsf{e}]$ is defined as follows:

$$\mathcal{R}[\mathsf{e}^{\tau}] = E_{\tau}^{+,-}(\bullet, \mathcal{R}[\mathsf{e}]) \qquad \mathcal{R}[(\mathsf{e}_1\ \mathsf{e}_2)] = (\mathcal{R}[\mathsf{e}_1]\ \mathcal{R}[\mathsf{e}_2]) \qquad \ldots$$

Theorem 4.5.9 (boundary replacement preserves termination)**.** *If* $;\vdash_S \mathsf{e} : \boldsymbol{TST}$*, then* $\mathsf{e} \mapsto^* v_1 \Leftrightarrow \mathcal{R}[\mathsf{e}] \mapsto^* v_2$*, where* $v_1 = \overline{n} \Leftrightarrow v_2 = \overline{n}$*.*

This claim is a special case of a more general theorem that requires us to consider open contracts. The term $\mathsf{v}^{\forall\alpha.\alpha\to\alpha}$ where v is a procedure value reduces as follows:

$$\begin{array}{lll}
\mathsf{v}^{\forall\alpha.\alpha\to\alpha} & = & (SM^{\forall\alpha.\alpha\to\alpha}({}^{\forall\alpha.\alpha\to\alpha}MS\ \mathsf{v})) \\
& \mapsto^3 & (SM^{\mathbf{L}\to\mathbf{L}}({}^{\langle\alpha;\mathbf{L}\rangle\to\langle\alpha;\mathbf{L}\rangle}MS\ \mathsf{v})) \\
& \mapsto^2 & \lambda\mathsf{x}.(SM^{\mathbf{L}}\ ((\lambda\mathbf{y}:\mathbf{L}.\ ({}^{\langle\alpha;\mathbf{L}\rangle}MS\ (\mathsf{v}\ (SM^{\langle\alpha;\mathbf{L}\rangle}\ \mathbf{y}))))\ ({}^{\mathbf{L}}MS\ \mathsf{x}))) \\
& = & \lambda\mathsf{x}.(SM^{\mathbf{L}}\ ({}^{\langle\alpha;\mathbf{L}\rangle}MS\ (\mathsf{v}\ (SM^{\langle\alpha;\mathbf{L}\rangle}\ ({}^{\mathbf{L}}MS\ \mathsf{x})))))
\end{array}$$

Notice that the two closed occurrences of α in the original contracts become two different configurations of boundaries when they appear open in the final procedure. These correspond to the fact that negative and positive occurrences of a type variable with respect to its binder behave differently. Negative occurrences, of the form $(SM^{\langle\alpha;\mathbf{L}\rangle}\ ({}^{\mathbf{L}}MS\ldots))$, act

as dynamic seals on their bodies. Positive occurrences, of the form $(SM^{\mathbf{L}}\,(^{\langle\alpha;\mathbf{L}\rangle}MS\ldots))$, dynamically unseal the values their bodies produce. So, we write open contract variables as $\alpha-$ (for negative occurrences) and $\alpha+$ (for positive occurrences).

Now we are prepared to define another logical relation, this time between contracted Scheme terms and $\lambda_{\texttt{seal}}$ terms. We define it as follows, where p owns the given expressions, q is the other party, and ρ maps type variables to seals and owners:

$$
\begin{array}{lcl}
p;q;\rho \vdash e_1 =^k_{seal} e_2 & \stackrel{\text{def}}{=} & \forall j<k.\ (e_1 \mapsto^j \textbf{Error}\text{: s} \Rightarrow e_2 \mapsto^* \textbf{Error}\text{: s}) \text{ and} \\
& & \quad (\forall v_1.\ e_1 \mapsto^j v_1 \Rightarrow \exists v_2.\ e_2 \mapsto^* v_2 \text{ and } p;q;\rho \vdash v_1 =^{k-j}_{seal} v_2) \\
& & \forall j<k.\ (e_2 \mapsto^j \textbf{Error}\text{: s} \Rightarrow e_1 \mapsto^* \textbf{Error}\text{: s}) \text{ and} \\
& & \quad (\forall v_1.\ e_2 \mapsto^j v_2 \Rightarrow \exists v_2.\ e_1 \mapsto^* v_1 \text{ and } p;q;\rho \vdash v_1 =^{k-j}_{seal} v_2) \\
p;q;\rho \vdash v_1{}^{\alpha-} =^k_{seal} \{v_2\}_{sv} & \stackrel{\text{def}}{=} & \rho(\alpha) = (sx,q) \text{ and } \forall j<k.\ p;q;\rho \vdash v_1 =^j_{seal} v_2 \\
& \vdots & \\
p;q;\rho \vdash (\lambda x.e_1) =^k_{seal} (\lambda x.e_2) & \stackrel{\text{def}}{=} & \forall j<k, v_1, v_2.\ q;p;\rho \vdash v_1 =^j_{seal} v_2 \Rightarrow \\
& & \quad p;q;\rho \vdash e_1[v_1/x] =^j_{seal} e_2[v_2/x]
\end{array}
$$

The rest of the cases are defined as in the Scheme relation of figure 4.3. An important subtlety above is that two sealed terms are related only if they are locked with a seal owned by the *other* party, and that the arguments to functions are owned by the party that does *not* own the function. The former point allows us to establish this lemma, after which we can build a new bridge lemma and then prove the theorem of interest:

Lemma 4.5.10.

1. *If* $p;q;\rho,\alpha:(sx,p) \vdash e_1 =^k_{seal} e_2$ *(and* α *not free in* e_1*), then* $p;q;\rho \vdash e_1 =^k_{seal} e_2$.
2. *If* $p;q;\rho \vdash e_1 =^k_{seal} e_2$, *then* $p;q;\rho,\alpha:(sx,p) \vdash e_1 =^k_{seal} e_2$.

Lemma 4.5.11. *For any two terms* e_1 *and* e_2 *such that* e_1*'s open type variables (and their ownership information) occur in* ρ*, and so do the open type variables in* τ*, then if* $(\forall k.p;q;\rho \vdash e_1 =^k_{seal} e_2)$ *then* $(\forall k.p;q;\rho \vdash e_1{}^{\tau} =^k_{seal} E^{p,q}_{\tau}(\rho,e_2))$.

Proof. By induction on τ. The $\forall\alpha.\tau$ case requires the preceding lemma. □

Theorem 4.5.12. *If* $\rho \vdash \gamma_1 =_{seal} \gamma_2 : \Gamma$, *e's open type variables occur in* ρ, $\Delta;\Gamma \vdash_S e : \textbf{\textit{TST}}$, *and* e *only uses boundaries as contracts, then* $\forall k.p;q;\rho \vdash \gamma_1(e) =^k_{seal} \gamma_2(\mathcal{R}[e])$.

Proof. Induction on the derivation $\Delta;\Gamma \vdash_S \mathsf{e} : \mathbf{TST}$. Contract cases appeal to lemma 4.5.11. □

This theorem has two consequences: first, contracts as we have defined them in this paper can be implemented by a variant on Sumii and Pierce's translation, and thus due to our earlier development their translation preserves parametricity; and second, since Sumii and Pierce's translation is itself a variant on Findler-and-Felleisen-style contracts, our boundary-based contracts are actually contracts in that sense.

Finally, one observation: when we replace ML boundaries with contracts, if we choose $\mathcal{E} = \mathsf{E}$ then there is no trace of ML left in the language we are considering; it is pure Scheme with a contract system. But, strangely, the contract system's parametricity theorem relies on the fact that parametricity holds in ML.

4.6 Theorems for low, low prices!

We have shown that Scheme terms with behavioral contracts attached are related to themselves by the relation of figure 4.4, a relation that can be used in much the same way as Reynolds' parametricity relation. In particular, we can follow Wadler (Wadler 1989) by interpreting σ as a function to extract free theorems about contracted terms. To demonstrate this we rework some examples from "Theorems for Free!" in this setting.

Example 4.6.1 (rearrangements (Wadler 1989, section 3.1))**.** Let e be a closed Scheme term, and let $r = \mathsf{e}^{\forall\alpha.\alpha^*\to\alpha^*}$. By corollary 4.5.7, we have that $\vdash r = r : \forall\alpha.\alpha^* \to \alpha^*$. By definition of the relation $\forall\alpha.\tau$, we have that:

$$\text{For all } \mathsf{R} \in \mathsf{Rel},\ \alpha \mapsto \mathsf{R} \vdash r = r : \alpha^* \to \alpha^*$$

By the definition of the relation at $\tau_1 \to \tau_2$, we have that:

$$\text{For all } \mathsf{R} \in \mathsf{Rel},\ \text{For all } \mathsf{v}_1, \mathsf{v}_2.\ \ \sigma \vdash \mathsf{v}_1 = \mathsf{v}_2 : \tau^* \Rightarrow \sigma \vdash (r\ \mathsf{v}_1) = (r\ \mathsf{v}_2) : \tau^*$$

Now consider the special case where σ is a total function a' from some domain A of Scheme values to some range A' of Scheme values and a is a Scheme term such that $a'(\mathsf{v}_d) = \mathsf{v}_r) \Rightarrow (a\ \mathsf{v}_d) \mapsto^* \mathsf{v}_r$. The above becomes

$$\begin{array}{l}\text{For all } a' : A \to A',\ \text{For all lists } xs \text{ with elements drawn from } A,\\ \quad (\mathsf{map}\ a\ xs) \mapsto^* xs' \Rightarrow \forall \mathcal{E}[]_S, v.\\ \quad\quad \mathcal{E}[(\mathsf{map}\ a\ (r\ xs))]_S \mapsto^* \mathcal{E}[v] \Leftrightarrow \mathcal{E}[(r\ (\mathsf{map}\ a\ xs))]_S \mapsto^* \mathcal{E}[v]\end{array}$$

Hence $\vdash (\mathsf{map}\ a\ (r\ xs)) = (r\ (\mathsf{map}\ a\ xs))$, thus $\vdash (\mathsf{map}\ a\ (r\ xs)) \simeq_{ctxt} (r\ (\mathsf{map}\ a\ xs))$.

Example 4.6.2 (sorting (Wadler 1989, section 3.3))**.** Let e be a closed Scheme term, and let $s = \mathsf{e}^{\forall\alpha.(\alpha\to\alpha\to\mathbf{Nat})\to\alpha^*\to\alpha^*}$. We have:

$$\text{For all relations } \mathsf{R} \in \mathsf{Rel},\ \ \alpha \mapsto \mathsf{R} \vdash s = s : (\alpha \to \alpha \to \mathbf{Nat}) \to \alpha^* \to \alpha^*.$$

Expanding and specializing R to a function as in the previous example, we have

$$\begin{array}{l}\text{For all functions } a : A \to A',\ \ \text{For all functions } \prec, \prec' \text{ that agree on effects such that}\\ \quad \text{for all } x, y \in A.\ x \prec y \mapsto^* \overline{n} \Leftrightarrow (ax) \prec' (ay) \mapsto^* \overline{n},\\ \quad\quad \text{For all } \mathcal{E}[]_S, \text{ lists } xs \text{ with elements drawn from } A,\\ \quad\quad\quad \mathcal{E}[(\mathsf{map}\ a\ (s\ (\prec)\ xs))]_S \ \mapsto^* \mathcal{E}[v] \ \Leftrightarrow\ \mathcal{E}[(s\ (\prec')\ (\mathsf{map}\ a\ xs))]_S \mapsto^* \mathcal{E}[v]\end{array}$$

which again implies $\vdash (\mathsf{map}\ a\ (s\ (\prec)\ xs)) \simeq_{ctxt} (s\ (\prec')\ (\mathsf{map}\ a\ xs))$.

Example 4.6.3 (an isomorphism (Wadler 1989, section 3.8))**.** Suppose τ is some arbitrary contract. Intuitively applying this contract to a term is isomorphic to as applying the contract $\forall\alpha.(\tau \to \alpha) \to \alpha$ (a type henceforth abbreviated τ_{cps}) to it. This apparent isomorphism is expressed by the functions:

$$i \ = \ (\lambda \mathsf{x}.(\lambda \mathsf{g}.\ (\mathsf{g}\ \mathsf{x})))^{\tau\to\tau_{cps}} \qquad\qquad j \ = \ (\lambda \mathsf{h}.(\mathsf{h}\ (\lambda \mathsf{x}.\ \mathsf{x})))^{\tau_{cps}\to\tau}$$

To prove this truly is an isomorphism, we must verify that $j \circ i$ and $i \circ j$ are identities for values that satisfy τ and τ_{cps}, respectively. The first holds by simple calculation. For the second, choosing h is any value that satisfies contract τ_{cps} (*i.e.*, does not signal an error when used in ways consistent with that contract), we have

$$\begin{array}{ll} & (i\ (j\ h))\\ = & (i\ ((\lambda \mathsf{h}.(\mathsf{h}\ (\lambda \mathsf{x}.\ \mathsf{x})))^{\tau_{cps}\to\tau}\ h))\\ \mapsto^2 & (i\ (h^{\tau_{cps}}\ (\lambda \mathsf{x}.\ \mathsf{x}))^{\tau})\\ \mapsto^2 & (\lambda \mathsf{g}.\ (\mathsf{g}\ (h^{\tau_{cps}}\ (\lambda \mathsf{x}.\ \mathsf{x}))^{\tau}))^{\tau_{cps}}\end{array}$$

Here is where corollary 4.5.7 helps. Using it, and by considering functions as a special case of relations, we have that for all $b : B \to B'$, and $f : A \to B$ (for any sets of Scheme values

A, B and B'),

$$\vdash (b\ (h^{\tau_{cps}}\ f)) \simeq_{ctxt} (h^{\tau_{cps}}\ (b \circ f))$$

Taking $b = g$ and $f = (\lambda x.\ x)$, we have

$$\begin{array}{ll} & (\lambda g.(g\ (h^{\tau_{cps}}\ (\lambda x.\ x)))) \\ \simeq_{ctxt} & (\lambda g.(h^{\tau_{cps}}\ (g \circ (\lambda x.\ x)))) \\ \simeq_{ctxt} & (\lambda g.\ (h^{\tau_{cps}}\ g)) \\ \simeq_{ctxt} & h^{\tau_{cps}} \\ \simeq_{ctxt} & h \end{array}$$

$$\mathbf{Rel}_{\tau_1,\tau_2} \;=\; \{\,\mathbf{R} \mid \forall(k,\mathbf{v}_1,\mathbf{v}_2)\in\mathbf{R}.\ \forall j\le k.\ (j,\mathbf{v}_1,\mathbf{v}_2)\in\mathbf{R} \text{ and } ;\vdash \mathbf{v}_1:\tau_1 \text{ and } ;\vdash \mathbf{v}_2:\tau_2\,\}$$

$$\Delta\vdash\delta \stackrel{\text{def}}{=} \Delta\subseteq \mathrm{dom}(\delta) \text{ and } \forall\alpha\in\Delta.\ \delta_R(\alpha)\in\mathbf{Rel}_{\delta_1(\alpha),\delta_2(\alpha)}$$

$$\delta\vdash\gamma_M\le^k\gamma'_M:\Gamma_M \stackrel{\text{def}}{=} \forall(\mathbf{x}:\tau)\in\Gamma_M.\ \gamma_M(\mathbf{x})=\mathbf{v}_1,\ \gamma'_M(\mathbf{x})=\mathbf{v}_2 \text{ and } \delta\vdash\mathbf{v}_1\lesssim^k_M\mathbf{v}_2:\tau$$

$$\delta\vdash\gamma_S\le^k\gamma'_S:\Gamma_S \stackrel{\text{def}}{=} \forall(\mathsf{x}:\mathbf{TST})\in\Gamma_S.\ \gamma_S(\mathsf{x})=\mathsf{v}_1,\ \gamma'_S(\mathsf{x})=\mathsf{v}_2 \text{ and } \delta\vdash\mathsf{v}_1\lesssim^k_S\mathsf{v}_2:\mathbf{TST}$$

$$\delta\vdash\gamma\le^k\gamma':\Gamma \stackrel{\text{def}}{=} \Gamma=\Gamma_M\cup\Gamma_S,\ \gamma=\gamma_M\cup\gamma_S,\ \gamma'=\gamma'_M\cup\gamma'_S \text{ and } \delta\vdash\gamma_M\le^k\gamma'_M:\Gamma_M \text{ and } \delta\vdash\gamma_S\le^k\gamma'_S:\Gamma_S$$

$$\Delta;\Gamma\vdash\mathbf{e}_1\lesssim_M\mathbf{e}_2:\tau \stackrel{\text{def}}{=} \forall k\ge 0.\ \forall\delta,\gamma_1,\gamma_2.\ \Delta\vdash\delta \text{ and } \delta\vdash\gamma_1\le^k\gamma_2:\Gamma\ \Rightarrow\ \delta\vdash\delta_1(\gamma_1(\mathbf{e}_1))\lesssim^k_M\delta_2(\gamma_2(\mathbf{e}_2)):\tau$$

$$\delta\vdash\mathbf{e}_1\lesssim^k_M\mathbf{e}_2:\tau \stackrel{\text{def}}{=} \forall j<k.\ (\mathbf{e}_1\hookrightarrow^j \mathbf{Error}\text{: s}\ \Rightarrow\ \mathbf{e}_2\hookrightarrow^*\mathbf{Error}\text{: s}) \text{ and } (\forall\mathbf{v}_1.\ \mathbf{e}_1\hookrightarrow^j\mathbf{v}_1\ \Rightarrow\ \exists\mathbf{v}_2.\ \mathbf{e}_2\hookrightarrow^*\mathbf{v}_2 \text{ and } \delta\vdash\mathbf{v}_1\lesssim^{k-j}_M\mathbf{v}_2:\tau)$$

$$\delta\vdash\mathbf{v}_1\lesssim^k_M\mathbf{v}_2:\alpha \stackrel{\text{def}}{=} (k,\mathbf{v}_1,\mathbf{v}_2)\in\delta_R(\alpha)$$

$$\delta\vdash {}^{\mathbf{L}}MS\ \mathsf{v}_1\lesssim^k_M {}^{\mathbf{L}}MS\ \mathsf{v}_2:\mathbf{L} \stackrel{\text{def}}{=} \forall j<k.\ \delta\vdash\mathsf{v}_1\lesssim^j_S\mathsf{v}_2:\mathbf{TST}$$

$$\delta\vdash\overline{n}\lesssim^k_M\overline{n}:\mathbf{Nat} \quad \textit{(unconditionally)}$$

$$\delta\vdash\lambda\mathbf{x}:\tau_1.\mathbf{e}_1\lesssim^k_M\lambda\mathbf{x}:\tau_1.\mathbf{e}_2:\tau_1\to\tau_2 \stackrel{\text{def}}{=} \forall j<k.\ \forall\mathbf{v}_1,\mathbf{v}_2.\ \delta\vdash\mathbf{v}_1\lesssim^j_M\mathbf{v}_2:\tau_1\ \Rightarrow\ \delta\vdash\mathbf{e}_1[\mathbf{v}_1/\mathbf{x}]\lesssim^j_M\mathbf{e}_2[\mathbf{v}_2/\mathbf{x}]:\tau_2$$

$$\delta\vdash\Lambda\alpha.\mathbf{e}_1\lesssim^k_M\Lambda\alpha.\mathbf{e}_2:\forall\alpha.\tau \stackrel{\text{def}}{=} \forall j<k.\ \forall \text{ closed } \tau_1,\tau_2.\ \forall\mathbf{R}\in\mathbf{Rel}_{\tau_1,\tau_2}.\ \delta,\alpha:(\tau_1,\tau_2,\mathbf{R})\vdash\mathbf{e}_1[\tau_1/\alpha]\lesssim^j_M\mathbf{e}_2[\tau_2/\alpha]:\tau$$

$$\delta\vdash[\mathbf{v}_1,\cdots,\mathbf{v}_n]\lesssim^k_M[\mathbf{v}'_1,\cdots,\mathbf{v}'_n]:\tau^* \stackrel{\text{def}}{=} \forall j<k.\ \forall i\in 1\ldots n.\ \delta\vdash\mathbf{v}_i\lesssim^j_M\mathbf{v}'_i:\tau$$

$$\Delta;\Gamma\vdash\mathsf{e}_1\lesssim_S\mathsf{e}_2:\mathbf{TST} \stackrel{\text{def}}{=} \forall k\ge 0.\ \forall\delta,\gamma_1,\gamma_2.\ \Delta\vdash\delta \text{ and } \delta\vdash\gamma_1\le^k\gamma_2:\Gamma\ \Rightarrow\ \delta\vdash\delta_1(\gamma_1(\mathsf{e}_1))\lesssim^k_S\delta_2(\gamma_2(\mathsf{e}_2)):\mathbf{TST}$$

$$\delta\vdash\mathsf{e}_1\lesssim^k_S\mathsf{e}_2:\mathbf{TST} \stackrel{\text{def}}{=} \forall j<k.\ (\mathsf{e}_1\hookrightarrow^j\mathbf{Error}\text{: s}\ \Rightarrow\ \mathsf{e}_2\hookrightarrow^*\mathbf{Error}\text{: s}) \text{ and } (\forall\mathsf{v}_1.\ \mathsf{e}_1\hookrightarrow^j\mathsf{v}_1\ \Rightarrow\ \exists\mathsf{v}_2.\ \mathsf{e}_2\hookrightarrow^*\mathsf{v}_2 \text{ and } \delta\vdash\mathsf{v}_1\lesssim^{k-j}_S\mathsf{v}_2:\mathbf{TST}$$

$$\delta\vdash\overline{n}\lesssim^k_S\overline{n}:\mathbf{TST} \quad \textit{(unconditionally)}$$

$$\delta\vdash(SM^{\langle\alpha;\tau_1\rangle}\ \mathsf{v}_1)\lesssim^k_S(SM^{\langle\alpha;\tau_2\rangle}\ \mathsf{v}_2):\mathbf{TST} \stackrel{\text{def}}{=} (k,\mathbf{v}_1,\mathbf{v}_2)\in\delta_R(\alpha)$$

$$\delta\vdash\lambda\mathsf{x}.\mathsf{e}_1\lesssim^k_S\lambda\mathsf{x}.\mathsf{e}_2:\mathbf{TST} \stackrel{\text{def}}{=} \forall j<k.\forall\mathsf{v}_1,\mathsf{v}_2.\ \delta\vdash\mathsf{v}_1\lesssim^j_S\mathsf{v}_2:\mathbf{TST}\ \Rightarrow\ \delta\vdash\mathsf{e}_1[\mathsf{v}_1/\mathsf{x}]\lesssim^j_S\mathsf{e}_2[\mathsf{v}_2/\mathsf{x}]:\mathbf{TST}$$

$$\delta\vdash\mathbf{nil}\lesssim^k_S\mathbf{nil}:\mathbf{TST} \quad \textit{(unconditionally)}$$

$$\delta\vdash(\mathbf{cons}\ \mathsf{v}_1\ \mathsf{v}_2)\lesssim^k_S(\mathbf{cons}\ \mathsf{v}'_1\ \mathsf{v}'_2):\mathbf{TST} \stackrel{\text{def}}{=} \forall j<k.\ \delta\vdash\mathsf{v}_1\lesssim^j_S\mathsf{v}'_1:\mathbf{TST} \text{ and } \delta\vdash\mathsf{v}_2\lesssim^j_S\mathsf{v}'_2:\mathbf{TST}$$

Figure 4.3: Logical approximation for ML terms (middle) and Scheme terms (bottom)

$$\mathsf{Rel} \quad = \quad \{\, \mathsf{R} \mid \forall (k, \mathsf{v}_1, \mathsf{v}_2) \in \mathsf{R}.\ \forall j \leq k.\ (j, \mathsf{v}_1, \mathsf{v}_2) \in \mathsf{R} \,\}$$

$$\begin{array}{lcl}
\sigma \vdash \mathsf{e}_1 \leq^k \mathsf{e}_2 : \tau & \stackrel{\text{def}}{=} & \forall j < k.\ (\mathsf{e}_1 \hookrightarrow^j \textbf{Error}\text{: s} \ \Rightarrow\ \mathsf{e}_2 \hookrightarrow^* \textbf{Error}\text{: s}) \ \text{and} \\
& & \quad (\forall \mathsf{v}_1.\ \mathsf{e}_1 \hookrightarrow^j \mathsf{v}_1 \ \Rightarrow \\
& & \qquad \exists \mathsf{v}_2.\ \mathsf{e}_2 \hookrightarrow^* \mathsf{v}_2 \ \text{and} \ \sigma \vdash \mathsf{v}_1 \leqq^{k-j} \mathsf{v}_2 : \tau) \\
\sigma \vdash \mathsf{v}_1 \leqq^k \mathsf{v}_2 : \alpha & \stackrel{\text{def}}{=} & (k, \mathsf{v}_1, \mathsf{v}_2) \in \sigma(\alpha) \\
\sigma \vdash \overline{n} \leqq^k \overline{n} : \textbf{Nat} & & \textit{(unconditionally)} \\
\sigma \vdash \lambda \mathsf{x}.\mathsf{e}_1 \leqq^k \lambda \mathsf{x}.\mathsf{e}_2 : \tau_1 \to \tau_2 & \stackrel{\text{def}}{=} & \forall j < k.\ \forall \mathsf{v}_1, \mathsf{v}_2.\ \sigma \vdash \mathsf{v}_1 \leqq^j \mathsf{v}_2 : \tau_1 \ \Rightarrow \\
& & \qquad \sigma \vdash \mathsf{e}_1[\mathsf{v}_1/\mathsf{x}] \leq^j \mathsf{e}_2[\mathsf{v}_2/\mathsf{x}] : \tau_2 \\
\sigma \vdash [\mathsf{v}_1, \cdots, \mathsf{v}_n] \leqq^k [\mathsf{v}'_1, \cdots, \mathsf{v}'_n] : \tau^* & \stackrel{\text{def}}{=} & \forall j < k.\ \forall i \in 1 \ldots n.\ \sigma \vdash \mathsf{v}_i \leqq^j \mathsf{v}'_i : \tau \\
\sigma \vdash \mathsf{v}_1 \leqq^k \mathsf{v}_2 : \forall \alpha.\tau & \stackrel{\text{def}}{=} & \forall j < k.\ \forall \mathsf{R} \in \mathsf{Rel}.\ \sigma, \alpha : \mathsf{R} \vdash \mathsf{v}_1 \leqq^j \mathsf{v}_2 : \tau
\end{array}$$

Figure 4.4: Behavioral specification for polymorphic contracts

CHAPTER 5
CALL-BY-NAME, CALL-BY-VALUE, AND THE GOLD STANDARD

5.1 What makes a foreign interface design good?

In chapters 3 and 4 we explored languages with similar operational semantics and different type systems. Now we turn our attention an embedding between two languages with the same type system but differing evaluation orders — simply-typed call-by-name and call-by-value variants of the lambda calculus. These will allow us to focus on an important question that we hinted at with the development of the preceding chapter: how can we characterize what makes a multi-language boundary design "good"?

Kennedy (Kennedy 2006), following Abadi (Abadi 1998), argues that we need to think about interoperability as a full abstraction problem: Given source language S and target language T (with contextual equivalence relations $\cong_s$ and $\cong_t$) and translation function $\mathcal{T} : S \to T$ that compiles the source into the target, he makes the case that we should try to establish that for any source-language terms e_1 and e_2,

$$\begin{array}{ccc} e_1 & \cong_s & e_2 \\ & \Updownarrow & \\ \mathcal{T}(e_1) & \cong_t & \mathcal{T}(e_2) \end{array}$$

His reasoning is that programmers — or at least, expert programmers — rely on their intuitive understanding of a programming language's contextual equivalence relation in building secure abstractions, and if his property held it would "ensure that C♯ programmers [could] reason about the security of their code by *thinking in C♯*" (emphasis in original). Conversely, places where it fails represent potential security risks since a language's defense mechanisms might be overridden.

This property is important to more than just security. It applies equally well to compiler optimization, refactoring, and any other domain in which we might want to take advantage of program equivalence in a language that has any kind of foreign interface. For this reason, we view Kennedy's property as getting at a 'gold standard' criterion that separates good interoperability facilities from bad: a good foreign interface allows programmers to forget about it, while a bad one does not. To take an obvious example, Haskell's foreign function interface is 'good' under this criterion because it uses the IO monad to sequester impure foreign function calls; if it did not, then such calls would disturb its equivalence relation and break programmers' ability to forget about them.

Kennedy's specific formulation in terms of fully abstract translations, though, is unwieldy for this purpose. First, it necessarily involves a compiler $\mathcal{T}$; in our setting we may not even have a compiler to reason about, and even if we do we are more interested in the semantics of the languages involved than how they are implemented. Second, purely psychologically, it suggests that in order to prove your language implementation secure you have to prove a universal property about all assembly-language programs that might get linked in with your compiled code; in many interoperability contexts, though, both interacting languages have richer equational properties that we may be able to exploit.

In this chapter we suggest a variation. Instead of using source and target languages S and T and the translation function $\mathcal{T}$, we imagine two peer languages L and K and multi-language system LK (with contextual equivalence relation $\cong_{l+k}$) which includes both L and K connected by boundaries. In this setting, the gold standard is that equivalence in source-language contexts implies equivalence in arbitrary contexts:

$$\begin{array}{ccc} e_1 & \cong_l & e_2 \\ & \Updownarrow & \\ e_1 & \cong_{l+k} & e_2 \end{array}$$

This formulation is actually stronger than Kennedy's in the sense that we could pick C$^\sharp$ for L and the .NET intermediate language for K, and then if our formulation holds then Kennedy's also holds for any semantics-preserving translation function $\mathcal{T}$. Furthermore it has the nice property that rather than focusing our attention on the translation, it focuses our

attention on the semantics of a computation flowing across boundaries, and in particular it calls to attention that appropriate boundaries may be able to defend against malicious foreign contexts.

In the rest of this paper, we argue our formulation's utility by pursuing an example: forming a multilanguage embedding that combines a simply-typed call-by-name language with a simply-typed call-by-value language. We begin with a straightforward natural embedding of the two languages. We show by the method of logical relations that we can compile the multilanguage system into a single language using Plotkin's call-by-name and call-by-value CPS transformations (Plotkin 1975); then we show that, nonetheless, there are pairs of terms that are contextually equivalent when considering only call-by-name contexts that can be distinguished by contexts that contain call-by-value portions. Considering one example leads us to a slight refinement of our semantic model and a somewhat larger change to our compilation strategy, after which we prove that the refined version does satisfy our criterion. We conclude by sketching several other examples of multilanguage systems involving more sophisticated language features whose designs could be informed by our criterion.

5.2 A first system

In this section we present a first attempt at a multilanguage system combining call-by-name and call-by-value. We show that it satisfies a number of nice properties but that it does not satisfy the gold standard.

5.2.1 Syntax and reduction rules

Figure 5.1 defines grammars and reduction rules for our call-by-name and call-by-value languages (hereafter Λ_n and Λ_v), with Λ_n on the left in bold red font and Λ_v on the right in sans-serif blue font. Each system is standard except that we give each language a single effect represented as the term $\Uparrow$ which inhabits all types and immediately terminates the program when evaluated (it is intended to suggest a diverging computation). We add $\Uparrow$ only so that we can write programs whose outcomes depend on evaluation order, which would

$$
\begin{array}{lcl}
\mathbf{e} & = & x \mid (\mathbf{e}\ \mathbf{e}) \mid \lambda x.\ \mathbf{e} \mid \overline{n} \mid \Uparrow \\
\mathbf{v} & = & \lambda x.\ \mathbf{e} \mid \mathbf{Nat} \\
\mathbf{E} & = & [\,]_N \mid (\mathbf{E}\ \mathbf{e})
\end{array}
$$

$$
\frac{(\mathbf{x} : \tau) \in \Gamma}{\Gamma \vdash_n \mathbf{x} : \tau} \quad \frac{\Gamma \vdash_n \mathbf{e}_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_n \mathbf{e}_2 : \tau_1}{\Gamma \vdash_n (\mathbf{e}_1\ \mathbf{e}_2) : \tau_2}
$$

$$
\frac{\Gamma, (\mathbf{x} : \tau_1) \vdash_n \mathbf{e} : \tau_2}{\Gamma \vdash_n \lambda \mathbf{x}.\ \mathbf{e} : \tau_1 \to \tau_2} \quad \frac{}{\Gamma \vdash_n \overline{n} : \mathbb{N}} \quad \frac{}{\Gamma \vdash_n \Uparrow : \tau}
$$

$$
\begin{array}{lcl}
\mathcal{E}[(\lambda x.\ \mathbf{e}_1)\ \mathbf{e}_2]_N & \mapsto & \mathcal{E}[\mathbf{e}_1\{x := \mathbf{e}_2\}] \\
\mathcal{E}[\Uparrow]_N & \mapsto & \Uparrow
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{e} & = & x \mid (\mathsf{e}\ \mathsf{e}) \mid \lambda x.\ \mathsf{e} \mid \mathbf{Nat} \mid \Uparrow \\
\mathsf{v} & = & \lambda x.\ \mathsf{e} \mid \mathbf{Nat} \\
\mathsf{E} & = & [\,]_V \mid (\mathsf{E}\ \mathsf{e}) \mid (\mathsf{v}\ \mathsf{E})
\end{array}
$$

$$
\frac{(\mathsf{x} : \tau) \in \Gamma}{\Gamma \vdash_v \mathsf{x} : \tau} \quad \frac{\Gamma \vdash_v \mathsf{e}_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_v \mathsf{e}_2 : \tau_1}{\Gamma \vdash_v (\mathsf{e}_1\ \mathsf{e}_2) : \tau_2}
$$

$$
\frac{\Gamma, (\mathsf{x} : \tau_1) \vdash_v \mathsf{e} : \tau_2}{\Gamma \vdash_v \lambda \mathsf{x}.\ \mathsf{e} : \tau_1 \to \tau_2} \quad \frac{}{\Gamma \vdash_v \overline{n} : \mathbb{N}} \quad \frac{}{\Gamma \vdash_v \Uparrow : \tau}
$$

$$
\begin{array}{lcl}
\mathcal{E}[(\lambda x.\ \mathsf{e})\ \mathsf{v}]_V & \mapsto & \mathcal{E}[\mathsf{e}\{x := \mathsf{v}\}] \\
\mathcal{E}[\Uparrow]_V & \mapsto & \Uparrow
\end{array}
$$

$$
\mathcal{E} = \mathbf{E}
$$

Figure 5.1: Call-by-name (left) and call-by-value (right) languages

be impossible otherwise since all programs would terminate and evaluation order does not change the final answer of a program that terminates. The languages are both simply-typed with the base type $\mathbb{N}$ (for "natural number") and arrow type $\tau_1 \to \tau_2$. We use $\Gamma \vdash_n \mathbf{e} : \tau$ for Λ_n's typing judgment and $\Gamma \vdash_v \mathsf{e} : \tau$ for Λ_v's typing judgment[1]. We assume that we can syntactically differentiate variables introduced by call-by-name terms and variables introduced by call-by-value terms.

As a first pass at combining the two languages, we extend the language grammars to include boundaries in the obvious way:

$$
\begin{array}{lcl}
\mathbf{e} & = & \cdots \mid (\mathbf{NV}^{\tau}\ \mathsf{e}) \\
\mathsf{e} & = & \cdots \mid (\mathsf{VN}^{\tau}\ \mathbf{e})
\end{array}
$$

1. To be very precise we could define isomorphic but separate alphabets, say τ and σ, for the types given to Λ_n and Λ_v languages respectively. This would add considerably to the weight of our notation, though, without providing any additional insight, so we choose to use the same types for both languages.

Next we modify evaluation contexts so that cross-language evaluation can take place. This is straightforward as well:

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

Third we add typing rules. Since both sides have the same type system, the typing rules at boundaries are straightforward:

$$\frac{\Gamma \vdash_v \mathsf{e} : \tau}{\Gamma \vdash_n (\mathbf{NV}^{\tau}\ \mathsf{e}) : \tau} \qquad \frac{\Gamma \vdash_n \mathbf{e} : \tau}{\Gamma \vdash_v (\mathsf{VN}^{\tau}\ \mathbf{e}) : \tau}$$

The last step of defining the embedding is to add new reduction rules for the new syntactic forms:

$$\begin{array}{lcll} \mathcal{E}[\mathbf{NV}^{\mathbb{N}}\ n]_N & \mapsto & \mathcal{E}[n] & (\text{NVNum}) \\ \mathcal{E}[\mathbf{NV}^{\tau_1 \to \tau_2}\ \mathsf{v}]_N & \mapsto & \mathcal{E}[\lambda x.\ (\mathbf{NV}^{\tau_2}\ (\mathsf{v}\ (\mathsf{VN}^{\tau_1}\ x)))] & (\text{NVFun}) \\ & & & \\ \mathcal{E}[\mathsf{VN}^{\mathbb{N}}\ n]_V & \mapsto & \mathcal{E}[n] & (\text{VNNum}) \\ \mathcal{E}[\mathsf{VN}^{\tau_1 \to \tau_2}\ \mathbf{v}]_V & \mapsto & \mathcal{E}[\lambda x.\ (\mathsf{VN}^{\tau_2}\ (\mathbf{v}\ (\mathbf{NV}^{\tau_1}\ x)))] & (\text{VNFun}) \end{array}$$

Here we have chosen straightforward reduction rules for boundaries that mirror the behavior of boundaries we have seen so far. We call the language formed this way $\Lambda_{(n+v)_1}$.

Theorem 5.2.1 (type soundness)**.** $\Lambda_{(n+v)_1}$ *is type-sound.*

Before we proceed, we should also take a moment to define the notion of contextual equivalence that we use throughout the paper. For a given language Λ_x, we will define contexts to be terms in language x "that have a hole in them", *i.e.* a single subtree removed. If a context C would have type τ under environment Γ if its hole were replaced by a term of language Λ_y that had type τ' under environment Γ', we write $C : (\Gamma' \vdash_y \tau') \rightsquigarrow (\Gamma \vdash_x \tau)$. We define whole-program contexts to be those contexts with type $C : (\Gamma \vdash_y \tau) \rightsquigarrow (\ \vdash_x \mathbb{N})$ for any Γ, y, and τ, and we define a contextual equivalence relation as follows:

Definition 5.2.2 (Kleene equivalence)**.** We say that $a =_{\text{kleene}} b$ iff $a \mapsto_x^* \Uparrow \Leftrightarrow\ b \mapsto_x^* \Uparrow$ and $a \mapsto^* \overline{n} \ \Leftrightarrow\ b \mapsto^* \overline{n}$.

$$
\begin{array}{lcl}
[\![\tau]\!]_N & = & ([\tau]_N \to \sigma) \to \sigma \\
[\mathbb{N}]_N & = & \mathbb{N} \\
[\tau_1 \to \tau_2]_N & = & [\![\tau_1]\!]_N \to [\![\tau_2]\!]_N \\
\\
[\![x]\!]_N & = & x \\
[\![n]\!]_N & = & \lambda k.\ k\ n \\
[\![\Uparrow]\!]_N & = & \lambda k.\ \Uparrow \\
[\![\lambda x.\ \mathbf{e}]\!]_N & = & \lambda k.\ k\ (\lambda x.\ [\![\mathbf{e}]\!]_N) \\
[\![(\mathbf{e}_1\ \mathbf{e}_2)]\!]_N & = & \lambda k.\ ([\![\mathbf{e}_1]\!]_N\ (\lambda f.\ (f\ [\![\mathbf{e}_2]\!]_N\ k))) \\
[\![\mathbf{NV}^\tau\ \mathsf{e}]\!]_N & = & (\mathcal{W}^\tau_{NV} \tau\ [\![\mathsf{e}]\!]_V)
\end{array}
\qquad
\begin{array}{lcl}
[\![\tau]\!]_V & = & ([\tau]_V \to \sigma) \to \sigma \\
[\mathbb{N}]_V & = & \mathbb{N} \\
[\tau_1 \to \tau_2]_V & = & [\tau_1]_V \to [\![\tau_2]\!]_V \\
\\
[\![x]\!]_V & = & \lambda k.\ k\ x \\
[\![n]\!]_V & = & \lambda k.\ k\ n \\
[\![\Uparrow]\!]_V & = & \lambda k.\ \Uparrow \\
[\![\lambda x.\ \mathsf{e}]\!]_V & = & \lambda k.\ k\ (\lambda x.\ [\![\mathsf{e}]\!]_V) \\
[\![(\mathsf{e}_1\ \mathsf{e}_2)]\!]_V & = & \lambda k.\ [\![\mathsf{e}_1]\!]_V\ (\lambda f.\ [\![\mathsf{e}_2]\!]_V\ (\lambda x.\ f\ x\ k)) \\
[\![\mathsf{VN}^\tau\ \mathbf{e}]\!]_V & = & (\mathcal{W}^\tau_{VN}\ [\![\mathbf{e}]\!]_N)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{W}^\tau_{NV} & : & [\![\tau]\!]_N \to [\![\tau]\!]_V \\
\mathcal{W}^{\mathbb{N}}_{NV} & = & \lambda c.\ c \\
\mathcal{W}^{\tau_1 \to \tau_2}_{NV} & = & \\
\multicolumn{3}{l}{\quad \lambda c.\ \lambda k.\ (c\ (\lambda f.\ (k\ (\lambda x.\ (\mathcal{W}^{\tau_2}_{NV}\ (f\ (\mathcal{W}^{\tau_1}_{VN}\ (\lambda k'.\ k'\ x))))))))}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{W}^\tau_{VN} & : & [\![\tau]\!]_V \to [\![\tau]\!]_N \\
\mathcal{W}^{\mathbb{N}}_{VN} & = & \lambda c.\ c \\
\mathcal{W}^{\tau_1 \to \tau_2}_{VN} & = & \\
\multicolumn{3}{l}{\quad \lambda c.\ \lambda k.\ (c\ (\lambda f.\ (k\ (\lambda c'.\ \lambda k'.\ (\mathcal{W}^{\tau_1}_{NV}\ c')\ (\lambda x.\ (\mathcal{W}^{\tau_2}_{VN}\ (f\ x)))\ k'))))}
\end{array}
$$

Figure 5.2: Boundary-elimination CPS transformation 1

Definition 5.2.3 (contextual equivalence)**.** We say $\Gamma \vdash a \cong_x b : \tau$ iff for all contexts C of Λ_x such that $C : (\Gamma \vdash_x \tau) \rightsquigarrow (\ \vdash_x \mathbb{N})$, we have that $C[a] =_{\text{kleene}} C[b]$.

5.2.2 Eliminating boundaries with CPS

$\Lambda_{(n+v)_1}$ is an intuitively simple way to describe an interaction between Λ_n and Λ_v that corresponds to an interpreter. We can also derive a semantically-equivalent compiler from it. To do so we combine both of Plotkin's original continuation-passing-style transformations (Plotkin 1975) in a single system; as one might hope, Plotkin's rules map over directly and to make the system work all we need to do is define translations for boundaries.

Figure 5.2 defines three related pieces that together form the CPS translation $[\![\ ]\!]_N$ (for Λ_n source terms) and $[\![\ ]\!]_V$ (for Λ_v source terms). Those pieces are the CPS type translation, the CPS term translation, and definitions for a type-indexed family of "wrapper" terms $\mathcal{W}^{\tau}_{NV}$ and $\mathcal{W}^{\tau}_{VN}$ used by the term translation for boundary cases.

The type translations $[\![\tau]\!]_N$ and $[\![\tau]\!]_V$ give the types for Λ_n and Λ_v computations of type τ, while $[\tau]_N$ and $[\tau]_V$ give the types for CPS-converted Λ_n and Λ_v values of type τ. Notice that while the term translations are functions, the type relations are not: any concrete type can be chosen for the abstract "answer" type σ. We choose to encode answer types this way so that we can avoid including polymorphic types in the languages themselves; in effect we are keeping the universal quantifier in the meta-level rather than encoding it at the term level. As a matter of notation, when we write types involving $[\![\tau]\!]_N$ or $[\![\tau]\!]_V$ (for instance, $[\![\tau]\!]_N \to [\![\tau]\!]_V$) we mean to indicate the *set* of types that can be formed by choosing a concrete answer type for σ. When we ascribe a term such a set of types (for instance, $\mathcal{W}^{\tau}_{NV} : [\![\tau]\!]_N \to [\![\tau]\!]_V$) we mean that the given term can be typed regardless of which concrete type σ represents.

The term translations are Plotkin's call-by-name and call-by-value CPS transformations, each extended with a translation for boundaries. These cases have the same simple pattern: they take a foreign computation and translate it using the appropriate wrapper function to a native computation. For the base case, note that $[\![\mathbb{N}]\!]_N$ and $[\![\mathbb{N}]\!]_V$ represent the same concrete type (*i.e.*, $(\mathbb{N} \to \sigma) \to \sigma$) and CPS terms of that type have the same concrete behavior (to yield an integer to the continuation) the wrappers at type $\mathbb{N}$ can be the identity. The wrappers for arrow types both essentially return a new computation that forces the input computation to a value and then returns a function value that performs the appropriate wrappings on its inputs and outputs.

The three portions of figure 5.2 are connected to each other by the following theorem, which says that for any term $\mathbf{e}$ of type τ, $[\![\mathbf{e}]\!]_N$ has type $[\![\tau]\!]_N$ and similarly for Λ_v terms:

Theorem 5.2.4 (CPS type-correctness)**.** *Taking* $[\![\Gamma]\!]$ *to mean* $\{(\boldsymbol{x} : [\![\tau]\!]_N) \mid (\boldsymbol{x} : \tau) \in \Gamma\} \cup \{(x : [\tau]_V) \mid (x : \tau) \in \Gamma\}$*, both of the following hold:*

1. *If* $\Gamma \vdash_n \boldsymbol{e} : \tau$ *then* $[\![\Gamma]\!] \vdash_n [\![\boldsymbol{e}]\!]_N : [\![\tau]\!]_N$
2. *If* $\Gamma \vdash_v e : \tau$ *then* $[\![\Gamma]\!] \vdash_v [\![e]\!]_V : [\![\tau]\!]_V$

The proof of this theorem is by simultaneous induction on the typing derivations. As one might expect, the only interesting cases are for boundaries, which transform into applications of $\mathcal{W}^{\tau}_{NV}$ and $\mathcal{W}^{\tau}_{VN}$. The $\mathcal{W}^{\tau}_{NV}$ and $\mathcal{W}^{\tau}_{VN}$ functions (or rather families of type-indexed functions) are the heart of the translation. Each converts a foreign computation into a native one: $\mathcal{W}^{\tau}_{NV}$ converts a Λ_n computation to a Λ_v computation, and $\mathcal{W}^{\tau}_{VN}$ does the opposite.

Lemma 5.2.5. *For all τ, both of the following hold:*

1. $\vdash_n \mathcal{W}^{\tau}_{NV} : [\![\tau]\!]_N \to [\![\tau]\!]_V$
2. $\vdash_v \mathcal{W}^{\tau}_{VN} : [\![\tau]\!]_V \to [\![\tau]\!]_N$

The two cases for numbers are straightforward. Numbers are assumed to convert straight across, just as they do in the operational semantics. Using this rule, we can see that

$$[\![\mathbf{NV}^{\mathbb{N}}\ \mathsf{e}]\!]_N = \lambda k.\ ([\![\mathsf{e}]\!]_V\ \lambda v.\ (k\ v))$$

The CPS translation is also connected to the operational semantics we gave previously by the fact that it is semantics-preserving: a program that terminates with a particular integer value under the original semantics compiles into a program that terminates with the same integer value, and a program that reduces to $\Uparrow$ in the original semantics compiles into another program that reduces to $\Uparrow$.

To show this we use the method of logical relations. First we define a logical relation $\mathbf{e}_{\text{src}} \sim_{\text{c}} \mathbf{e}_{\text{cps}} : \tau$ between terms. The left side of the relation is intended for source-language terms and comprises both Λ_n and Λ_v terms. To remind the reader of this we annotate variables representing these terms with subscript src (for instance, $\mathbf{e}_{\text{src}}$). The right side is intended for Λ_n terms in continuation-passing style. To suggest this we annotate variables

representing these terms with subscript cps (for instance, $\mathbf{e}_{\text{cps}}$). The logical relation is:

$$
\begin{array}{rcl}
\mathbf{e}_{\text{src}} \sim_{\text{c}} \mathbf{e}_{\text{cps}} : \tau & \stackrel{\text{def}}{=} & \vdash_n \mathbf{e}_{\text{src}} : \tau, \vdash_n \mathbf{e}_{\text{cps}} : [\![\tau]\!]_N \\
& \text{and} & \mathbf{e}_{\text{src}} \hookrightarrow^* \Uparrow \text{ and } \forall k.(\mathbf{e}_{\text{cps}}\ k) \hookrightarrow^* \Uparrow \\
& \text{or} & \mathbf{e}_{\text{src}} \hookrightarrow^* \mathbf{v}_{\text{src}}, \\
& & \forall k.(\mathbf{e}_{\text{cps}}\ k) \hookrightarrow *(k\ \mathbf{v}_{\text{cps}}), \\
& \text{and} & \mathbf{v}_{\text{src}} \sim_{\text{c},v} \mathbf{v}_{\text{cps}} : \tau
\end{array}
$$

$$
\begin{array}{rcl}
n_{\text{src}} \sim_{\text{c},v} n_{\text{cps}} : \mathbb{N} & \stackrel{\text{def}}{=} & n_{\text{src}} = n_{\text{cps}} \\
\mathbf{v}_{\text{src}} \sim_{\text{c},v} \mathbf{v}_{\text{cps}} : \tau_1 \to \tau_2 & \stackrel{\text{def}}{=} & \forall \mathbf{e}'_{\text{src}}, \mathbf{e}'_{\text{cps}} \\
& \text{where} & \mathbf{e}'_{\text{src}} \sim_{\text{c}} \mathbf{e}'_{\text{cps}} : \tau_1 \\
& \text{we have} & (\mathbf{v}_{\text{src}}\ \mathbf{e}'_{\text{src}}) \sim_{\text{c}} \lambda k.\ (\mathbf{v}_{\text{cps}}\ \mathbf{e}'_{\text{cps}}\ k) : \tau_2
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{e}_{\text{src}} \sim_{\text{c}} \mathbf{e}_{\text{cps}} : \tau & \stackrel{\text{def}}{=} & \vdash_v \mathsf{e}_{\text{src}} : \tau, \vdash_n \mathbf{e}_{\text{cps}} : [\![\tau]\!]_V, \\
& \text{and} & \mathsf{e}_{\text{src}} \hookrightarrow^* \Uparrow \text{ and } \forall k.(\mathbf{e}_{\text{cps}}\ k) \hookrightarrow^* \Uparrow \\
& \text{or} & \mathsf{e}_{\text{src}} \hookrightarrow^* \mathsf{v}_{\text{src}}, \\
& & \forall k.(\mathbf{e}_{\text{cps}}\ k) \hookrightarrow^* (k\ \mathbf{v}_{\text{cps}}), \\
& \text{and} & \mathsf{v}_{\text{src}} \sim_{\text{c},v} \mathbf{v}_{\text{cps}} : \tau
\end{array}
$$

$$
\begin{array}{rcl}
n_{\text{src}} \sim_{\text{c},v} n_{\text{cps}} : \mathbb{N} & \stackrel{\text{def}}{=} & n_{\text{src}} = n_{\text{cps}} \\
\mathsf{v}_{\text{src}} \sim_{\text{c},v} \mathbf{v}_{\text{cps}} : \tau_1 \to \tau_2 & \stackrel{\text{def}}{=} & \forall \mathsf{v}'_{\text{src}}, \mathbf{v}'_{\text{cps}} \\
& \text{where} & \mathsf{v}'_{\text{src}} \sim_{\text{c},v} \mathbf{v}'_{\text{cps}} : \tau_1 \\
& \text{we have} & (\mathsf{v}_{\text{src}}\ \mathsf{v}'_{\text{src}}) \sim_{\text{c}} \lambda k.\ (\mathbf{v}_{\text{cps}}\ \mathbf{v}'_{\text{cps}}\ k) : \tau_2
\end{array}
$$

As is often done with logical relations, we extend the relation to term substitutions with the relation $\gamma_{\text{src}} \sim_{\text{c}} \gamma_{\text{cps}} : \Gamma$:

Definition 5.2.6 (related term substitutions)**.** We say that $\gamma_{\text{src}} \sim_{\text{c}} \gamma_{\text{cps}} : \Gamma$ iff:

1. For each $(\mathbf{x} : \tau) \in \Gamma$, there exist terms $\mathbf{e}_{\text{src}}$ and $\mathbf{e}_{\text{cps}}$ such that $\gamma_{\text{src}}(\mathbf{x}) = \mathbf{e}_{\text{src}}$, $\gamma_{\text{cps}}(\mathbf{x}) = \mathbf{e}_{\text{cps}}$, and $\mathbf{e}_{\text{src}} \sim_{\text{c}} \mathbf{e}_{\text{cps}} : \tau$.

2. For each $(\mathsf{x} : \tau) \in \Gamma$, there exist terms v_{src} and $\mathbf{e}_{\text{cps}}$ such that $\gamma_{\text{src}}(\mathsf{x}) = \mathsf{v}_{\text{src}}$, $\gamma_{\text{cps}}(\mathsf{x}) = \mathbf{e}_{\text{cps}}$, and $\mathsf{v}_{\text{src}} \sim_{\text{c}} \mathbf{e}_{\text{cps}} : \tau$.

With that, we are ready to prove that any well-typed term is related to its own CPS conversion:

Lemma 5.2.7 (CPS-correctness lemma 1)**.** *Both of the following are true:*

1. *If* $\Gamma \vdash_n \boldsymbol{e} : \tau$ *and* $\gamma_{\text{src}} \sim_{\text{c}} \gamma_{\text{cps}} : \Gamma$*, then* $\gamma_{\text{src}}(\boldsymbol{e}) \sim_{\text{c}} \gamma_{\text{cps}}([\![\boldsymbol{e}]\!]_N) : \tau$
2. *If* $\Gamma \vdash_v \mathsf{e} : \tau$ *and* $\gamma_{\text{src}} \sim_{\text{c}} \gamma_{\text{cps}} : \Gamma$*, then* $\gamma_{\text{src}}(\mathsf{e}) \sim_{\text{c}} \gamma_{\text{cps}}([\![\mathsf{e}]\!]_V) : \tau$

Lemma 5.2.8 (CPS-correctness bridge lemma 1)**.** *Both of the following hold:*

1. *If* $\boldsymbol{e}_{\text{src}} \sim_{\text{c}} \boldsymbol{e}_{\text{cps}} : \tau$ *then* $(\mathit{VN}^{\tau}\ \boldsymbol{e}_{\text{src}}) \sim_{\text{c}} (\mathcal{W}^{\tau}_{\mathit{NV}}\ \boldsymbol{e}_{\text{cps}}) : \tau$.
2. *If* $\mathsf{e}_{\text{src}} \sim_{\text{c}} \boldsymbol{e}_{\text{cps}} : \tau$ *then* $(\mathit{NV}^{\tau}\ \mathsf{e}_{\text{src}}) \sim_{\text{c}} (\mathcal{W}^{\tau}_{\mathit{VN}}\ \boldsymbol{e}_{\text{cps}}) : \tau$.

Theorem 5.2.9 (compiler correctness 1)**.** *If* $\vdash_n \boldsymbol{e} : \mathbb{N}$ *then* $\boldsymbol{e} =_{\text{kleene}} ([\![\boldsymbol{e}]\!]_N\ (\lambda I.\ I))$.

Proof. Special case of lemma 5.2.7. □

5.2.3 Equivalence preservation

So far we have shown $\Lambda_{(n+v)_1}$ to be type-sound and we have given a compiler that reduces it to pure-Λ_n terms using the CPS transformation. Unfortunately, despite these facts, it does not satisfy our gold standard. To demonstrate this, we characterize equality in call-by-name and call-by-value, then show that some things are equal in call-by-name that are not equal in the combined calculus.

To do so, first we establish a logical equivalence relation, which we prove is sound with respect to contextual equivalence in the call-by-name lambda-calculus when complete programs always have type $\mathbb{N}$. We do this by establishing that it is a consistent, symmetric, reflexive and transitive congruence; soundness follows from the fact that contextual equivalence is the largest consistent, congruent equivalence relation.

$$\begin{aligned}
\mathbf{e}_1 \simeq_n \mathbf{e}_2 : \mathbb{N} &\stackrel{\text{def}}{=} \exists \mathbf{v}_1, \mathbf{v}_2 \in \Lambda_n.\ \mathbf{e}_1 \hookrightarrow^* \mathbf{v}_1 \Leftrightarrow \mathbf{e}_2 \hookrightarrow^* \mathbf{v}_2 \\
&\qquad \text{and } \mathbf{v}_1 = \mathbf{v}_2 \\
\mathbf{e}_1 \simeq_n \mathbf{e}_2 : \tau_1 \to \tau_2 &\stackrel{\text{def}}{=} \forall \mathbf{e}_1', \mathbf{e}_2' \in \Lambda_n.\ \mathbf{e}_1' \simeq_n \mathbf{e}_2' : \tau_1 \Rightarrow \\
&\qquad \mathbf{e}_1(\mathbf{e}_1') \simeq_n \mathbf{e}_2(\mathbf{e}_2') : \tau_2
\end{aligned}$$

We extend the relation to open terms by saying that $\Gamma \vdash \mathbf{e}_1 \simeq_n \mathbf{e}_2 : \tau$ if for all substitutions $\gamma_1 \simeq_n \gamma_2 : \Gamma$ we have $\gamma_1(\mathbf{e}_1) \simeq_n \gamma_2(\mathbf{e}_2) : \tau$.

Lemma 5.2.10 (consistency)**.** *For all $\boldsymbol{e}_1, \boldsymbol{e}_2$, if $\boldsymbol{e}_1 \simeq_n \boldsymbol{e}_2 : \mathbb{N}$ then $\boldsymbol{e}_1 \simeq \boldsymbol{e}_2$ (where $\simeq$ is Kleene equality).*

Proof. Immediate from the definition. □

Note the subtlety here: if complete programs could have arbitrary types rather than needing the type $\mathbb{N}$ this property would not hold.

Lemma 5.2.11 (logical equivalence symmetry)**.** *If $\boldsymbol{e}_1 \simeq_n \boldsymbol{e}_2 : \tau$, then $\boldsymbol{e}_2 \simeq_n \boldsymbol{e}_1 : \tau$.*

Lemma 5.2.12 (logical equivalence transitivity)**.** *If $\boldsymbol{e}_1 \simeq_n \boldsymbol{e}_2 : \tau$ and $\boldsymbol{e}_2 \simeq_n \boldsymbol{e}_3 : \tau$, then $\boldsymbol{e}_1 \simeq_n \boldsymbol{e}_3 : \tau$.*

To prove this lemma we strengthen the induction hypothesis by adding that a term that is related to any other term is also related to itself. With this strengthening the proof goes through straightforwardly.

Lemma 5.2.13. *Both of the following hold:*

1. *If $\boldsymbol{e} \simeq_n \boldsymbol{e}' : \tau$ and $\boldsymbol{e}' \simeq_n \boldsymbol{e}'' : \tau$, then $\boldsymbol{e} \simeq_n \boldsymbol{e}'' : \tau$.*
2. *If $\exists \boldsymbol{e}'.\ \boldsymbol{e} \simeq_n \boldsymbol{e}' : \tau$ then $\boldsymbol{e} \simeq_n \boldsymbol{e} : \tau$.*

Lemma 5.2.14 (reflexivity / fundamental theorem)**.** *If $\Gamma \vdash_n \boldsymbol{e} : \tau$, then $\Gamma \vdash \boldsymbol{e} \simeq_n \boldsymbol{e} : \tau$.*

Lemma 5.2.15 (congruence)**.** *If $\Gamma \vdash \boldsymbol{e}_1 \simeq_n \boldsymbol{e}_2 : \tau$ and $\boldsymbol{C} : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')$, then $\Gamma' \vdash \boldsymbol{C}[\boldsymbol{e}_1] \simeq_n \boldsymbol{C}[\boldsymbol{e}_2] : \tau'$.*

Putting these together gives us the complete equivalence result:

Theorem 5.2.16. *If* $\Gamma \vdash \boldsymbol{e}_1 \simeq_n \boldsymbol{e}_2 : \tau$ *then* $\Gamma \vdash \boldsymbol{e}_1 \cong_n \boldsymbol{e}_2 : \tau$.

Corollary 5.2.17. $\Uparrow \cong_n (\lambda x.\, \Uparrow) : \tau_1 \to \tau_2$ *for all* τ_1, τ_2.

Theorem 5.2.18. *If* $\Gamma \vdash \boldsymbol{e}_1 \cong_n \boldsymbol{e}_2 : \tau$, *then* $\Gamma \vdash \boldsymbol{e}_1 \simeq_n \boldsymbol{e}_2 : \tau$.

We prove this theorem by way of Mason and Talcott's notion of ciu-equivalence (Mason and Talcott 1991), where ciu stands for "closed instantiations of uses"; intuitively it expresses the idea that two (possibly open) terms e_1 and e_2 are equivalent if for any possible evaluation context E and any possible assignment of free variables to terms γ, $E[\gamma(e_1)]$ coterminates with $E[\gamma(e_2)]$ and if the two expressions evaluate to numbers then they evaluate to the same number. (We can think of $\gamma(e_1)$ and $\gamma(e_2)$ as "closed instantiations" and E as a "use", hence the name.[2]

Our strategy is to show that operational equivalence implies ciu-equivalence and that ciu-equivalence implies logical equivalence; combining the two we have the desired result that operational equivalence implies logical equivalence.

Definition 5.2.19 (ciu-equivalence)**.** We write that $\Gamma \vdash \mathbf{e}_1 =_{\text{ciu}} \mathbf{e}_2 : \tau$ if for all substitutions γ such that $\Gamma \vdash \gamma$ and for all evaluation contexts $\mathbf{E}$ such that $\mathbf{E} : (\Gamma \vdash \tau) \rightsquigarrow (\emptyset \vdash \mathbf{Nat})$, we have that $\mathbf{E}[\gamma(\mathbf{e}_1)] \simeq \mathbf{E}[\gamma(\mathbf{e}_2)]$ (where $\simeq$ is Kleene equality).

Claim 5.2.20 (ciu-equivalence is a congruence)**.** *If* $\Gamma \vdash \boldsymbol{e}_1 =_{\text{ciu}} \boldsymbol{e}_2 : \tau$ *and* $\boldsymbol{C} : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')$, *then* $\Gamma' \vdash \boldsymbol{C}[\boldsymbol{e}_1] =_{\text{ciu}} \boldsymbol{C}[\boldsymbol{e}_2] : \tau'$.

Claim 5.2.21 (ciu-equivalence and substitution)**.** *If* $\Gamma \vdash \boldsymbol{e}_1 =_{\text{ciu}} \boldsymbol{e}_2 : \tau$, $(\boldsymbol{x} : \tau') \in \Gamma$, *and* $\vdash_n \boldsymbol{e}' : \tau'$, *then* $\Gamma - \boldsymbol{x} \vdash \boldsymbol{e}_1[\boldsymbol{e}'/\boldsymbol{x}] =_{\text{ciu}} \boldsymbol{e}_2[\boldsymbol{e}'/\boldsymbol{x}] : \tau$.

Lemma 5.2.22 ($\cong_n \subseteq =_{\text{ciu}}$)**.** *If* $\Gamma \vdash \boldsymbol{e}_1 \cong_n \boldsymbol{e}_2$, *then* $\Gamma \vdash \boldsymbol{e}_1 =_{\text{ciu}} \boldsymbol{e}_2$.

---

2. From this perspective the name "closed instantiations of uses" seems a little backwards, since we are producing closed instantiations of e_1 and e_2 and *then* using them, rather than using e_1 and e_2 and then taking closed instantiations of those uses. As Harper points out(Harper 2000, remark 27.10), we can reconcile this seeming mismatch between name and definition by imagining it to be an open extension of a "uses" equivalence relation that only relates closed terms. Still, presumably to bypass this possible confusion, Harper renames this notion of equivalence to uci-equivalence, for "uses of closed instantiations."

Proof. The definition of ciu-equivalence is a special case of the definition of operational equivalence where $\mathbf{C} = \mathbf{E}$. □

Lemma 5.2.23 ($=_{\text{ciu}} \subseteq \simeq_n$)**.** *If* $\Gamma \vdash_n \boldsymbol{e}_1 : \tau$, $\Gamma \vdash_n \boldsymbol{e}_2 : \tau$, *and* $\Gamma \vdash \boldsymbol{e}_1 =_{\text{ciu}} \boldsymbol{e}_2 : \tau$, *then* $\Gamma \vdash \boldsymbol{e}_1 \simeq_n \boldsymbol{e}_2 : \tau$.

5.3 Equivalence in call-by-value

We can establish a logical relation for equivalence in the call-by-value language by following the same process we did in the call-by-name case, using a slightly different relation:

$$\begin{aligned} \mathsf{e}_1 \simeq_v \mathsf{e}_2 : \sigma &\stackrel{\text{def}}{=} \exists \mathsf{v}_1, \mathsf{v}_2 \in \Lambda_v.\ \mathsf{e}_1 \hookrightarrow^* \mathsf{v}_1 \Leftrightarrow \mathsf{e}_2 \hookrightarrow^* \mathsf{v}_2 \\ &\qquad \text{and } \mathsf{v}_1 \simeq_v \mathsf{v}_2 : \sigma \end{aligned}$$

$$\begin{aligned} \mathsf{v}_1 \simeq_v \mathsf{v}_2 : \mathbb{N} &\stackrel{\text{def}}{=} \mathsf{v}_1 = \mathsf{v}_2 \\ \mathsf{v}_1 \simeq_v \mathsf{v}_2 : \sigma_1 \to \sigma_2 &\stackrel{\text{def}}{=} \forall \mathsf{v}_1', \mathsf{v}_2' \in \Lambda_v.\ \mathsf{v}_1' \simeq_v \mathsf{v}_2' : \sigma_1 \Rightarrow \\ &\qquad \mathsf{v}_1(\mathsf{v}_1') \simeq_v \mathsf{v}_2(\mathsf{v}_2') : \sigma_2 \end{aligned}$$

Since the technique is similar, rather than work through the development again we simply state the results. We refer the interested reader to Harper (Harper 2000).

Theorem 5.3.1. *If* $\Gamma \vdash_v \boldsymbol{e} : \sigma$ *then* $\Gamma \vdash \boldsymbol{e} \simeq_v \boldsymbol{e} : \sigma$.

Theorem 5.3.2. $\Gamma \vdash \boldsymbol{e}_1 \simeq_v \boldsymbol{e}_2 : \sigma$ *if and only if* $\Gamma \vdash \boldsymbol{e}_1 \cong_v \boldsymbol{e}_2 : \sigma$.

Now we are in a position to prove that the embedding does not preserve equivalence.

Theorem 5.3.3. $\boldsymbol{e}_1 \cong_n \boldsymbol{e}_2 : \tau_1 \to \tau_2$ *does not imply that* $\boldsymbol{e}_1 \cong_{n+v} \boldsymbol{e}_2 : \tau_1 \to \tau_2$.

Proof. By counterexample. We know from corollary 5.2.17 that $\Uparrow \cong_n \lambda x.\ \Uparrow : \mathbb{N} \to \mathbb{N}$. Now consider the context $\mathbf{C}_1 = \mathbf{NV}^{\mathbb{N}}\ ((\lambda f.\ 12)\ (\mathsf{VN}^{\tau_1 \to \tau_2}\ [\,]_N))$. Plugging the two terms

in, we have

$$
\begin{array}{ll}
 & \mathbf{C}_1[\Uparrow] \\
= & \mathbf{NV}^{\mathbb{N}}\ ((\lambda f.\ 12)\ (\mathsf{VN}^{\tau_1 \to \tau_2}\ \Uparrow)) \\
\mapsto & \Uparrow
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{C}_1[\lambda x.\ \Uparrow] \\
= & \mathbf{NV}^{\mathbb{N}}\ ((\lambda f.\ 12)\ (\mathsf{VN}^{\mathbb{N} \to \mathbb{N}}\ \lambda x.\ \Uparrow)) \\
\mapsto & \mathbf{NV}^{\mathbb{N}}\ ((\lambda f.\ 12)\ (\lambda y.\ \mathbf{NV}^{\mathbb{N}}\ ((\lambda x.\ \Uparrow)\ (\mathsf{VN}^{\mathbb{N}}\ y)))) \\
\mapsto & 12
\end{array}
$$

Hence the two are not contextually equivalent. □

5.4 Another choice for boundaries

Based on theorem 5.3.3 we know that the boundaries we defined in section 5.2 are not equation-preserving and thus $\Lambda_{(n+v)_1}$ does not satisfy the gold standard. Our counterexample suggests where the embedding's extra observational power comes from: since boundaries force their contents on evaluation, they can be used to directly observe termination at arrow types, an observation that is otherwise impossible. Using this intuition, we can build an alterate set of boundaries that are equation-preserving.

5.4.1 Syntax and reduction rules

Instead of the extensions to the core languages made in section 5.2, we instead make the extensions of figure 5.3 to form $\Lambda_{(n+v)_2}$. Evaluation contexts (listed on the third and fourth lines of that figure) are now sensitive to the type at a boundary: they still directly force boundaries that contain call-by-name terms of type $\mathbb{N}$, but do not evaluate inside boundaries that contain call-by-name terms with arrow types. Instead, the last listed reduction rule directly reduces a VN boundary with an arrow type to a function value without forcing the term it contains.

$$
\begin{array}{rcl}
\mathbf{e} & = & \cdots \mid (\mathbf{NV}^{\tau}\ \mathsf{e}) \\
\mathsf{e} & = & \cdots \mid (\mathsf{VN}^{\tau}\ \mathbf{e}) \\
\mathbf{E} & = & (\mathbf{NV}^{\tau}\ \mathsf{E}) \\
\mathsf{E} & = & (\mathsf{VN}^{\mathbb{N}}\ \mathbf{E})
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E}[\mathbf{NV}^{\mathbb{N}}\ n]_N & \mapsto & \mathcal{E}[n] \\
\mathcal{E}[\mathbf{NV}^{\tau_1\to\tau_2}\ \lambda x.\ \mathsf{e}]_N & \mapsto & \mathcal{E}[\lambda x.\ (\mathbf{NV}^{\tau_2}\ ((\lambda x.\ \mathsf{e})\ (\mathsf{VN}^{\tau_1}\ x)))] \\
\mathcal{E}[\mathbf{NV}^{\tau_1\to\tau_2}\ (\mathsf{VN}^{\tau_1\to\tau_2}\ \mathbf{e})]_N & \mapsto & \mathcal{E}[\mathbf{e}] \\
\mathcal{E}[\mathsf{VN}^{\mathbb{N}}\ n]_V & \mapsto & \mathcal{E}[n] \\
\mathcal{E}[(\mathsf{VN}^{\tau_1\to\tau_2}\ \mathbf{e})]_V & \mapsto & \mathcal{E}[\lambda x.\ (\mathsf{VN}^{\tau_2}\ (\mathbf{e}\ (\mathbf{NV}^{\tau_1}\ x)))]
\end{array}
$$

Figure 5.3: Extensions to form equation-preserving system

5.4.2 Eliminating boundaries with CPS

As before, we can eliminate boundaries by performing a CPS transformation of the program. The transformation is defined as before, but with new wrapping procedures as in figure 5.4. Also as before we can establish that this translation is semantics-preserving through a logical relation between source terms and their translations. The relation is nearly the same as before, except that we modify the definition of the relation at call-by-name function types to relate source and CPS terms if all possible applications to related terms are related, regardless of whether the source term's function position coterminates with the CPS term's function computation. The relation is defined in figure 5.5.

Lifting that relation to apply to substitution using the same procedure as in definition 5.2.6, we prove correctness using the same method as before:

Lemma 5.4.1 (CPS-correctness lemma 2)**.** *Both of the following hold:*

1. *If* $\Gamma \vdash_n \mathbf{e} : \tau$ *and* $\gamma_{\text{src}} \sim_{c_2} \gamma_{\text{cps}} : \Gamma$*, then* $\gamma_{\text{src}}(\mathbf{e}) \sim_{c_2} \gamma_{\text{cps}}([\![\mathbf{e}]\!]_N) : \tau$

2. *If* $\Gamma \vdash_v \mathsf{e} : \tau$ *and* $\gamma_{\text{src}} \sim_{c_2} \gamma_{\text{cps}} : \Gamma$*, then* $\gamma_{\text{src}}(\mathsf{e}) \sim_{c_2} \gamma_{\text{cps}}([\![\mathsf{e}]\!]_V) : \tau$

Lemma 5.4.2 (CPS-correctness bridge lemma 2)**.** *Both of the following hold:*

1. *If* $\mathbf{e}_{src} \sim_{c_2} \mathbf{e}_{cps} : \tau$*, then* $(\mathsf{VN}^{\tau}\ \mathbf{e}_{src}) \sim_{c_2} (\mathcal{W}^{\tau}_{NV}\ \mathbf{e}_{cps}) : \tau$.

$$
\begin{array}{lcl}
[\![x]\!]_N & = & x \\
[\![n]\!]_N & = & \lambda k.\, k\, n \\
[\![\Uparrow]\!]_N & = & \lambda k.\, \Uparrow \\
[\![\lambda x.\, \mathbf{e}]\!]_N & = & \lambda k.\, k\, (\lambda x.\, [\![\mathbf{e}]\!]_N) \\
[\![(\mathbf{e}_1\, \mathbf{e}_2)]\!]_N & = & \lambda k.\, ([\![\mathbf{e}_1]\!]_N\, (\lambda f.\, (f\, [\![\mathbf{e}_2]\!]_N\, k))) \\
[\![\mathbf{NV}^\tau\, \mathbf{e}]\!]_N & = & (\mathcal{W}^\tau_{NV}\, [\![\mathbf{e}]\!]_V)
\end{array}
\qquad
\begin{array}{lcl}
[\![x]\!]_V & = & \lambda k.\, k\, x \\
[\![n]\!]_V & = & \lambda k.\, k\, n \\
[\![\Uparrow]\!]_V & = & \lambda k.\, \Uparrow \\
[\![\lambda x.\, \mathsf{e}]\!]_V & = & \lambda k.\, k\, (\lambda x.\, [\![\mathsf{e}]\!]_V) \\
[\![(\mathsf{e}_1\, \mathsf{e}_2)]\!]_V & = & \lambda k.\, [\![\mathsf{e}_1]\!]_V\, (\lambda f.\, [\![\mathsf{e}_2]\!]_V\, (\lambda x.\, f\, x\, k)) \\
[\![\mathsf{VN}^\tau\, \mathbf{e}]\!]_V & = & (\mathcal{W}^\tau_{VN}\, [\![\mathbf{e}]\!]_N)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{W}^{\mathbb{N}}_{NV} = \lambda c.\, c \\
\mathcal{W}^{\tau_1 \to \tau_2}_{NV} = \\
\lambda z.\, \lambda k_1.\, (k_1\, (\lambda x_1.\, (\mathcal{W}^{\tau_2}_{NV}\, (\lambda k_2.\, (z\, (\lambda v_1.\ (v_1\, (\mathcal{W}^{\tau_1}_{VN}\, (\lambda k_3.\, (k_3\, x_1)))\, k_2)))))))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{W}^{\mathbb{N}}_{VN} = \lambda c.\, c \\
\mathcal{W}^{\tau_1 \to \tau_2}_{VN} = \\
\lambda z.\, \lambda k_1.\, z\, (\lambda f.\ (k_1\, (\lambda x_1.\, \lambda k_2.\, ((\mathcal{W}^{\tau_1}_{NV}\, x_1)\, (\lambda v_1.\, (\mathcal{W}^{\tau_2}_{VN}\, (f\, v_1)\, k_2))))))
\end{array}
$$

Figure 5.4: Boundary-elmination CPS transformation 2

2. *If* $e_{src} \sim_{c_2} e_{cps} : \tau$, *then* $(\boldsymbol{NV}^\tau\ e_{src}) \sim_{c_2} (\mathcal{W}^\tau_{VN}\, e_{cps}) : \tau$.

Lemma 5.4.3. *If* $\forall \mathcal{E}[].\ \mathcal{E}[\boldsymbol{e}_{\text{src}}] \mapsto^* \Uparrow$ *and* $\forall k.\, (\boldsymbol{e}_{\text{cps}}\ k) \mapsto^* \Uparrow$ *then* $\boldsymbol{e}_{\text{src}} \sim_{c_2} \boldsymbol{e}_{\text{cps}} : \tau$.

Theorem 5.4.4 (compiler correctness 2)**.** *If* $\vdash_n \boldsymbol{e} : \mathbb{N}$ *then* $\boldsymbol{e} =_{\text{kleene}} ([\![\boldsymbol{e}]\!]_N\, (\lambda I.\, I))$.

5.4.3 Equivalence preservation

We can develop a notion of equivalence in $\Lambda_{(n+v)_2}$ again by means of logical relations, this time using the relation defined in figure 5.6. We establish that this relation is sound with respect to contextual equivalence in the combined language by establishing that it is a consistent, congruent, reflexive relation. Consistency is immediate:

Lemma 5.4.5 (consistency)**.** *If* $\boldsymbol{e}_1 \sim_n \boldsymbol{e}_2 : \mathbb{N}$ *then* $\boldsymbol{e}_1 =_{\text{kleene}} \boldsymbol{e}_2$.

Proof. Immediate by the definition of logical equivalence. □

Next we prove that reflexivity holds:

$$
\begin{array}{rcl}
\mathbf{e}_{\text{src}} \sim_{c_2} \mathbf{e}_{\text{cps}} : \mathbb{N} & \stackrel{\text{def}}{=} & \text{either } \mathbf{e}_{\text{src}} \hookrightarrow^* \Uparrow \text{ and } \forall k.(\mathbf{e}_{\text{cps}}\ k) \hookrightarrow^* \Uparrow \\
& & \text{or } \mathbf{e}_{\text{src}} \hookrightarrow^* \mathbf{v}_{\text{src}}, \forall k.(\mathbf{e}_{\text{cps}}\ k) \hookrightarrow^* (k\ \mathbf{v}_{\text{cps}}) \\
& & \text{and } \exists n \in \mathbb{N}.\mathbf{v}_{\text{src}} = \mathbf{v}_{\text{cps}} = \overline{n} \\
\mathbf{e}_{\text{src}} \sim_{c_2} \mathbf{e}_{\text{cps}} : \tau_1 \to \tau_2 & \stackrel{\text{def}}{=} & \forall \mathbf{e}'_{\text{src}}, \mathbf{e}'_{\text{cps}} \text{ such that } \mathbf{e}'_{\text{src}} \sim_{c_2} \mathbf{e}'_{\text{cps}} : \tau_1. \\
& & (\mathbf{e}_{\text{src}}\ \mathbf{e}'_{\text{src}}) \sim_{c_2} (\lambda k.\ (\mathbf{e}_{\text{cps}}\ (\lambda f.\ (f\ \mathbf{e}'_{\text{cps}}\ k)))) : \tau_2
\end{array}
$$

$$
\begin{array}{rcl}
e_{\text{src}} \sim_{c_2} \mathbf{e}_{\text{cps}} : \tau & \stackrel{\text{def}}{=} & \text{either } e_{\text{src}} \hookrightarrow^* \Uparrow \text{ and } \forall k.(\mathbf{e}_{\text{cps}}\ k) \hookrightarrow^* \Uparrow \\
& & \text{or } e_{\text{src}} \hookrightarrow^* v_{\text{src}}, \\
& & \quad \forall k.(\mathbf{e}_{\text{cps}}\ k) \hookrightarrow^* (k\ \mathbf{v}_{\text{cps}}), \\
& & \text{and } v_{\text{src}} \sim_{c_2,v} \mathbf{v}_{\text{cps}} : \tau \\
\\
n_{\text{src}} \sim_{c_2,v} n_{\text{cps}} : \mathbb{N} & \stackrel{\text{def}}{=} & n_{\text{src}} = n_{\text{cps}} \\
v_{\text{src}} \sim_{c_2,v} \mathbf{v}_{\text{cps}} : \tau_1 \to \tau_2 & \stackrel{\text{def}}{=} & \forall \mathbf{v}'_{\text{src}}, \mathbf{v}'_{\text{cps}} \\
& & \text{where } \mathbf{v}'_{\text{src}} \sim_{c_2,v} \mathbf{v}'_{\text{cps}} : \tau_1 \\
& & \text{we have } (v_{\text{src}}\ \mathbf{v}'_{\text{src}}) \sim_{c_2} \lambda k.\ \mathbf{v}_{\text{cps}}\ \mathbf{v}'_{\text{cps}}\ k : \tau_2
\end{array}
$$

Figure 5.5: CPS logical relation 2

Lemma 5.4.6 (reflexivity / fundamental theorem of the logical relation)**.** *Both of the following are true:*

1. *If* $\Gamma \vdash_n \boldsymbol{e} : \tau$, *then* $\Gamma \vdash \boldsymbol{e} \sim_n \boldsymbol{e} : \tau$
2. *If* $\Gamma \vdash_v e : \tau$, *then* $\Gamma \vdash e \sim_v e : \tau$

The proof of this lemma is standard except that it requires a bridge lemma that establishes that relation in one language implies relation in the other:

Lemma 5.4.7 (reflexivity bridge lemma)**.** *Both of the following are true:*

1. *If* $\Gamma \vdash \boldsymbol{e}_1 \sim_n \boldsymbol{e}_2 : \tau$, *then* $\Gamma \vdash \mathit{VN}^\tau\ \boldsymbol{e}_1 \sim_v \mathit{VN}^\tau\ \boldsymbol{e}_2 : \tau$
2. *If* $\Gamma \vdash e_1 \sim_v e_2 : \tau$, *then* $\Gamma \vdash \boldsymbol{NV}^\tau\ e_1 \sim_n \boldsymbol{NV}^\tau\ e_2 : \tau$

We use this lemma to prove congruence, which is otherwise standard, though note that we must unfortunately check all combinations of call-by-name or call-by-value contexts

$$
\begin{array}{lcl}
\mathbf{e}_1 \sim_n \mathbf{e}_2 : \mathbb{N} & \stackrel{\text{def}}{=} & \exists \mathbf{v}_1, \mathbf{v}_2 \in \Lambda_{(n+v)_1}. \\
 & & \forall \mathcal{E}.\ \mathcal{E}[\mathbf{e}_1] \mapsto^* \mathcal{E}[\mathbf{v}_1] \Leftrightarrow \mathcal{E}[\mathbf{e}_2] \mapsto^* \mathcal{E}[\mathbf{v}_2] \text{ and } \mathbf{v}_1 = \mathbf{v}_2 \\
\mathbf{e}_1 \sim_n \mathbf{e}_2 : \tau_1 \to \tau_2 & \stackrel{\text{def}}{=} & \forall \mathbf{e}'_1, \mathbf{e}'_2 \in \Lambda_{(n+v)_1}. \\
 & & \mathbf{e}'_1 \sim_n \mathbf{e}'_2 : \tau_1 \Rightarrow (\mathbf{e}_1\, \mathbf{e}'_1) \sim_n (\mathbf{e}_2\, \mathbf{e}'_2) : \tau_2 \\
 & & \\
\mathsf{e}_1 \sim_v \mathsf{e}_2 : \tau & \stackrel{\text{def}}{=} & \exists \mathsf{v}_1, \mathsf{v}_2 \in \Lambda_{(n+v)_1}. \\
 & & \forall \mathcal{E}. \mathcal{E}[\mathsf{e}_1] \mapsto^* \mathsf{v}_1 \Leftrightarrow \mathcal{E}[\mathsf{e}_2] \to^* \mathsf{v}_2 \text{ and } \mathsf{v}_1 \sim_v \mathsf{v}_2 : \tau \\
\mathsf{v}_1 \sim_v \mathsf{v}_2 : \mathbb{N} & \stackrel{\text{def}}{=} & \mathsf{v}_1 = \mathsf{v}_2 \\
\mathsf{v}_1 \sim_v \mathsf{v}_2 : \tau_1 \to \tau_2 & \stackrel{\text{def}}{=} & \forall \mathsf{v}'_1, \mathsf{v}'_2 \in \Lambda_{(n+v)_1}. \\
 & & \mathsf{v}'_1 \sim_v \mathsf{v}'_2 : \tau_1 \Rightarrow (\mathsf{v}_1\, \mathsf{v}'_1) \sim_v (\mathsf{v}_2\, \mathsf{v}'_2) : \tau_2
\end{array}
$$

Figure 5.6: Logical equivalence relation for terms in the second embedding

with call-by-name or call-by-value holes. (This may seem strange at first, but notice that $(\mathbf{NV}^\tau\ [\,]_V)$ and $(\mathsf{VN}^\tau\ [\,]_N)$ are contexts whose top-level language and hole language differ.)

Lemma 5.4.8 (congruence)**.** *All of the following are true:*

1. *If* $\Gamma \vdash \boldsymbol{e}_1 \sim_n \boldsymbol{e}_2 : \tau$ *and* $\boldsymbol{C} : (\Gamma \vdash_n \tau) \rightsquigarrow (\Gamma' \vdash_n \tau')$ *then* $\Gamma' \vdash \boldsymbol{C}[\boldsymbol{e}_1] \sim_n \boldsymbol{C}[\boldsymbol{e}_2] : \tau'$
2. *If* $\Gamma \vdash \boldsymbol{e}_1 \sim_n \boldsymbol{e}_2 : \tau$ *and* $C : (\Gamma \vdash_n \tau) \rightsquigarrow (\Gamma' \vdash_v \tau')$ *then* $\Gamma' \vdash C[\boldsymbol{e}_1] \sim_v C[\boldsymbol{e}_2] : \tau'$
3. *If* $\Gamma \vdash e_1 \sim_v e_2 : \tau$ *and* $\boldsymbol{C} : (\Gamma \vdash_v \tau) \rightsquigarrow (\Gamma' \vdash_n \tau')$ *then* $\Gamma' \vdash \boldsymbol{C}[e_1] \sim_n \boldsymbol{C}[e_2] : \tau'$
4. *If* $\Gamma \vdash e_1 \sim_v e_2 : \tau$ *and* $C : (\Gamma \vdash_v \tau) \rightsquigarrow (\Gamma' \vdash_v \tau')$ *then* $\Gamma' \vdash C[e_1] \sim_v C[e_2] : \tau'$

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

Theorem 5.4.9. *Both of the following are true:*

1. *If* $\boldsymbol{e}_1 \sim_n \boldsymbol{e}_2 : \tau$ *then* $\boldsymbol{e}_1 \cong_{n+v} \boldsymbol{e}_2 : \tau$
2. *If* $e_1 \sim_v e_2 : \tau$ *then* $e_1 \cong_{n+v} e_2 : \tau$

Proof. Special case of lemma 5.4.8. □

In section 5.2.3 we showed that $\Uparrow$ and $(\lambda x.\ \Uparrow)$, which are equivalent in Λ_n, could be distinguished using the $\Lambda_{(n+v)_1}$ context $\mathbf{C}_1 = (\mathbf{NV}^{\mathbb{N}}\ ((\lambda f.\ 12)\,(\mathsf{VN}^{\tau_1\to\tau_2}\ [\,]_N)))$. Under $\Lambda_{(n+v)_2}$, however, $\mathbf{C}_1$ no longer distinguishes the terms:

$$\begin{aligned}\mathbf{C}_1[\Uparrow] &= (\mathbf{NV}^{\mathbb{N}}\ ((\lambda f.\ 12)\,(\mathsf{VN}^{\tau_1\to\tau_2}\ \Uparrow)))\\ &\mapsto (\mathbf{NV}^{\mathbb{N}}\ ((\lambda f.\ 12)\,(\lambda x.\ (\mathsf{VN}^{\tau_2}\ (\Uparrow\ (\mathbf{NV}^{\tau_1}\ x))))))\mapsto(\mathbf{NV}^{\mathbb{N}}\ 12)\mapsto 12\end{aligned}$$

$$\begin{aligned}\mathbf{C}_1[(\lambda x.\ \Uparrow)] &= (\mathbf{NV}^{\mathbb{N}}\ ((\lambda f.\ 12)\,(\mathsf{VN}^{\tau_1\to\tau_2}\ (\lambda x.\ \Uparrow))))\\ &\mapsto (\mathbf{NV}^{\mathbb{N}}\ ((\lambda f.\ 12)\,(\lambda x.\ (\mathsf{VN}^{\tau_2}\ ((\lambda x.\ \Uparrow)\ (\mathbf{NV}^{\tau_1}\ x))))))\mapsto(\mathbf{NV}^{\mathbb{N}}\ 12)\mapsto 12\end{aligned}$$

In fact, using theorem 5.4.9 we can see that those two terms are not distinguishable by any $\Lambda_{(n+v)_2}$ contexts at all:

Proposal 5.4.10. $\Uparrow \cong_{n+v} \lambda x.\ \Uparrow : \mathbf{Nat}\to\mathbf{Nat}$.

Proof. Fix $\mathbf{e}_1, \mathbf{e}_2$ such that $\mathbf{e}_1 \simeq_n \mathbf{e}_2 : \mathbf{Nat}$. We have that $(\Uparrow\ \mathbf{e}_1)\mapsto\Uparrow$ and $((\lambda x.\ \Uparrow)\ \mathbf{e}_2)\mapsto\Uparrow$. Hence by definition of the logical relation, $\Uparrow \sim_n \lambda x.\ \Uparrow : \mathbf{Nat}\to\mathbf{Nat}$. Thus by theorem 5.4.9, $\Uparrow\cong_{n+v}\lambda x.\ \Uparrow : \mathbf{Nat}\to\mathbf{Nat}$ as required. □

5.5 The gold standard

The fact that proposal 5.4.10 holds is a good sign, but to establish the gold standard we need to show that *any* two terms that are equal in Λ_n alone are also equal in $\Lambda_{(n+v)_2}$.

Our strategy is to prove the gold standard's contrapositive, *i.e.* that if there exists a $\Lambda_{(n+v)_2}$ program context that distinguishes two purely-Λ_n terms, then there is also a Λ_n context that distinguishes them[3]. We can almost show this by construction: suppose $\mathbf{C}[\,]_N$

3. At this point a reader may wonder why this result is not a near-immediate implication of the fact that the relation given in figure 5.6 corresponds to contextual equivalence given that its two halves look very similar to those given previously. The answer is that the similarity is only superficial. Consider the two cases for arrow types: in a standard logical equivalence relation, two call-by-name (or call-by-value) terms are be related at arrow types if their applications to related *purely call-by-name* (or purely call-by-value) arguments are related. In our relation, though, two call-by-name (or call-by-value) terms are related at arrow types if their applications to related call-by-name terms *that potentially contain call-by-value boundaries* (or call-by-value terms that

is a $\Lambda_{(n+v)_2}$ program context. It is equivalent by a single beta-reduction to the context $((\lambda t.\ \mathbf{C}[t])\ [\,]_N)$ (where t does not appear in $\mathbf{C}$). We would like to now use the CPS translation turn this into a pure-call-by-name term something like $((\lambda t.\ [\![\mathbf{C}[t]]\!]_N)\ [\,]_N\ (\lambda I.\ I))))$ — to adapt it to open terms we could simply perform the equivalent of lambda-lifting and generalize the context to $((\lambda t.\ [\![\mathbf{C}[(t\ x_1 \cdots)]]\!]_N)\ (\lambda x_1 \cdots.\ [\,]_N)\ (\lambda I.\ I))))$ where $x_1 \cdots$ is a list of all variables bound by the context at the point where the hole occurs.

This strategy does not quite work, though, because it would mean we were substituting arbitrary terms into contexts that were expecting CPS computations. Fortunately, though, we can dynamically convert from a plain call-by-name term into an equivalent CPS computation (and vice versa, as turns out to be necessary) using the following type-indexed functions that are defined directly as call-by-name terms:

$$
\begin{array}{lcl}
exp^{\tau} & : & \tau \to [\![\tau]\!]_N \\
exp^{\mathbb{N}} & \stackrel{\text{def}}{=} & \lambda x.\ \lambda k.\ k(x+0) \\
exp^{\tau_1 \to \tau_2} & \stackrel{\text{def}}{=} & \lambda f.\ \lambda k.\ k\ (\lambda x.\ (exp^{\tau_2}\ (f\ (imp^{\tau_1}\ x))))
\end{array}
$$

$$
\begin{array}{lcl}
imp^{\tau} & : & [\![\tau]\!]_N \to \tau \\
imp^{\mathbf{N}} & \stackrel{\text{def}}{=} & \lambda x.\ (x\ (\lambda I.\ I)) \\
imp^{\tau_1 \to \tau_2} & \stackrel{\text{def}}{=} & \lambda c.\ c\ (\lambda f.\ \lambda x.\ (imp^{\tau_2}\ (f\ (exp^{\tau_1}\ x))))
\end{array}
$$

We establish that these functions are correct by proving that if two terms $\mathbf{e}_1$ and $\mathbf{e}_2$ are logically equivalent (under the combined call-by-name and call-by-value equivalence relation) at type τ, then $\mathbf{e}_1$ is CPS-related to $(exp^{\tau}\ \mathbf{e}_2)$ at type τ and simultaneously that if $\mathbf{e}_1$ is CPS-related to $\mathbf{e}_2$ at type τ then $\mathbf{e}_1$ is logically equivalent to $(imp^{\tau}\ \mathbf{e}_2)$ at type τ.

Lemma 5.5.1 (exp^{τ} and imp^{τ} correctness)**.** *Both of the following hold:*

1. *If* $\boldsymbol{e}_1 \sim_n \boldsymbol{e}_2 : \tau$, *then* $\boldsymbol{e}_1 \sim_{c_2} (exp^{\tau}\ \boldsymbol{e}_2) : \tau$.

2. *If* $\boldsymbol{e}_{\text{src}} \sim_{c_2} \boldsymbol{e}_{\text{cps}} : \tau$, *then* $\boldsymbol{e}_{\text{src}} \sim_n (imp^{\tau}\ \boldsymbol{e}_{\text{cps}}) : \tau$.

---

potentially contain call-by-name boundaries) are related. We can think of this section as an attempt to show that the latter implies the former.

Lemma 5.5.2. *If* $\mathbf{e}_{\text{src}} \sim_{c_2} \mathbf{e}_{\text{cps}} : \tau$ *and* $\mathbf{e}_{\text{cps}} \cong_{n+v} \mathbf{e}'_{\text{cps}} : [\![\tau]\!]_N$ *then* $\mathbf{e}_{\text{src}} \sim_{c_2} \mathbf{e}'_{\text{cps}} : \tau$.

Given exp^{τ}, we use the context $((\lambda \mathbf{x}.\ ([\![\mathbf{C}[\mathbf{x}]]\!]_N\ (\lambda I.\ I)))\ (exp^{\tau}\ [\ ]_N))$ (henceforth $\mathbf{C}'$) to complete the proof outline above. By construction, $\mathbf{C}'[\mathbf{e}]$ terminates with a number $\overline{n}$ if and only if $\mathbf{C}[\mathbf{e}]$ terminates. Furthermore, it is a purely call-by-name context. Since we have placed no restrictions on $\mathbf{C}$, we can build this device for an arbitrary context. The following theorem codifies that intuitive argument.

Lemma 5.5.3 (The gold standard contrapositive)**.** *If* $\Gamma \vdash \mathbf{e}_1 \ncong_{n+v} \mathbf{e}_2 : \tau$ *where* $\mathbf{e}_1$ *and* $\mathbf{e}_2$ *are entirely call-by-name terms, then* $\Gamma \vdash \mathbf{e}_1 \ncong_n \mathbf{e}_2 : \tau$.

Proof. By definition, $\exists \mathbf{C} : (\Gamma \vdash \tau) \rightsquigarrow (\vdash \mathbf{N})$ such that $\mathbf{C}[\mathbf{e}_1]$ disagrees with $\mathbf{C}[\mathbf{e}_2]$. Furthermore, by beta-equivalence this context is equivalent to $((\lambda x.\ \mathbf{C}[(x\, x_1 \cdots)])\ (\lambda x_1 \cdots.\ [\ ]_N))$ where $x_1 \cdots$ indicates a sequence consisting of the variables in Γ in some canonical order (we will use $\tau_1 \cdots$ as the sequence consisting of those variables' types). By lemma 5.4.1, we have that $(\lambda x.\ \mathbf{C}[(x\, x_1 \cdots)]) \sim_{c_2} [\![(\lambda x.\ \mathbf{C}[(x\, x_1 \cdots)])]\!]_N : (\tau_1 \cdots \to \tau) \to \mathbf{N}$. By lemma 5.4.6 we have that $(\lambda x_1 \cdots.\ \mathbf{e}_1) \sim_n (\lambda x_1 \cdots.\ \mathbf{e}_1) : \tau_1 \cdots \to \tau$ and hence by lemma 5.5.1 that $(\lambda x_1 \cdots.\ \mathbf{e}_1) \sim_{c_2} (exp^{\tau_1 \cdots \to \tau}\ (\lambda x_1 \cdots.\ \mathbf{e}_1)) : \tau_1 \cdots \to \tau$. Applying the definition of the CPS logical relation, we have that

$$((\lambda t.\ \mathbf{C}[(t\, x_1 \cdots)])\ (\lambda x_1 \cdots.\ \mathbf{e}_1) \sim_{c_2} \lambda k.\ ([\![(\lambda x.\ \mathbf{C}[(x\, x_1 \cdots)])]\!]_N\ (\lambda f.\ f\ (exp^{\tau_1 \cdots \to \tau}\ (\lambda x_1 \cdots.\ \mathbf{e}_1)))\ k) : \mathbf{N}$$

Thus by definition if $\mathbf{C}[\mathbf{e}_1] \mapsto^* \overline{n}$ then the CPSed version is a computation that reduces to the same number when applied to $\lambda I.\ I$, and furthermore if $\mathbf{C}[\mathbf{e}_1] \mapsto^* \Uparrow$ then the CPSed version does as well when applied to $\lambda I.\ I$. Hence we have that

$$\mathbf{C}[\mathbf{e}_1] =_{\text{kleene}} \lambda k.\ ([\![(\lambda t.\ \mathbf{C}[(t\, x_1 \cdots)])]\!]_N\ (\lambda f.\ f\ (exp^{\tau_1 \to \cdots \to \tau_n \to \tau}\ (\lambda x_1 \cdots.\ \mathbf{e}_1)))\ k)\ (\lambda I.\ I)$$

The same line of reasoning applies to $\mathbf{C}[\mathbf{e}_2]$; thus

$$\mathbf{C}[\mathbf{e}_2] =_{\text{kleene}} \lambda k.\ ([\![(\lambda t.\ \mathbf{C}[(t\, x_1 \cdots)])]\!]_N\ (\lambda f.\ f\ (exp^{\tau_1 \cdots \to \tau}\ (\lambda x_1 \cdots.\ \mathbf{e}_2)))\ k)\ (\lambda I.\ I)$$

and hence

$$\lambda k.\ ([\![(\lambda t.\ \mathbf{C}[(t\ x_1 \cdots)])]\!]_N\ (\lambda f.\ f\ (exp^{\tau_1 \to \cdots \to \tau}\ (\lambda x_1 \cdots.\ \mathbf{e}_1)))\ k)\ (\lambda I.\ I)$$
$$\neq_{\text{kleene}}$$
$$\lambda k.\ ([\![(\lambda t.\ \mathbf{C}[(t\ x_1 \cdots)])]\!]_N\ (\lambda f.\ f\ (exp^{\tau_1 \cdots \to \tau}\ (\lambda x_1 \cdots.\ \mathbf{e}_2)))\ k)\ (\lambda I.\ I)$$

Therefore the context

$$\lambda k.\ ([\![(\lambda t.\ \mathbf{C}[(t\ x_1 \cdots)])]\!]_N\ (\lambda f.\ f\ (exp^{\tau_1 \cdots \to \tau}\ (\lambda x_1 \cdots.\ [\,]_N)))\ k)\ (\lambda I.\ I)$$

distinguishes $\mathbf{e}_1$ and $\mathbf{e}_2$. This is an entirely call-by-name context; thus $\Gamma \vdash \mathbf{e}_1 \ncong_n \mathbf{e}_2 : \tau$. □

Theorem 5.5.4 (The gold standard)**.** *For entirely call-by-name terms* $\boldsymbol{e}_1$ *and* $\boldsymbol{e}_2$, $\Gamma \vdash \boldsymbol{e}_1 \cong_n \boldsymbol{e}_2 : \tau$ *if and only if* $\Gamma \vdash \boldsymbol{e}_1 \cong_{n+v} \boldsymbol{e}_2 : \tau$.

Proof. The "if" direction is a corollary of lemma 5.5.3; the "only if" direction holds trivially because entirely call-by-name contexts are a subset of call-by-name plus call-by-value contexts. □

It is important to point out that though we have used CPS and thus a "compilition-based" argument in support of the gold standard claim, the result is a property that is completely independent of any compiler.

5.6 Related work

At the beginning of this chapter we discussed our relation to Kennedy. Felleisen (Felleisen 1991, theorems 3.9 and 3.15) establishes that Λ_n cannot macro-express call-by-value lambda abstractions. This does not contradict our results in section 5.4: in Felleisen's system, call-by-value lambda abstractions always force their arguments, exactly the problem that caused $\Lambda_{(n+v)_1}$ to fail. He also notes that while a programming language extension that can tell apart two otherwise indistinguishable terms necessarily adds to the language's expressive power, it is possible for extensions that add expressive power to preserve the base language's equations (Felleisen 1991, theorem 3.14ii). Our proposed gold standard, then,

suggests that well-designed foreign interfaces should either be macro-expressible in the host language or fall into this second case.

Riecke gives another result that at first glance appears to contradict the present work (Riecke 1991, theorem 15). Again, there is no contradiction, but this time for a more subtle reason that points at the additional insight we gain from the framework we have developed in this dissertation. The translations Riecke considers must be performed in a vacuum: the translation of any term must behave in an observably identical way to the original term and be insertable into an arbitrary foreign context. Our translations do not behave this way; $[\![\tau]\!]_N$ and $[\![\tau]\!]_V$ produce computations that are only suitable for composing with other computations of the same kind. Call-by-name computations, but not call-by-value computations, can be made implicit with imp^τ in a meaning-preserving way, and computations can be converted from one language to another using $\mathcal{W}^\tau_{NV}$ and $\mathcal{W}^\tau_{VN}$ but these conversions behave as though they had boundaries around them. (The technical version of this point is that it does not appear that within our system there exist $proj^\tau$ and inj^τ functions that satisfy the constraints of Riecke's definition 13 — certainly neither exp^τ and imp^τ nor $\mathcal{W}^\tau_{NV} \circ exp^\tau$ and $imp^\tau \circ \mathcal{W}^\tau_{VN}$ fit the bill — thus it appears that our system is not a "functional translation" as per his definition, and thus theorem 15 does not apply.)

5.7 Conclusions and further applications

We believe that the process we have been through in this chapter — choose two languages, connect them with boundaries, and then vary those boundaries' semantics until you find semantics that preserve the gold standard — is an effective way to design foreign interfaces between general-purpose languages, even if you do not establish the gold standard formally. To illustrate this, we conclude by sketching designs for multilanguage systems involving languages with different features. In each example, we describe a naive embedding that we argue does not preserve the gold standard, then describe another embedding we argue does preserve it.

5.7.1 State

Suppose that language L is a strict language with no imperative state features. In L, we have that $(\lambda fg.\ g\ (f\ 1)\ (f\ 1) \cong_l \lambda fg.$ **let** $y =$ **f** 1 **in** $g\ y\ y$ **end**$)$: even in the presence of nontermination or error signaling, in this language running $(f\ 1)$ and using its result twice is the same as running $(f\ 1)$ twice. Now suppose that K is another language that has the features of L plus mutable state (in this example, using ML-style ref cells).

Under a naive embedding, the single language equation does not hold in multilanguage contexts, *i.e.* $(\lambda fg.\ g\ (f\ 1)\ (f\ 1) \ncong_{l+k} \lambda fg.$ **let** $y =$ **f** 1 **in** $g\ y\ y$ **end**$)$. To see why, consider the context

$$(\textbf{let } x = \textbf{ref } 0 \textbf{ in } ((\textbf{KL } [\,])\ (\lambda z.\ (x := !x + 1; !x))\ (\lambda ab.\ a == b)) \textbf{ end})$$

This context provides a function that returns a different value each time it is called, and uses it to observe the difference between the two terms.

This can be fixed in at least two ways, both of which involve restricting the types of boundaries. The first way is to restrict boundary types monadically *a la* Haskell, so that any function that enters L from K consumes and produces an additional "world state" argument. Under this scheme enemy context we are thinking of contains a boundary whose type is illegal and is not a counterexample. The second way is to add linear types to L (Wadler 1990) and add a global value and make the type of every boundary linear; this would prevent any boundary from being used more than once which would again make our proposed counterexample ill-typed.

5.7.2 Exceptions

Suppose that language L now has mutable state. In this language it seems clear that $(\lambda f.$ **let** $x =$ **ref** 0 **in** $(f(\lambda g.\ (x := 1; g(); x := 0)); !x)$ **end**$)$ is contextually equivalent to $(\lambda f.$ **let** $x =$ **ref** 0 **in** $(f(\lambda g.\ (x := 1; g(); x := 0)); 0)$ **end**$)$. In other words, suppose we have a ref cell whose intitial value is 0 and function that takes a thunk g and does nothing other set x to 1, call g for effect, and then set x back to 0. If we provide that function to some

arbitrary context-chosen function f (that has no access to x), then no matter what f does the value of x after it returns (if it returns at all) will be 0.

But if K contains exceptions, then in a naive embedding this equation does not hold, *i.e.* $\lambda f.$ **let** x **= ref** 0 **in** $(f(\lambda g.\ (x := 1; g(); x := 0)); !x)$ **end** is not contextually equal to $\lambda f.$ **let** x **= ref** 0 **in** $(f(\lambda g.\ (x := 1; g(); x := 0)); 0)$ **end**. This is because a context that can throw and catch exceptions can arrange to skip parts of the stack, for instance with the context

$$((\mathbf{KL}\ [\,])\ (\lambda h.\ \mathbf{try}\ h(\lambda\_.\ \mathbf{throw}\ ())\ \mathbf{catch}\ e\ \Rightarrow\ ()))$$

By having g throw an exception and then catching it, f causes x to be incremented but not decremented.

This can be fixed by making boundaries catch exceptions and then either aborting the program (as we did in chapter 3.5)r converting them into values (as we did in chapter 3.6) This strategy appears to allow the attacker context without allowing it to distinguish the two terms.

5.7.3 Thread systems

Finally, imagine that L now has mutable state and exceptions. We have that $(\lambda x.\ (x := 1; !x) \cong_l \lambda x.\ (x := 1; 1))$ but if it is embedded naively into a language K with threads, then $(\lambda x.\ (x := 1; !x) \ncong_{l+k} \lambda x.\ (x := 1; 1))$. This is because it introduces a race condition, as could be exposed by the context

$$\mathbf{let}\ x = (\mathbf{KL}\ \mathbf{ref}\ 0)\ \mathbf{in}\ (\mathbf{spawn}(\mathbf{KL}\ \lambda\_.\ x := 2); (\mathbf{KL}\ [\,]\ (\mathbf{LK}\ x)))\ \mathbf{end}$$

which might set x to 2 after the two terms set x to 1 but before they return either $!x$ or 1.

We can fix this by demanding that whenever L runs it does so in a single-threaded mode (which can be encoded formally by restricting evaluation contexts on boundaries) or less conservatively by only allowing a single thread at a time to evaluate L code.

CHAPTER 6
APPLICATION: TOPSL

6.1 Introduction

In this chapter we present a final example of how we have exploited our multi-language framework to gain insight on problems by applying it to the design of a domain-specific embedded language for writing online surveys.

The demand for web-based surveys has grown significantly in recent years as social scientists have become more aware of the practical and theoretical benefits of gathering information online (Birnbaum 2000). Unfortunately, this growing demand has not been met by correspondingly mature technologies. While many domain-specific languages (DSLs) and wizards exist for on-line surveys — for instance SuML (Barclay et al. 2002) and QPL (Office) — every one we have found falls into the common trap for DSLs of being "80% solutions", with rich facilities for asking individual questions but only rudimentary support for the other tasks a programming language must perform, such as controlling flow and performing arbitrary computations. Simple surveys are easy to implement, but sometimes even seemingly minor extensions are impossible.

When studies run into these limitations, programmers resort to implementing them in a general-purpose language (GPL) such as PHP or Perl that allow them to express anything they want (as evidence that this is a popular approach, Fraley recently published a how-to guide on the subject for psychologists (Fraley 2004)). Unfortunately, if they make that choice, they become responsible for handling HTML generation, CGI, and data storage, all of which is unrelated to the specific survey being written. In the authors' direct experience, on-line surveys are plagued by bugs in this non-domain-specific code. For instance, my introduction to the problems with online surveys was as a student in a sociology class. The class designed an online survey and hired independent consultants to implement it in ColdFusion; unfortunately after a few hundred participants had already completed the

survey the class discovered that program had a major bug in its answer-saving routines that caused it to lose all answers to about half of the questions on the survey. When the bug was discovered, the class had to contact all the participants and ask them to fill the survey out again; only about 10% of the original participants actually did. Such incidents are common, but they are an unacceptable risk in expensive research.

It is natural that these two general strategies for solving the survey problem should emerge. Survey programs exist to collect answers to questions that will then be put into rows in a database or analyzed by a statistics program, and that might be printed out for copy-editing or for handing out to off-line survey participants. To make those operations possible, it must be possible to identify (before runtime) every question that could a particular survey possibly ask. Of course if a survey program had complete freedom at runtime to generate questions, that identification would be impossible. So, the problem must be made easier, and two simple ways to make it easier are to restrict the language in which programmers write surveys to the point where questions are statically identifiable, or restrict analysis to one particular survey and do the analysis by hand.

Both available options have serious problems, though: current DSLs afford too little flexibility in their models of flow control, and GPLs make programmers implement substantial amounts of non-domain-specific code for each survey. In this paper, we demonstrate a way to take the middle path with Topsl, a domain-specific language for writing on-line surveys embedded into the general-purpose language PLT Scheme. We follow the tradition of domain-specific embedded languages (DSELs) (Hudak 1996, 1998; Shivers 1996) by embedding a domain-specific sublanguage for survey-specific tasks into a general-purpose language, PLT Scheme, whose full power is available when necessary. We depart from tradition, however, in that rather than approaching the problem as one of programming appropriate combinators, instead we approach it as a multi-language system operational semantics design problem. Specifically, we design a small-step operational semantics that combines a calculus for a special-purpose survey language with a variant of the untyped lambda calculus that also models web interaction using ideas from Krishnamurthi et al's webL (Graunke et al. 2003; Krishnamurthi et al. 2006). After motivating our design with a description of some non-obvious design requirements we have found (section 6.2), we step back and develop a new design for Topsl by repeatedly refining a basic model of a

domain-specific survey language into a full-featured multi-language system (sections 6.3, 6.4, 6.5, 6.6, and 6.7). These models presented an unusual challenge for PLT Redex in that they require two kinds of nondeterminism; we discuss how we addressed that problem in section 6.8. Finally in section 6.9 we recount how the development we present here led us to a much simpler and more flexible version of Topsl than we had found in previous work (MacHenry and Matthews 2004).

6.2 Web surveys

The author was introduced to the problem of designing a language for online surveys by Dr. Eli Finkel, assistant professor of psychology at Northwestern University, in Fall 2003. At that time Finkel had written a paper design for a recurring online survey that he intended to use as a key component in a year-long longitudinal study[1]. Figure 6.1 shows a simplified version of his survey that will work for our purposes. At each two-week interval, every participant answered set 1, a set of general questions that assessed the status of his or her relationship. Then, depending on the answers the participant gave, he or she was categorized as either having entered a relationship, exited a relationship, both, or neither since the last time he or she completed the survey. Depending on the answer, the participant was guided to answer some relevant set of questions, some subset of question sets 2 through 4. Finally, all participants no matter their status were asked to answer the questions in set 5 once for *each* relationship they had been involved in during the study.

To his surprise Finkel found that while he could easily find survey-authoring services that would suffice to implement almost his entire design, none would implement set 5's looping behavior because none of them could handle surveys in which a question was asked once for each distinct way another question had been answered in the past, effectively creating a loop whose iteration count grew over the course of the study. Hearing about Finkel's problem, we immediately thought that his problem could be solved with a simple use of *map* in an implementation based around PLT Scheme's continuation-based web

1. A longitudinal study is a study in which the researcher tracks a group of participants over time; in Finkel's study, this meant asking a group of freshmen to complete a survey once every two weeks for half a year.

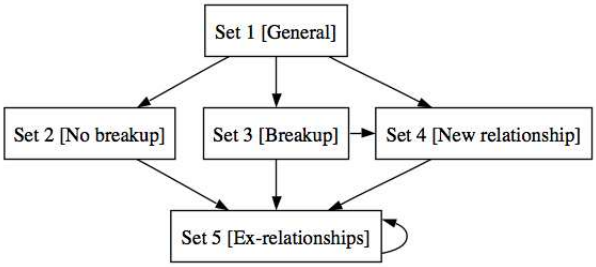

Figure 6.1: Finkel's survey

server (Graunke et al. 2001; Krishnamurthi et al. 2007), a technology we were eager to get practical experience with anyway. So, we decided to solve his problem by building an embedded domain-specific language for constructing on-line surveys using PLT Scheme web server. We quickly decided on two guiding design principles:

1. The system had to be easy to use correctly. Specifically we wanted it to be impossible for a programmer to write a buggy survey that fails to save a participant's responses to questions, in the same way that a traditional type system makes it impossible for a programmer to write a buggy program that can cause a segmentation fault.

2. The system had to be flexible enough to handle unforeseen designs. It would not be a particularly robust general solution if we just implemented another domain-specific language that added a new construct to allow us to write Finkel's loop, since the next survey that came along might have some other new twist that would put us back at the drawing board. Instead, we wanted a language that would be flexible enough to allow Finkel's loop but also new experimental designs that we hadn't even heard of yet.

At first blush this did not seem to be a hard problem. However, as we worked on the problem, we uncovered several non-obvious requirements that could not all be met simultaneously using a naive design strategy. The next three subsections discuss them in turn.

TST	: "The Scheme Type"
page	: *page-element ...* → *survey*
(**?** *question-type question-id string*)	: syntax[*page-element*]
sequence	: *survey ...* → *survey*
(**branch** *question-id* [(*TST* → *bool*) *survey*] ⋯)	: syntax[*survey*]
(**bind** *var symbol survey survey*)	: syntax[*survey*]

Figure 6.2: Topsl primitives, attempt 1

6.2.1 Row-passing style

A basic question that our implementation needed to answer was how to arrange the basic plumbing that receives user answers, allows them to be inspected and used during the program, and eventually saves them as completed survey answers. Unfortunately, this brings up a subtle implementation problem. Consider the function *ask*, a hypothetical function that Topsl could use to ask questions of a user and receive their answers. How should we arrange for it it to store those answers so that they cannot be lost? An obvious choice would be to use mutation of global state, but unfortunately mutation does not have quite the right semantics for our needs in the presence of *send/suspend*. When a participant, in the middle of some long multi-page survey, realizes she has made a mistake, say, three pages back, she may well press the back button to get back to the page with the error, resubmit it with a correct answer, and continue. This might cause the survey program to present her with the same two pages she just answered, but it may also cause the program to present some entirely new path of pages. In the former case, participant probably expects the old answers from the incorrect path to disappear, and in either case the researcher looking at the answers later definitely expects answers from paths not finished to disappear always since otherwise the set of survey answers might not correspond to any legal path through the survey. If *ask* mutates a global table to record new answers, the continuations *send/suspend* uses will capture control information and lexical bindings but not the current values in the store; thus when the participant goes back to an old page the global table will continue to hold now-invalid answers.

To avoid using mutation, though, we would have to make *ask* produce a row of answers,

and make every survey end with a survey-finalization procedure that took a list of answer rows that represented all answers that had been given over the entire survey and blessed them as a complete set of answers ready for data analysis. This causes another problem: left to their own devices, programmers could very easily write a buggy program that received a row of answers from *ask* and then simply ignored them, never including them in the final result set at the end of the program. In effect, programmers would have to write surveys in a monadic "result-row-passing style" that would be easy to get wrong. To alleviate that burden, we wrote several combinators that consumed and produced surveys of which pages were the ground elements, and threaded row-passing through automatically.

6.2.2 Abstraction

A subtle but important requirement we found when implementing Finkel's survey was that function abstractions play a more interesting role than we had initially thought they would. We had expected to write functions that abstracted over pages, since often pages needed to be dynamically generated: in Finkel's survey, many questions took the form "When is the last time you talked to *[name]*?" where *[name]* was the answer to a question on a previous page, or drawn from a previous session. In particular, his set 5, the set that had caused him so many problems, relied on this ability.

Another use of abstraction that we had not anticipated was to satisfy the need to define a standard, repetitive group of questions that appeared with variations many times on a survey. For instance, Finkel's survey included many 4-question blocks of the form, "How do you predict you will feel about *[subject]* in . . . one week? . . . two weeks? . . . one month? . . . two months?" where *[subject]* varied and each question was answered separately. For this pattern, we found it very useful to write question-producing functions that factored out the repeated elements.[2]

---

2. In fact, after completing the survey we learned that this form of abstraction was even more important than we had realized, since researchers in the social sciences take great pains to ensure that question types are standard from use to use and from survey to survey. They do this to cut down potential sources of imprecision in data analysis: for instance if some questions asked about how the participant would feel in "four weeks" and others asked how the participant would feel in "one month," analyses that used answers from both sets would need to take into account that

6.2.3 Static analysis

A requirement that might at first seem innocuous, but in fact has become the major technical challenge of Topsl's development, is the need to know all of the pages and questions that could be asked of a participant independently of any participant taking the survey and answering the questions (and in fact before participants even saw the survey at all). We have encountered this need in two ways: first, to build a database back-end laid out laid out with one table per page, each table column corresponding to a particular question and each row corresponding to a particular submission of answers, it is necessary to know each possible question and a SQL type that can hold answers to that question before the survey executes. Second, Finkel and his colleagues wanted documentation that was in principle derivable from the program itself: a printed, paper copy of the entire survey for proofreading, and a "code book" listing each question's ID, question type (*e.g.*, Likert-type[3] or free-response), and question text.[4] To meet these needs, we needed to know all possible questions a survey could produce, independently of any actual participant taking the survey.

This requirement is incompatible with using abstractions as a way to build up surveys and their answers. All of the previous requirements we have discuss tend to suggest that a survey ought be built up by composing functions that produce pages and questions. Some of these functions could be provided by a standard library and some of them were intended to be written directly by the Topsl programmer; some could be evaluated statically to predict their runtime effect but some could not even in principle be applied until a participant actually responded to some questions to produce inputs — without having any way to look

---

the participant might — perhaps subconsciously — think of "four weeks" and "one month" as significantly different durations.

3. A Likert-type question is the technical name for a question in which respondents are presented with a scale from 1 to 7, 1 labeled "Strongly disagree" and 7 labeled "Strongly agree", and are asked to rank where they fall on that scale with respect to some particular proposal. Likert-type questions are probably the most common question type in surveys conducted by social scientists due to their quantitative nature and broad applicability.

4. Every survey of any substantial size requires a code book. Surveys often comprise many hundreds of questions and researchers often share datasets with their colleagues or make them publically available, and a code book is the most compact and useful way to record all the information about a survey that a researcher would need to interpret its dataset.

inside these abstractions other than running them, we cannot not possibly perform a useful static analysis. Even applying them to dummy data was not workable, because a page's question set, or the particular phrasings of questions, could depend on the input data, and pages could produce or rely on effects — for instance, many pages in Finkel's survey took a participant ID and queried a database for information about that participant's responses in previous iterations of the survey.

It is possible to solve this problem by building a function-like mechanism that also supported some kind of inspection, for instance an object that had a *run* method and an *analyze* method. These abstractions could behave like a function when encountered at survey runtime by the other Topsl primitives. The construct could evaluate its body to a list of questions before survey runtime, allowing string constants in question text to be substituted with survey runtime arguments; static analysis could then simply read this list of questions without substituting arguments. Since the body is evaluated, we could use native Scheme functions to abstract over common question patterns. This is the solution documented previously (MacHenry and Matthews 2004). That approach has a limitation too, however, in that not only functions, but also every other way of putting functions together — **if**, **let**, *for-each*, and so on must be reimplemented so that they also support both *run* and *analyze* methods, or we will not be able to properly pipe the static analysis of their subparts to the top level. For instance, the survey

(**define-page** (*page1*)
  (**?** likert q_1 "Text 1 ...")
  (**?** free q_2 "Text 2 ...")
  (**?** likert q_3 "Text 3 ..."))

(**define-page** (*page2 txt*)
  (**?** free q_4 "Question pertaining to " *txt*))

(**bind** ([*txt* 'q_2] [*pivot* 'q_3]) (*page1*)
  (**if** (= *pivot* 7)
    (**sequence** (*page2 txt*))
    (**sequence**)))

would generate two database tables, one for *page1* and one for *page2*; the table for *page1* would contain columns q_1 of type number, q_2 of type text, and q_3 of type number, and the table for *page3* would contain a single column for question q_4 of type text. The only information relevant to the decision of how to construct tables is the **define-page** statements and their contents; any survey flow control code (for instance the last expression in the above code fragment, which controls evaluation of the survey) is completely ignored when generating a survey's static summary. The control information does, however, need to implement row-passing style, and each construct in the control is responsible for gathering up any rows produced by its subexpressions and delivering them in a reasonable way. For instance, that is why the **if** clause's second alternative has to be an empty **sequence** instead of (*void*) — if we had put (*void*) there instead, it would not observe row-passing style and thus the survey would not save its results.

While this system is workable, and in fact we and others implemented two more large-scale surveys using variations of it, it retains little of the simplicity of our initial idea. Ironically, the idea that had led us to take on the project in the first place — the idea that Finkel's set 5 loop corresponded to Scheme's *map* — did not make it into the final implementation, because Scheme's built-in *map* did not conform to row-passing style and did not work with our static analysis technique; instead we had to implement our own *map*-like construct. In fact, the impact of this design choice rippled through the entire language design, essentially requiring us to implement a Topsl version of every control construct we wanted to use. Furthermore, the language we had designed did not match our initial design goals very well: while we offered support for building surveys in row-passing style, in practice computations that needed to access an answer during survey runtime often had to deal with row-passing style manually, leading to the possibility of data loss errors; and while the system was flexible enough allow us to write Finkel's loop, the set of control constructs we provided was somewhat ad-hoc and limited to those constructs we could analyze statically, giving us no guarantee that we could implement the next survey we encountered without rebuilding the entire system.

These problems left us unsatisfied with our solution. We decided to go back to the drawing board, this time taking a formal approach to the same problem. In the next several sections, we define a formal model for a very simple survey-constructing DSL, and

then extend it. First we combine it with Scheme using boundaries as a simple technique to let us talk formally about the power that Scheme adds to the system. Thinking about the problem as a multilanguage problem gives us the insight required to see how we can borrow Scheme's existing control and binding constructs as a way of extending Topsl's power. Second we add a formal model for web-based interaction (including the ability for survey participants to submit pages multiple times) based on webL (Graunke et al. 2003; Krishnamurthi et al. 2006).

One important goal of this redesign is to ensure that our new language is safe by design, in that participants' responses can never be dropped or mixed improperly by a participant's use of the back button. Our strategy for achieving that goal is to check several properties at every step of the way: what we call the *analysis*, *consistency*, and *coherence* theorems. Analysis and consistency, taken together, say that every run of the survey must result in all the participant's responses stored in a database whose layout can be determined ahead of time: analysis says that we can safely approximate the set of possible pages and questions a survey could produce, and consistency says that the responses to every page and question will be remembered. The coherence property is only relevant to the web-based Topsl model, and guarantees that guarantees that any web-based interaction (including normal page-to-page interaction but also possibly uses of the back button or cloned browser windows) always leads to a set of answers that could have been the result of a single, linear traversal through a survey.

6.3 Base Topsl

We start with a model of as simple survey language as we can (though our model itself is slightly more complicated than at first appears necessary so that we can easily extend it into a multi-language system later). Figure 6.3 presents the grammar for 'base Topsl,' a language with no features other than the ability to ask a sequence of pages, each of which may contain any number of Likert-scale and free-response questions. The *s* (for "survey") nonterminal generates a sequence of survey elements (nonterminal *s-elt*), which in turn may be a page (nonterminal *p*) or another survey. Furthermore, because we want to think of base Topsl as a small-step semantics, we need a term representation for partially-completed

```
complete-survey ::= s
              s ::= (seq s-elt ...)
          s-elt ::= p | r | s | a
              p ::= (page x q ...)
              r ::= (response x (x rty ans) ...)
              a ::= (r ...)
              q ::= (? x rty q-elt ...)
            rty ::= likert | free
            ans ::= string | number
```

Figure 6.3: Base Topsl grammar

surveys as well; for this reason a survey-element may also be a page response (nonterminal *r*) corresponding to a page that has been asked and answered, and an answer sequence (nonterminal *a*) that corresponds to a sequence of such page responses. Evaluation reduces a survey down to flat sequence of answers that we interpret as a completed response to the survey.

A page is straightforward: every page has a name *x*, which we represent as an arbitrary symbol with the side constraint that no name may be used on more than one page, and a sequence of questions (nonterminal *q*). Each question has a name, a question type (nonterminal *rty*), and a sequence of elements, each of which must be a string.

Base Topsl's value terms, evaluation contexts, and reductions are shown in figure 6.4. We include several notions of values as nonterminals here that we do not yet use, but that will become important later on: the *qv* ("question value") and *q-eltv* ("question element value") nonterminals, which represent fully-evaluated, ready-to-ask questions and strings within question text, respectively. Here, they only serve to distinguish uses of nonterminals *q* and *q-elt* respectively that we consider "fully-evaluated"; by making this distinction we can extend *q* and *q-elt* in subsequent sections to introduce new ways to generate a question without also extending what constitutes a question that is ready to be printed on a web page. Also note that a survey contains two notions of a completely evaluated page: a page that is ready to be presented to a user (an element of *pv*), and a page that has actually been asked and answered and has become a set of responses (an element of *r*).

A top-level Topsl evaluation context (nonterminal *TE*) is simply a sequence evaluation context (*SeqE*); a sequence evaluation context can be a hole named *seq*, corresponding to direct evaluation of a sequence by either the [seq demotion 1] or [seq demotion 2] reduc-

$$
\begin{array}{rl}
pv & ::= (\text{page } x\ qv \ldots) \\
qv & ::= (?\ x\ rty\ q\text{-}eltv \ldots) \\
q\text{-}eltv & ::= \text{string} \\
TE & ::= SeqE \\
SeqE & ::= []_{seq} \mid (\text{seq } r \ldots PE\ s\text{-}elt \ldots) \\
PE & ::= []_{page} \mid SeqE \mid (\text{page } x\ qv \ldots QE\ q \ldots) \\
QE & ::= []_{q}
\end{array}
$$

$$
\begin{array}{ll}
TE_1[(\text{page } x_1\ (?\ x_q\ rty_q\ q\text{-}eltv_q \ldots) \ldots)]_{page} \longrightarrow & \text{[page response]} \\
TE_1[(\text{response } x_1\ (x_q\ rty_q\ \mathsf{A}[\![(x_1\ (?\ x_q\ rty_q\ q\text{-}eltv_q \ldots))]\!]\ ) \ldots)] & \\
TE_1[(\text{seq } r_1 \ldots)]_{seq} \longrightarrow & \text{[seq demotion 1]} \\
TE_1[(r_1 \ldots)] & \\
TE_1[(\text{seq } r_1 \ldots (r_2 \ldots)\ s\text{-}elt_3 \ldots)]_{seq} \longrightarrow & \text{[seq demotion 2]} \\
TE_1[(\text{seq } r_1 \ldots r_2 \ldots s\text{-}elt_3 \ldots)] &
\end{array}
$$

Figure 6.4: Base Topsl reductions

tion (explained below), or a survey to which a participant has partially responded, allowing evaluation on the left-most uncompleted page element. A page element evaluation context is either a hole labeled page, corresponding to direct evaluation of a page by the [page response] reduction (explained below) or evaluation of a page element. While the evaluation contexts further subdivide pages, allowing left-to-right evaluation of page elements by use of a hole named q, these productions are not yet useful because there are no reductions that make use of them.

The reductions themselves are very simple. Rule [page response] represents a survey participant answering a page; we model that participant as an unspecified relation A (for "answers") that maps a question and the ID of the page it came from to an answer of the appropriate result type (string for free response questions, number between 1 and 7 for Likert-type questions). It is important that A can be a general relation, rather than needing to be a function, to represent the fact that the same question can be freely answered in many different ways.

With all these rules in place, surveys do something quite simple: from left to right, the rewriting system applies the page response rule and the two demotion rules, which do nothing but flatten nested responses. These rules successively replace each page of questions with a response, flattening response sequences as it goes, until the entire survey has been rewritten into a single, flat answer sequence which we interpret as the survey's final result.

Theorem 6.3.1 (base Topsl safety)**.** *For each survey cs in base Topsl,* $cs \mapsto^* (r \ldots)$.

Proof. Similar to standard type-soundness theorems, though here preservation is trivial and only progress (*i.e.*, that every term either reduces to another term or is an element of $(r \ldots)$) needs to be established. This holds by induction on the structure of the term. □

Furthermore, static analysis of the flavor we described in section 6.2.3 is extremely simple: all pages and all questions asked during the execution of the survey must correspond to some page that syntactically appears in the source survey term.

Definition 6.3.2 (Topsl contexts)**.** An unrestricted Topsl context, written *TC*, is a Topsl term with a hole in an arbitrary position.

With this definition in place, we establish the first of what we call our "analysis theorems," theorems that give us a simple, sound static analysis technique; the analysis theorem is the critical property we need to hold for the all of the languages we introduce in order for static analysis to work. Theorem 6.3.3 is the simplest of these theorems; it establishes the entirely straightforward fact that under the semantics we have defined, if during execution the survey asks a question, then that question appeared textually in the original source code.

Theorem 6.3.3 (base Topsl analysis)**.** *For each survey cs in base Topsl, if* $cs \mapsto^* TE[(\textbf{page}\ x_p\ (\textbf{?}\ x_q\ rty\ qv \ldots)\ldots)]$ *then there exists some context TC such that* $cs = TC[(\textbf{page}\ x_p\ (\textbf{?}\ x_q\ rty\ qv \ldots)\ldots)]$.

Proof. It suffices to show the related claim that if $cs \mapsto TC[(\textbf{page}\ x_p\ (\textbf{?}\ x_q\ rty\ qv \ldots)\ldots)]$ for any *TC* then there exists some context TC' such that $cs = TC'[(\textbf{page}\ x_p\ (\textbf{?}\ x_q\ rty\ qv \ldots)\ldots)]$. This holds by inspection of the reduction rules. □

Because this theorem holds, we can implement static analysis for a given base Topsl survey very simply by just reading off every page that appears syntactically within it.

complete-survey ::= ((survey: *s*) (answers: *a*))
TE ::= ((survey: *SeqE*) (answers: *a*))

((survey: PE_1[(page x_1 (? x_q rty_q $q\text{-}eltv_q$ ...) ...)]$_{page}$) ⟶ [page response]
(answers: (r_1 ...)))

((survey: PE_1[(response x_1 answer ...)])
(answers: (r_1 ... (response x_1 answer ...))))
where (answer ...) = (term A⟦(page x_1 (? x_q rty_q $q\text{-}eltv_q$ ...) ...)⟧)

TE_1[(seq r_1 ...)]$_{seq}$ ⟶ [seq demotion 1]
TE_1[(r_1 ...)]

TE_1[(seq r_1 ... (r_2 ...) $s\text{-}elt_3$ ...)]$_{seq}$ ⟶ [seq demotion 2]
TE_1[(seq r_1 ... r_2 ... $s\text{-}elt_3$ ...)]

Figure 6.5: Figures 6.3 and 6.4 modified for imperative Topsl

6.4 Imperative Tospl

At this point, we can understand the final design we settled on in section 6.2 as an attempt to scale the language to more complex systems by adding new forms to base Topsl — a branch form, a binding form, and so on. We take this as our departure point from that solution. Instead, we will explore the possibility of making Topsl as small as possible, letting programmers reuse Scheme's flow control and binding constructs instead of building new, redundant ones. By taking a semantic approach, we can (for the moment) ignore the problems we described in the obvious implementation strategy. We proceed in two steps. First, we define imperative Topsl, a variant on base Topsl that stores page results in a global store in addition to returning them as values. Second, we use the technical devices introduced by that refinement to make a convenient Scheme interface; with the Scheme interface in place we can use its native control-flow and binding constructs rather than needing to implement special-purpose forms for Topsl.

Figure 6.5 shows how we extend base Topsl to form imperative Topsl. Only two nonterminals need to change. First, a complete survey must now consist of two parts: a program, and an accumulated store of results with the invariant that all pages that have been asked and answered during a survey's execution have their results contained in this store. Second, the top-level evaluation context must now allow for the store as well.

The changes to base Topsl's reduction rules are similarly small. The only rule that

differs from its base Topsl counterpart is [page response], which now puts the participants answers in the survey's top-level results section in addition to making it the result of a page's evaluation. (One might reasonably wonder at this point why we reduce a page to any meaningful value, rather than void or a unit value, since the results are stored in a global store anyway. More on this shortly.)

Those changes complete imperative Topsl. We can establish a straightforward soundness property for it: for any program, all responses that occur during a run are included in the store and all inclusions in the store (except those that were already in the store at the beginning of the program's execution) arise because they were responses to questions asked at some point. More formally, if a survey term reduces to a term that represents a page being asked and answered, then any further term to which it reduces will contain the supplied answer in its store, and similarly every answer contained in the store is the result of the survey asking a question that yielded that answer at some point.

Theorem 6.4.1 (Imperative Topsl consistency)**.** *For any Topsl program plus store* $cs = ((\mathsf{survey}{:}\, s)\ (\mathsf{answers}{:}\, a))$ *and any* $r \notin a$, $cs \mapsto^* TE[pv_i] \mapsto TE'[r] \mapsto^* ((\mathsf{survey}{:}\, s')\ (\mathsf{answers}{:}\, a'))$ *iff* $a' \supseteq a + r$.

Proof. It suffices to prove the related claim that for all s, a, s', and a', $((\mathsf{survey}{:}\, s)\ (\mathsf{answers}{:}\, a)) \mapsto ((\mathsf{survey}{:}\, s')\ (\mathsf{answers}{:}\, a'))$ holds if and only if either:

1. $a = a'$ and there do not exist any qv and r such that $s = TE[qv]$ and $s' = TE[r]$, and $a' = a, r$; or

2. there exists some qv and r such that $s = TE[qv]$, $s' = TE[r]$, and $a' = a, r$.

This claim holds by inspection of the cases of the reduction relation. Note that the restriction in the theorem statement that $r \notin a$ is necessary for the "only if" part of the theorem, because if the desired response is already in the response set in the initial term, then it will also appear in the final term even if it is never asked during the rest of the survey's execution. □

We also establish another analysis theorem. Again it says that any question that is asked during any execution of a survey appeared textually in the survey's source.

Theorem 6.4.2 (Imperative Topsl analysis)**.** *For each complete survey cs in imperative Topsl, if* $cs \mapsto^* TE[(\textbf{page}\ x_p\ (\textbf{?}\ x_q\ rty\ qv \ldots)\ldots)]$ *then there exists some context TC such that* $cs = TC[(\textbf{page}\ x_p\ (\textbf{?}\ x_q\ rty\ qv \ldots)\ldots)]$.

Proof. As the proof of theorem 6.3.3. □

6.5 Topsl + Scheme

At this point we can easily add Scheme constructs by introducing Scheme as a separate language that interacts with imperative Topsl through boundaries. (It is important that we use imperative Topsl rather than base Topsl for the embedding, because otherwise any Scheme fragment that ran embedded Topsl code but did not return the answers produced would lose data.) First we define a model for a simple Scheme-like language; it is presented on its own in figure 6.6. The model we have chosen is just the call-by-value lambda calculus extended with booleans, numbers, strings, and cons cells as base values and `if` and `eq?` as new syntactic forms. Its grammar, evalation contexts, and reductions are standard with the exception that we use a hole named `S` for holes in which Scheme reductions may occur, and we use a *TE* top-level evaluation context rather than an *SE* evaluation context in the definitions of our reduction rules.

With Scheme defined, we can consider how to define appropriate boundaries between it and imperative Topsl. Figure 6.7 shows extensions to the grammars of our models of Scheme and imperative Topsl to add cross-language boundaries, and figure 6.8 shows the new reduction rules. There are two essential extensions: a survey element may now be a `TS` ("Topsl-to-Scheme") boundary, which allows Topsl to run a Scheme expression that can perform arbitrary computation, and an `ST` ("Scheme-to-Topsl") boundary that allows Scheme to call back into Topsl to ask further questions.

`TS` boundaries may appear in two different places in Topsl code: first, they may be survey elements, in which case the Scheme value produced will be converted to an answer by one of the [`ST ans`] rules. Since all answers are recorded whenever they are submitted, it is not particularly important for a Scheme fragment that asks questions to gather up all of its answers and return them here. In fact, generally speaking Scheme code will not return any

e ::= (e e) | x | (if e e e) | (eq? e e)
| (λ (x) e) | number | string | #t | #f | null | (cons e e) | car | cdr
v ::= (λ (x) e) | number | string | #t | #f | null | (cons v v) | car | cdr
SE ::= $[]_S$ | (SE e) | (v SE) | (eq? SE e)
| (eq? v SE) | (if SE e e) | (cons SE e) | (cons v SE)

$TE_1[((\lambda\ (x_1)\ e_1)\ v_1)]_S \longrightarrow$ [beta]
TE_1[subst[[(x_1 v_1 e_1)]]]

$TE_1[(\text{if \#f } e_1\ e_2)]_S \longrightarrow$ [if #f]
$TE_1[e_2]$

$TE_1[(\text{if } v_1\ e_1\ e_2)]_S \longrightarrow$ [if #t]
$TE_1[e_1]$
where $v_1 \neq$ #f

$TE_1[(\text{eq? } v_1\ v_1)]_S \longrightarrow$ [eq? #t]
TE_1[#t]

$TE_1[(\text{eq? } v_1\ v_2)]_S \longrightarrow$ [eq? #f]
TE_1[#f]
where $v_1 \neq v_2$

$TE_1[(\text{car (cons } v_1\ v_2))]_S \longrightarrow$ [car]
$TE_1[v_1]$

$TE_1[(\text{cdr (cons } v_1\ v_2))]_S \longrightarrow$ [cdr]
$TE_1[v_2]$

Figure 6.6: Simple Scheme model

particularly useful answer and the default empty answer sequence produced by rule [ST ans 2] (which converts any non-answer value that Scheme produces to an empty answer sequence) will appear here, though if Scheme wants to return a particular answer sequence it may do so using rule [TS ans 1] (which converts Scheme answer values to their Topsl representations). Second, they appear as question elements (where in base and imperative Topsl only strings were allowed), in which case Scheme's result will be converted to a string by rule [TS question]. This feature allows us to make questions whose text varies programmatically based on runtime input, but that are still amenable to static analysis in the sense that we can separate out the parts of a question that never change from those that depend on computation done while a particular participant is taking the survey. (We will revisit other kind of programmatically-generated questions we identified in section 6.2.2, questions that are programmatically generated purely to take advantage of abstraction to factor out common patterns, in section 6.7.) Notice here that since we were careful to make a distinction between question elements and question element values and between questions and question values in base Topsl, we do not have to rework those definitions here. Instead

s ::= .... | (TS Ans *e*)
q-elt ::= .... | (TS str *e*)
SeqE ::= .... | (TS Ans *SE*)
QEltE ::= .... | (TS str *SE*)

e ::= .... | (ST τ *s*)
v ::= .... | (ST Ans *a*)
τ ::= (from *x x*)

| Ans

SeqE ::= .... | (TS Ans *SE*)
QEltE ::= .... | (TS str *SE*)
SE ::= .... | (ST τ *SeqE*)

Figure 6.7: Extensions to figures 6.5 and 6.6 to form Topsl + Scheme grammar

we can simply extend what constitutes a question without extending what constitutes a question value, and the rest of the system including the reduction rules we have already written will be able to properly distinguish between questions with unevaluated portions and those that are fully evaluated.

ST boundaries can appear in Scheme as normal expressions, each of which is intended to evaluate the survey element to an answer. They may be annotated with either of two different conversion strategies (identified by the τ nonterminal), either (from x_p x_a) or Ans. A boundary with conversion strategy (from x_p x_a) evaluates the embedded Topsl program to an answer sequence, then finds the answer to the question named x_a on the page named x_p in that answer sequence and returns the answer to the question named x_a on that page as a string or number as appropriate to the question. (It is an error to select the answer from a page if that page is represented multiple times in the same answer sequence. This is not a particularly odious restriction sinceout of an answer A boundary with conversion strategy Ans also evaluates its embedded survey element to an answer sequence, but then simply holds it in its entirety as a Scheme value. These two conversion strategies work together to allow a Scheme program to extract multiple values from the same page: for instance (relying on **let** and *max*, two very straightforward extensions to our Scheme model), the Scheme fragment

(**let** ((*a* (**ST** (**page** *p* (**?** likert q_1 ...) (**?** likert q_2 ...)))))

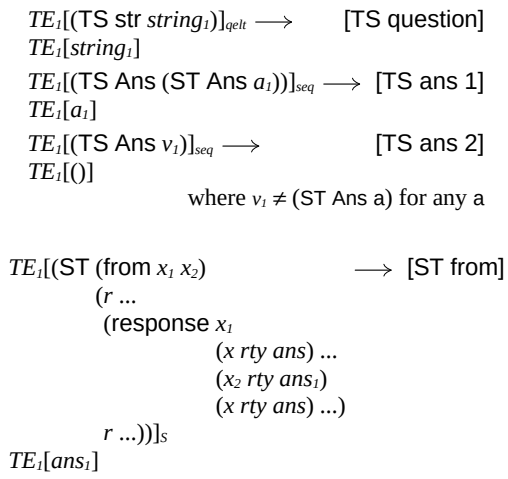

Figure 6.8: Topsl + Scheme boundary reduction rules

(*max* (**ST** (**from** *p* q_1) (**TS Ans** *a*))
(**ST** (**from** *p* q_1) (**TS Ans** *a*))))

extracts the answers to both q_1 and q_2 from the same answer sequence and computes their maximum. More generally, in this model we can consider the **bind** mechanism we discussed in section 6.2 as syntactic sugar: (**bind** (([x_p x_q] ...) *s*) *e*) becomes

(**let** ((*ans* (**ST Ans** *s*)))
(**let** ((x_q (**ST** (**from** x_p x_q) (**TS Ans** *ans*))) ...)
e))

Many other idioms are conveniently expressible in this language as well. For instance, a branching survey can be written using Scheme's **if**:

(**let** ((*pivot* (**ST** (**from** x_p *xpivot*) *s*)))
(**if** (*eq? pivot* [some value])
s1
s2))

A survey that iterates over arbitrary input is also easy to build:

(**let** ((*strings* [arbitrary Scheme computation]))
(*for-each*

(λ (*s*) (**ST Ans** (**page** *P* (**?** free "question about " (**TS** *str s*)))))
strings))

In fact, at this point the system is sufficiently powerful to express the entire survey suggested by figure 6.1. Figure 6.9 is a sketch of how it would be implemented.

Since we record the answers to every question on every page in a global register, we could have designed a different semantics for *from* that drew from that global store and ignored the answer sequence provided by Topsl altogether. We chose this design instead because we like to think of Topsl programs as essentially functional programs that have an imperative backup system that programmers trust to save answers but otherwise ignore. In practical terms, we expect that because of this decision a Topsl programmer will not have to reason about the global state of a survey ("Am I guaranteed that page p1 has been asked at this point?") which depends on whole-program reasoning, but only about its local value flow ("Must the expression inside this boundary evaluate to an answer sequence that includes page p1?") which is a local property.

Owing to the unrestricted computation Scheme allows, this combined language is powerful enough to let us author surveys that loop and have interesting runtime behavior without giving up on the guarantees we want. In fact the consistency theorem does not need to be changed at all:

Theorem 6.5.1 (Topsl + Scheme consistency)**.** *For any Topsl + Scheme program* $cs = ((\mathsf{survey}: s)\ (\mathsf{answers}: a))$ *where* $r \notin a$, $cs \mapsto^* TE[pv_i] \mapsto TE'[r] \mapsto^* ((\mathsf{survey}: s')\ (\mathsf{answers}: a'))$ *iff* $a' \supseteq a + r$.

Proof. As the proof of theorem 6.4.1. □

The analysis theorem needs to be weakened very slightly, because Topsl questions may now contain Scheme code that evaluates to an arbitrary string at runtime rather than appearing literally in the source text. So, instead of demanding that any question asked a runtime appear literally in the source text, we only demand that some source text question *describes* any question that is asked at runtime.

Definition 6.5.2.

- (**page** x_p q_1 ... q_n) describes (**page** $x_p{}'$ q_1' ... q_m) if $x_p = x'_p$, $n = m$, and for each $i \in \{1 \ldots n\}$, q_i describes q'_i.

- (**?** x_q *rty* *q-elt*$_1$ ... *q-elt*$_n$) describes (**?** $x_q{}'$ *rty*$'$ *q-elt*$'_1$ ... *q-elt*$'_m$) if $x_q = x'_q$, *rty* = *rty*$'$, $n = m$, and for each $i \in \{1 \ldots n\}$, if $q - elt_i = str$ for any string *str*, then $q - elt'_i = str$.

Theorem 6.5.3 (Topsl + Scheme analysis)**.** *For each complete survey cs in imperative Topsl, if cs* $\mapsto^*$ *TE*[*pv*] *then there exists some context TC such that cs* = *TC*[*p*] *and p describes pv.*

Proof. As the proof of theorem 6.3.3. □

6.6 Web Topsl

Up to this point, we have only been dealing with sequential surveys. This is a reasonable model for surveys that take place in person or over the phone. It is not, however, a good model of our intended application domain, that of surveys conducted over the web, since web users may use the back button or clone browser windows and submit answers multiple times to the same page of questions. These features are of particular concern to us because they correspond to re-entering a continuation, and since our Topsl + Scheme model makes important use of state we are obligated to extend it in such a way that submitting a page twice does not violate our assumptions about that state.

To do so, we borrow aspects of Krishnamurthi *et al*'s webL model of client/server web interaction (Graunke et al. 2003; Krishnamurthi et al. 2006), adapting it and paring it down as we go to focus on the aspects of web interaction that concern us here. In particular, webL models the client as well as the server; since we are not interested in the client we replace it with a rule that nondeterministically chooses to submit answers to any survey page it has seen.

Concretely, we extend a complete survey so that in addition to a survey program and a list of answers already recorded, it also holds a set of resumption points. Conceptually, a resumption point is a "save point" in a survey; it contains the entire state of the survey saved at the moment the survey sent a particular page of questions to the participant, which is necessary so that we can start the survey back up exactly where it left off when (and

if) the participant submits answers to that page. Each resumption point (nonterminal *rp*) contains three items: the first, *pv*, represents the page value sent to the participant. The second, *s*, is the code representing the continuation to run should the user submit answers. The third, *a*, is a record of the current value of answers: at the point when the survey sent this page to the participant. To understand its role, it will be helpful for us to look at web Topsl's reduction rules, which appear in figure 6.11.

Web Topsl adds two new reduction rules beyond the ones we have seen before. Essentially they break the process of asking and answering a question, which have both been contained in a single rule in prior models, into two reductions: [page to web] for asking questions, and [web to page] for answering them. The first, [page to web], applies as before whenever a page appears in an evaluation context. However, instead of rewriting directly to an answer, it rewrites to a new configuration in which there is no survey or current sequence of answers, only the set of resumption points. Furthermore, to that set of resumption points it adds a new one corresponding to the page just asked; the new point has the page to ask, the evaluation context in which the page appeared (which corresponds to its continuation (Felleisen and Hieb 1992)). No reduction rules apply to this configuration except [web to page].

The [web to page] rule selects a resumption point arbitrarily, chooses some arbitrary set of answers to the questions on the page it represents, and then starts the survey running again, restoring the suspended evaluation context and replacing its hole with those answers. Furthermore, it sets answers: to be the answers sequence stored by the selected resumption point. By following this procedure, the reduction rules maintain the invariant that the answers sequence in answers: always corresponds to some sequential path through the program, or more precisely, to a state that could be reached by an interaction with the same survey run with sequential Topsl + Scheme semantics.

Definition 6.6.1.

$$s_1 \mapsto_{st} s_2 \quad \stackrel{\text{def}}{=} \quad s_1 \mapsto s_2 \text{ under the rules of section 6.5}$$
$$s_1 \mapsto_{wt} s_2 \quad \stackrel{\text{def}}{=} \quad s_1 \mapsto s_2 \text{ under the rules of section 6.6}$$

Theorem 6.6.2 (web Topsl coherence)**.** *For any survey element s, if*

$$\begin{array}{rl} & ((\textsf{survey: } s)\ (\textsf{answers: } ())\ (\textsf{resumption-points: } ())) \\ \mapsto_{wt}^{*} & ((\textsf{survey: } s')\ (\textsf{answers: } a)\ (\textsf{resumption-points: } (rp \ldots))) \end{array}$$

then there exists some s'' such that

$$\begin{array}{rl} & ((\textsf{survey: } s)\ (\textsf{answers: } ())) \\ \mapsto_{ts}^{*} & ((\textsf{survey: } s'')\ (\textsf{answers: } a)) \end{array}$$

This theorem's proof is slightly more complicated than the proofs of prior theorems. First, we observe that all web Topsl reduction sequences that do not involve the [page to web] or [web to page] reduction rules correspond to reduction sequences in Topsl + Scheme:

Lemma 6.6.3. *If*

$$\begin{array}{rl} & ((\textsf{survey: } s)\ (\textsf{answers: } a)\ (\textsf{resumption-points: } (rp\ \ldots)) \\ \mapsto_{wt}^{*} & ((\textsf{survey: } s')\ (\textsf{answers: } a')\ (\textsf{resumption-points: } (rp \ldots))) \end{array}$$

and no reduction used in the reduction sequence is generated by [page to web] or [web to page], then $((\textsf{survey: } s)\ (\textsf{answers: } a)) \mapsto_{ts}^{*} ((\textsf{survey: } s')\ (\textsf{answers: } a'))$.

Proof. By induction on the length of the reduction sequence. The base case is immediate; for the inductive case, observe that each rule $\mapsto_{wt}$ other than the two excluded rules is also present in $\mapsto_{ts}$. □

Using this lemma we can establish that any resumption point contains an answer sequence and a continuation that could be found on a valid Topsl + Scheme reduction sequence.

Lemma 6.6.4. *Suppose*

$$\begin{array}{rl} & ((\textsf{survey: } s)\ (\textsf{answers: } ())\ (\textsf{resumption-points: } ()) \\ \mapsto_{wt}^{*} & ((\textsf{survey: } s')\ (\textsf{answers: } a)\ (\textsf{resumption-points: } (rp_1 \ldots rp_n))) \end{array}$$

Then for each $i \in \{1 \ldots n\}$, *if* $rp_i = (pv_i\ s_i\ a_i)$ *then*

$$\begin{array}{rl} & ((\textit{survey:}\ s)\ (\textit{answers:}\ ())) \\ \mapsto_{ts}^{*} & ((\textit{survey:}\ TE_i[pv_i])\ (\textit{answers:}\ a_i)) \end{array}$$

where $TE_i[(\textbf{TS Ans}\ x)] = s_i$.

Proof. Observe that since each reduction using rule [page to web] increases the number of resumption points in the subsequent term by exactly one and no other rules modify resumption points, the number of resumption points in any term in the sequence is equal to the number of uses of the rule [page to web] in the subsequence that precedes it. Now we can use induction on the number of such uses in the reduction sequence. If it is zero, the theorem trivially holds. Otherwise, consider subsequence that ends with the last use of [page to web] and begins with the rightmost use of [web to page] that precedes it (or the beginning of the entire sequence if no use of [web to page] precedes it). By induction, that use of [web to page] restores some resumption point — say, $(pv_i\ s_i\ a_i)$ — that corresponds to a sequential trace. Furthermore, by inspection the contents of the survey: and answers: fields in the term produced by rule [web to page] correspond to a step from $((\text{survey: } TE_i[pv_i])\ (\text{answers: } a_i))$ using the Topsl + Scheme rule [page response].

Now by appealing to lemma 6.6.3 and straightforward fact that Topsl + Scheme reduction sequences compose, we have that the term immediately to the left of the last use of the rule [page to web] corresponds to an Topsl + Scheme sequence. Thus the resumption point [web to page] adds has the desired property; by induction all the others do as well, so the lemma holds. □

Now the proof of the main theorem is straightforward:

Proof. Consider the last use of rule [page to web] in the reduction sequence. By lemma 6.6.4, that use results in a term that corresponds to a legal Topsl + Scheme reduction sequence. Then the theorem holds by application of lemma 6.6.3 to that term and the final term in the sequence. □

Theorem 6.6.5 (web Topsl analysis)**.** *For each complete survey cs in web Topsl, if* $cs \mapsto^{*} TE[pv]$ *then there exists some context TC such that* $cs = TC[p]$ *and p describes pv.*

Proof. As the proof of theorem 6.3.3. □

6.7 User + Core Topsl

At this point we have established some reasonable safety properties for a language that seems reasonably close to what we had in mind from the outset. The one feature on our desiderata that we have not yet modeled is the ability to programmatically generate questions purely so that we can abstract out repeated question patterns. In fact we will not do so at all. The reason is that by using Scheme as our host language and decomposing the problem as we have in the previous sections, we have made such a feature unnecessary.

As the analysis theorems have shown, it is sound for our static analysis to consist of simply reading out every page that syntactically occurs in the initial web Topsl source term. However, since we are embedding our survey language in Scheme and thus have access to its rich macro system, we are free to use them to programmatically compute the Topsl source code itself at compile time. The analysis theorems along with our ability to compute dynamic text for questions using boundaries draws a bright line: questions generated by abstractions for convenience that should appear separately in static analysis must be computed at compile time using macros, and questions generated by abstractions in order to account for information acquired at run time must be generated by boundaries.

We call this the line between user Topsl and core Topsl. User Topsl is version of Topsl that a programmer will actually interact with, and it allows arbitrary computation in generating a survey to allow the programmer to abstract commonalities in questions or to do anything else he or she wants to do; for instance to automatically generate unique page and question names for the dozens or hundreds of pages and questions that do not play any special role in computation and thus whose names are a nuisance to have to write by hand. When a user Topsl program is executed, it does not go to the web or do any interaction with a user; instead, it produces a core Topsl program. A core Topsl program is a Topsl program as in the previous sections; all of its static computation has already been performed and it can be statically analyzed or executed to interact with a survey participant.

For instance, a survey-writer might want abstract out common pattern in asking questions while still having all the computed questions appear in the survey's static summary.

For this purpose, regular Scheme functions are unsuitable: for one thing, it is not grammatically possible in the language we have defined for questions to appear outside the lexical context of a page, and for another, even if it were possible those questions would not be able to appear in the survey's static summary, since they would not be computed until after the static summary was already final. So instead, the programmer can compute them before runtime the same way Scheme programmers always compute things before runtime, using macros. Here, for instance, is how a programmer could abstract a series of two related questions:

```
(define-syntax related-questions
  (syntax-rules ()
    [(related-questions (id1 id2) shared-text)
     (begin
       (? free id1 shared-text)
       (? free id2 shared-text " ... part 2"))]))
```

This macro can be used like any Scheme macro:

```
(page P
  ...
  (related-questions (q1 q2) "shared text 1")
  (related-questions (q3 q4) "shared text 2")
  ...)
```

As this example shows, our method allows Topsl to usefully reuse Scheme's macro facilities for survey construction in the same way it connects Scheme control and binding mechanisms to intuitively-similar survey tasks. This is not automatic, and in fact it did not hold of our previous design: just as new Scheme binding and control constructs could not be used to extend Topsl survey binding and control, Scheme macros had no particularly useful interaction with static analysis.

6.8 Using PLT Redex

We have implemented all of the term-rewriting systems discussed in the preceeding sections as PLT Redex programs. This presents an unusual challenge, since PLT Redex is designed to be good at nondeterministically matching input terms against patterns and infinitely-long chains of reductions, but not to be good at handling rules that nondeterministically select different outputs based on the same input or reductions in which a term reduces in a single step to infinitely many successors.

Our semantics depend on both kinds of nondeterminism. They use the former to represent a survey participant's ability to go back to any previously-asked web page and resubmit answers (see the ambiguous evaluation-context grammar in the web-to-page rule in figure 6.11), and the latter to represent the participant's ability to answer any question with an arbitrary response (via the unspecified A relation). For that reason, we have to make a compromise to implement our semantics as Redex programs by giving a concrete, function representation for A.

Here we have a few choices. The first and easiest option is to make A a constant function that maps every question to a constant dummy answer. This is actually more effective than it might at first seem, since when we design test cases we can arrange to test all of the rules of the operational semantics by knowing what the dummy variables are, and most automated tests that me might want to perform do not depend critically on particular answer values anyway.

The second option is a variant on the first: rather than using a single constant function, design a test harness that reruns the same test case with many different functions for A in successive runs. By changing A itself rather working around it behaving like a constant, we can directly test programs that do interesting things based on the answers to a question. It is also useful to have a function that returns two different answers to the same question during the same test, for instance to test the web coherence property (theorem 6.6.2). Since PLT Redex is embedded in PLT Scheme we are not really limited to pure mathematical functions for our implementation of A — we can use any Scheme function we like — so we can use state in conjunction with this strategy to make functions that behave this way.

The final option we have explored is to exploit the fact that PLT Redex is embedded in PLT Scheme further by having the A function present the user with a page of questions take his or her answers as its result. Since this option requires user supervision it is not a good choice for automatic testing, but it works well as an interactive frontend for experimenting. In fact, by using the PLT web server we can even have the user interact with Redex through a web browser, making our model of Topsl into a makeshift implementation, providing an excellent sense of how the language might work in practice.

6.9 Implementation

After languishing for a year and a half, our Topsl implementation effort has been revitalized due to the semantic exploration of the last section, which we conducted by building and refining PLT Redex models. The analysis in section 6.7 indicates our overall compilation strategy, which we have summarized in figure 6.12: a user Topsl program is a module that compiles down to a core Topsl program, which in turn compiles to a PLT Scheme web server-compatible servlet that additionally provides static analysis information.

Static analysis

We perform the static analysis when transforming a core Topsl program into a servlet. A sound and very simple static analysis technique is to record the compilation of each **page** and **?** form that appears in a survey, generating a list of all pages and questions that textually appear anywhere. Because of the way we have built the language, and in particular because it satisfies the static analysis theorems (theorems 6.3.3, 6.4.2, 6.5.3, and 6.6.5), we know this method is sound.

Imperative state

The implementation-analogue of our models' answers: sections is less obvious. Here again, though, formal methods indicate a simple solution: as long as a survey saves the current answers sequence with the continuation whenever it asks the participant to respond

to a page, restores that saved answers sequence when it returns, the current answers sequence will remain consistent with sequential ordering. So the following implementation sketch has the proper behavior, which we hope the reader will appreciate our calling the "caller-save strategy" in analogy to the assembly-level function calling convention it resembles:

```
(define *current-answers* '())
(define (ask-page p)
  (let ((saved-current-answers (copy-list *current-answers*)))
    (let ((results (request→results (send/suspend (html-derived-from-page p)))))
      (begin
        (set! *current-answers* (cons results saved-current-answers))
        results))))
```

In fact at one point we used this strategy. However recent versions of the PLT Scheme web server have introduced a more general facility called web cells (McCarthy and Krishnamurthi 2006) that serve our purpose; the current implementation uses those instead.

Our implementation effort is still ongoing, particularly with regard to user Topsl. However, preliminary results are very encouraging: when we rewrote our existing implementation using the new design, we were able to reduce the total size of the implementation by a third (1410 lines versus 2198) while continuing to be able to run a large, complicated survey that had been developed and deployed for use in a psychologist's research project.

6.10 Related work

There are a considerable number of mechanisms for creating on-line surveys apart from implementing them in a general-purpose language. Two domain-specific languages, SuML and QPL, stand out as being the closest to the goals the authors set for Topsl.

SuML is an XML/Perl-based survey language in which the programmer describes a survey in an XML document which follows the SuML Schema. The SuML Schema has a *question* element which contains question text and a sequence of allowable responses, much like Topsl. The root *survey* element contains any number of questions and a *routing*

element that describes control flow. The *routing* element contains any number of **if** and *ask* elements which are composed to ask questions in the survey and branch on their responses.

The programmer creates two files in addition to the content of the survey: an XSLT stylesheet and a front-end Perl CGI program. The style sheet is responsible for describing what a survey will look like when presented on the web to a participant, and multiple stylesheets can be written for different mediums. The front-end is a Perl CGI program that acts as the entry point to the survey.

SuML's most significant problem for our purposes is that owing to its need for static analysis, its notion of control-flow is very limited, providing its users with only an **if** statement with which to branch to different parts of the survey. Furthermore, the test position of the **if** is written in a domain-specific language that only allows access to the results of the current survey execution, which makes it unsuitable for longitudinal studies or other more exotic study designs.

In addition, SuML is somewhat too generic for our purposes. The user-written Perl CGI is in charge of driving the survey by using SuML's Perl API to get the next questions to be asked and then present them as well as storing the results of the questions asked. Putting the burden on the programmer makes survey development more difficult, time-consuming, and error-prone.

QPL is another domain-specific language for creating surveys that offers static analysis by restricting what programmers can express, and as a result it suffers from problems very similar to SuML's. QPL's semantics are reminiscent of BASIC: it is an imperative language using **if** and **goto** for control flow along with a large set of built-in predicates used for conditional testing. Current distributions provides users with a large set of comparison functions for use with **if**; however, it lacks an means of growing to meet programmers' changing needs.

Finally, Queinnec's library for quizzes (Queinnec 2002) is similar in spirit to Topsl in that it provides a library of combinators for generating questionnaires in a high-level way, with implementation based on a continuation-based web server. Queinnec's library is quite a bit lighter-weight than Topsl and offers neither Topsl's safety guarantees nor its static analysis, but a design close to Queinnec's is probably appropriate for some lighter-weight social science surveys.

```
;; gets the initials of the person the given participant reported seeing last week, or #f if none
(define (get-last-partner-inits-from-db participant) ...)

;; gets the initials of all previous partners of the given participant
(define (get-all-partners name) ...)

;; gets the initials of the person the given participant reported seeing last week, or #f if none
(define (get-last-partner-inits-from-db participant) ...)

;; gets the initials of all previous partners of the given participant
(define (get-all-partners name) ...)
                (define (general-set)
                  (ST Ans
                    (page Set1
                      (? free name "Your name?") ...
                      (? yes/no seeing-anyone? "Currently seeing anyone?")
                      (? free partner-inits "Current partner's intials?") ...)))
                  (define (no-breakup-set) (ST Ans ...))
                  (define (breakup-set) (ST Ans ...))
                  (define (new-relationship-set) (ST Ans ...))
                  (define (ex-relationship name) (ST Ans ...))
          (bind ([(Set1 name)
                  (Set1 seeing-anyone?)
                  (Set1 partner-inits)]
                 (general-set))
            (let ((old-partner-inits (get-last-partner-inits-from-db name)))
              (cond
                [(and seeing-anyone? (eq? old-partner-inits partner-inits))
                 (no-breakup-set)]
                [(and seeing-anyone? (not (eq? old-partner-inits partner-inits)))
                 (begin
                   (breakup-set)
                   (new-relationship-set))]
                [(and (not seeing-anyone?) old-partner-inits)
                 (breakup-set)]
                [(and (not seeing-anyone?) (not old-partner-inits))
                 (void)]))
            (for-each ex-relationship (get-all-partners name)))
```

Figure 6.9: Sketch implementation of the survey described in figure 6.1

complete-survey ::= ((survey: *s*)
(answers: *a*)
(resumption-points: (*rp* ...)))
rp ::= (*pv s a*)
TE ::= ((survey: *SeqE*)
(answers: *a*)
(resumption-points: (*rp* ...)))

Figure 6.10: Modifications to figure 6.7 to form web Topsl grammar

((survey: PE_1[(page x_1 (? x_q rty_q $q\text{-}eltv_q$ ...) ...)]$_{page}$) ⟶ [page to web]
(answers: (r_1 ...))
(resumption-points: (rp_1 ...)))
(resumption-points: (rp_1 ...
((page x_1 (? x_q rty_q $q\text{-}eltv_q$ ...) ...)
PE_1[(TS Ans x)]
(r_1 ...))))
where x fresh
(resumption-points: (rp_1 ... ⟶ [web to page]
((page x_i (? x_q rty_q $q\text{-}eltv_q$ ...) ...)
PE_1[(TS Ans x_k)]$_{page}$
(r_{i1} ...))
rp_{i+1} ...))
((survey: PE_1[(response x_i (x_q rty_q A⟦(x_i (? x_q rty_q $q\text{-}eltv_q$ ...))⟧) ...)])
(answers: (r_{i1} ... (response x_i (x_q rty_q A⟦(x_i (? x_q rty_q $q\text{-}eltv_q$ ...))⟧) ...)))
(resumption-points: (rp_1 ...
((page x_i (? x_q rty_q $q\text{-}eltv_q$ ...) ...)
PE_1[(TS Ans x_k)]
(r_{i1} ...))
rp_{i+1} ...)))

Figure 6.11: Web Topsl reductions

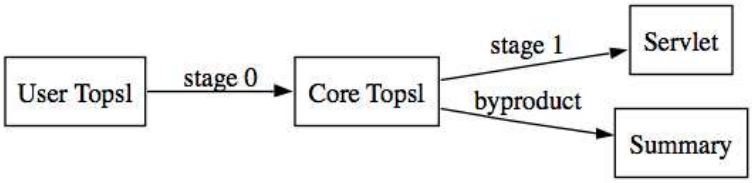

Figure 6.12: Macro-expansion stages for combining static analysis with question abstraction

REFERENCES

Martín Abadi. Protection in programming-language translations. In *International Conference on Automata, Languages and Programming*, 1998.

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991a. URL `citeseer.ist.psu.edu/abadi89dynamic.html`.

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991b. URL `citeseer.ist.psu.edu/abadi89dynamic.html`.

Hal Abelson and Gerald Jay Sussman. LISP: A language for stratified design. *BYTE*, 13(2):207–218, 1988. URL `http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-986.pdf`. Appeared in May 1987 as MIT AI Lab Memo AIM-986.

Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, pages 69–83, 2006. URL `http://ttic.uchicago.edu/~amal/papers/lr-recquant.pdf`. Extended version appears as Harvard University Technical Report TR-01-06, available online as `http://ttic.uchicago.edu/~amal/papers/lr-recquant-techrpt.pdf`.

Joshua Auerbach and Mark Chu-Carroll. The Mockingbird system: A compiler-based approach to maximally interoperable distributed programming. Technical Report RC 20718, IBM Research Division, February 1997.

Joshua Auerbach, Charles Barton, Mark Chu-Carroll, and Mukund Raghavachari. Mockingbird: Flexible stub compilation from pairs of declarations. Technical Report RC 21309, IBM Research Division, September 1998.

John Backus. The history of FORTRAN I, II, and III. In *Proceedings of the first ACM SIGPLAN Conference on the History of Programming Languages (HOPL)*, 1978. URL `http://doi.acm.org/10.1145/800025.808380`.

Ilya Bagrak and Olin Shivers. trx: regular-tree expressions, now in Scheme. In *Workshop on Scheme and Functional Programming*, 2004.

Henry G. Baker. Pragmatic parsing in Common Lisp; or, putting `defmacro` on steroids. *SIGPLAN Lisp Pointers*, 4(2):3–15, 1991. ISSN 1045-3563. doi: http://doi.acm.org/10.1145/121983.121984.

Barclay, Lober, Huq, Dockery, and Karras. SuML: A survey markup language for generalized survey encoding. In *AMIA Annual Symposium*, 2002. URL \url{http://cirg.washington.edu/public/cirg/suml.php}.

Daniel J. Barrett. *Polylingual Systems: an approach to seamless interoperability*. PhD thesis, University of Massachusetts Amherst, February 1998.

Daniel J. Barrett, Alan Kaplan, and Jack C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *Proceedings of the Fourth Symposium on the Foundations of Software Engineering (FOSE)*, 1996. URL http://portal.acm.org/citation.cfm?id=239098.239123.

Joel F. Bartlett. Scheme→C: a portable Scheme-to-C compiler. Technical Report 89/1, Digital Equipment Corporation Western Research Laboratory, January 1989. URL http://citeseer.ist.psu.edu/bartlett89schemec.html.

Eli Barzilay and John Clements. Laziness without all the hard work. In *Workshop on Functional and Declarative Programming in Education (FDPE)*, 2005.

Eli Barzilay and Dmitry Orlovsky. Foreign interface for PLT Scheme. In *Workshop on Scheme and Functional Programming*, 2004.

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

David Beazley. Pointers, constraints, and typemaps, 1997. URL http://www.swig.org/Doc1.1/HTML/Typemaps.html. In SWIG 1.1 Users Manual.

Nick Benton. Embedded interpreters. *Journal of Functional Programming*, 15:503–542, July 2005.

Nick Benton and Andrew Kennedy. Interlanguage working without tears: Blending SML with Java. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 126–137, 1999. URL citeseer.ist.psu.edu/benton99interlanguage.html.

Nick Benton, Andrew Kennedy, and Claudio V. Russo. Adventures in interoperability: the SML.NET experience. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 215–226, 2004.

Brian N. Bershad, Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

Michael Birnbaum, editor. *Psychological Experiments on the Internet*. Academic Press, 2000.

Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984. ISSN 0734-2071. http://doi.acm.org/10.1145/2080.357392.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1998.

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

Per Bothner. Kawa: Compiling Scheme to Java. In *Lisp Users Conference: Lisp in the Mainstream (40th Anniversary of Lisp)*, November 1998. URL http://www.cygnus.com/bothner/kawa.html.

Don Box. *Essential COM*. Addison-Wesley, 1998.

Martin Bravenboer and Eelco Visser. Concrete syntax for objects. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2004.

Martin Bravenboer, René de Groot, and Eelco Visser. MetaBorg in action: examples of domain-specific language embedding and assimilation using Stratego/XT. In *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, June 2005.

Burroughs Algebraic Compiler: a representation of ALGOL for use with the Burroughs 220 data-processing system. Burroughs Corporation, bulletin 220-21011-D edition, January 1961. URL http://www.bitsavers.org/pdf/burroughs/B220/220-21011-D_BALGOL_Jan61.pdf.

William E. Byrd and Daniel P. Friedman. From variadic functions to variadic relations: a miniKanren perspective. In *Workshop on Scheme and Functional Programming*, 2006.

Manuel M. T. Chakravarty. The Haskell 98 foreign function interface 1.0, 2002. URL http://www.cse.unsw.edu.au/~chak/haskell/ffi/.

Tyson Dowd, Fergus Henderson, and Peter Ross. Compiling Mercury to the .NET Common Language Runtime. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

ECMA International. Common language infrastructure (CLI). Technical Report ECMA-335, ECMA International, June 2006. URL http://www.ecma-international.org/publications/standards/Ecma-335.htm.

Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1997. URL `citeseer.ist.psu.edu/elliott97functional.html`.

Conal Elliott, Sigbjörn Finne, and Oege de Moor. Compiling embedded languages. In *Workshop on semantics, applications and implementation of program generation (SAIG)*, 2000.

Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

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

Matthias Felleisen, D.P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, pages 205–237, 1987.

Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming (FLOPS)*, 2006.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2002.

Sigbjörn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: A binary foreign language interface to Haskell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1998. URL `http://research.microsoft.com/users/daan/download/papers/hdirect.ps`.

Sigbjörn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 114–125, 1999. URL `citeseer.ist.psu.edu/finne99calling.html`.

Kathleen Fisher, Riccardo Pucella, and John Reppy. A framework for interoperability. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

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

Matthew Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. `http://www.plt-scheme.org/software/mzscheme/`.

R. C. Fraley. *How to conduct behavioral research over the Internet: A beginner's guide to HTML and CGI/Perl*. Guilford, 2004.

Alexander Friedman and Jamie Raymond. PLoT Scheme. In *Workshop on Scheme and Functional Programming*, 2003.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. MIT Press, July 2005. ISBN 0-262-56214-6.

Michael Furr and Jeffrey S. Foster. Polymorphic type inference for the JNI. In *European Symposium on Programming (ESOP)*, March 2006. URL `http://www.cs.umd.edu/~jfoster/papers/esop06.html`.

Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 62–72, 2005.

Fausto Giunchiglia. Multilanguage systems. In *Proceedings of AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*, 1991.

Fausto Giunchiglia and Luciano Serafini. Multilanguage hierarchical logics, or how we can do without modal logics. *Artificial Intelligence*, 65(1):29–70, 1994.

Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 248–260, 2001. doi: \url{http://doi.acm.org/10.1145/360204.360228}.

Paul Graham. *On Lisp*. Prentice Hall, 1993. ISBN 0130305529. URL `http://www.paulgraham.com/onlisptext.html`.

Graunke, Krishnamurthi, Van der Hoeven, and Felleisen. Programming the web with high-level programming languages. In *European Symposium on Programming (ESOP)*, 2001. URL `\url{http://www.ccs.neu.edu/scheme/pubs/esop2001-gkvf.ps.gz}`.

Paul Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Modeling web interactions. In *European Symposium on Programming (ESOP)*, 2003. URL `http://www.cs.brown.edu/~sk/Publications/Papers/Published/gfkf-modeling-web-inter/`.

David N. Gray, John Hotchkiss, Seth LaForge, Andrew Shalit, and Tony Weinberg. Modern languages and Microsoft's component object model. *Communications of the ACM*, 41 (5):55–65, May 1998.

Kathryn E Gray, Robert Bruce Findler, and Matthew Flatt. Fine grained interoperability through mirrors and contracts. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2005.

Mark Grechanik, Don Batory, and Dewayne Perry. Design of large-scale polylingual systems. In *International Conference on Software Engineering (ICSE)*, 2004.

Jurg Gutknecht. Active Oberon for .NET: an exercise in object model mapping. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

Robert Harper. Type systems for programming languages. Unpublished course notes, 2000.

Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

Fritz Henglein and Jakob Rehof. 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)*, 1995.

Michi Henning. The rise and fall of CORBA. *ACM Queue*, 4(5), June 2006. URL `http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=396&page=1`.

David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 16–27, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-905-5. doi: \url{http://doi.acm.org/10.1145/1016850.1016857}.

Hans-Jürgen Hoffman. The IBM OS/360 ALGOL 60-compiler. Unpublished notes, 1998. URL `http://www.pu.informatik.tu-darmstadt.de/docs/HJH-199804xx-Algol60.rtf`.

Peter Housel, Christian Stork, Vivek Haldar, Niall Dalton, and Michael Franz. Language-agnostic approaches to mobile code. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28 (4es):196–196, 1996.

Paul Hudak. Modular domain specific languages and tools. In *International Conference on Software Reuse*, pages 134–142, 1998.

Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3), 1996.

Lorenz Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical report, Bell Laboratories, 1995. URL `http://www.cs.bell-labs.com/who/lorenz/papers/smlnj-c.pdf`.

Rosziati Ibrahim and Clemens Szyperski. The COMEL language. Technical Report FIT-TR-97-06, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, 1997.

J. D. Ichbiah. Preliminary Ada reference manual. *SIGPLAN Notices*, 14(6a):1–145, 1979. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/956650.956651.

FORTRAN II for the IBM 704 Data Processing System. International Business Machines Corporation, c28-6000-2 edition, 1958. URL http://www.bitsavers.org/pdf/ibm/704/C28-6000-2_704_FORTRANII.pdf.

David Jeffery, Tyson Dowd, and Zoltan Somogyi. MCORBA: a CORBA binding for Mercury. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*, 1999. URL http://www.cs.mu.oz.au/research/mercury/information/papers.html#mcorba_padl99.

Samuel Kamin. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science*, 14, 1998.

Alan Kaplan and Jack C. Wileden. Toward painless polylingual persistence. In *Seventh International Workshop on Persistence Object Systems*, 1996.

Alan Kaplan, John Ridgway, and Jack C. Wileden. Why IDLs are not ideal. In *Proceedings of the 9th International Workshop on Software Specification and Design*, 1988.

Andrew Kennedy. Securing the .NET programming model. *Theoretical Computer Science*, 364(3):311–317, November 2006. URL http://research.microsoft.com/~akenn/sec/.

Leif Kornstaedt. 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)*, 2001.

Justin Koser, Haakon Larsen, and Jeffery A. Vaughan. SML2Java: a source to source translator. In *Declarative Programming in the Context of Object-Oriented Languages (DP-COOL)*, 2003.

Shriram Krishnamurthi. Automata via macros. *Journal of Functional Programming*, 16(3): 253–267, May 2006. URL http://www.cs.brown.edu/~sk/Publications/Papers/Published/sk-automata-macros/.

Shriram Krishnamurthi. *Linguistic Reuse*. PhD thesis, Rice University, 2001.

Shriram Krishnamurthi, Robert Bruce Findler, Paul Graunke, and Matthias Felleisen. Modeling web interactions and errors. In Dina Goldin, Scott Smolka, and Peter Wegner, editors, *Interactive Computation: The New Paradigm*, chapter 11. Springer-Verlag, September 2006. URL http://www.cs.brown.edu/~sk/Publications/Papers/Published/kfgf-model-web-inter-error/.

Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Journal of Higher-Order and Symbolic Computation (HOSC)*,

2007. URL http://www.cs.brown.edu/~sk/Publications/Papers/Published/khmgpf-impl-use-plt-web-server-journal/. To appear.

David Alex Lamb. IDL: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9(3):297–318, July 1987.

David Alex Lamb. *Sharing intermediate representations: the interface description language*. PhD thesis, Carnegie-Mellon University, May 1983.

P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3): 157–166, March 1966.

Daan Leijen. Functional components: COM components in Haskell. Master's thesis, Department of Computer Science, University of Amsterdam, September 1998.

Daan Leijen. *The λ Abroad – A Functional Approach to Software Components*. PhD thesis, Department of Computer Science, Universiteit Utrecht, The Netherlands, November 2003.

Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

Xavier Leroy. *Camlidl user's manual, version 1.05*. INRIA Rocquencourt, 2001.

Xavier Leroy. Efficient data representation in polymorphic languages. In *Programming Language Implementation and Logic Programming*, 1990.

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Professional, second edition, April 1999. URL http://java.sun.com/docs/books/vmspec/.

Mike MacHenry and Jacob Matthews. Topsl: a domain-specific language for on-line surveys. In *Workshop on Scheme and Functional Programming*, 2004. URL http://repository.readscheme.org/ftp/papers/sw2004/machenry.pdf.

Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, July 1991.

Jacob Matthews. Equation-preserving multilanguage systems, 2007.

Jacob Matthews. *Reduction Semantics and Redex*, chapter Topsl: DSEL as Multi-Language System. MIT Press, 2008.

Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing, or, theorems for low, low prices!, 2007.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007. Extended version appears as University of Chicago Technical Report TR-2007-8, under review.

Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA)*, 2004.

Jay McCarthy and Shriram Krishnamurthi. Interaction-safe state for the web. In *Workshop on Scheme and Functional Programming*, 2006. URL `http://www.cs.brown.edu/~sk/Publications/Papers/Published/mk-int-safe-state-web/`.

Erik Meijer, Nigel Perry, and Arjan van Yzendoorn. Scripting .NET using Mondrian. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 150–164, London, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4.

Philippe Meunier and Daniel Silva. From Python to PLT Scheme. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, pages 24–29, 2003. URL `\url{http://spyweb.hopto.org/}`.

Philippe Meunier and Daniel Silva. From Python to PLT Scheme. In *Workshop on Scheme and Functional Programming*, 2004. URL `http://repository.readscheme.org/ftp/papers/sw2003/SPY.pdf`.

James H. Morris, Jr. Types are not sets. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1973. URL `http://portal.acm.org/citation.cfm?id=512938`.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Introduction to Scala. `http://scala.epfl.ch/docu/files/ScalaIntro.pdf`, 2005.

U.S. General Accounting Office. QPL. Software: `http://www.gao.gov/qpl/`.

Atsushi Ohori and Kazuhiko Kato. Semantics for communication primitives in an polymorphic language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 99–112, 1993. URL `citeseer.ist.psu.edu/article/ohori93semantics.html`.

Atsushi Ohori, Kiyoshi Yamatodani, Nguyen Huu Duc, Liu Bochao, and Katsuhiro Ueno. SML# project. `http://www.pllab.riec.tohoku.ac.jp/smlsharp/`.

Scott Owens, Matthew Flatt, Olin Shivers, and Benjamin McMullan. Lexer and parser generators in Scheme. In *Workshop on Scheme and Functional Programming*, 2004.

parrot. Parrot virtual machine. http://www.parrotcode.org/.

Simon Peyton Jones and Phil Wadler. Imperative functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1994.

Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components in Haskell. In *ACM SIGPLAN Haskell Workshop*, 1997.

Simon Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: weak pointers and stable names in Haskell. In *International Workshop on Implementation of Functional Languages (IFL)*, 1999.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

Pedro Pinto. Dot-Scheme: A PLT Scheme FFI for the .NET framework. In *Workshop on Scheme and Functional Programming*, November 2003. URL http://repository.readscheme.org/ftp/papers/sw2003/Dot-Scheme.pdf.

David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management (IWMM)*, pages 211–249, September 1995. URL http://www-sor.inria.fr/publi/SDGC_iwmm95.html.

G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, pages 223–255, 1977.

Riccardo Pucella. The design of a COM-oriented module system. In *Proceedings of the Joint Modular Languages Conference (JMLC)*, 2000. URL http://arxiv.org/abs/cs.PL/0405083.

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

Christian Queinnec. A library for quizzes. In *Workshop on Scheme and Functional Programming*, 2002.

Norman Ramsey. Embedding an interpreted language using higher-order functions and types. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 6–14, 2003. URL http://www.complang.tuwien.ac.at/anton/ivme03/proceedings/ramsey.ps.gz.

Fermin Reig. Annotations for portable intermediate languages. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

Claus Reinke. FunWorlds — functional programming and virtual worlds. In *International Workshop on Implementation of Functional Languages (IFL)*, 2001.

John Reppy and Chunyan Song. Application-specific foreign-interface generation. In *International Conference on Generative Programming and Component Engineering (GPCE)*, 2006.

John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

Jon G. Riecke. Fully abstract translations between functional languages. In *POPL: The 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1991.

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 1993.

Michel Schinz and Martin Odersky. Tail call elimination on the Java virtual machine. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

Zhong Shao. Typed common intermediate format. In *USENIX Conference on Domain-Specific Languages*, 1997.

Olin Shivers. A universal scripting framework or Lambda: the ultimate 'little language'. *Concurrency and Parallelism, Programming, Networking, and Security, Lecture Notes in Computer Science*, 1179:254–265, 1996. URL `http://citeseer.ist.psu.edu/shivers96universal.html`.

Olin Shivers. The anatomy of a loop: a story of scope and control. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2005.

Dorai Sitaram. Programming in Schelog. `http://www.ccs.neu.edu/home/dorai/schelog/schelog.html`, June 2003.

Paul Steckler. Component support in PLT Scheme. In *Workshop on Scheme and Functional Programming*, 2000. http://www.ccs.neu.edu/home/matthias/Scheme2000/steckler.ps.

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

Guy L. Steele and Richard P. Gabriel. The evolution of Lisp. *History of programming languages*, pages 233–330, 1996. doi: http://doi.acm.org/10.1145/234286.1057818.

Ejiro Sumii and Benjamin Pierce. A bisimulation for dynamic sealing. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004. URL citeseer.ist.psu.edu/sumii04bisimulation.html.

Don Syme. ILX: extending the .NET common IL for functional language interoperability. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, 2001.

Clemens Szyperski. *Component Software*. Addison-Wesley, second edition, 1998.

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

Inc. The Object Management Group. Common Object Request Broker Architecture: Core Specification, version 3.0.3. Technical Report formal/04-03-12, March 2004. URL http://www.omg.org/cgi-bin/doc?formal/04-03-12.

Andrew Tolmach and Dino P. Oliva. From ML to Ada(!?!): strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998. URL http://web.cecs.pdx.edu/~apt/jfp98.ps.

Valery Trifonov and Zhong Shao. Safe and principled language interoperation. In *European Symposium on Programming (ESOP)*, pages 128–146, 1999.

Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990.

Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, 1989.

Mark Weiser, Alan Demers, and Carl Hauser. The portable common runtime approach to interoperability. In *ACM Symposium on Operating Systems Principles*, 1989. URL http://portal.acm.org/citation.cfm?id=74862.

Noel Welsh, Francisco Solsona, and Ian Glover. SchemeUnit and SchemeQL: Two little languages. In *Proceedings of the Third Workshop on Scheme and Functional Programming*, 2002.

Frank Wierzbicki, Charles Groves, Otmar Humbel, Alan Kennedy, Samuele Pedroni, and Khalid Zuberi. Jython, 2007. URL http://www.jython.org/. Last visited September 17, 2007.

Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.

Kazuki Yasumatsu and Norihisa Doi. SPiCE: A system for translating Smalltalk programs into a C environment. *IEEE Transactions on Software Engineering*, 21(11):902–912,

1995. ISSN 0098-5589. URL http://doi.ieeecomputersociety.org/10.1109/32.473219.

Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1999.