TWIG: A CONFIGURABLE DOMAIN-SPECIFIC LANGUAGE

by

GEOFFREY C. HULETTE

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2012

DISSERTATION APPROVAL PAGE

Student: Geoffrey C. Hulette

Title: Twig: A Configurable Domain-Specific Language

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Dr. Allen D. Malony	Chair
Dr. Michal Young	Member
Dr. Zena Ariola	Member
Dr. Shawn Lockery	Outside Member

and

Kimberly Andrews Espy	Vice President for Research & Innovation/ Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2012

© 2012 Geoffrey C. Hulette

DISSERTATION ABSTRACT

Geoffrey C. Hulette

Doctor of Philosophy

Department of Computer and Information Science

June 2012

Title: Twig: A Configurable Domain-Specific Language

Programmers design, write, and understand programs with a high-level structure in mind. Existing programming languages are not very good at capturing this structure because they must include low-level implementation details. To address this problem we introduce Twig, a programming language that allows for domain-specific logic to be encoded alongside low-level functionality. Twig's language is based on a simple, formal calculus that is amenable to both human and machine reasoning. Users may introduce rules that rewrite expressions, allowing for user-defined optimizations. Twig can also incorporate procedures written in a variety of low-level languages. Our implementation supports C and Python, but our abstract model can accommodate other languages as well. We present Twig's design and formal semantics and discuss our implementation. We demonstrate Twig's use in two different domains, multi-language programming and GPU programming, and compare Twig against a well-known typemapping system, SWIG.

CURRICULUM VITAE

NAME OF AUTHOR: Geoffrey C. Hulette

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:
University of Oregon, Eugene, OR
University of California, San Diego, CA
Tufts University, Somerville, MA

DEGREES AWARDED:
Doctor of Philosophy in Computer Science, 2012, University of Oregon
Master of Science in Computer Science, 2007, University of California, San Diego
Bachelor of Arts in Computer Science, 2000, Tufts University

AREAS OF SPECIAL INTEREST:
Programming languages, multi-language programming, high-performance computing.

PUBLICATIONS:

G. C. Hulette and J. Solis. On source code transformations for steganographic applications. In Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 03, WI-IAT 11, pages 261264, Washington, DC, USA, 2011. IEEE Computer Society.

G. C. Hulette, M. J. Sottile, R. Armstrong, and B. Allan. OnRamp: enabling a new component-based development paradigm. In Proceedings of the 2009 Workshop on Component-Based High Performance Computing, CBHPC 09, pages 110, New York, NY, USA, 2009. ACM.

M. J. Sottile, G. C. Hulette, and A. D. Malony. Workflow representation and runtime based on lazy functional streams. In Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, WORKS 09, pages 110, New York, NY, USA, 2009. ACM.

G. C. Hulette, M. J. Sottile, and A. D. Malony. WOOL: A workflow programming language. In Proceedings of the 2008 Fourth IEEE International Conference on eScience, ESCIENCE 08, pages 7178, Washington, DC, USA, 2008. IEEE Computer Society.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisors, Prof. Allen Malony and Dr. Matthew Sottile, for their support and guidance during my Ph.D work. Their patience, enthusiasm, and deep knowledge of my chosen areas were instrumental in my research.

I would also like to thank my committee: Prof. Zena Ariola, Prof. Michal Young, and Prof. Shawn Lockery, for their insightful questions and feedback.

I would like to acknowledge my colleagues at Sandia National Laboratories, who supported me throughout this process. I would especially like to thank my mentor, Dr. Robert Armstrong, and my very patient manager, Dr. Keith Vanderveen.

Last but not the least, I would like to thank my friends and family. I could not have written this dissertation without their help. My deepest and most sincere thanks go to my wife, Dr. Annmarie Hulette, for her understanding, patience, encouragement, and love. She has been an inspiration to me, in both my work and life.

My research was supported in part by the Department of Energy Office of Science, Advanced Scientific Computing Research. I very grateful to my colleagues who were part of that grantwriting effort.

I dedicate this dissertation to my parents, Richard and Mary Hulette. Their love, generosity, and guidance have enabled me to pursue my dreams, and for that I am deeply grateful.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

Benefits of Capturing High-level Structure 1
Existing Approaches 2
Twig 3
Contributions 7

II. BACKGROUND 9

Multi-language Programming 9
Typemap Tools 57
Term Rewriting 63
Workflow Programming 67

III. CODE GENERATION 72

Abstract Code Generation 72
Generating C 76

Chapter Page

IV. THE TWIG LANGUAGE 80

Formal Semantics 80

V. USER-DEFINED EXPRESSION REDUCTIONS 95

Expression Normalization 96

Implementation 98

Discussion 99

VI. THE DESIGN OF TWIG'S INTERPRETER 101

The `twigc` Application 101

Embedded Design 101

Code Generation 103

Expression Reductions 105

VII. EVALUATION OF TWIG 106

Twig Compared 106

Multi-language Programming 112

GPU programming 122

Chapter Page

VIII. FUTURE WORK 130

Implementation 130

Theory 132

IX. CONCLUSION 137

Summary of Results 137

Advantages 138

Limitations 139

Twig in Context 140

APPENDIX: PROOFS 141

Associative Operators 141

Expression Identities 147

REFERENCES CITED 155

LIST OF FIGURES

Figure Page

1. A sequence of two permutation blocks. 77
2. Two basic blocks. 78
3. Sequential block composition. 78
4. Parallel block composition. 79
5. Permutation of tuples. 94
6. Typemap identities. 120

CHAPTER I

INTRODUCTION

Programmers design, write, and understand programs with a high-level structure in mind. In most cases, however, that structure is not captured in the artifacts resulting from the coding process. This occurs because programming languages are not very good at expressing the high-level and/or domain-specific logic of a program. There are simply too many details to capture in the lower-level logic of any non-trivial program, and these details obscure and possibly even subvert the higher-level logic. This is especially true in the very common case where a program undergoes multiple modifications, rewrites, extensions, and refactoring over the course of its lifetime – even if the high-level logic were captured, perhaps in natural language comments or design documents, its relationship to the actual code weakens over time.

The kind of high-level structure we are concerned with typically arises from the programmer's domain-specific thought process and reasoning. Often, the logic may be quite simple, at least conceptually – sequences of operations, conditional branches, cycles, and the like, although the exact semantics of these structures within the logic of a particular domain may differ and be more complex when compared against the versions found within a mainstream programming language.

Benefits of Capturing High-level Structure

If we could somehow capture this high-level logic, what might we do with it? First, since it expresses domain-specific structure, it could be used for *domain-specific optimization*. For example, in a GPU code, if the high-level logic captures

memory copying operations, we could recognize a redundant copy and eliminate it. This optimization might be quite difficult to do if the algorithm were written in a language that does not capture high-level structure. In C, for example, GPU memory copies are performed via API calls that appear as regular functions and are therefore outside the scope of the compiler to analyze.

Second, in some cases we can design reusable software tools, such as code generators or runtime engines, that enable programmers or domain experts to work with the high-level structure and leave the low-level details to an automated tool. *Workflow systems* are an interesting and successful example of this kind of tool.

Third, if the high-level logic is somehow associated with the final code, it may serve as a high-quality and high-fidelity form of *documentation*, enabling a programmer to grasp the high-level interpretation of a program more easily.

Existing Approaches

Programmers are, of course, aware that this kind of high-level information is both important and difficult to capture, and so there have been several attempts to improve the situation.

The most common method for capturing high-level information is informal documentation based on natural language descriptions of the program components and their intended functionality. In this case, the information may be embedded in code comments, perhaps using a structured comment tool such as Doxygen or JavaDoc. Or, it may be captured in an entirely separate document. Either way, the informal nature of this sort of documentation does not capture much (if any) logical structure, and does not lend itself to the kind of reasoning we are interested in. Furthermore, the loose connection between documentation and the code it describes

infamously leads to out-of-date, misleading, or completely inaccurate descriptions. This phenomenon is often referred to as “version skew,” as the version of software referred to in the documentation gradually deviates from the most current version. In many cases, for a programmer attempting to understand code, this scenario is worse than having no documentation at all.

Formal models, like those used by model checking tools such as SPIN [45], do capture the kind of high-level program structure we are interested in. Indeed, a model is essentially a high-level abstraction of a program’s behavior. However, these kinds of models are not tied to the code and they cannot be used to alter or optimize it directly. Also, as with informal documentation, there is significant potential for version skew if the program evolves and the model is not updated.

Workflow systems allow for the expression of high-level structure in a domain-agnostic fashion, but are typically informal and involve complex external requirements. A workflow-based program’s behavior is essentially defined by the workflow runtime used to execute it, and this can make such programs brittle and hard to maintain as the software evolves. Workflow programming systems also tend to focus on coordination of relatively large components, such as whole applications, which often have little in common in terms of their design. Since the components tend to be so heterogeneous, it is uncommon for these systems to provide ways to reason about the semantics of the workflow as a whole.

We discuss these approaches further in Chapter II.

Twig

Ideally, we would like a programming paradigm that allows for high-level, domain-specific logic to be encoded alongside the low-level implementation details

in a way that binds the two together. The high-level structure would organize the low-level details, and be available for automated analysis, reasoning, and formal verification. The low-level logic, meanwhile, would be available for modification and tuning by the programmer. Ideally, this hypothetical system would also allow programmers to incorporate their own domain- or application-specific rules, allowing them to reason about and exploit the exposed high-level structure. Such a system would lead to software that is easier to develop and maintain, since it would be easier for both humans and machines to reason about than today's typical programming languages allow. It would also be much easier for humans to interpret and understand programs written in this way, since the high-level structure would be provided explicitly, and would never become out-of-date. Through these benefits, we argue that this type of system would represent an important advancement in the way that programs are written.

Twig is a language for writing *typemaps* – small, declarative programs used to translate data from one representation to another. Twig represents a step in the direction for programming we envision above, by uniting two levels of semantic information in one program. In Twig, programmers are able to express programs in terms of a combined high- and low-level logic, with the final program being synthesized automatically. Twig's high-level logic is based on a simple, formal calculus that is amenable to either human or machine reasoning, and can be adapted with extra rules for reasoning in domain-specific contexts. The low-level domain code is expressible, in theory, in any mainstream programming language. Our implementation allows for C and Python, and can be extended to support other languages.

Twig is, in many ways, quite similar to typemap-based tools like SWIG, but it equips SWIG's typemap language design with extra structure and expressiveness. The result is a language that is flexible while remaining broadly applicable to practical problems. We exploit its structure by introducing the idea of *reductions*, which allow for user-defined, domain-specific typemap optimizations and transformations.

Twig addresses the issues identified above:

1. **The high-level, domain-specific structure of a program is lost in the process of writing low-level code.** Twig's language, while restricted to relatively small programs called *typemaps*, is a move in the direction of providing high-level structure over low-level program code. The high-level aspect of a Twig program is based on just a few combinators, described in Chapter IV. Twig's relatively simple primitives and combinators restrict the program's semantics, but allow for easier reasoning, analysis, and transformation. In particular, Twig's high-level combinators are adaptable enough to express domain-specific constructions and transformations; we demonstrate this with examples in Chapter VII.

2. **Programs written at a very high level lose the flexibility that low-level coding affords.** Twig aims to provide a bridge between high-level abstraction and low-level code. Twig allows for low-level code, written in languages such as C or Python, to be embedded in primitives. The programmer is then able to work with the primitives at a high-level, using Twig's combinators. The output of a Twig program is the low-level code, combined and synthesized according to the high-level rules. Thus, the low-

level representation is available for flexibility, but is still subject to the structure imposed at the high level.

Domain-specific languages are generally a purely high-level approach, and are typically tied to a particular domain. Twig, by comparison, can be customized and applied to many different domains. We review some approaches to domain-specific programming in Chapter II, and show how Twig can be applied in two application domains, GPU programming and multi-language programming, in Chapter VII.

Twig's set of high-level combinators can be used to build program structures similar to those commonly found in workflow systems. Unlike workflow systems, Twig generates code instead of managing its execution at runtime. This approach has the advantage of decoupling the workflow implementation from the abstraction presented to the programmer. We review some existing workflow systems in Chapter II.

3. **Existing approaches to capturing the high-level structures of low-level code are fragile, because the two representations evolve independently and can become inconsistent.** Twig's language associates low-level code, written in languages like C or Python, with primitive structures in its high-level logic. These primitives are combined via a set of simple combinators. In addition, unlike formal modeling tools, in Twig neither the low nor high levels of the program can be changed without impacting the overall program's behavior, and thus the high-level and low-level semantics are tightly coupled. We describe the lower-level semantics for code block synthesis in Chapter III, and our implementation in Section 6.3.

These primitives can be combined using Twig's high-level semantics, described in Chapter IV.

4. **Programs that lack explicit high-level structure can be difficult to optimize in domain-specific ways.** Twig programs can be *reduced*, via both built-in and user-defined rules for reorganizing and combining expressions. Twig's simple high-level logic makes such rules easy to describe and express. By providing the proper domain-specific rules, domain-specific optimizations may be designed, even by end-users. The process of reduction is described in Section V, and we give the details of our implementation in Section 6.4. Sections 7.2 and 7.3 demonstrate practical examples.

5. **Programs that lack explicit high-level structure force programmers to work at an inappropriate level of abstraction.** Twig allows programmers to abstract their own low-level code, and then work with it in a domain abstraction. It also exposes the high-level structure for analysis. We exploit this capability with reductions, presented in Section V.

Contributions

To summarize, with Twig we have made the following novel contributions:

1. Designed a simple model for low-level code synthesis;

2. Extended the semantics of System S to incorporate this model;

3. Shown how term rewriting tools can be used to reduce typemaps written in our language;

4. Implemented a prototype of this language;

5. Demonstrated the utility of typemap reductions in a domain where they had not previously been applied: that of GPU programming.

CHAPTER II

BACKGROUND

Multi-language Programming

Foreign function interfaces

A foreign function interface (FFI) allows a program written in one language to invoke routines and/or access data in another language. The term FFI is somewhat misleading; in addition to function calls, FFIs may allow method invocations for object-oriented languages, or hooks for a low-level language to work with high-level data.

FFIs allow interoperation between exactly two languages. This restriction differentiates them from approaches that allow more than two languages, such as language-neutral intermediate representations (Section 2.1) or interface definition languages (Section 2.1). FFIs are directional, and we say that the *source* language initiates calls to the *target* language. Many FFIs, however, also support callbacks from the target language to the source language.

FFIs are a common feature in mainstream programming languages because they provide two important capabilities. First, FFIs allow high-level languages to use low-level functions and capabilities not definable in the high-level language itself. For example, a high-level language may not be able to call the operating system directly, or may require low-level access to hardware features for performance-critical routines. Second, FFIs are used to wrap and expose existing libraries of routines so that they need not be rewritten in the high-level language.

FFIs must reconcile the runtime environments and application binary interfaces of two different languages. This may present a variety of challenges, depending on the features of those languages. We will now consider some examples of FFI systems and the issues they face.

Examples

Fortran Bind(C) Fortran 2003 introduced an FFI that allows bidirectional interoperability between Fortran and C [76, 75]. Since Fortran and C are similar, the FFI is straightforward. BindC introduces a set of *interoperable types* in Fortran that represent C's primitive types (e.g., `int`, `float`, and so on). Values of these types may be passed back and forth between C and Fortran with no conversion needed. Derived types, such as structures, arrays, and pointers are interoperable if their component types are interoperable. To expose a C function to Fortran, the C function is defined as usual and linked into the Fortran program. The Fortran program must declare a correspondingly named function marked with the `BIND(C)` keyword. The function's signature (i.e., its argument and return types) must consist only of interoperable types, and it is the programmer's responsibility to ensure that the declared signature in Fortran matches the C definition.

`BIND(C)` is a bi-directional FFI, so this process also works in reverse. If a Fortran function is defined with an interoperable signature, then the C program can access it by linking in the Fortran code and declaring a C function with the corresponding name and signature.

Java Native Interface The Java Native Interface (JNI) [60] is an FFI from Java to C. The JNI makes extensive use of *glue code*, that is, code that does not contribute to the core program functionality, but rather serves to "glue" otherwise

incompatible code together (see Section 2.1, below). In the JNI, glue code is used within C functions to provide access to Java object data, and to invoke methods. When it is invoked, the C function is passed a special pointer (called `JNIEnv`) that acts as a reference to the JVM. This pointer is used by special JNI functions that allow C code to look up classes and types by name, instantiate objects, invoke methods, and so on. Interaction occurs dynamically, and no static type checking is performed. One benefit of this dynamic approach is binary compatibility. Because the JNI uses Java's type reflection mechanism [39] to work with JVM entities by name, the C code will not need to be recompiled even if the JVM implementation changes.

The JNI uses glue code extensively. Primitive Java types are passed by value and mapped to corresponding C primitive types, while objects are passed by reference (except arrays, see below). The references appear to C as opaque pointers and must be manipulated exclusively through glue code. There are two kinds of references: local and global. Local references are valid only until the C function returns to Java. When control has returned to the JVM, locally referenced objects may be garbage collected. Global references are never be garbage collected until they are explicitly released. This implies that the C code must take care to release global references or else incur memory leaks. A global reference is created from a local one. Object-type arguments to JNI functions are always local.

Because objects are accessed indirectly through references and glue code, the garbage collector is free to move objects around in memory at any time. The data may even be moved during the C call if the garbage collector is running on a separate thread – the JNI standard requires that implementations take this possibility into account.

Java arrays are treated specially for performance reasons. Native code is able to access arrays directly, circumventing the usual glue code. This makes array manipulation fast. But, it means the garbage collector must take care not to move the data while the native code is accessing it. There are three ways to accomplish this task. First, the programmer can ask the JNI to "pin" the array; this tells the garbage collector to leave it in place until the programmer unpins it. This method is effective and easy to program, but can significantly complicate the garbage collection algorithm. Therefore not all JVM implementations support pinning. Second, the JNI can copy the array's contents to a buffer so that the C code can work on it locally, and then copy it back. While the copied array is being modified, the garbage collector is free to relocate the original array. This method works when pinning is not available, but large or frequent array copies may be costly. Third, the native code can enter a critical region that temporarily suspends the Java garbage collection thread. While the garbage collector is suspended, the native code can work on the array undisturbed. However, within the critical region the native code is restricted from blocking.

Extra care must be taken when using arrays in multi-threaded JNI programs. For example, consider what would happen if two Java threads tried to invoke a native method on the same array concurrently, using the method of copying the array to a local buffer. The program would contain a race condition – the first method would complete and write the array back to Java, and then the second method would complete, and overwrite the first array. Or, if instead the program uses the critical region method, it must take care that other threads will not exhaust Java's available memory while the garbage collector is suspended. The JNI enables native code to acquire and release locks via a glue code interface to

the usual Java synchronization mechanism, but otherwise does not provide special support for multi-threaded code.

The JNI supports callbacks from C to Java. The C program can use glue code to acquire object references, and pass these references to a Java method. The method signatures are obtained and invoked dynamically, via the `JNIEnv` data structure.

The JNI allows C code to throw Java exceptions. When an exception is thrown from C, the native function exits immediately, control returns to Java, and the exception is processed normally. Because the JNI allows callbacks, it is possible that an exception thrown from a Java method will be encounter a C function on its way up the stack. In this case, control returns to C. The exception does not interrupt the native code, but is recorded in the `JNIEnv` data structure. Therefore, it is the responsibility of the C function to check whether there are any pending exceptions after control returns from a callback. If an exception is detected, the C function can handle it or else re-throw it.

Python Ctypes Ctypes [55, 4] is one of several Python FFIs to C; it is interesting because it uses dynamic instead of static libraries. Dynamic libraries are files containing compiled C functions, formatted in such a way that the code can be loaded and executed by other programs at runtime. Dynamic libraries include a symbol table so that functions can be found by name.

The Ctypes Python module is initialized with a dynamic library, and it generates Python wrapper code to expose each function in the library as a Python method. Dynamic libraries do not contain type information, so the programmer must explicitly set the number and types of the arguments as well as the return type for each function.

Ctypes knows how to convert simple data representations between C and Python. Primitive C types are converted to equivalent Python types. Structures in C are converted recursively into Python objects. Arrays may be converted if they contain only primitive types. Structures and arrays are passed to Python as references to the C heap or stack. The references are just pointers, but Python is prohibited from manipulating them directly. Instead, Ctypes provides Python glue code to manipulate each kind of data structure through the pointer.

Ctypes can construct a C function pointer from a Python routine, allowing C to call back into Python. The documentation suggests that Python objects passed to C should have references held in Python, to prevent them from being garbage collected [4].

Matlab MEX files The Matlab programming environment provides MEX files [3], a simple but effective FFI that allows Matlab to call C, C++, or Fortran functions. MEX foreign functions are just C functions, written using the MEX API to decode and manipulate Matlab's matrix values. It is the programmer's responsibility to ensure that the MEX functions they provide conform to Matlab's requirements; for example, failing to properly parse an argument list should result in the function returning a MEX-specified error. Matlab makes no effort to check that the functions conform in these ways.

SML/NJ Standard ML of New Jersey (SML/NJ) provides an FFI to C [48]. It is an interesting example because the two languages are quite different. The representations even of basic data types are different in SML. In particular, data types in C depend on the compiler, where in SML they are fixed by the standard. The FFI avoids this problem by parameterization, using meta-information about

the C compiler. The meta-information includes details such as the size of an integer, byte ordering, calling convention, and so on. The parameterized FFI then exposes a set of types to ML programs representing those used by the C compiler, along with conversion routines to and from common SML types.

SML/NJ manages memory with a garbage collector. As we saw in the JNI, this can cause problematic interactions. For example, a C program using a pointer into the ML heap must ensure that the dereferenced data will not be moved or disposed of. SML/NJ's implementation supports "pinning" memory, i.e., explicitly instructing the garbage collector to temporarily leave the data in place. So, C functions that wish to work with ML data directly must pin the data first.

Functions exported to SML from a C file must be marked with a special macro, which enables SML/NJ to look up the function's address by name. Since compiled C function libraries do not include type information, C functions must be registered in SML at runtime, and assigned the appropriate argument and return types. The FFI provides an ML type representing a pointer onto the C heap. It also provides C glue code to manipulate ML data structures via a pointer onto the ML heap. These pointer types allow complex data to pass across the language boundary.

SML/NJ's FFI provides callbacks to ML functions. A callback's arguments and return types are restricted to the C-compatible data types, including pointers to ML data. ML callbacks are created from regular ML functions, i.e., closures. The conversion is accomplished by dynamically registering an ML closure with the FFI – this creates a "bundle" at a fixed address, which contains code to invoke the closure. The bundle's address is presented to C as a function pointer. With this scheme, the closure may be relocated by the garbage collector – when this happens,

the bundle's contents are updated to reflect the closure's new address. The bundle itself remains at its original address, and so the function pointer remains valid in C.

Haskell 98 The Haskell 98 FFI [27] is part of the Haskell 98 standard. The FFI includes some placeholders intended to facilitate calls from Haskell to any external language, but only the binding to C is defined in detail. Haskell presents some interesting challenges to integration with C. Like ML, Haskell features first-class functions, a strong and static type system, and automatic memory management. In addition, it has call-by-need semantics, and distinguishes functions that may cause side effects from so-called "pure" functions.

A foreign function is exposed by declaring its type signature in the Haskell source code, along with the keywords `foreign import`. This tells the Haskell compiler that the function definition will be found in a C library.

The Haskell 98 FFI restricts the type signature of foreign functions to a set of *basic types* that map unambiguously between Haskell and C. Basic types include the usual primitive types, such as integers and floating point numbers. Basic types also include a set of "raw" types such as `Word32`, that are independent of the machine architecture and C compiler. Finally, basic types include several varieties of pointers. First, there are regular pointers to the C heap, parameterized with another basic type describing the dereferenced data. Second, there are C function pointers. Third, there are "stable" pointers, which are references to Haskell expressions guaranteed to never be deleted or moved by the garbage collector until they are explicitly released. Stable pointers may be safely stored in C, without worry that they will be invalidated when control returns to Haskell. Finally, there are "foreign" pointers, a type representing a pair of a regular pointer onto the C heap along with a function pointer. The function pointer should point to a finalizer

function, to be called by the Haskell when the object is garbage collected. Foreign pointers allow C data objects to be memory managed by Haskell.

The Haskell 98 FFI permits callbacks. A Haskell callback must be a function declared with the `foreign export` keywords, and defined in Haskell. The callback function is always evaluated strictly (not lazily) if invoked from C. To invoke a Haskell callback, the calling C function should be declared "safe" using the `safe` keyword. Safe foreign functions entail some extra overhead to call, but they guarantee that the Haskell runtime will be in a consistent state if a callback is invoked. Unsafe foreign functions are faster, but callback behavior is undefined. Haskell functions used as callbacks should not throw exceptions; the runtime behavior in this case is undefined.

Haskell uses monadic types to represent functions that may cause side effects. Using type inference, the Haskell compiler can (usually) construct this type information from the Haskell code. For foreign functions, however, Haskell cannot infer the type information. So, by default, the FFI must conservatively assume that all foreign functions may cause side effects. The programmer may override this assumption, asserting that an imported foreign function is pure. Pure functions have two benefits. First, the FFI assigns the foreign function a less restrictive type (i.e., it does not wrap the return type in the IO monad). Second, the Haskell runtime is free to memoize invocations of pure function, including pure foreign functions.

GreenCard GreenCard [53] is a different approach to designing a Haskell FFI to C. GreenCard focuses on providing an easy way to generate *wrappers*, i.e., layers of code that expose pre-existing libraries of C functions. Usually, such libraries are provided as a pair of files – one a pre-compiled library of functions; the other a

"header" file of C declarations, including the names, arguments, and return types of each function. Where the Haskell 98 FFI requires that programmers manually declare foreign function signatures, GreenCard uses the header file to automatically generate appropriate types and glue code for each function.

Ideally, GreenCard could extract all the required information from only the header file. Unfortunately there are ambiguities. For example, C programmers generally use the type `char *` to represent null-terminated strings. But `char *` may also represent a pointer to a single character, or to an array of bytes. C does not distinguish between these cases, but Haskell, due to its strict type system, does. To resolve these ambiguities, GreenCard augments the header file with *annotations*. An annotations contains a Haskell type declaration for a C function, and the correspondence between the declared C and Haskell types resolves any mapping ambiguities. GreenCard includes a default mapping of types from C types to Haskell. If the defaults are satisfactory for a particular function, then no annotation is required.

GreenCard uses a type translation mechanism (see Section 2.2) called "Data Interface Schemes" (DIS). DISs are like macros or functions that describe how a type in C is converted to its Haskell equivalent. DISs have a flexible syntax, with common cases handled by simple directives, and uncommon cases with arbitrary code. For example, there is a DIS directive for converting a C `enum` to an equivalent Haskell variant type.

Another DIS directive is used for C pointer return values. Many C functions that return a pointer will return the value `NULL` in case of a failure, or else a valid pointer. The DIS maps the pointer type to a Haskell `Maybe` type. The value will be

`None` in case the pointer is `NULL`, and `Just x` otherwise, where `x` is the dereferenced pointer value.

GreenCard assumes by default that C functions may cause side effects, and wraps them in the `IO` monad. The programmer may override this assumption if they know the function is pure.

Since its purpose is wrapping existing C libraries, GreenCard restricts itself to calling C from Haskell. So, GreenCard does not include callbacks. It therefore avoids issues with garbage collection and exceptions.

CHASM CHASM [74] is a restricted kind of FFI between C++ and Fortran 90 (F90). F90 programs have a compiler-dependent interface, because the F90 standard leaves many decisions to the implementation. CHASM generates wrappers around F90 procedures that present a consistent and well-defined interface. This allows C++ code to call F90 procedures, without having to modify the calls to suit a particular F90 compiler. CHASM's static analysis uses the Program Database Toolkit (PDT) [63] to parse and query Fortran and C++ programs.

There are two important facets of F90's procedure interface that are left to the implementation. The first is the way that arrays are passed to functions. The F90 standard declares that arrays are passed by "descriptor," without specifying the descriptor's exact representation. So, depending on the F90 compiler, a procedure might expect a simple pointer for a descriptor, or an integer handle, or some data structure. The second compiler-dependent aspect of F90 is procedure names. Some compilers, for example, store F90 procedures names using only capital letters, while others precede names with an underscore. If a caller does not know the naming convention, it cannot find the F90 procedure.

CHASM solves this problem by abstracting each Fortan 90 procedure, using a compiler-independent wrapper function. The wrapper uses a well-defined, compiler-independent naming scheme for Fortran procedures, and within the wrapper it calls the compiler-dependent name.

The wrapper function also presents a compiler-independent interface for passing arrays to procedures. The wrapper procedure has the same number and types of arguments as the wrapped procedure and except for arrays, these are passed through unchanged. The wrapper replaces each array descriptor argument, however, with an integer handle representing the array. This handle is an index into a globally-scoped table, maintained by CHASM. The table relates integer handles to array references. An array only needs to be registered in the table if it passes through one of the wrapper functions. Therefore, the wrappers contain all the logic needed to maintain the tables.

CHASM also includes a C++ class that encapsulates a Fortran array. CHASM will generate stubs for calling F90 procedures in C++ as well, and these wrappers automatically convert the integer handle to the array class.

Glue code

The examples in this section have illustrated that we can categorize FFI systems according to the way they use *glue code*. Glue code is a term for program logic that helps to connect or reconcile two different representations of data or code. In general, frameworks endeavor to minimize the need for glue code. As we have seen, FFIs are rarely able to exclude glue code altogether, but there are ways to mitigate the burden this places on programmers.

Glue code in FFIs serves to overcome semantic ambiguities where the two languages interact. These issues include pinning arrays as in the JNI, or registering the appropriate C function type signatures at runtime, as seen in SML/NJ and Ctypes. The most common use of glue code, though, is to allow the data types of one language to be interpreted and manipulated by the other. In the JNI, for example, programmers must use an API to pick apart Java objects passed to C functions. This kind of glue code is often used to address the problem of *type mapping* (see Section 2.2), where the "natural" data types of one language are mapped to a convenient and/or natural analog in the other. For example, many FFIs map between some high-level data type for strings and C's `char *` representation.

Many systems that we will examine in Sections 2.2 and 2.1, such as [44], attempt to automatically or semi-automatically generate FFI glue code.

FFIs may require glue code in the source language, the target language, or occasionally both. The former is useful for generating high-level "wrapper" functions for existing libraries, because the library source code may be inconvenient or impossible to modify. This is the approach taken by Haskell GreenCard and the Haskell 98 FFI, Ctypes, and SML/NJ. The latter option, glue code in the target language, may be preferable when foreign functions are written with interoperability in mind. It allows low-level functions to work directly on complex, high-level data types, possibly avoiding data conversion. This approach is taken by the JNI and Matlab's MEX files. Fortran BindC requires little glue code because Fortran and C are fairly similar.

Type safety

A FFI cannot make strong guarantees about type safety if one of its interacting languages is unsafe. In particular, a program in an otherwise type-safe language employing a FFI to C is no safer than C itself.

Glue code should, in principle, be able to check for some kinds of safety violations at runtime. In practice, this does not seem to be a popular FFI feature. This is probably because one of the foremost benefits of a C FFI is the speed of C code, which runtime checks could degrade. None of the FFIs above include runtime safety checks.

Garbage collection

Automatic memory management, i.e., a garbage collector [16], in one or both languages presents challenges for FFIs. The most common problem is notifying a foreign garbage collector that a reference to one of its objects is held, so that the object will not be released. Usually, this must be done explicitly. Depending on the scheme, the object may have to be explicitly deallocated as well.

In addition, garbage collectors may move objects around in memory. If the garbage collector runs on a separate thread (e.g., in Java), the object could even be moved while the main thread is operating on the data in a foreign function. In this case, the result would be a disaster – the data would literally be moved out from under the running program.

A seemingly simple solution is to suspend garbage collection for the duration of an FFI call. This is insufficient in general. Pointers to foreign objects stored across separate FFI invocations may still be invalidated (i.e., moved or deallocated) when control returns from the FFI invocation. If the FFI includes callbacks, the

FFI must usually resume garbage collection when the foreign function calls back; in this case, even local pointers might be invalidated by the time control returns to the foreign code.

A better solution, used in the JNI and SML/NJ, is to "pin" data objects used by the foreign language. This tells the garbage collection algorithm that the data should not be moved or deallocated, until the pin is released. Unfortunately, pinning may complicate the garbage collector implementation. In cases where pinning is not or cannot be implemented, an alternative is to copy the data from the source language to a buffer in the target language, and then copy it back after the function completes. However, the overhead of copying may be substantial for large objects, or if it is called frequently. Moreover, copying becomes more complicated if the target language stores a pointer to the buffer; the FFI must then reconcile the two buffers on every entry or exit from the foreign runtime.

Exceptions

Exceptions are a common feature of high-level languages that can be difficult to map properly into low-level language semantics. Usually, FFIs connect higher-level languages to lower-level ones, so exceptions only cause problems if the FFI permits callbacks. In the absence of callbacks, there is no way for a thrown exception to reach a foreign function.

If the FFI has callbacks and the high-level language has exceptions, there are two approaches. The first, expedient option is to declare the program's behavior undefined when an exception reaches a foreign function. This approach is used in Ctypes and the Haskell 98 FFI. The second option, used for example in the JNI, is to set an exception flag when control returns to a foreign function. The

foreign function must check the flag to see if the callback returned normally, or via an exception. If an exception was thrown, the foreign function may re-throw the exception. Or it may handle it, resetting the flag to indicate the exceptional condition was resolved, and returning normally.

Discussion

The primary goal of most FFIs is to provide efficient access to C from a higher-level language. This goal is practical – most higher-level languages otherwise would sacrifice some or all of the low-level capabilities that C offers.

Efficient access to C is generally at the cost of the other interoperability goals. Targeting C precludes strong type checking and safety guarantees; in fact, use of an FFI will generally compromise these properties in a high-level language that features them. Furthermore, as we have seen, the programming model for FFIs may not be as natural as possible. While function calls seem like a good enough abstraction, FFIs frequently require glue code to reconcile language differences. This glue code generally obscures the main program logic, and ensures that the interoperability is not seamless. Finally, it is clear that FFIs are not scalable, because by definition an FFI connects exactly two languages.

Another reason to use FFIs is to decompose a program or algorithm across two languages, with the programmer writing different parts in the language most suited to the task. With the exception of using C for performance, as previously discussed, this use of FFIs is not seen very often. This may be because the design and implementation of an FFI is labor-intensive, and it is easier to use C as an intermediate *lingua franca* than to write a different FFI for every language.

Interface definition languages

Interface definition languages (IDLs) are a popular approach to interoperability [54]. An IDL describes an interface to a software *component* (see below), in terms that are abstracted as much as possible from the underlying implementation language, operating system, architecture, network protocol, and so on. The goal of this abstraction is to allow the software to be reused in many contexts. Here, we are interested in the ways that IDLs permit components written in different languages to interact.

A key concept in IDL-based systems is *marshalling*. IDLs are used to generate code that implements the abstract interface they describe in some target language. Among other things, this involves deciding which types in the target map to those of the IDL. IDLs usually specify some binary format for data in their type system so that it can be moved from place to place and interpreted in different languages or systems. The process of translating data from a language's native representation to that of the IDL is called `marshalling`. The reverse process, translating from the IDL representation to a native data format, is called `unmarshalling`.

Although details vary, IDL systems often work by generating *skeleton* and *stub* code. Skeleton code implements a template of the IDL-specified interface in some target language. The skeleton handles unmarshalling the arguments, invoking the correct function, and marshalling the return value. The skeleton contains *hooks*, i.e., spaces for an implementation of the interface functions to be filled in by the programmer. The stub code presents the interface in the target language, usually as a set of callable functions. Stub code marshals the arguments, finds and invokes the corresponding skeleton function, and unmarshals the return value. The stub

and skeleton code work together to hide the complexities of the framework from the programmer.

There are many different IDLs, a fact which in itself deters from the IDLs goal of maximal interoperability. This proliferation reflects the challenge of designing an abstract interface language that is interoperable with many different languages, while easy to use in any given language.

In this section we give an overview of components and remote procedure calls, two branches of software design where IDLs play an important role. Then we provide examples of several IDL-based systems, and conclude with a discussion of the reasons why IDLs are popular and some of the weaknesses they entail.

Components

A software component, generally, is a piece of code that is intended to be reused. The exact definition is a matter of some debate [46]. A reasonable, inclusive definition might be "a physical packaging of executable software with a well-defined and published interface" [46]. Software components are generally designed to be composed and reused by third parties, i.e., by programmers other than those who wrote the component itself.

Most component systems use IDLs to describe component interfaces [84]. IDLs allow components to be written in whatever language is most appropriate or convenient, while still presenting an interface that other components can consume. This approach to multi-language programming potentially allows many different languages to interoperate, in contrast to FFIs which allow only two [40].

Component-oriented software engineering describes a method for building software that consists entirely of composing systems of components [17, 56].

Components in this model are classified by their role and origins [84]. The most general-purpose components are those needed by many different kinds of applications (e.g., database access services, graphics, and so on). An application programmer rarely creates these kinds of components from scratch, since they are often available from third parties, and may be difficult to create. A middle tier of component generality encompasses those components that are needed by many applications within a particular domain. For example, many medical applications may need to access DICOM format images [2]. These components may expose a somewhat less general interface, if widely-used standards exist for the domain. These kinds of components are also rarely written solely by an application programmer, but large application developers may often contribute to or improve the components. Finally, there are application-specific components. These are always written by the application developer, and may be difficult to reuse outside the application context for which they are created. These components may implement things like a specific graphical user interface for the application.

A *component framework* is required to instantiate components, manage their execution and interoperation, and provide semantics for component composition [69]. As we will see in the examples, the exact definition of a component is usually tied to the framework in which it operates [46].

Component-oriented programming has several important advantages over traditional techniques [69]. First, it has been shown to simplify the design of large applications by decomposing them into smaller parts. Second, it increase flexibility, as component-based applications may be recomposed and/or augmented in response to changing requirements. The drawback to component-based software is the increased time and effort required to design component interfaces, and to

ensure that components conform to the framework's requirements. Component frameworks may also entail some performance costs. Finally, use of an IDL often restricts the form of composed interfaces [54]. We examine this last drawback in more detail in Section 2.1.

Examples of component frameworks

In this section we examine component frameworks that provide connections between components written in multiple languages. There are other important component frameworks, including Enterprise Java Beans (EJB) [31, 72] and Microsoft's Component Object Model (COM) [1] that we omit here because they do not support multi-language programming.

CORBA The Common Object Request Broker Architecture (CORBA) [84] is a large and popular component framework standard. Components in CORBA are called "objects," although they are quite different than the notion of objects in object-oriented programming languages. In particular, CORBA components have a unique identifier, are created once, and then are accessed only through an interface which is defined in CORBA's IDL (described below).

The CORBA standard describes a framework that is based on an "Object Request Broker" (ORB), which acts as a backplane for inter-component communication. Every component in a CORBA system has access to the ORB. Many CORBA ORB implementations provide support inter-ORB communication, even to other ORB implementations [84]. The ORB has many functions, including services that allow components to be looked up either by name or by interface.

The ORB encapsulates almost all aspects of a component, isolating them from other components except for the IDL-defined interface. Hidden properties

include the network location of the component, implementation details including the programming language used to write the component, and the execution state of the component (uninitialized, idle, or currently servicing a request).

CORBA's IDL is simple by design, so that as many languages as possible may be used to write CORBA components. The IDL is used to construct an *interface*, which components may then choose to *provide*. An interface which can be thought of a set of functions with names and associated argument and return types. When a component provides an interface, it implements the functions described in the interface, using the appropriate types. CORBA can generate skeleton and stub code from an interface for any language that the implementation supports.

The IDL allows functions to take any fixed number of arguments, and to return a single value. Each argument and the return value must have a type, and the available types are specified by the IDL. The types include a set of precisely specified primitive numeric types (e.g., 8-, 16-, 32-, and 64-bit integers, booleans, 32- and 64-bit floating point numbers), and both ASCII and Unicode characters. There are fixed- and variable-length strings and lists. There are also constructed types, such as structure and union types similar to those in C, and a special "any" wildcard type. Finally, there is an interface reference type for CORBA interfaces, which can be used to pass typed references to components. Since many languages do not support pointers, the IDL intentionally lacks explicit pointer types.

The CORBA IDL requires that each function argument be marked as `in`, `out`, or `inout`, to indicate its directionality. Arguments marked `in` are effectively passed by value, and changes to the value within the function will not be propagated back to the caller. Arguments marked `out` are references; their initial value from the caller is ignored, but changes within the function will modify the referenced value

in the caller's context. Arguments marked as `inout` are treated like `out` arguments, but their value at input is not ignored.

For a programming language to support CORBA, a mapping must be defined from CORBA's type system to the language's type system, and vice-versa. Because the CORBA IDL specifies a fairly limited and common set of types, this mapping is straightforward for many languages. For example, in a C compiler, a CORBA string maps to a `char *`. In C++, it may be mapped to a `std::string`.

Language interoperability is realized by the CORBA IDL's type system, which acts essentially as a commonly-understood, intermediate format for data exchange across languages. Note that since the IDL includes a notion of references to other components, it also allows CORBA to act as a kind of FFI between any two languages that the implementation supports, allowing function calls from one language to another.

In practice, many popular languages provide support CORBA integration in the form of a mapping from their basic types to CORBA's IDL [84]. This support makes CORBA a practical choice for multi-language programming scenarios that require interoperation of two languages which do not have a dedicated FFI. It is also useful for situations where more than two languages are required.

CCA The Common Component Architecture (CCA) [17] is a component framework for high-performance scientific applications. CCA is strongly influenced by CORBA, and the two systems have much in common.

Scientific application developers are good candidates to adopt component-based software engineering. Reuse of highly complex and specialized scientific codes is highly desirable, and the nature of scientific research, especially the need for

repeatable experiments, encourages sharing [56]. The CCA was created to address these requirements.

CCA describes component interfaces using an IDL called called SIDL [80]. SIDL is essentially an extension of CORBA's IDL that adds types of particular interest to scientists. These include multi-dimensional arrays (with either fixed or dynamic size) and complex numbers. A tool called Babel implements a SIDL parser, and can generate stub and skeleton code in C, C++, Fortran 77, Fortran 90, and Python.

In CCA, interfaces are called *ports*. A component may *provide* a port, which means that the component implements functions that match those described in the port. A component may also declare that it *uses* a port. This means that the component requires an implementation of a component that provides that port to be loaded in the component environment. This system allows for different components that provide the same functionality (i.e., provide the same port) to easily be swapped in and out of an application [17].

CCA does not currently support type mapping (see Section 2.2) beyond the default primitive and structure maps in CORBA, although recent work has been moving towards this goal [49].

Cactus Cactus [37, 38] is a component framework with a focus on scientific applications. It does support components written multiple languages, but it does not use an IDL. Instead, it limits the languages that components may be written in to C, C++, and Fortran, and uses their existing interoperability facilities for inter-language component communication. In particular, C++ can call C functions directly since C++ is derived from C and shares much of its architecture, and Fortran has a bi-directional FFI to C (see Section 2.1).

Remote Procedure Calls

Remote procedure call (RPC) is an approach to inter-process communication, where each participating processes is assumed to be running on a separate computer connected by a network [43, 24, 83, 42]. The mechanism disguises itself as a procedure call, but after the call is made, the RPC system packages the function and its arguments in a binary message, and sends it to be handled by some other process. How the receiving process is chosen and located is a feature of the particular RPC system, and in general the receiver may be running on a different machine. After the message is received, the function and arguments are decoded, the function is executed, and the return value is passed back to the calling process. When the caller receives the response, it decodes the value and returns it via the regular function call return mechanism. From the caller's point of view, this entire process is indistinguishable from a regular function call.

RPC's straightforward semantics have made it a popular approach to distributed programming [83]. Also, and more importantly for our purposes, RPC systems usually allow for multi-language programming [43]. Their use of IDLs to describe exposed procedures has the effect of abstracting language-specific details, allowing for RPC to work across any language that supports the particular RPC protocol.

In fact, RPC can be thought of as a general approach to handling heterogeneity in computing systems [70], since it can be designed to abstract different operating systems, machine architectures, programming languages, and networking, all under the guise of a simple procedure call.

Both component frameworks and most RPC systems make use of IDLs [43]. In RPC, the units of interoperability described by the IDL are procedures or functions rather than components, but the approach is very similar.

Two models of RPC semantics are popular: blocking and non-blocking [83]. The blocking model uses strives to emulate the semantics of a normal procedure call, i.e., from the point of view of the caller, execution is suspended until the call returns. In the non-blocking model, the call is still made normally, but for the caller, execution continues immediately and does not wait for the remote call to return. Later, the caller can use the RPC's API to check whether or not a response has been received, and to collect the return value if one is available. Many systems provide both blocking and non-blocking calls. For the purposes of language interoperability, the distinction is immaterial.

RPC designs generally favor hiding the details of the network communication from the caller. Ideally, the caller need not even be aware of whether a call is remote or local. This abstraction principle is somewhat leaky, however. First, RPC calls must generally avoid passing arguments that are tied to the local address space (e.g., a pointer to an array), since the callee might be located in a separate address space. Second, the possibility of network errors implies that RPC systems must handle failures that regular procedure calls do not [24, 70]. For example, the correct semantics for RPC are unclear if the network stops functioning entirely, and must be defined by the implementation or standard. More subtle network failures are also possible, and RPC systems must be careful in the design of their protocols to ensure that, for example, failures do not cause a single call to result in more than one execution of a remote function [81].

There are several issues in RPC systems that we will not examine here because they are not relevant to our discussion of multi-language programming. In particular, RPC systems are often concerned with their performance under different network configurations and parameters, as well as the security implications of exposing program functions on the network [83].

Examples of RPC systems

In this section we will examine several RPC systems and show how they facilitate multi-language programming. One important RPC system, Java Remote Method Invocation (RMI), is omitted because it is tied to Java and does not directly support multiple languages [89].

Sun RPC Sun RPC introduced the "External Data Representation" (XDR), an influential and at the time innovative IDL design [29]. Sun RPC was originally designed to enable network distributed function calls to and from C, and not necessarily for inter-language function calls. So, XDR's data types look a lot like those of C. In addition to the usual primitive types (integers, floating point, characters, and so on), XDR supports C-style structures and unions, although these may only be one level deep (e.g., no `struct`s within `struct`s). XDR also supports fixed-length strings, as well as fixed- and variable-length arrays of primitives. Finally, XDR supports an "opaque" data type, that represents a sequence bytes guaranteed not to be modified by marshalling and unmarshalling [82]. XDR does not support explicit pointer types, since a raw pointer value has no valid interpretation outside its local address space. The mappings to and from these types and their representations in C are fixed by XDR.

XDR was designed for C, and C's native data representation is tied to the machine architecture and compiler. This explains why XDR's binary data representation is so precisely specified, since it must accommodate marshalling between data formats that may have different integers lengths, different endianess, different ways of packing `struct`s, and so on. This is also why XDR is a useful system for interlanguage communication – by abstracting the data transport representation from the in-language representation, and keeping its supported set of types minimal and fairly universal, XDR becomes a language-independent standard.

ILU Inter-Language Unification (ILU) goes beyond traditional RPC systems in two ways. First, it explicitly focuses on facilitating multi-language programming through an RPC-based approach. Second, it creates a system where objects may be passed by reference through the IDL interface [52]. Objects in ILU are not merely static collections of data; as in OOP languages, object instances have a unique identity, may have methods, and those methods may be invoked anywhere a valid reference to the object is held, even from other languages.

ILU works much like Sun RPC, but adds a notion of *modules*. In an application, there may be only one instance of a module, and a module has exactly one interface, specified in the IDL. Modules reside permanently in one address space, on one machine, and each module is written using one language. An application consists of a set of module instances.

In addition to regular procedures, modules may expose objects through their interface. Objects have a type, which is specified by another IDL interface with a set of methods. Object instances are owned by the module that exported the object's type. However, a *reference* to the object may be obtained by other

modules. This reference may be used to invoke the object's methods, which are executed via RPC to the owner module.

ILU provides a garbage collection scheme for objects. The module that owns an object instance is responsible for disposing of the object when there are no longer any references to the object. To manage this in ILU's distributed environment, ILU must arrange for the owner module to keep track of external modules holding references to its object, and periodically query them over the network to see if the reference is still held.

The ILU implementation supports modules written in Modula-3, FORTRAN 77, C++, Lisp, and Python [52].

XML-RPC XML-RPC [8] was designed to be a simple RPC protocol that would be easy to support from a number of languages. It uses an XML format instead of binary to represent a marshalled data, and provides two formats: one for function call requests and another for responses. XML-RPC does not provide a dedicated IDL; instead it describes a set of data representations in terms of XML, and expects implementations to define their own mapping from the language to those representations.

XML-RPC provides XML encodings for the usual primitive number, boolean, and character types, as well as variable-length strings and arrays of primitive types, and structures of arbitrary nested depth.

XML-RPC specifies an extra field in the response encoding, the presence of which indicates that an error occurred during the function execution. The interpretation of the error code depends on the implementation.

Web Services Web services are a general-purpose approach to "distributed services" [5, 30]. In practice, web services are used to provide RPC functionality on top of web protocols and standards like XML and HTTP.

The design of web services encompasses three orthogonal components required for distributed services. These are a communication protocol to encode data (analogous to a format for marshalled data in RPC), a way to describe services (analogous to IDLs), and a method for discovering services on the network.

SOAP is the most common communication protocol used in web services [5, 30]. Like XML-RPC, it is based on XML, and describes formats for requests (function invocations) and responses (function returns). SOAP is more complex than XML-RPC, and specifies formats for meta-data like security credentials.

The XML standard has a specification, called XSD [23], for encoding primitive and structured data in an XML format. These include numbers, booleans, structures, arrays, and so on, as well as some higher-level types like dates and times. XSD also supports construction of new data types from this primitive set using sequencing, discriminated union, and restriction. This last mechanism is unique, and interesting: it specifies or restricts ranges of valid values that the underlying type may take. While decidedly more complex and verbose than the data encoding used by XML-RPC, XSD has the advantage that encoded data can be checked for correctness at runtime using XML processing tools that are available for a wide variety of platforms and languages. For example, integers can be verified to be within the representable range, strings can be checked to contain only valid Unicode characters, and so on.

A second XML format, called WSDL [30] provides a way to describe the web service interface. WSDL, then, acts like the IDL in a traditional RPC mechanism.

A WSDL specification defines a set of valid *messages* in terms of XSD types. These messages are used by a set of *ports*, each of which contains a set of *operations*. Each operation describes a sequence of valid request and response messages. These operations might be similar to regular RPC semantics (i.e., a caller request, followed the callee's response), asynchronous messaging (i.g. just the request, with no response expected), or some other protocol entirely. WSDL therefore trades simplicity for flexibility; it can describe very complex protocols.

A WSDL description also contains a concrete binding, that tells the service what encoding and transport protocols to use (e.g., SOAP over HTTP). The concrete binding also maps operations to URLs, which gives them a globally unique endpoint for communication.

Like an IDL, the WSDL specification is processed through a tool that generates a skeleton for the implementation of the service in the desired target language. A stub implementation can also be generated from the WSDL. Typically, WSDL for a service is made available on the internet, and clients who wish to use the service may download the WSDL interface and generate stubs in the language of their choice.

While complex, the XML standards that web services are built upon are standardized and implemented in a wide variety of languages and systems.

Thrift Thrift [79] is a recent RPC system developed at Facebook, with a focus on multi-language interoperability. It uses an IDL that, in addition to the usual primitive and structured types, includes data structures, such as maps and sets, common in modern "scripting" languages (e.g., Python, Ruby).

Thrift abstracts the marshalled representation of this data with a functional interface. So, marshalled data structures may be constructed or picked apart using

and API with functions like `writeInt`, `writeStruct`, or `endStruct`. This API has been ported to C++, Java, Python, PHP, and Ruby, so Thrift can marshall data to and from a variety of languages.

Instead of using the API functions, most users of Thrift describe data structures in an IDL. The IDL is processed in the usual way, producing marshalling and unmarshalling routines for the desired language. These routines are generated with calls to the API. The benefit of this approach is that the API routines may be rewritten, and the underlying data representation altered, without requiring any changes to the client code.

The IDL can describe functions, and these have the same semantics as RPC. Functions that have no return value may be marked with the `async` keyword, which allows callers of that function to continue execution without waiting for the call to return.

Discussion

Relatively little effort is required, in general, to implement a mapping of a language's basic data types to those of an IDL, and to enable the IDL tools to generate stub and skeleton code for that language. This allows IDL-based approaches to multi-language programming to scale well – if you have n IDL-mapped languages, then you have n^2 language bindings, since any language in the set can interoperate with any other [54].

The cost of this approach is a lack of specificity to any particular language and set of types. In particular, domain-specific data types (e.g., complex numbers, matrices, images, and so on), will need to be expressed in terms of the simple primitives and structures that most IDLs offer, and not the more natural high-level

types that some languages may offer. IDLs cannot easily get around this limitation, because the types they define must constitute, in some sense, a *lowest common denominator* type system across any and all languages that wish to participate [54]. If an IDL included say, a type for images, then each language would have to be able to decode that image type into meaningful data (or accept an incomplete mapping, but this defeats the purpose of interoperability). As we will see in Section 2.2, customizable *type maps* can help to resolve this issue.

IDL-based interoperability systems usually require marshalling for complex types, even for communication between components written in the same language. Therefore, use of an IDL may be inefficient compared to other approaches.

Language-neutral intermediate representations

Language-neutral intermediate representations (neutral IRs) are, in some sense, similar to IDL-based approaches. Both work by constructing a "common ground" that participating languages must be able to interface with. This approach connects each language, by transitivity, to every other language that targets the same IR.

In the language-neutral IR approach, each language is compiled to some target language (the IR) that is general enough to encode the data representations and semantics for a variety of languages. The IR serves as the mechanism for interoperability. The nature of the IR determines how interoperability mechanisms are exposed in each languages. Neutral IRs, then, can be seen as providing a framework or mechanism for multi-language interoperability, on top of which other strategies (including FFIs or language integration) can be applied.

Examples

The following systems all feature multi-language interoperability facilitates by a language-neutral IR.

UNCOL As far back as 1958, there was interest in developing a "universal compiler IR," called UNCOL [65]. UNCOL was more a concept than an actual proposal, and it was never successfully designed. The idea was proposed as a means to reduce the effort required to write compilers, which was at the time considerable. Machine architectures at the time were not standardized, and language features like records, pointers, and data types were still novel. A universal IR was thought to be a partial solution – languages could target the IR, which would then be portable to many hardware architectures. Language interoperability was hardly considered, but was mentioned as a potential extra benefit.

Microsoft Common Language Infrastructure Microsoft's Common Language Runtime (CLR) [9, 41] is a language-neutral IR and execution specification, similar to Java's virtual machine bytecode [62], but designed to support many different languages. It is one of the foundational technologies for Microsoft's ".NET" platform. The CLR supports compilation and execution of procedural, functional, or object-oriented paradigms languages. CLR's language-neutral bytecode format is called Common Intermediate Language (CIL), and it has a language-neutral type system called the Common Type System (CTS).

Like the JVM, the CLR provides high level services like garbage collection, a class loader with security features, and an extensive (and language-independent) class library providing many common data structures and algorithms.

CLR programs are self-describing, with extensive annotations describing the types, fields that are read-only, and so on. These annotations are packaged along with the compiled CIL code. This allows compiled CIL code to be distributed independently of the source program that generated it, but still inspected and used by other CIL modules.

When executed, the platform-independent CIL is translated to platform-specific native code. CIL is a stack-based model, and although it retains type annotations as meta-data to be used by compilers and other tools, the execution engine ignores the types entirely.

The CIL is essentially a stack-based form of assembly language. Unlike assembly languages, however, it is not tied to a particular ISA, and is designed to be interpretable on any modern CPU.

Although similar to JVM bytecode, CIL has some important differences that facilitate its language neutrality. Unlike the JVM, unsafe CIL codes can be generated and are permitted to be executed. This enables languages like C++ that feature unsafe pointer arithmetic to interoperate. Also unlike JVM bytecode, CIL permits global variables, function pointers, and the ability to pass primitive parameters by reference. The garbage collection algorithm is required to support pinning data (see Section 2.1), which guarantees that pointers may be used on memory managed data.

The CTS provides a language-neutral type system for the CLR. There are two kinds of types in CTS: value and reference types. Value types are bit sequences (e.g., integers), allocated on the stack. The value type describes what operations are appropriate, but values do not carry their type information, so type checking must be done statically. References types are similar to object references in Java.

Values of reference types carry their type information with them, and so can be checked dynamically.

Like Java, reference types are part of an inheritance hierarchy, and must inherit from exactly one parent. Value types exist outside the hierarchy. The CLR has a notion of *interfaces*, which are sets of methods. Inheriting from an interface implies that the reference type implements all the methods described in that interface. Inheritance from multiple interfaces is allowed, and the sub-typing rules permit a reference type that inherits an interface to be used wherever that interface is expected.

Notably, although the CTS defines inheritance behavior, it intentionally omits method overloading. This is because different languages have different overloading rules. It is relatively easy for languages that wish to support overloading to implement it themselves by mangling method names.

To support language interoperability, CLR defines a Common Language Spec (CLS), which is a subset of the CTS type system. At a minimum, CLS compliant languages must be able to import and use CLS types, which include object types as well as a set of primitives. Notably, CLS-compliant languages need not be able to *extend* object types via inheritance, or even to define new object types. CLS specifies some other constraints as well, for example it mandates an interoperable naming scheme for identifiers, and disallows architecture-specific primitive types. Compliance with CLS is not required, it guarantees interoperability with any other CLS compliant code. In some ways CLS is like an IDL, but an exceptionally rich one.

Error handling in the CLR is done through exceptions, so compliant languages are required to either handle or tolerate exceptions that are thrown

to them. Exceptions are implemented by a two-pass stack unwinding: first the stack is searched for a handler, and then the second pass performs cleanup before invoking the handler. Exception information is stored in fixed-format table that precedes each method entry in the CIL format. The table contains a pointer to the handler as well as a discriminator indicating whether the handler may re-throw the exception or simply terminate. Exception handling is late-bound; no work is done until the exception is thrown, and then the stack is searched. With this system, an exception can be raised in one language and caught in another.

Targeting a particular language to the CLR can be easy or difficult, depending on how well the language's concepts map onto the CLR's infrastructure. For example, the CLR does not support nested functions, multiple inheritance, or callcc, so languages with these facilities must either omit the feature, or transform the feature into CLR-compatible terms. Moreover, in order to interoperate with other languages in the CLR, a language must support the minimum requirements of the CLS. This can be awkward. For example, support for SML required an extension to the language to support object types.

LLVM LLVM [58] specifies an intermediate code representation based on a 3-address, RISC-like architecture, but abstracted from actual hardware. It also provides a compiler infrastructure to transform that representation, including compilation to real hardware.

LLVM's representation is language independent; it is only slightly richer than a RISC-like assembly language. It does include a type system, but the types may be treated as annotations, and do not prevent definition of programs that ignore the type information. LLVM does not impose any particular runtime requirements on programs, and does not provide high-level runtime features (e.g., garbage

collection) directly. This makes LLVM a very different kind of system than the CLR or JVM, which provide many high-level features but also usually require that compliant languages use them.

The LLVM IR has just 31 opcodes, but most have overloaded semantics based on the types of their arguments. LLVM does not permit type coercion; types must be cast explicitly to other types if desired, so language features like implicit coercions must be compiler directed. LLVM's IR uses an infinite set of virtual registers. The definition of a register always dominates its use, i.e., registers have single-assignment semantics. The IR includes load and store opcodes to access a heap. Control flow in LLVM is explicit: functions are basic blocks, and each block ends in either a branch, a return, or one of two exception opcodes (see below).

LLVM's type system is designed to be language-independent. Every register and heap object has an explicit type, and opcodes work differently with different types. The type system includes the usual set of primitive types as well as four kinds of derived types. These are pointers, arrays, structures, and functions.

LLVM is designed to support weakly-typed languages (e.g., C) so declared type information in an LLVM program may not be reliable. In particular, there is an unsafe opcode that will cast any type to any other type. The cast opcode is the only way to convert types; this implies that programs without the cast opcode are typesafe. [1] Heap address arithmetic uses a dedicated opcode (instead of the general-purpose addition opcodes) that preserves type information.

LLVM uses a flexible, language-independent scheme for exceptions. The IR includes two special opcodes called invoke and unwind. Invoke works like a regular function call, but takes an extra basic block argument that represents an exception

[1] Lack of the cast opcode does not prevent memory errors, such out-of-bounds memory or array accesses, from occurring.

handler. When the unwind opcode is executed, it works up through the call stack until an invoke opcode is found. Execution then transfers control to the exception handler block of that invoke.

Type information allows a range of aggressive transformations that would not otherwise be possible, e.g., reordering fields; optimizing memory management. These can only be done with *reliable* type information however, so they include an algorithm (Data Structure Analysis) that uses the declared types as speculative, and conservatively checks whether load/stores are consistent.

Moby Moby is a functional programming language that supports interoperability with C through its compiled intermediate representation, called BOL [34, 77]. BOL is more expressive than Moby itself; in particular, it can also be used as an IR for C. The two languages have very different features, but can interoperate using BOL.

The BOL IR framework serves as a *mechanism* for interoperability, but does not define a *policy*. The distinction is important. The policy determines how low-level data structures are represented and manipulated in the high-level language. The mechanism, by contrast, exists to reify the policy. Ideally, if the interoperability mechanism is both flexible and powerful, it may support many different kinds of policies.

BOL is an extended, low-level lambda calculus. It has a weak type system, designed to be almost equivalent to that of C, but lacking C's recursive types. Type constructors in BOL include enumerations, pointers, arrays, and structures. BOL code can make C function calls and work with C data types directly. No marshalling is required because there is a direct mapping from BOL's types to those of C.

Types in Moby can be defined in terms of BOL types; this is Moby's primitive types are defined. Primitive Moby functions can also be defined in BOL; this can be used, for example, to wrap the C standard library for use in Moby, since BOL can call C functions directly.

There are two separate ways to access C from Moby, representing two distinct interoperability policies. The first is an IDL-based approach. Tools are used to parse a C header file and map the function signatures to Moby's type system. The header file may include some extra annotations to disambiguate the mapping of pointer types. Stubs are generated in BOL code that call the appropriate C functions, but map the types to Moby's high-level representations. Some marshalling code, also in BOL, may be generated if needed to implement the mapping.

The second interoperability policy is called Charon, which implements a type-safe embedding of C into the Moby language using phantom types. This allows Moby to manipulate C data structures and call C functions directly, without sacrificing type safety. Charon was inspired by, and is very similar to, NLFFI [25], which is discussed in Section 2.1.

Whirl Whirl is the IR used in the Pro64/Open64 compiler suite [7, 67, 61] and its many offshoots. Whirl is designed to be used as the input and output of every internal compiler pass. This makes it easy to reorder optimization passes in the compiler, which is important because the optimal order of the optimization passes with respect to some performance criteria may be different for different applications.

Whirl was designed to support a fixed set of languages, namely C, C++, Java, and FORTRAN 77, and it may be adequate for other languages as well.

Whirl code may be designated as being at a "level" with higher levels being closer to the source language, and lower levels closer to the machine architecture. The process of compilation, then, is a gradual translation of a program from the highest level to the lowest one. Each level is well-defined, i.e., there are constructs in the higher-level IR that must be eliminated before the level can be reduced.

At the highest level, Whirl code is very close the original language, and at this stage it is even possible to translate from Whirl back to the original language. Whirl supports this high-level representation through constructs that are specific to different language semantics. For example, Whirl contains a `DO_LOOP` element that corresponds to Fortran's loop semantics, as well as `DO_WHILE` and `WHILE_DO` elements for C. At the highest level, language-specific types are preserved as well.

While Whirl does provide a language-neutral IR for the languages it supports, it would have to be extended for an additional language to participate. Moreover, Whirl does not offer any special support for interoperability between the languages it supports. For example, Whirl does not bother to define an IR representation of C code calling a Java function, since it does not accept a source language that would admit this construct in the first place.

SUIF SUIF [90] is a compiler infrastructure that includes a flexible IR component. SUIF is particularly focused on facilitating parallelizing transformations. To detect the data dependencies required for parallel transformations, SUIF uses a fairly high-level intermediate representation. Lower-level IRs often erase high-level representations of loops and conditionals, and array accesses are expressed as pointer arithmetic, and this makes certain data dependence analyses more difficult or impossible.

The SUIF IR includes standard RISC-like operations, but also higher-level representations for various loops, conditional statements, and array accesses. The loops and conditionals representations are similar to those in an abstract syntax tree, but are designed to be independent of a particular language.

Because it preserves high-level constructs, SUIF may be a good choice for a language-independent IR. At the moment, the SUIF distribution includes front-ends for both C and Fortran.

Discussion

C is something of a *lingua franca* among programming languages, not unlike a universal intermediate IR. As we saw in Section 2.1, most mainstream languages have some kind of ability to interoperate with C. C enjoys this status for at least two reasons. First, it is a highly desirable language to interoperate with owing to the vast numbers of existing libraries it can access. Second, C may be used for programming tasks that high-level languages are ill-suited for, such as coding of performance-sensitive algorithms that can take advantage of specialized hardware. Third, C is a very low-level program representation while remaining platform-independent. It also has some convenient higher-level features, such as types. In some ways, though, C makes for a poor language-neutral IR, in particular because it lacks high-level features like garbage collection and exceptions.

In contrast to C, frameworks like the CLR have lots of high-level features, and support a more coherent form of interoperability because those features can be shared across languages. By the same token, however, requiring that these features be supported (or at least tolerated) by participating languages may constrain language implementations, and prevent different but useful approaches. This in

turn may limit the utility of multi-language programming, reducing language specialization to a matter of syntax in the most extreme scenario. In addition many higher-level systems such as the CLR or JVM require a large, complex runtime system to support their execution. The runtime must be ported, possibly at great cost and/or effort, to each platform where programs will be run.

Language-neutral IRs provide a mechanism that facilitates multi-language interoperability and programming. Some systems, like the CLR, use this mechanism to provide multi-language interoperability by designing participating languages with certain unified language semantics. Other systems, like BOL and Moby, use the neutral IR as a framework for standard approaches like FFIs. Each case, therefore, meets our goals for multi-language interoperability differently, but the use of a neutral IR impacts the goal of scalability. Like IDLs, neutral IRs allow any number of languages to interact through a single common representation – this reduces the number of required interfaces for n languages to $2n$. However, the cost to add each language amounts to the cost of writing a full compiler that targets the neutral IR.

Whether a neutral IR can accommodate multi-language type safety, offer a natural programming model, and operate efficiently depends on the system in question. The CLR, for example, goes to great lengths to fulfill these goals. It largely succeeds, at the cost of restricting language features. Systems like Moby take a different approach, allowing different interoperability policies to be layered on top of BOL. These policies, then, determine whether and how interoperability goals are fulfilled.

Language Integration

Integrating two or more languages involves reconciling the syntax and semantics of those languages. The combination may produce an entirely new language, or one language may be entirely expressible in terms of another.

Examples

The following systems are examples of systems that achieve language interoperability through an integration.

NLFFI NLFFI embeds C's type system within SML, allowing for data-level interoperability with C code [25]. It allows SML to call C functions, without the need to map types or marshall data, since ML can just pass the arguments in the representation that C expects. The tradeoff is that, since C's type system is not safe, ML programs that use NLFFI are no longer guaranteed to be safe either.

NLFFI uses a type system "trick" (see below) to set up a one-to-one correspondence between types in C and a set of types defined in ML. The ML compiler is modified to generate C code for expressions that have these types. The embedding is rather complicated; here, we give a few examples of the kinds of techniques they use.

NLFFI relies heavily on *phantom types*, which are type constructors used only to construct other types, and not assigned values. Consider the following ML type:

```
sig type bin
  type binary
  type 'a dg0
  type 'a dg1
```

```
end
```

These types can be used as a kind of "type language," to construct binary numbers. For example, the type `binary dg1 dg0 dg0` represents 100_2, or 4 in base 10. In NLFFI the technique is only slightly more complex. Specifically, NLFFI uses a decimal number representation instead of binary, and introduces extra type parameters to prohibit leading zeros, which ensures that each number corresponds to a unique type.

For example, to encode C's fixed-size array types, NLFFI uses phantom types in a one-to-one relation to non-negative integers to type fixed-length arrays, by using the phantom type to indicate the array's length. With this scheme, C arrays of the same length will have the same ML type, which conforms to C's typing rules.

As another example, NLFFI encodes C's `const` pointer type parameter by including a tag in the C pointer type. The tag itself is a phantom type that corresponds to two values, `const` or non-`const`. NLFFI provides functions to read and write data from a pointer, and these functions take the pointer type as an argument. While the read function is polymorphic in the `const` tag, writing is strict. This technique allows ML to statically reject code that attempts to write to a `const` pointer.

NLFFI does not address some of the usual aspects of multi-language programming. In particular, it does not allow C to access SML's data or functions. NLFFI is therefore a one-way integration.

Jeannie Jeannie [44] combines the syntax and semantics of C and Java, and allows each to be embedded in the other recursively. Programmers switch between languages using a special operator (Jeannie uses the backtick character to switch

from Java to C, or from C to Java). The Jeannie compiler produces two separate programs, one in Java and the other in C, and connects them with automatically generated JNI code (see Section 2.1).

The complete syntax of each language is supported, but the switch operator must be used explicitly to change from one language to the other. The switch operator may be applied to either statements or expressions. Jeannie adds some extra syntax for convenience, such as synchronized blocks and exception constructs like `try`, `throw`, and `catch` for C, as well as extra functions, such as a version of `memcpy` that copies a block of memory from Java's heap to C's.

Jeannie statically type checks the separate Java and C code according to their own typing rules. It also adds static checks across language boundaries, which are not normally performed on JNI code. For example, there are cross-language checks to ensure that checked exceptions are either caught locally or declared, and that private and protected Java methods are not called from C. Because it incorporates C, Jeannie is not type safe. Cross-language checks are an improvement over hand-written JNI in this regard, however.

Jeannie uses the same type equivalences defined in the JNI, but also checks that they are used correctly. C constructs like explicit pointers, `struct`s and `union`s have no equivalents in Java, and so Jeannie checks to make sure they do not cross over to Java without being explicitly marshalled.

Operators like `break` and `continue` cannot divert control flow across language boundaries, since the JNI does not support it. Other control flow operators, such as `throw`, are able to divert control flow because they have a semantics in C that is defined by the JNI (see Section 2.1).

MLj MLj [22] is a Standard ML (SML) compiler that generates Java bytecode. It includes an extension to SML, allowing it to call Java code. The "semantic gap" between the two languages is fairly small. Java and SML both have strong typing, similar basic types, and checked bounds. They use similar exception semantics. Both prohibit explicit pointers and have automatic memory management. There are differences, however. Java's objects and inheritance subtyping have no counterpart in ML. Java lacks parametric polymorphism and support for closures.

The goal of MLj's extension is to allow ML to call Java methods and handle Java objects as first-class entities, i.e., store them and use them as arguments and/or return values from ML functions. Furthermore, they allow Java classes to be constructed within MLj, and these classes may use ML objects freely. The complete formal semantics are given in [22]. Here, we highlight some of the interesting points.

MLj extends SML with Java types and terms. Primitive types in Java are equated with their respective ML types. So, in MLj, a ML `int` and a Java `int` refer to the same type. Some basic classes, like ML's `string` and Java's `java.lang.String` are also equated. Java's package hierarchy is equated with ML's modules, and Java's `import` syntax is mapped to ML's module `open`. Conveniently, Java's package syntax and ML's module syntax are identical, and so they work without modification.

Unlike in ML, which requires that values be bound when they are declared, Java allows reference types (i.e., objects) to be null. Therefore, MLj wraps Java object types within an option[2] type. In this scheme, a null reference is given the

[2]ML's option type is a variant with two constructors: `None` or `Just` α.

value `None`, while valid values are constructed with `Just`. This approach is similar to how the Haskell 98 FFI and GreenCard represent null pointer return values in C.

MLj allows certain implicit *coercions* to make the code more natural. Here, a coercion is an implicit type cast and conversion from one datatype to another. In particular, MLj allows a widening coercion on Java objects that implements Java inheritance rules. Specifically, for Java object types `T1` and `T2`, if `T1 <: T2`[3], then whenever `T1` is expected, it will be coerced to `T2`. This coercion is always well-defined for Java objects. Explicit casts, notably downcasts (i.e., a cast from `T2` to `T1`, where `T1 <: T2`), are supported for Java objects. As in Java, an invalid cast will throw an exception.

MLj also permits coercion from an option type `option` α to α, which simplifies programming with the null pointer mapping described above. This coercion will throw a Java `NullPointerException` if the value is `None`.

Java methods may be "overloaded," i.e., share the same identifier within a single class, and distinguished by their argument types. At an overloaded method call site, the specific method is determined by examining the supplied argument types. Overloading can be ambiguous, however. Consider the following Java class `A`, with overloaded method `foo`:

```
class A {
  void foo(String s) {System.out.println("1");}
  void foo(Object o) {System.out.println("2");}
  public static void main(String [] argv) {
    A a = new A();
    a.foo("hello");
```

[3]The notation `<:` is read "is a subtype of."

```
  }
}
```

The call to `foo` in `main` is ambiguous; the value `"hello"` can be typed as either a `String` and an `Object`, because `String <: Object`. Java handles this ambiguity by choosing the most specific method with respect to an ordering on types (roughly, the ordering is determined by the inheritance hierarchy, with extra rules for primitives). So, in Java, the example above will output `1`. Note that a *unique* most specific method may not exist, and in this case the Java compiler will terminate with an error.

Unfortunately, ML's type inference algorithm cannot accommodate the "most specific" technique of selecting overloaded methods, and so MLj discards it. The specific overloaded method must be unambiguously selected by the programmer, using ML's type annotation syntax if necessary to disambiguate argument types.

MLj's compiler produces Java bytecode, which is executed within a single Java virtual machine (JVM). Therefore, MLj's ML and Java portions share a garbage collection system, and so the special rules for pinning arrays and so on that we saw in Section 2.1 are not needed. By the same token, in MLj Java and ML share a call stack, and since ML's `exn` type and Java's `java.lang.Exception` type are equated, exceptions may flow freely up the stack from one language to the other.

Discussion

Integrated languages are a bit of a catch-all category for systems that achieve interoperability by combining two or more languages. The approach is clearly not scalable, because considerable effort is required even to define the syntax and

semantics of a language that combines two other languages, let alone implement a compiler. However, the result can accommodate a very natural programming style. It is especially interesting that the examples in this section are able to go beyond function calls as the main abstraction for interaction. They focus, instead, on expressions. MLj is even able to maintain the type safety guarantees of both ML and Java. Whether or not a given integrated language system is efficient depends on the system in question. Jeannie, for example, generates JNI code, and so is no more efficient than the JNI itself.

Typemap Tools

In multi-language programming, there is always the question of how to relate the data types in one language to the types in another. For low-level, primitive types such as integers and floating point numbers, the mapping is usually obvious because both languages are likely to have to types that correspond very closely, if not exactly. The question becomes more complicated, however, for higher-level data types. For example, consider the following C `struct` and function declaration:

```
struct Point {
  double x, y;
};


double norm(struct Point *p);
```

How would an IDL or FFI expose `norm` to, say, Java? One possibility would map the `Point` structure to the `java.awt.Point` class, and construct a `norm` method in Java like so:

```
double norm(java.awt.Point p);
```

There are other options, as well. For example, `Point` could be mapped to `java.awt.geom.Point2D.Double`, or even to a new point class, defined by recursively mapping individual elements of the structure to corresponding new instance variables. A *type map* is some data structure or algorithm that allows these cases to be disambiguated. Type mapping is a common feature across many kinds of multi-language interoperability systems, because it is difficult in general to match types across languages. These systems may be quite simple and/or inflexible. In particular, many systems require that user-defined types by translated to some simpler type with a known cross-language mapping, in order to be passed across language boundaries. The examples we cover here are more involved. In particular, high-level type mapping systems allow arbitrary user-defined types to be mapped to across language boundaries in ways that are, to some extend, customizable by the programmer.

Examples

In this section we will examine some examples of systems that have popular or interesting approaches to high-level type mapping.

SWIG

SWIG [21] is a "wrapper generator", intended to generate tedious glue code needed for higher-level languages to interact with lower-level ones. It has a particular focus on connecting so-called "scripting" languages, such as Python, Ruby, and Perl, to pre-existing libraries written in C. SWIG can be configured to generate any kind of glue code, so the exact output will depend on the FFI system

used. To call a C function from these languages, glue code must be generated that converts the function arguments from the scripting language representation to some representation in C, invokes the C function, checks for possible error codes, converts the return value to a high-level representation, and finally transfers control back to the high-level language.

To direct the code generation, the SWIG tool takes an interface file as input. The interface consists of ANSI C function prototypes and variable declarations. SWIG handles simple primitive type conversions automatically. By default, SWIG does not perform sophisticated type mapping. Instead, it simply converts pointer types in C to an "opaque pointer" type in the high-level language. Opaque pointers are represented as the pointer's contents (i.e., an address in memory) encoded as hexadecimal strings. Null pointers are converted to the string value "NULL". This scheme treats pointers as opaque handles, which may only be used by passing them to (appropriately typed) wrapped C functions.

If a more involved mapping of pointer types is desired, SWIG has a notion of type mapping functions [6]. A type mapping function is a block of glue code that implements a conversion from a low-level C data type to some high-level type in the scripting language, or vice-versa. In general, to convert from C to a high-level language and back requires a pair of symmetric type maps. The code in a typemap can be arbitrarily complex. For example, it could pick apart a C structure, field by field, instantiate a new object in Python, set some fields in the Python object instance, and then return the object.

To insert the code block that implements a type conversion, SWIG must be able to recognize the data types to which it applies. SWIG uses a regular expression pattern, paired with each type map definition, to match types lexically.

SWIG extends the pattern matching to account for things like aliased types. Once a type is matched, the type mapping code will be inserted into the generated glue code at the appropriate point to convert the data type.

Parameterizing SWIG with a set of typemaps defines a type mapping *policy*. By changing the typemaps, the policy will be changed. SWIG is distributed with a set of default type maps, but users are free to define their own and augment or replace the defaults.

We compare a typemap written in SWIG to an equivalent one written in our own language, Twig, in Section 7.1.

FIG

FIG is a tool used to wrap existing C libraries for Moby [77]. Since Moby already uses BOL as a language-neutral IR (see Section e.g.,), the mechanism for interoperability with C is already in place. Fig's job is to fix a policy that maps types in C to types in Moby, and is able to generate the BOL code that implements the conversion.

The input to FIG is a C header file along with a script that is used to guide the translation of types. Internally, FIG represents the contents of the header file (i.e., function type declarations) as a set of typed terms, and transforms the terms from C to Moby through the application of *term rewriting* rules (see Appendix 2.3). Typemaps in FIG are built from rewriting rules whose input terms are BOL types, and whose output is either another BOL type or $\bot$, which indicates failure. The use of BOL's language-neutral type representations allows type terms to be uniformly represented in BOL, while still translating between Moby and C.

A typemap in FIG is a 4-tuple, consisting of a high-level type T_m, a low-level type T_c, a marshalling function that converts a value of type T_m to a value of type T_c, and an unmarshalling function that converts values from T_c back to T_m. The marshalling and unmarshalling functions are defined in BOL. When FIG needs to generate conversion code for a type, it simply checks to see if a registered typemap takes that type as input, and then inserts the marshalling or unmarshalling code to perform the conversion.

FIG also defines a combinator language that allows simple typemaps to be composed into more complex conversions. The composition rules are based on term rewriting *strategies*, and System S in particular (see Section 2.3). Strategies allow flexible composition of typemaps, including simple constructions like sequencing (i.e., the output of one typemap is sent to the input of another), or more complex rules like mapping a typemap across the elements of tupled types. Typemap combinators are a useful and powerful feature that distinguishes FIG from other approaches to typemapping. We adopt FIG's combinators in Twig's language, as described in Chapter IV.

FIG also has a pattern matching feature, not unlike that of SWIG, that can select terms based on identifiers or other lexical constructs. This can be used, for example, to disambiguate cases when two typemaps may apply to the same type, or to apply certain typemaps only when particular naming conventions are being used in the header file.

Twig was partially inspired by FIG. We designed Twig to be similar to FIG in terms of its language, but with the ability to generate code for target languages other than Moby.

PolySPIN

PolySPIN is a multi-language interoperability system designed for object-oriented languages [20]. It seeks to make interoperability for any number of OOP languages totally *seamless*, i.e., such that a programmer need never be aware that they are working with objects written in a different language. PolySPIN rejects the IDL approach, since it requires programmers to define an interface in terms of the IDL. Even if the interface is automatically generated, it does supports only a limited type system, and so using the interface is unnatural when high-level types, including objects, are involved.

PolySPIN's approach to interoperability is based on automatically generated mappings between high-level types, using a process called "type matching". Matching objects are pairs of inter-language object types that have equivalent, or roughly equivalent, interfaces. PolySPIN implements this check using *signature matching* [92]. Strict signature matching works by equating a set of primitive types across languages (e.g., a Java `int` is defined to match a C++ `int`), and then recursively equating constructed types, including method signatures and objects. Matching can also be relaxed in various ways; for example, an object may match another object if they match strictly except that the first object contains additional methods that are not in the second object. Matching criteria in PolySPIN can be defined by programmers, and may include lexical criteria such as a type's identifier.

To make use of matching, PolySPIN introduces a system that binds a unique name to each object in a multi-language application. The methods of each object are modified so that they consult this binding whenever they are invoked. If the underlying object is instantiated locally in the calling language, the method is invoked normally. But it is also possible that the object is acting a facade to a

matched object in another language. In this case, when a method is invoked, PolySPIN redirects the method call back to the underlying object in the original language. Facade objects may be obtained by passing regular objects across language boundaries, for example in a method's arguments. In any case, when an object is passed to a different language, a matching type is found, and is used as a facade to the original object.

It is worth noting one drawback of PolySPINs approach – because PolySPIN must modify the methods of each class, the full program source code must be available. So, existing libraries for which only the compiled code is available cannot be used with PolySPIN.

Discussion

The systems in this section illustrate the use of type maps in multi-language programming. Type maps are best viewed as a feature of these systems, rather than a general approach to interoperability. The goal of high-level type maps, generally, is to make programming in multi-language applications more natural by allowing programmers to use familiar high-level types instead of a restricted, predefined set interoperable types. As with all marshalling code, high-level type maps may be at odds with efficiency, since they potentially introduce extraneous transformations.

Term Rewriting

Term rewriting is a formal technique for reducing *terms* to some normal form with respect to a set of rules [18]. Much of the interest in term rewriting stems from its utility in program transformation [86].

Terms are built from *variables* and *constructors* [18]. Variables are placeholders for other terms. Constructors are functions from terms to a single term, with some unique constant symbol and arity $n \geq 0$. We adopt the convention that constructors symbols begin with lowercase letters (e.g., `cons`), while variables are usually represented by a single uppercase letter (e.g., X). If a constructor has zero arity, we refer to it as a *constant symbol.* For example, x and y are both terms, and if `cons` has arity 2, then $\texttt{cons}(x, y)$ is also a term. A *signature* is a set of function symbols with associated arities. Notably, the terms of a properly defined signature can be used to represent the abstract syntax tree of a program.

Rewrite rules have the form $t_1 \rightarrow t_2$, and define a transformation of a term matching the *pattern* t_1 to a term t_2. If the pattern t_1 contains variable terms, these will be bound as a result of a successful pattern match. These bindings may then be used in the construction of t_2.

Given a term t and a set of rewrite rules R, we say that t contains a *redex* s if s is a subterm of t and if s matches one or more rewrite rules in R. A term is in *normal form* with respect to R if it contains no redices.

The task of most term rewriting systems is the reduction of terms to a normal form with respect to some set of rules R by applying rewrite rules to subterms until no more rules may be applied. In general, a term may contain many redices; one challenge for term rewriting implementations is to devise a *strategy* for deciding when and where to apply each rule within a term [86].

If repeated application of a set of rewrite rules is guaranteed to eventually reduce to a normal form, then the rules are said to *terminate.* For example, consider the rule set containing the single rule $\texttt{f(X)} \rightarrow \texttt{f(f(x))}$. It is easy to see that this rule will never terminate, because every application grows the term and

the redex still applies. Unsurprisingly then, for a given a set of rules termination is undecidable in general [18].

If a set of rules is terminating, and repeated application of a set of rules will always produce a *unique* normal form, then the rules are said to be *confluent*. Like termination, confluence is undecidable in general. However, for rules which are known to terminate, confluence is decidable [18].

System S

System S is a language that provides the basic machinery for term rewriting [87]. It can be used to implement different term rewriting strategies, that is, methods for deciding which redices to reduce, and when to reduce them relative to other redices. As we will discuss in Chapter IV, System S is the basis for Twig's language.

Terms in System S are defined formally, but the rules for constructors, variables, and pattern matching are very similar to the canonical definitions given above, so we omit the details here.

In System S, a rule s is a binary relation $\xrightarrow{s} \subseteq \mathcal{S} \times (\mathcal{S} \cup \bot)$, where $\mathcal{S}$ is a set of terms and $\bot$ is a special term indicating failure. Specifically, we say that $t \xrightarrow{s} t'$ succeeds if and only if $(t, t') \in \xrightarrow{s}$, and $t' \neq \bot$. If $(t, t') \in \xrightarrow{s}$, but $t' = \bot$, we say it fails. If $(t, t') \notin \xrightarrow{s}$ then $t \xrightarrow{s} t'$ is undefined in s.

Atomic rules are just primitive defined rules $t \rightarrow t'$, and rules may be defined to fail. System S introduces a set of compositions on rules that allow new rules to be created from existing ones. The complete semantics for these compositions are given in [87]. The compositions include a sequencing operator ;, where for rules $s1$ and $s2$, $s1; s2$ applies $s1$ first and, if it succeeds, applies $s2$ to the result (and fails

otherwise). Other combinators include deterministic and non-deterministic choice, a fixed-point operator that allows recursive composition, as well as a set of operators to traverse subterms by enumerating them. Twig makes use of these operators as well, although we extend their semantics to support code generation.

Twig (Aho et al.)

Aho's Twig [13] is a language (not to be confused with our own language, also called Twig) that uses tree rewriting rules to specify code generators for compiler back-ends. Twig's rewriting rules operate on intermediate representation trees, which encode a program at the level of the target machine. Rules specify a *template* tree, a *replacement* node, a *cost function*, and a block of object code that will be emitted as a side-effect of the rule being applied. Emitting code in this way is similar to our own scheme, although in our language the ordering of side-effects is controlled by the user through rewriting strategies, whereas in Twig the ordering is decided through a combination of depth-first traversal and dynamic programming based on the cost function for each rule. The cost function allows Twig to generate object code that is optimal, up to the accuracy of the given costs.

Twig differs from our own language in that the rewrite rules are applied to program trees. In our Twig, rewrites are applied to types. Our Twig is able to take a somewhat looser approach to rewrite rules, e.g. rules need not reduce to a single node and may contain variables. Our Twig has this freedom since we rely on user-provided strategies to control rule application. Their Twig's use of cost functions to control application provides an interesting metric for comparing application strategies – it would be quite interesting to try to incorporate something similar into our own language.

Workflow Programming

Workflow programming refers to a rather broad category of languages and tools. Very generally speaking, workflows are programs constructed as graphs by connecting the inputs and outputs of various *components*. The components act in some way on their inputs to produce outputs, and the wiring determines the flow of data through the program.

We take special note of workflow programming in Twig because our code generation semantics, described in Chapter III, are based in part on our own previous work in designing the WOOL workflow programming language [50]. Twig builds upon and expands this work, and in particular provides a more robust formalization of the programming model.

In this section we review some popular workflow languages and systems, including WOOL, and describe how WOOL influenced Twig's code generation model.

Workflow Languages

There are many existing languages and tools for describing workflows. Most provide complete systems that include languages for describing workflows, a way of programming activities, and a runtime engine for executing a workflow.

Many earlier workflow systems were designed to address the needs of business users, and employed coordinated web services as a enactment back-end. Examples of this style of workflow system include BPEL4WS [15] and WSFL [59]. These languages were designed to support and coordinate activities accessed through XML-based web services, and so while they are good at describing and enacting

workflows in this particular domain, they are not very useful for abstract workflow representation.

Other workflow systems have focused on scientific workflows. These are intended for use by scientists who want to focus on their problem domain and leave the low-level details to the workflow system. Scientific workflow tools delegate the often-tedious programming needed to connect and orchestrate series of computational steps to the language and/or runtime system.

Triana [28][66] uses a visual scientific workflow language that includes both data- and control-flow constructs. Triana, like other XML web service-oriented workflow systems, has a type system based on XSD schema datatypes [23]. XSD datatypes are powerful and flexible, but introduce additional complexity.

Taverna [71], part of the myGrid project, is a scientific workflow system focused on supporting life sciences experiments. Activities are implemented either as web services or Java classes. Taverna relies on an XML-based language called SCUFL for workflow specification. SCUFL has a type system, but data types are restricted to MIME-types, names from the myGrid bioinformatics ontology, and free form text.

VisTrails [26] is another system support scientific workflows. It is interesting because it keeps extensive provenance information for both the data being processed as well as workflows themselves. This allows VisTrails to treat workflows as a kind of scientific notebook, documenting the evolving scientific process. VisTrail uses a visual workflow language, and is focused on workflows intended to be executed immediately and interactively.

Kepler [14] inherits a visual environment and the Modeling Markup Language (MoML) from Ptolemy [51], and adds scientific workflow features like the ability to

test a workflow without needing to completely program all its activities, distributed execution with a web-services framework or Globus grid [36], database access, and other specialized actors.

Other workflow systems are designed to let users easily harness the power of grid computing [35]. One example is WFEE [91], which uses a relatively simple workflow description language (called xWFL) with grid-specific constructs. WFEE features support for workflow parameterization using filenames, ranges of number, and constants, which is important for scientific workflow applications. Another example is GSFL [57], designed for Globus OGSA-based grids.

The Abstract Grid Workflow Language (AGWL) [32] was designed to specify workflows in a way that balances abstract representation with enough information to execute the workflow in a real environment. AGWL makes parallelism an explicit construct in the language. This allows for a high degree of programmer control at the expense of abstractness of the workflow specification. Explicit parallelism may also increase the required level of sophistication for workflow programmers. AGWL workflows are executed on a portable back-end system called CGWL [33]. CGWL must be ported to a particular platform, and acts as an interface between the platform and the workflow.

WOOL

We designed WOOL [50] as a language for specifying workflows abstractly, similar in spirit to AGWL. To this end, the language deliberately excludes information related to the runtime system. WOOL has an intentionally simple syntax and semantic interpretation. Workflows are composed of “activities,” which are basic “units” of computation. Each activity has a type which assigns it a set

of input and output ports and other properties. Connections between the ports on activities, from outputs to inputs, establish data-flow relationships. WOOL workflows can be composed hierarchically, with sub-workflows treated as activity types in a higher-level workflow. Activity ports are typed such that WOOL can check that the kinds of values flowing across ports match up.

WOOL includes a standard library of activity types, available to all workflows and providing helpful functionality such as control flow primitives. The WOOL language provides syntactic sugar to make certain control flow idioms easier to type and read. These idioms are normalized at compilation time to the equivalent sequence of language primitives.

Truly abstract workflow specifications must avoid making assumptions about the architecture(s) where they will be run. As such, WOOL adopts a minimal set of assumptions about the semantics of its data-flow execution model. In theory, this makes WOOL workflows portable to almost any workflow execution system.

In many ways, Twig's code generation scheme (described in Chapter III) was inspired and influenced by our work on WOOL. Twig's *blocks* are conceptually similar in principle to WOOL's *activities* – they are, essentially, functions that transform one or more inputs to one or more outputs. In WOOL, inputs and outputs could be "wired together" arbitrarily. In Twig we adopt a more formal model. In particular, Twig's blocks are composed via the sequence (Section 3.1) and parallel (Section 3.1) composition operators. This gives Twig's blocks more structure than a WOOL workflow, and this is motivated by Twig's need to match the semantics of its code generation scheme to the semantics of its language based on term rewriting.

In WOOL, the meaning of connections between activities was somewhat informal. A connection from the output of one activity to the input of another implied a dataflow relationship but, in keeping with WOOL's abstract focus, the exact meaning was left up to the implementation and runtime. Twig adopts a similar approach for blocks – we leave the exact interpretation of blocks and the connections between them up to the implementation for a particular target language. In this way, we avoid tying ourselves to a particular target language.

CHAPTER III

CODE GENERATION

Twig is able to generate code in different target languages by relying on an abstract, language-independent model with a small number of basic operations. To implement a new target language, it suffices to implement these operations only. In particular, there is no need to modify the core Twig interpreter, which assumes only the language-independent model. This model for code generation was partly inspired by our own previous work on the WOOL workflow programming language, which we discuss in Section 2.4.

We make use of the code generation model in describing Twig's semantics in Chapter IV. It is also helpful in clarifying the precise operations which Twig supports, without getting bogged down in the rather complicated details of rendering code for a particular target language.

In Section 3.1, we describe the code generation model abstractly, and apart from any specific target language. Then, in Section 3.2, we show how the model can be specialized to generate C code.

Abstract Code Generation

We call a single unit of generated code a *block*. A block is an abstract representation of some code in a target language, which accepts inputs and produces outputs. We denote the set of all blocks M, and provide functions

$$\text{in} : M \to \mathbb{N}$$

$$\text{out} : M \to \mathbb{N}$$

which map a block in M to the number of its inputs and outputs, respectively.

Sequential Composition

The first binary operation on blocks is *sequential composition*, which we represent as "addition" on the elements of M, i.e.

$$+ : M \times M \to M$$

Sequencing represents connecting two blocks "vertically," feeding the outputs of the first block to the inputs of the second. The block $x + y \in M$ is defined if and only if $\text{out}(x) = \text{in}(y)$. The outputs of the first element must be equal in number to the inputs of the second element because they are "fused" pairwise in the sequence operation. We define

$$\text{in}(x + y) = \text{in}(x)$$

since the inputs of the first block will become the inputs of the combined block. Similarly,

$$\text{out}(x + y) = \text{out}(y)$$

for the outputs.

Implementations must ensure that $+$ is associative. That is, they must enforce the fact that

$$\forall a, b, c \in M \;:\; (a + b) + c = a + (b + c)$$

Parallel Composition

The second block operator is *parallel composition*. We represent this operation as "multiplication" on the elements of M, i.e.,

$$\times : M \times M \to M$$

Parallel composition attaches two blocks "horizontally," where each block executes independently of one another, but they appear as a single block with combined inputs and outputs. For the block $x \times y \in M$, we define

$$\text{in}(x \times y) = \text{in}(x) + \text{in}(y)$$

and

$$\text{out}(x \times y) = \text{out}(x) + \text{out}(y)$$

Implementations must ensure that $\times$ is associative, that is that for all $a, b, c \in M$, $(a \times b) \times c = a \times (b \times c)$.

Permutation and Identity Blocks

We define a set of special blocks in M called *permutation* blocks. These blocks represent the primitive operation of "wiring" m inputs to n outputs in arbitrary order, without altering the values. We call the block permuting m inputs to n outputs $\Pi_m(i_1, \ldots, i_n)$, where

$$i_1, \ldots, i_n \in \{i \mid 1 \leq i \leq m\}$$

Identity blocks are a subset of the permutation blocks. The simplest of these is $\Pi_1(1)$, which acts as an identity transformation with one input and one output. That is, the block $\Pi_1(1)$ takes its single input and passes it unchanged to its single output. We refer to this block as I_1. There are an unlimited number of identity transformations which take n inputs to n outputs without reordering. We refer to these blocks as I_n, where $1 \leq n$, and $I_n = \Pi_n(1, 2, \ldots, n)$. By definition, $\text{in}(I_n) = \text{out}(I_n) = n$.

When n is implied from the context, we will sometimes write I for I_n. For example, when we write $x + I$, we mean $x + I_n$ where it is understood that $n = \text{out}(x)$.

Since the blocks I represent identity operations, we assign them a special meaning in the semantics. Namely, I acts as both a left- and right-identity under the sequence operator. So, for all $x \in M$, $x + I = x$ and $I + x = x$. We usually use I as a "no-op" block.

It is worth noting one further identity, namely that I_n is equivalent to the n-way parallel composition of I_1, that is

$$I_n = \underbrace{I_1 \times \ldots \times I_1}_{\text{n}}$$

Permutation blocks may be used to “drop” values by not wiring an input to any output, or to “duplicate” a value by wiring an input to more than one output. We leave the exact meaning of dropping or duplicating values depends on the implementation. The reason for this design choice is that duplicating a value may mean different things in different target languages. For example, depending on the value the implementation may need to allocate memory and perform a copy operation. Or, it might simply copy a reference to the value's underlying data. Therefore, Twig ignores some potential identity relationships among permutation blocks. Figure 1 shows the block $\Pi_1(1,1) + \Pi_2(2)$, a sequence of two permutation blocks. The top block “duplicates” its single input, while the second block “drops” its first input and outputs its second. The combination should pass a value through unchanged, i.e., be equivalent to I. Twig does not assign semantics to duplicating or dropping elements and thus cannot consider the illustrated block to be equivalent to I.

Note that *any* object that provides and conforms to the operations above can be “generated” by Twig. Because the system is so general, this could include trivial or non-sensical implementations. The code generation implementation should conform to the intuitive interpretation of blocks and their composition.

Generating C

We have adapted the model described above to generate C code. Our implementation must provide a way to construct a primitive block from an arbitrary chunk of C code since the abstract model, by design, does not provide

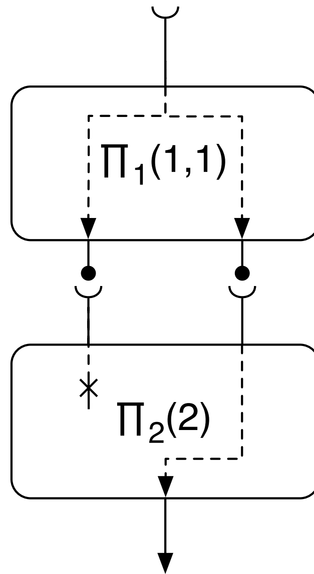

FIGURE 1. A sequence of two permutation blocks.

this facility. In our implementation for C, a primitive block is a string of C code with some specially-named variables indicating inputs and outputs. In fact, our implementation makes no attempt to parse the C language *per se* – it treats code as plain text with the aforementioned special variables.

To use the block's inputs, the code references escaped variables named `$in1`, `$in2`, and so on. Similarly, the variables `$out1`, `$out2`, and so on represent the outputs. We allow `$in` as a synonym for `$in1`, and `$out` for `$out1`, for the common case where a block has just one input and/or output. When the code is rendered, these variables will be replaced with unique, generated variable names. For example, the code

```
$out = foo($in);
```

represents a primitive C block with one input and one output. Figure 2 shows a visual representation of two primitive blocks of C code, labeled A and B, with inputs on top and outputs on the bottom.

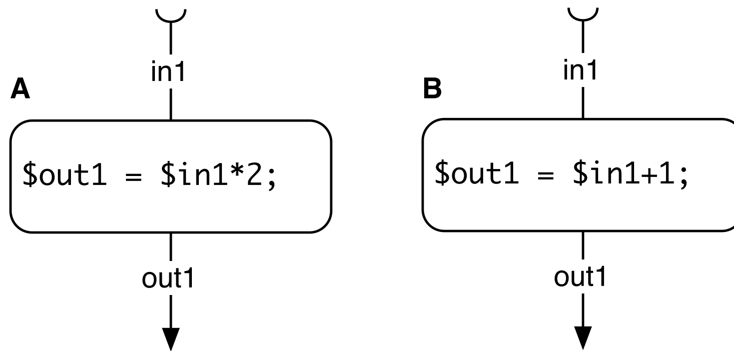

FIGURE 2. Two basic blocks.

To implement block sequencing in C, Twig generates variable names such that the output(s) of the first block in the sequence are the same as the inputs(s) of the second, and the text is concatenated. Figure 3 shows the two blocks from Figure 2 composed sequentially. The variable "tmp" is created, and renaming performed, so that the outputs of block A flows to the inputs of block B.

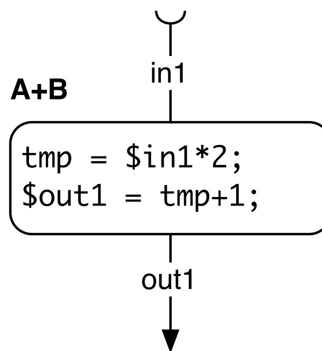

FIGURE 3. Sequential block composition.

Parallel composition for C is implemented similarly; Twig generates independently-named variables for the inputs and outputs of the two blocks, and then concatenates the text. Figure 4 shows the two blocks from Figure 2 composed in parallel. Renaming is performed such that the composed block has two inputs and two outputs.

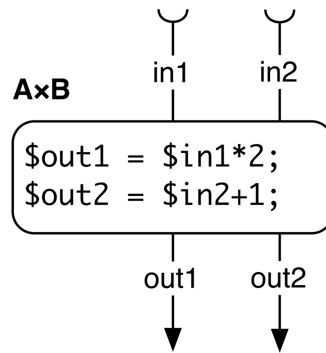

FIGURE 4. Parallel block composition.

Implementing the permutation and identity blocks is a matter of performing the appropriate bookkeeping and renaming on the variable names. Note that this implementation does not perform resource management, such as allocating or free memory, as part of the permutation operations. The generated code will follow C's semantics for passing data by value.

CHAPTER IV

THE TWIG LANGUAGE

Formal Semantics

Twig is based on *System S* [88], which was originally designed as a core language for term rewriting systems [19]. A review of term rewriting and System S is given in Section 2.3. In Twig, use the operators of System S to combine primitive *rules* into complex expressions. An expression is applied to an input term, which represents some type in the target language. The expression may transform the given type, generating code as a side effect, or the transformation may fail. In this way, different code can be generated depending on the input term. Twig's semantics are inspired by Fig [78], but extended to incorporate our code generation model. In the following sections we describe this process more formally.

Terms

Twig programs operate on values called *terms*. Terms represent tree-structured data with labeled internal nodes, a versatile data type useful in a variety of applications. For example, abstract syntax trees are naturally represented as terms.

We define the grammar for constructing terms as follows:

$$t := x \mid c \mid f(t_1, \ldots, t_n)$$

A term is either a variable x, a constant c, or the application of a *constructor* f to at least one other term. In Twig's syntax, constants and constructors can

be any string of characters beginning with a lower-case letter. The only string excluded from this condition is the special constructor `tuple` (see Section 4.1, below). Variables are represented by strings of characters beginning with a capital letter; in our presentation we will typically use a single capital letter only, e.g., `X` or `Y`. We use a fixed-width font when presenting terms, e.g., the constant term `myterm`, or the constructed term with a variable `cons(myterm1,X)`.

Following the notation of System S, we denote the set of all variables $\mathcal{X}$, and the set of all terms containing variables $T(\mathcal{X})$. We denote the set of all terms without variables, known as *ground terms*, as T.

In many systems, terms are typed through the use of *signatures* [19]. That is, certain terms can be defined as valid in a particular domain, and terms not found to conform can be flagged as errors. Twig could be extended to support term signatures, and indeed this might be useful. In our current work, however, we consider only untyped terms.

For the purposes of Twig's semantics, the meaning of a particular term (other than `tuple`s) is abstract. Terms are defined by their use in the program's *rules*, described below.

Tuples

Twig recognizes a special kind of term: tuples. The tuple elements are represented as the sub-terms of a term with a special constructor: `tuple`. Tuples may have any length. Twig's syntax equates the absence of any constructor with the presence of the `tuple` constructor. For example, the syntax `(string,int)` is interpreted as the `tuple(string,int)`. This term represents a tuple of length two, whose first element is `string` and whose second element is `int`.

The *size* of a tuple is simply the cardinality of its children. We will sometimes write $\texttt{tuple}_n(\ldots)$ to indicate a tuple of length n, where the length is not otherwise clear from the context.

One complication arises since we permit tuples to be nested to arbitrary depth. For example the term

$$\texttt{tuple}(\texttt{tuple}(\texttt{int}, \texttt{float}), \texttt{tuple}(\texttt{double}))$$

is a nested tuple. In our semantics, we will require the *width* of a tuple, defined as

$$\text{width}(t) = \begin{cases} \sum_{n}^{i=1} \text{width}(t_i) & \text{if } t = \texttt{tuple}(t_1, \ldots, t_n) \\ 1 & \text{otherwise} \end{cases}$$

Intuitively, the width of a tuple corresponds to its size after being "flattened," where the elements of nested tuples are pushed up, recursively, to the top level. If we flattened the tuple in the example above, we would get

$$\texttt{tuple}(\texttt{int}, \texttt{float}, \texttt{double})$$

and its width would be three.

Terms representing types

In Twig, terms are used to represent types in a target language. For example, we use terms such `int` and `float` to represent primitive types in C. Terms built with constructors can represent types with some structure, e.g., the term `ptr(int)`

can represent a C pointer to an integer. More complicated terms may involve multiple children, and may be nested to any depth. For example, the term

$$\texttt{struct}(\texttt{int}, \texttt{float}, \texttt{struct}(\texttt{ptr}(\texttt{char})))$$

can represent a structure with three fields: an `int`, a `float`, and a second structure with a single string (pointer to `char`) field.

The mapping between terms and types in the target language is a configuration option, customizable for a particular domain. The mapping need not be injective, that is, multiple terms in Twig may represent a single type in the target language. For example, you might have the distinct terms `string` and `ptr(char)` both map to a `char` pointer in C.

Expressions

Twig *expressions* can be either *primitive rules* (Section 4.1) or else built from other expressions using operators (Section 4.1). We denote the set of all expressions S. An expression $s \in S$ maps ground terms T to elements of the set $(T \times M) \cup \{\bot\}$, i.e., either a pair (t', m) where $t' \in T$ and $m \in M$ is a *block* of generated code in the set M (see Chapter III), or else the special, distinguished value $\bot$. Formally, $s \in S$ is a function:

$$s : T \rightarrow ((T \times M) \cup \{\bot\})$$

Following Fig's notation, we use $\bot$ to denote "failure." In particular, $\bot$ is used in the semantics for the operators described in Section 4.1.

As with System S and Fig, Twig allows expressions to be named. An expression's name may be used in place of itself within other expressions. The syntax is

$$v \texttt{ = } s$$

where v is an expression identifier and $s \in S$ is an expression of the form described below. A Twig program is a list of such name/expression assignments. To prevent circular references, expressions may only reference names that have been previous defined (i.e., appearing before the expression in the program text). For the same reason, expressions may not reference their own name. For example:

```
foo = foo ; bar
```

is not allowed, because the definition of `foo` references `foo` itself. Programs requiring recursive expressions should use the `#fix` fixed-point operator instead.

There is a special expression name, `main`, which designates the top-level expression for the program.

Primitive Rules

The simplest Twig expressions are *primitive rules*, which describe a single step of a type transformation. Since Twig terms represent types in a target language, a primitive rule in Twig describes how to transform an instance of one type into an instance of another in that language.

The syntax for primitive rules is

$$\texttt{[}p_1 \texttt{ -> } p_2\texttt{]<<<} \; m \; \texttt{>>>}$$

where $p_1, p_2 \in T(\mathcal{X})$, i.e. p_1 and p_2 are non-ground terms potentially having variables, with the condition that variables appearing in p_2 must also appear in p_1. These syntactic elements are called the *input pattern* and *output pattern* of the rule, respectively. The element $m \in M$ is a block representing some code in the target language (see Section III). We can write the semantics of primitive rules formally with two statements:

$$\overline{t \xrightarrow{\texttt{[}p_1\ \texttt{->}\ p_2\texttt{]<<<}\ m\ \texttt{>>>}} (t', m)} \quad \text{if} \quad \exists \sigma : \sigma(p_1) = t \wedge \sigma(p_2) = t' \tag{4.1}$$

$$\overline{t \xrightarrow{\texttt{[}p_1\ \texttt{->}\ p_2\texttt{]<<<}\ m\ \texttt{>>>}} \perp} \quad \text{if} \quad \nexists \sigma : \sigma(p_1) = t \tag{4.2}$$

The statement in (4.1) says that the rule `[`p_1 `->` p_2`]<<<` m `>>>` rewrites t to (t', m) if there exists a substitution *sigma* mapping variables to ground terms such that the substitution of σ in p_1 is t and the substitution of σ in p_2 is t'. The statement in (4.2) says that the rule `[`p_1 `->` p_2`]<<<` m `>>>` fails if there exists no substitution mapping p_1 to t.

Note that these semantics allow for some relatively sophisticated matching tasks. For example, if the same variable appears more than once in the input pattern, it must be bound to the same sub-term. For example, the input pattern `foo(X,X)` would match `foo(bar,bar)` but not `foo(bar,baz)`. This can be useful when we want to test that a term has some symmetric properties, without necessarily needing to care about the specific sub-terms involved. In addition, variables appearing in the input pattern need not appear in the output pattern; extraneous sub-terms can be matched and eliminated in this way.

Informally, the primitive rule above transforms t to (t', m) if and only if

1. t successfully *matches* the input pattern p_1, binding terms to variables in *sigma* and

2. t' is *built*, by substituting the bound values in σ into the variables of p_2;

and otherwise fails.

In our implementation, when a term is *matched* we mean that Twig attempts to unify [19] it with the input pattern. In fact, Twig's matching algorithm is simpler than full unification, since there is no equational theory and the input term may not contain variables (i.e., must be a ground term). If unification is successful, variables are bound to their corresponding terms in an environment. The environment is then used to construct the output term by substitution into the output pattern. See [19, 88] for details on unification algorithms.

As an example, consider the following primitive rule. In C it is easy to convert an integer value to floating point. Twig's syntax for writing this rule is as follows:

```
[int -> float] <<< $out = (float)$in; >>>
```

In this example, if the input term matches the input pattern `int`, then the output will be the term `float` along with the code block. If the input term does not match `int` then the output will be $\perp$.

As mentioned, input and output patterns can have *variables* in place of terms or sub-terms. For example the rule

```
[ptr(X) -> X] <<< $out = &$in; >>>
```

describes a transformation of any C pointer type to its referent. The variable `X` is bound to the corresponding value of the matched input on the right, and

that value is substituted for the variable where it appears on the left. Variables may stand in place of a single term only, not constructors; e.g., patterns such as `[X(int) -> X]` are not allowed.

Operators

Expressions can be combined using Twig's operators. In the following semantics, let t range over terms, m range over blocks, and s range over expressions, i.e., either a primitive rule, or else another expression built with operators.

The *sequence* operator, written as an infix semi-colon (`;`), chains the application of two rules together by sending the output of the first to the input of the second. The combined expression fails if either sub-expression fails. With this operator, simple rules can be composed into multi-step transformations. Upon success, the result blocks are combined sequentially using the block sequence operation (see Section 3.1). The formal semantics are:

$$\frac{t \xrightarrow{s_1} (t', m_1) \quad t' \xrightarrow{s_2} (t'', m_2)}{t \xrightarrow{s_1;s_2} (t'', m_1 + m_2)} \tag{4.3}$$

$$\frac{t \xrightarrow{s_1} \bot}{t \xrightarrow{s_1;s_2} \bot} \tag{4.4}$$

$$\frac{t \xrightarrow{s_1} (t', m) \quad t' \xrightarrow{s_2} \bot}{t \xrightarrow{s_1;s_2} \bot} \tag{4.5}$$

The sequence operator is associative, that is for any expressions f, g, h, $(f; g); h$ is equivalent to $f; (g; h)$. This allows us to write expressions like $f; g; h$ without ambiguity. We prove this fact in Section A.1.

Left-biased choice, written as a vertical bar (`|`), will attempt to apply the first rule expression to the input, and if it succeeds then its output is the result. If it fails, it attempts to apply the second rule instead. This operator allows different code to be generated depending on the input type. Formally:

$$\frac{t \xrightarrow{s_1} (t', m_1)}{t \xrightarrow{s_1|s_2} (t', m_1)} \tag{4.6}$$

$$\frac{t \xrightarrow{s_1} \bot \quad t \xrightarrow{s_2} (t', m_2)}{t \xrightarrow{s_1|s_2} (t', m_2)} \tag{4.7}$$

$$\frac{t \xrightarrow{s_1} \bot \quad t \xrightarrow{s_2} \bot}{t \xrightarrow{s_1|s_2} \bot} \tag{4.8}$$

Like the sequence operators, left-biased choice is associative. For any expression f, g, h, $(f|g)|h$ is equivalent to $f|(g|h)$, allowing us to write $f|g|h$ without ambiguity. We prove this fact in Section A.1.

Twig includes a variety of other basic operators. The *identity* (`T`) expression will always succeed, returning its input and an identity block. Conversely, *failure* (`F`) will always return $\bot$.

$$\frac{}{t \xrightarrow{\mathtt{T}} (t, I)} \tag{4.9}$$

$$\frac{}{t \xrightarrow{\texttt{F}} \bot} \tag{4.10}$$

The unary operator *test* (?) succeeds only if its argument succeeds, returning the original term and discarding the result term and block. This operator is useful for examining a term's structure without actually making use of its sub-terms.

$$\frac{t \xrightarrow{s} (t', m)}{t \xrightarrow{?s} (t, I)} \tag{4.11}$$

$$\frac{t \xrightarrow{s} \bot}{t \xrightarrow{?s} \bot} \tag{4.12}$$

Negation (¬) also takes a single expression argument. It succeeds only if its argument fails on the input, returning the original term.

$$\frac{t \xrightarrow{s} (t', m)}{t \xrightarrow{\neg s} \bot} \tag{4.13}$$

$$\frac{t \xrightarrow{s} \bot}{t \xrightarrow{\neg s} (t, I)} \tag{4.14}$$

Twig also provides some operators especially for tuples.

The *congruence* operator applies a tuple of expressions to the elements of a tuple term, pairwise, and returns a tuple of results. It fails in case any of the individual rule applications fail. Upon success, the result block is the parallel composition (see Section 3.1) of the individual result blocks. The formal semantics are as follows:

$$\frac{t_1 \xrightarrow{s_1} (t'_1, m_1) \quad \cdots \quad t_n \xrightarrow{s_n} (t'_n, m_n)}{\texttt{tuple}(t_1, \ldots, t_n) \xrightarrow{(s_1,\ldots,s_n)} (\texttt{tuple}(t'_1, \ldots, t'_n), m_1 \times \ldots \times m_n)} \tag{4.15}$$

$$\frac{t_i \xrightarrow{s_i} \bot}{\texttt{tuple}(\ldots, t_i, \ldots) \xrightarrow{(\ldots,s_i,\ldots)} \bot} \tag{4.16}$$

$$\frac{}{\texttt{tuple}(t_1, \ldots, t_m) \xrightarrow{(s_1,\ldots,s_n)} \bot \quad \text{if } m \neq n} \tag{4.17}$$

$$\frac{}{t \xrightarrow{(s_1,\ldots,s_n)} \bot \quad \text{if } t \neq \texttt{tuple}(\ldots)} \tag{4.18}$$

The family of unary *branch* operators apply a single expression to one, all, or some of a tuple's elements, depending on the variant.

The branch operator `#one` attempts to apply its parameter s to a single element: the first element, from left to right, for which s does not fail. The other elements of the tuple are unchanged. The expression fails if s fails for each element. The formal semantics for `#one` are as follows:

$$\frac{t_i \xrightarrow{s} (t'_i, m_i)}{\texttt{tuple}(\ldots, t_i, \ldots) \xrightarrow{\#\texttt{one}(s)} (\texttt{tuple}(\ldots, t'_i, \ldots), (I \times \ldots \times m_i \times \ldots \times I))} \tag{4.19}$$

$$\frac{t_1 \xrightarrow{s} \bot \quad \cdots \quad t_n \xrightarrow{s} \bot}{\texttt{tuple}(t_1, \ldots, t_n) \xrightarrow{\#\texttt{one}(s)} \bot} \tag{4.20}$$

$$\frac{}{t \xrightarrow{\#\texttt{one}(s)} \bot \quad \text{if } t \neq \texttt{tuple}(\ldots)} \tag{4.21}$$

The branch operator `#all` applies its parameter s to each element of a tuple. The expression fails if s fails for any element. The formal semantics for `#all` are:

$$\frac{t_1 \xrightarrow{s} (t'_1, m_1) \quad \cdots \quad t_n \xrightarrow{s} (t'_n, m_n)}{\texttt{tuple}(\ldots, t_i, \ldots) \xrightarrow{\#\texttt{all}(s)} (\texttt{tuple}(\ldots, t'_i, \ldots), (m_1 \times \ldots \times m_n))} \tag{4.22}$$

$$\frac{t_i \xrightarrow{s} \perp}{\texttt{tuple}(\ldots, t_i, \ldots) \xrightarrow{\#\texttt{all}(s)} \perp} \tag{4.23}$$

$$\frac{}{t \xrightarrow{\#\texttt{all}(s)} \perp \quad \text{if } t \neq \texttt{tuple}(\ldots)} \tag{4.24}$$

The branch operator `#some` applies its parameter s to at least one element of a tuple. The expression fails if s fails for each elements. The formal semantics for `#some` are:

$$\frac{\begin{array}{c} P(t) = \begin{cases} t' & \text{if } t \xrightarrow{s} (t', m) \\ t & \text{if } t \xrightarrow{s} \perp \end{cases} \\ Q(t) = \begin{cases} m & \text{if } t \xrightarrow{s} (t', m) \\ I & \text{if } t \xrightarrow{s} \perp \end{cases} \\ \exists i : i \in \{1..n\} \ \wedge \ t_i \xrightarrow{s} (t'_i, m) \end{array}}{\texttt{tuple}(t_1, \ldots, t_n) \xrightarrow{\#\texttt{some}(s)} (\texttt{tuple}(P(t_1), \ldots, P(t_n)), Q(t_1) \times \ldots \times Q(t_n))} \tag{4.25}$$

$$\frac{t_1 \xrightarrow{s} \perp \quad \cdots \quad t_n \xrightarrow{s} \perp}{\texttt{tuple}(t_1, \ldots, t_n) \xrightarrow{\#\texttt{some}(s)} \perp} \tag{4.26}$$

$$\frac{}{t \xrightarrow{\#\texttt{some}(s)} \bot \quad \text{if } t \neq \texttt{tuple}(\ldots)} \tag{4.27}$$

A *projection* extracts a single indexed element from a tuple, discarding the other tuple elements. The formal semantics are:

$$\frac{}{\texttt{tuple}(\ldots, t_i, \ldots) \xrightarrow{\#i} (t_i, \Pi(i))} \tag{4.28}$$

$$\frac{}{\texttt{tuple}(t_1, \ldots, t_n) \xrightarrow{\#i} \bot \quad \text{if } i > n} \tag{4.29}$$

$$\frac{}{t \xrightarrow{\#i} \bot \quad \text{if } t \neq \texttt{tuple}(\ldots)} \tag{4.30}$$

The *path* unary operator applies a rule to a single indexed tuple element, leaving the other elements unchanged.

$$\frac{t_i \xrightarrow{s} (t'_i, m_i)}{\texttt{tuple}(\ldots, t_i, \ldots) \xrightarrow{\#i(s)} (\texttt{tuple}(\ldots, t'_i, \ldots), I \times \ldots \times m_i \times \ldots \times I)} \tag{4.31}$$

$$\frac{t_i \xrightarrow{s} \bot}{\texttt{tuple}(\ldots, t_i, \ldots) \xrightarrow{\#i(s)} \bot} \tag{4.32}$$

$$\frac{}{\texttt{tuple}(t_1, \ldots, t_n) \xrightarrow{\#i(s)} \bot \quad \text{if } i > n} \tag{4.33}$$

$$\frac{}{t \xrightarrow{\#i(s)} \bot \quad \text{if } t \neq \texttt{tuple}(\ldots)} \tag{4.34}$$

The *permutation* operator allows arbitrary permutation of a tuple's elements, including duplicating or dropping elements. The operator is parameterized by the width of the input tuple and a list of indices into the tuple. The output will rearrange the elements of the tuple in the order of the indices. The expression will fail if the input does not have the given width, or if the input is not a tuple.

$$\frac{}{\texttt{tuple}(t_1,\ldots,t_n) \xrightarrow{\#\texttt{permute}_n(x_1,\ldots,x_m)} (\texttt{tuple}(t_{x_1},\ldots,t_{x_m}), \Pi_w(y_{x_1},\ldots,y_{x_m}))}$$

$$\text{where} \quad \begin{aligned} & \{x_1..x_m\} \in \mathbb{N} \\ w &= \textstyle\sum_{j=1}^{n} \text{width}(t_i) \\ b_i &= \begin{cases} 0 & \text{if } i = 1 \\ \sum_{j=1}^{i-1} \text{width}(t_i) & \text{if } i > 1 \end{cases} \\ y_i &= b_i + 1, \ldots, b_i + \text{width}(t_i) \end{aligned} \tag{4.35}$$

$$\frac{}{\texttt{tuple}(t_1,\ldots,t_m) \xrightarrow{\#\texttt{permute}_n(\ldots)} \bot \quad \text{if } m \neq n} \tag{4.36}$$

$$\frac{}{t \xrightarrow{\#\texttt{permute}_n(\ldots)} \bot \quad \text{if } t \neq \texttt{tuple}(\ldots)} \tag{4.37}$$

The `#permute` operator has relatively complex rules for constructing the result block. This is because blocks must account for each element of a tuple with a separate input and output. The `#permute` operator, by contrast, rearranges the top-level tuple but must preserve the ordering of elements within sub-tuples. Figure 5 shows the application of $\#\texttt{permute}_4(4,3,2,1)$ to the tuple term

$((t1, t2, t3), t4, (t5, t6), t7)$ – the top-level tuple elements are permuted, while in the interior tuples, ordering is preserved.

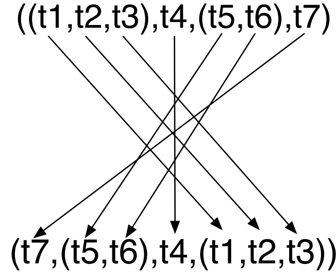

FIGURE 5. Permutation of tuples.

The `#fan` operator takes a integer parameter n, and replicates the input term n times in an n-element tuple. It is similar to `#permute` but does not require that its input be a tuple.

$$\frac{}{t \xrightarrow{\#\mathtt{fan}(n)} (\mathtt{tuple}(t, \overset{n}{\ldots}, t), \Pi_w(y, \overset{n}{\ldots}, y))} \quad \text{where} \quad w = \text{width}(t) \quad y = 1, \ldots, w \tag{4.38}$$

The fixed-point operator, `#fix`, allows Twig to express rules for handling recursively defined data types like lists and trees. An application of x within the expression $\#\mathtt{fix}_x(s)$, that is, x appearing within s, is essentially a recursive call to the expression $\#\mathtt{fix}_x(s)$.

$$\frac{t \xrightarrow{s[x \mapsto \#\mathtt{fix}_x(s)]} (t', m)}{t \xrightarrow{\#\mathtt{fix}_x(s)} (t', m)} \tag{4.39}$$

$$\frac{t \xrightarrow{s[x \mapsto \#\mathtt{fix}_x(s)]} \bot}{t \xrightarrow{\#\mathtt{fix}_x(s)} \bot} \tag{4.40}$$

CHAPTER V

USER-DEFINED EXPRESSION REDUCTIONS

In addition to the code generation and core semantics discussed in Chapters III and IV, we have outfitted Twig with semantics that allow users to add a customized optimization phase. We call this facility *expression rewriting* or *expression reduction*.

Expression reductions are an addition to the core semantics of Twig, not a necessary feature. Our implementation of Twig can be directed to ignore reduction directives.

An expression reduction is defined by a *directive*, written like so:

$$\texttt{@reduce} \quad e_1 \Rightarrow e_2$$

Where e_1 and e_2 are Twig expressions (see Chapter IV). Reduction directives can appear anywhere within a Twig program.

Informally, the directive above will cause the expression e_1 to be replaced with the expression e_2 anywhere it appears in the top level `main` expression, before the program is evaluated on its input. Expression names are substituted with their values within the `main` expression and, for each directive, within e_1 and e_2 before substitution begins. Directives are applied in the order in which they appear within the program text, and the main expression is transformed iteratively.

In order to match expressions for substitution, we need a notion of equality among expressions. Unfortunately, in the case of primitive rules, this is somewhat tricky. For one, we have only defined syntactic equality among the terms in $T(\mathcal{X})$

used for the input and output patterns. A better solution might involve something like alpha-equivalence among terms, allowing us to equate patterns like `foo(X)` with `foo(Y)`. Additionally, we do not define equality among abstract blocks because such a notion depends on the target language and thus the implementation.

We sidestep these issues by stipulating that there is no equality among primitive rules. That is, each primitive rule that appears in the program is assigned a unique identifier (having a well-defined notion of equality), and then matched by that identifier. This allows users to assign primitive rules to expression names, reference those names in reduction directives, and have the name stand for the primitive rule in a well-defined way. It also means that primitive rules should not appear on the left-hand side of reduction directives, since they will be assigned a unique identifier distinct from any appearing within the program, and thus never matched.

Expression Normalization

In addition to expression reduction directives provided by the user, Twig is able to apply some reductions based on the characteristics of its expression semantics. This is known as *normalization* of the expression.

The goal of the normalization procedure is to maximize the length of sequence expressions. This goal is motivated by our intuition that it is relatively easier for people to reason about sequence than the other operators, and so most user-defined reductions are likely to be in terms of sequences. For example, we have found in practice that reduction directives such as

$$@\texttt{reduce} \quad f; g => h$$

are quite common in our own work. Therefore, we would like to transform the program without otherwise altering its meaning to maximize the length of sequences, so that user-defined expression reductions based on sequences have the best chance of matching.

To that end, before user-defined reduction directives are applied, Twig can attempt to rewrite expression according to some built-in rules. Currently, we have identified two expression transformations which preserve the meaning of the program while rearranging expressions to prefer sequence over other operators.

First, sequence distributes over choice from the left. Stated formally:

$$\forall f, g, h \in S \ : \ f;(g|h) = (f;g)|(f;h)$$

We exploit this identity by rewriting any expression of the form $f;(g|h)$ to $(f;g)|(f;h)$, which preserves its meaning but creates longer sequences.

Second, congruence distributes over sequence. Stated formally,

$$\forall r_1..r_n, s_1..s_n \in S \ : \ \{r_1, \ldots, r_n\};\{s_1, \ldots, s_n\} = \{r_1; s_1, \ldots, r_n; s_n\}$$

Again, we can exploit this identity by rewriting the left-hand side of the equation to the right, which preserves the meaning of the expression but gives longer sequences.

We prove these identities in Section A.2, below. Note that these built-in identities could not be expressed by a user as reduction directives in the code, because we do not allow variables in the expressions. This is an area of future work.

Implementation

Our current implementation of reduction directives is rather ad hoc. We simply export the `main` expression as a string after substituting expression names for their values and primitive rules for their unique identifiers, as described above. Then, we process the string in a separate term rewriting tool, and import it back into Twig's internal expression representation. We have used the tool Maude [64] successfully for rewriting, although there is no particular reason that other tools, such as Stratego [85], could not be used instead.

To perform normalization we use Maude's functional modules. Here is a (partial) listing for the functional module we use to rewrite expressions of the form $f;(g|h)$ to $(f;g)|(f;h)$:

```
fmod TWIG-EXPR is
  protecting QID .
  sort Expr .
  op rule : Qid -> Expr [ctor] .
  op _;_ : Expr Expr -> Expr [ctor assoc] .
  op _|_ : Expr Expr -> Expr [ctor assoc] .
  vars F G H : Expr .
  eq F ; (G | H) = (F ; G) | (F ; H) .
endfm
```

As an example, we can export a very simple Twig expression to a string, and substituting unique identifiers `rule('a)`, `rule('b)`, and so on for primitive rules as described above, yielding something like this:

```
rule('a) ; (rule('b) | rule('c) | rule('d)) .
```

After applying the rewriting specified in the Maude module above, we get:

```
(rule('a) ; rule('b)) | (rule('a) ; rule('c))
                      | (rule('a) ; rule('d))
```

which we can import back into Twig and evaluate as usual.

One intriguing alternative implementation would use Twig itself to rewrite its own expressions. Twig's rules operate on terms, and Twig expressions can be represented as terms, so the approach would be straightforward to implement. This is a project we plan to pursue in the future.

Discussion

As we will demonstrate in Chapter VII, expression reductions are a powerful tool that allows users a high degree of control over how Twig evaluates programs. In particular, they allow users to introduce domain-specific optimizations for a given set of rules. Other tools, such as the Glasgow Haskell Compiler, have introduced similar facilities in their tools with promising results [73].

As with all powerful tools, however, caution must be exercised when using reductions. Because they allow arbitrary, user-defined rewriting of expressions, it is easy to introduce errors or even rules which will cause non-termination of the Twig tool.

For example, if a user introduces a directive such as

```
@reduce f => f;f
```

then Twig will keep rewriting `f` to `f;f`, then to `f;f;f;f`, and so on, ad infinitum. Other cases may be more subtle, especially when reduction directives interact. For example, the two reduction directives:

```
@reduce a => b;c
```

```
@reduce c => a
```

are individually harmless but, if they are both introduced in the same program, they will cause a chain of infinite expansion.

In general, the set of expression reductions including the rules described in Section 5.1 and any user-defined directives, should be *normalizing* for expressions. That is, application of the set of rules should rewrite any expression to a unique form, and terminate. For more details on normal forms and rewriting, see Section 2.3.

CHAPTER VI

THE DESIGN OF TWIG'S INTERPRETER

The `twigc` Application

Our implementation of Twig is called `twigc`. This command line application expects as input a `.twig` file containing a list of named rule expressions along with a `main` rule expression, as described in Section 4.1. It also expects an initial value (i.e., a term, representing a type in the target language), which will be used as the input to the main rule expression. As discussed below, our implementation supports both C and Python as target languages.

To render C, Twig must be configured with a mapping from terms to C types. Currently, this mapping is provided with a simple key/value text file. Note that this mapping is not necessary for Python, since types do not need to be declared in that language.

If the input value can be successfully rewritten using the main rule expression provided, then Twig will output the rewritten term along with the generated block of code. If desired, this code block may be redirected to a separate file. In C, for example, the file may then be included in a program using an `#include` directive.

Embedded Design

The `twigc` tool is written in Haskell. We have found Haskell very well suited to this kind of interpreter development. This is due primarily to Haskell's ability to embed domain-specific languages within itself [47].

In Haskell, we can easily describe the structure of Twig's expressions as a datatype. Here is a simplified version of our expression datatype in Haskell:

```
data RuleExpr = Rule Pattern Pattern String
              | Seq RuleExpr RuleExpr
              | LeftChoice RuleExpr RuleExpr
```

Our "interpreter" is then a function that takes a `RuleExpr`, and yields another function having the type

$$\texttt{Term}-> \texttt{Maybe}(\texttt{Term},\texttt{Block})$$

This is the type of a function takes a term (represented by the datatype `Term`) and returns a value of type `Maybe (Term,Block)`. This type represents a choice (the `Maybe` constructor) between either a pair of a term and a block, or else the single distinguished value `Nothing`, representing $\perp$ or failure in our semantics. Implementing the interpreter is straightforward. For example, here is the (somewhat simplified) code for Twig's sequence operator:

```
eval (Seq e1 e2) t =
  case eval e1 t of
    Nothing -> Nothing
    Just (t',m1) ->
      case eval e2 t' of
        Just (t'',m2) -> Just (t'',m1 `seqn` m2)
        Nothing -> Nothing
```

And here is left-biased choice:

```
eval (LeftChoice e1 e2) t =
  case eval e1 t of
    Just (t',m) -> Just (t',m)
    Nothing ->
      case eval e2 t of
        Just (t',m) -> Just (t',m)
        Nothing -> Nothing
```

We find this code quite elegant.

Code Generation

Our implementation supports generation of two different target languages: C and Python. We have described the C generation scheme and our implementation in Section 3.2. Our Python implementation is quite similar. In fact, generating Python is a bit simpler, because we do not have to worry about declaring generated variables before they are used.

We have added an experimental feature where the programmer can select between C and Python in the same Twig program file using a special directive. We note this feature because we make use of it in the example in Section 7.2.

Consider the Twig file:

```
@language{Python}

r1 = [py(int) -> py(float)] <<<
  $out = float($in)
>>>
```

```
@language{C}

r2 = [py(float) -> double] <<<
  $out = PyFloat_AsDouble($in);
>>>

main = r1;r2
```

Note the directives `@languagePython` and `@languageC`. These directives inform Twig that the code blocks in the primitive rules following the directive are written in the language indicated. If we evaluate this program on the input term `py(int)`, the output term will be a `double`, and Twig will generate two code files, one for C, and another for Python. The generated Python file wraps the Python output in a function, like this:

```
# Filename: output.python
def gen1_py(in):
  gen1 = float(in)
  return gen1
```

And the C file automatically calls the Python function using the Python/C API [12]:

```
void gen(PyObject *in) {
  PyObject *gen1;
  double gen2;
  gen1 = call_python("output","gen1_py",in);
  gen2 = PyFloat_AsDouble(gen1);
```

```
}
```

The C program invokes the generated Python function via the `call_python` function. The function accepts a single Python value as an input argument, along with some parameters indicating which Python function to invoke, and returns a single Python value. The code for our `call_python` function is as follows:

```
PyObject *call_python(char *module, char *f, PyObject *in) {
  PyObject *module,*dict,*func,*out;
  module = PyImport_ImportModule(module);
  dict   = PyModule_GetDict(module);
  func   = PyDict_GetItemString(dict,f);
  out    = PyEval_CallObject(func,in);
  return out;
}
```

This approach is promising, but at the moment is relatively ad hoc. We are able to automatically combine the two languages in this way only for simple data types, and in particular tuple terms cannot be passed from Python to C. Ideally, we would provide a way for user's to customize the `call_python` function, so that multi-language facilities other than the Python/C API could be used.

Expression Reductions

We have described our implementation of Twig's facility for user-defined expression reductions in Section 5.2. Here, we note only that expression reductions are implemented in a way that is orthogonal to the rest of the interpreter – they can be turned on or off as needed.

CHAPTER VII

EVALUATION OF TWIG

In this chapter we will show how Twig may be used for a variety of applications. First, in Section 7.1, we compare Twig against SWIG, the best-known system that uses typemaps. In Section 7.2, we demonstrate how Twig's abstract code generation model can be used to work with multiple target languages in the same typemap. Finally, in Section 7.3, we show how GPU programming, a seemingly dissimilar problem, can also benefit from Twig's approach to multi-targeted programming.

Twig Compared

We now walk through the construction of a simple typemap in Twig. In this example, our goal is to convert a set of C structures representing polar coordinates to a suitable representation in Python. The C structure comes in both a `float` and `double` variety. The Python code expects a Cartesian coordinate system, not polar, so we must perform this conversion as well. The C structures we will convert are defined in a header file, like so:

```
struct PolarD {
  double r;
  double theta;
};
struct PolarF {
  float r;
  float theta;
```

```
};
```

The first step is to unpack each polar structure into a Twig tuple. We define two rules to do exactly this:

```
unpackd = [polard -> (double,double)] <<<
  $out1 = $in.r;
  $out2 = $in.theta;
>>>


unpackf = [polarf -> (float,float)] <<<
  $out1 = $in.r;
  $out2 = $in.theta;
>>>
```

Next, we define a rule for casting `float`s to `double`s, and use the congruence operator to lift it to a conversion on tuples. This cast is sequenced after `unpackf` so that that rule will produce `double`s instead of `float`s. We combine that conversion with `unpackd` using the choice operator, and name the new rule `unpack`. This new rule will accept either a `polarf` or a `polard`, and produce a 2-tuple of `double`s.

```
f2d = [float -> double] <<<
  $out = (double)$in;
>>>


unpack = (unpackf;{f2d,f2d}) | unpackd
```

Next, we define the conversion from polar to Cartesian coordinates.

```
polarToX = [(double,double) -> double] <<<
  $out = $in1 * cos($in2);
>>>

polarToY = [(double,double) -> double] <<<
  $out = $in1 * sin($in2);
>>>
```

These two rules take a pair of `double`s, which represent a polar radius and angle, and convert the pair to the x (respectively, y) component of the equivalent Cartesian representation. But, we need both the x and y components, and we only have one polar pair. We use the *fanout* operator to duplicate the pair, and then sequence it with a congruence of the x and y rules, like so:

```
polarToCart = #fan(2);{polarToX,polarToY}
```

The rule `polarToCart` will convert a polar coordinate pair of `doubles` to a Cartesian pair of `double`s.

Next, we define rules to convert from C types to Python. We use Python's C interface API [12], which allows us to work with Python values in C.

```
d2pyf = [double -> pyfloat] <<<
  $out = PyFloat_FromDouble($in);
>>>

mkpytuple = [(pyfloat,pyfloat) ->
             pytuple(pyfloat,pyfloat)]
<<<
```

```
  $out = PyTuple_Pack(2,$in1,$in2);
>>>
```

```
pack = {d2pyf,d2pyf};mkpytuple
```

The first rule, `d2py` converts a C `double` to Python's floating-point type, which we call `pyfloat`. [1] The next rule, `mkpytuple` will combine a pair of `pyfloats` into a single Python tuple object (*not* a Twig tuple). The `pack` rule combines these in the usual way to convert a pair of C `doubles` to a Python tuple.

Finally, by placing these parts in sequence, we achieve our goal: a single rule which will convert either a `PolarD` or `PolarF struct` in C into a Cartesian coordinate in Python. We call the final rule `convert`.

```
convert = unpack;polarToCart;pack
```

We can invoke Twig with this typemap as its program. To generate the C code to perform the transformation, we apply `convert` to one of the terms `polarf` or `polard`. If we choose `polarf`, Twig will generate the code to convert a `PolarF struct`, like so:

```
PyObject *convert(struct PolarF gen1) {
  float gen2,gen3;
  double gen4,gen5,gen6,gen7;
  PyObject *gen8,*gen9,*gen10;
  gen2 = gen1.r;
```

[1] In the API, a `pyfloat` is actually mapped to a more general `PyObject *`; one interesting benefit of Twig is that it can potentially track more detailed type information than would be available from API itself.

```
  gen3 = gen1.theta;
  gen4 = (double)gen2;
  gen5 = (double)gen3;
  gen6 = gen4 * cos(gen5);
  gen7 = gen4 * sin(gen5);
  gen8 = PyFloat_FromDouble(gen6);
  gen9 = PyFloat_FromDouble(gen7);
  gen10 = PyTuple_Pack(2,gen8,gen9);
  return gen10;
}
```

Versus SWIG

It is interesting to contrast Twig's implementation of the typemap example above with the equivalent typemaps in SWIG. In that system, programmers are required to construct two separate typemaps by hand, like so:

```
%typemap(out) struct PolarD %{
  double r = $1.r;
  double theta = $1.theta;
  double x = r * cos(theta);
  double y = r * sin(theta);
  PyObject *px = PyFloat_FromDouble(x);
  PyObject *py = PyFloat_FromDouble(y);
  $result = PyTuple_Pack(2,px,py);
%}
```

```
%typemap(out) struct PolarF %{
  float fr = $1.r;
  float ftheta = $1.theta;
  double r = (double)fr;
  double theta = (double)ftheta;
  double x = r * cos(theta);
  double y = r * sin(theta);
  PyObject *px = PyFloat_FromDouble(x);
  PyObject *py = PyFloat_FromDouble(y);
  $result = PyTuple_Pack(2,px,py);
  }
%}
```

Even in this simple example, there is a considerable amount of duplicated code across the two typemaps. This duplication is unnecessary in Twig since simple typemaps, such as those to convert polar to Cartesian coordinates or convert C `double`s to Python, can be recombined and reused. In addition, the choice operator helps to reduce the overall number of typemaps needed, since one typemap can be used to generate different code depending on the input.

Twig has a few other advantages over SWIG. First, Twig allows sets of types to be mapped together using tuples. This is a common problem – consider function argument lists, or a pointer paired with a length to form an array.

Second, Twig has greater flexibility with respect to target languages. In particular, Twig can be extended to generate target languages other than C. Our implementation, for example, is able to generate Python as well as C. SWIG is currently able to generate only C.

Versus Fig

Twig shares much of its core semantics with Fig, but Twig aims to serve a more general role as a typemapping language for a variety of applications. In particular, Fig is intimately tied to its target language, Moby. Twig, by contrast, can support a large range of mainstream languages. Our own implementation supports C and Python, and Twig's abstract formal semantics for code generation are designed to allow other languages to be incorporated easily.

In addition, unlike Fig, Twig's primitive rules may be polymorphic, allowing variables in place of types. For example, certain primitive rules in Twig such as

```
deref = [ptr(X) -> X] <<<
  $out = *$in;
>>>
```

would be difficult to express with as much generality in Fig. Instead, a different primitive rule would be required for each specific type.

Multi-language Programming

In this section, we demonstrate how Twig can be used for multi-language programming. In a short example program, we will show how Twig can describe typemaps that convert between three different type systems – C, Python, and JSON – and how it allows us to automatically reason about identity relationships among them. JSON is Java-Script Object Notation, an intermediate representation and serialization format commonly used in web programming [11]. We describe a set of expressions similar to the kind of typemaps found in systems such as SWIG, but which leverage Twig's operators to build complex typemaps from simple ones.

We then show how user-defined expression reductions (see Chapter V) can be used to optimize the composed typemaps.

Imagine that we have a Python object representing some (very) simple personal contact information:

```
class Contact():
  def __init__(self,name,age):
    self.name = name
    self.age = age
```

We would like to be able to use this object from a C program, and we can use a typemap to perform the conversion. We can convert the data in a number of different ways. One way is to convert the data directly from Python to C using the Python/C API [12]. Another way, less direct but more flexible, is to first marshall the Python object to a JSON string and then unmarshal the JSON into C. We might prefer to convert via JSON if, for example, we were writing typemaps intended to work with a variety of languages, thus avoiding the n^2 binding problem described in Section 2.1.

In this example, we use the multi-language generation scheme described in Section 6.3. First, we write some primitive rules that unpack the object into its fields. Note that we tell Twig that the target language in this section of code is Python, using the `@language(Python)` directive.

```
@language(Python)


name = [contact -> py(string)] <<<
  $out = $in.name
```

```
>>>

age = [contact -> py(int)] <<<
  $out = $in.age
>>>
```

The terms `py(string)` and `py(int)` represent a string and an integer in the context of Python. The term `contact` represents the `Contact` object defined above.

Next, we define some primitive rules for marshalling the Python data types in JSON. Note that for the JSON types, like `py(json(int))`, we retain the outer `py` constructor – this is to indicate that the JSON string is still represented in the context of the Python language.

```
py_int_to_json = [py(int) -> py(json(int))] <<<
  $out = int_to_json($in)
>>>

py_string_to_json = [py(string) -> py(json(string))] <<<
  $out = string_to_json($in)
>>>

py_tuple_to_json =
  [(py(json(X)),py(json(Y))) -> py(json(pair(X,Y)))] <<<
    $out = '[' + ','.join($in1,$in2) + ']'
>>>
```

The rule `py_tuple_to_json` lifts a pair of JSON objects, represented by a Twig tuple, to a pair within JSON.

Now, using these rules, we can write a more general-purpose Python-to-JSON typemap:

```
py_to_json_step =
  py_int_to_json | py_string_to_json | py_tuple_to_json


py_to_json = #fix(X, (#all(X) | T) ; py_to_json_step)
```

The rule py_to_json uses the #fix operator to recursively convert nested tuples of Python objects to JSON.

Finally, to convert our Python Contact object to JSON, we can write the following rule:

```
py_address_to_json = #fan(2);{name,age};py_to_json
```

This rule sends the single contact object to a congruence extracting the name and age, resulting in a 2-tuple of Python data types. Then, it converts that tuple to JSON.

Next, we need primitive rules to convert from JSON into C. We use the Jansson library [10] to manipulate JSON data in C. Note that at this point we use the directive @language(C) to tell Twig to use C as the target language.

```
@language(C)


json_to_tuple = [json(pair(X,Y)) -> (json(X),json(Y))] <<<
  $out1 = json_array_get($in,0);
  $out2 = json_array_get($in,1);
>>>
```

```
json_to_int = [json(int) -> int] <<<
  $out = json_int_value($in);
>>>

json_to_string = [json(string) -> string] <<<
  $out = json_string_value($in);
>>>

from_json_step =
  json_to_int | json_to_string | json_to_tuple

from_json = #fix(X, from_json_step ; (#all(X) | T))
```

Notice that, in the rules above, we are using terms like `json(int)` to represent JSON types, rather than `py(json(int))`. This is because in these rules, we are working with JSON data in a C context (here, we equate the absence the `py` constructor to indicate that the type is in C). In both Python and C, with the libraries we chose, JSON data is represented by a string. So, in order to move from Python to C, we need to be able to convert a Python JSON string to a C JSON string. The following rule accomplishes this, using the Python/C API to convert the string.

```
py_json_to_c = [py(json(X)) -> json(X)] <<<
  json_error_t $tmp1;
  char *$tmp2 = PyString_AsString($in);
  $out = json_loads($tmp2, JSON_DECODE_ANY, &$tmp1);
```

```
>>>
```

In py_json_to_c, we use a variable X in the rule's input and output pattern. In this case, we do not care what the underlying JSON data represents, and we can convert it regardless. This kind of polymorphism is a powerful feature of Twig.

Putting our rules together, we can define a main expression which will convert a Python Contact first to JSON, and then to C.

```
main = py_address_to_json;py_json_to_c;from_json
```

If we run Twig with this program on the input term contact (the only term for which this program will succeed), the output will be the tuple term (string,int) representing a pair of C types. Twig will generate the following Python code:

```
def gen1_py(in):
  x1 = in.name
  x2 = in.age
  x3 = string_to_json(x1)
  x4 = int_to_json(x2)
  x5 = tuple_to_json(x3,x4)
  return x5
```

and this C code:

```
struct gen1_tuple {
  char *x1;
  int x2;
}
```

```
gen1_tuple gen1_c(PyObject *in) {
  gen1_tuple ret;
  PyObject *x1;
  char *x2;
  json_t *x3, *x4, *x5;
  char *x6;
  int x7;
  x1 = call_python("target","gen1_py",in);
  json_error_t tmp1;
  x2 = PyString_AsString(x1);
  x3 = json_loads(x2, JSON_DECODE_ANY, &tmp1);
  x4 = json_array_get(x3,0);
  x5 = json_array_get(x3,1);
  x6 = json_string_value(x4);
  x7 = json_integer_value(x5);
  ret.x1 = x6;
  ret.x2 = x7;
  return ret;
}
```

The generated C function needs to return a pair of values, so we automatically generate a C struct type for this purpose. Our implementation currently handles this case in an ad hoc manner, and we are working on a more robust solution.

Now, let us suppose we add some rules that leverage the Python/C API to convert data directly, without marshaling through JSON.

```
py_string_to_c = [py(string) -> string] <<<
  $out = PyString_AsString($in);
>>>


py_int_to_c = [py(int) -> int] <<<
  $out = (int)PyInt_AsLong($in);
>>>


py_to_c_step = py_string_to_c | py_int_to_c


py_to_c = #fix(X, (py_to_c_step | #all(py_to_c_step)) ; (#all(X) | T))
```

We can introduce the following expression reduction directive to automatically replace the JSON marshaling process with calls to the Python/C API.

```
@reduce py_to_json;json_to_c;from_json => py_to_c
```

This directive exploits the isomorphic relationship between conversions from Python to JSON, and then JSON to C, and those directly from Python to C. This relationship is illustrated in Figure 6, where f, g, and h are Twig expressions transforming Python to JSON, JSON to C, and Python to C, respectively. In the figure $f;g = h$, which is the relationship we wish to exploit using an expression reduction.

The original version of `main` was

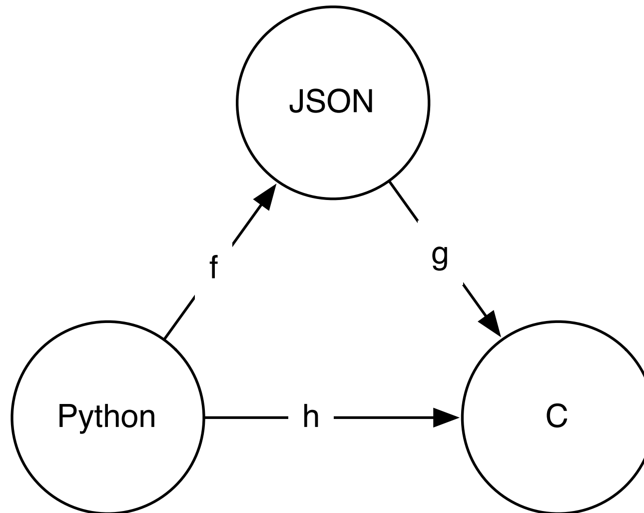

FIGURE 6. Typemap identities.

```
main = py_address_to_json;py_json_to_c;from_json
```

which is equivalent to

```
main = #fan(2);{name,age};py_to_json;py_json_to_c;from_json
```

after substituting for the value of `py_address_to_json`. After applying the expression reduction directive, `main` is rewritten to

```
main = #fan(2);{name,age};py_to_c
```

Applying this new expression to the term `contact` still yields the output term `(string,int)`, but the generated code is much simpler and more efficient. The Python code is now:

```
def gen1_py(in):
```

```
  x1 = in.name
  return x1


def gen2_py(in):
  x1 = in.age
  return x1
```

and the C code is:

```
gen1_tuple gen1_c(PyObject *in) {
  gen1_tuple ret;
  PyObject *x1,*x2;
  char *x3;
  int x4;
  x1 = call_python("target","gen1_py",in);
  x2 = call_python("target","gen2_py",in);
  x3 = PyString_AsString(x1);
  x4 = (int)PyInt_AsLong(x2);
  ret.x1 = x3;
  ret.x2 = x4;
  return ret;
}
```

It is notable that systems such as SWIG, based on monolithic typemaps, could not perform reductions of this kind. This is because, unlike Twig expressions, monolithic typemaps have no structure that SWIG can rewrite.

GPU programming

In this section we present an example Twig program that targets C code interacting with a graphics processing unit (GPU). This style of program is known as "hybrid" computing, since the program is intended to run on a hybrid system, incorporating more than one device. Hybrid computing has some interesting features in common with multi-language programming, including reasoning about types in different contexts.

CUDA [68] is a well-known and relatively simple API for interacting with GPUs from C. We use this API in our example below. In the interest of clarity, we simplify our code by omitting CUDA's setup and teardown logic and assuming that values such as `N_BLOCKS`, `BLOCK_SIZE`, `SIZE`, and `N` are constants defined elsewhere. A real application might pass these values within the rules, perhaps using tuples or custom data structures.

First, we define two primitive rules for moving data to and from the GPU:

```
copyToGPU=[array(float) -> gpu(array(float))] <<<
  cudaMalloc((void **)&$out,SIZE);
  cudaMemcpy($out,$in,SIZE,cudaMemcpyHostToDevice);
>>>


copyFromGPU=[gpu(array(float)) -> array(float)] <<<
  $out = malloc(SIZE);
  cudaMemcpy($out,$in,SIZE,cudaMemcpyDeviceToHost);
>>>
```

The rule `copyToGPU` copies an array of floating point numbers from main memory to the GPU's separate memory hierarchy. First, it allocates memory on the device, then it invokes the appropriate CUDA call to copy the data. In a similar fashion, `copyFromGPU` copies an array of `float`s from GPU back to the system.

Next, we define primitive rules for invoking two different GPU *kernels* on the data, after it has been moved to the GPU. A kernel is essentially a function on arrays, performed on the GPU.

```
kernelFoo = [gpu(array(float)) -> gpu(array(float))] <<<
  foo \<\<\<N_BLOCKS,BLOCK_SIZE/>/>/>($in, N);
  $out = $in;
>>>


kernelBar = [gpu(array(float)) -> gpu(array(float))] <<<
  bar \<\<\<N_BLOCKS,BLOCK_SIZE/>/>/>($in, N);
  $out = $in;
>>>
```

These two rules do almost exactly the same thing, except that `kernelFoo` invokes a kernel called "foo" while `kernelBar` invokes a kernel called "bar." The kernel names refer to CUDA functions defined elsewhere. Note that we needed to escape the triple angle brackets in CUDA's syntax; otherwise, they would interfere with Twig's own block syntax.

Next, we can define our main program, using some auxiliary expressions.

```
runFoo = copyToGPU;kernelFoo;copyFromGPU
```

```
runBar = copyToGPU;kernelBar;copyFromGPU
main   = runFoo;runBar
```

The expressions `runFoo` and `runBar` will perform a single logical function on the GPU. Note that these expressions will be semantically valid in any context where they appear, since they ensure that the data is moved onto the GPU before the kernel is executed, and then that the data is copied back. The programmer may effectively ignore the fact that `runFoo` and `runBar` interact with the GPU; they appear to perform a function on a local array. This abstraction is considerably simpler than that presented by raw CUDA calls.

The `main` expression is the top level of the program. It executes the two kernels `foo` and `bar` in sequence. As noted above, the invocations of `runFoo` and `runBar` would normally result in a copy to and from the GPU. This is a conservative design, hiding the details of the interaction with the GPU from the programmer.

Evaluating this expression on the input term `array(float)` will output the result term `array(float)`, along with the following generated code:

```
float *twig_gen_fun(float *in) {
  float *tmp01,*tmp02,*tmp03,*tmp04,*tmp05,*tmp06,*tmp07;
  tmp01 = in;
  cudaMalloc((void **)&tmp02,SIZE);
  cudaMemcpy(tmp02,tmp01,SIZE,cudaMemcpyHostToDevice);
  foo <<<N_BLOCKS,BLOCK_SIZE>>> (tmp02,N);
  tmp03 = tmp02;
  tmp04 = (float *)malloc(SIZE * sizeof(float));
  cudaMemcpy(tmp04,tmp03,SIZE,cudaMemcpyDeviceToHost);
```

```
  cudaMalloc((void **)&tmp05,SIZE);
  cudaMemcpy(tmp05,tmp04,SIZE,cudaMemcpyHostToDevice);
  bar <<<N_BLOCKS,BLOCK_SIZE>>> (tmp05,N);
  tmp06 = tmp05;
  tmp07 = (float *)malloc(SIZE * sizeof(float));
  cudaMemcpy(tmp07,tmp06,SIZE,cudaMemcpyDeviceToHost);
  return tmp07;
}
```

The generated function is assigned the unique name `twig_gen_fun`. It takes a pointer to an array of floating point numbers somewhere in system memory and returns a pointer to the transformed array.

This code is correct, but unfortunately it contains a redundant memory copy. That is, the program copies an array from the GPU to system memory, and then immediately copies it back to the GPU without modification. Copying across devices is a relatively expensive operation. For large arrays, or for code in a tight loop, this kind of redundancy will significantly reduce performance.

To see why the redundant copy is introduced, we substitute the names `runFoo` and `runBar` with the expressions they denote in `main`. Now we can see that `main` is equivalent to this expression:

```
main = copyToGPU;
       kernelFoo;
       copyFromGPU;copyToGPU;
       kernelBar;
       copyFromGPU
```

Notice that in the middle, the sequence `copyFromGPU;copyToGPU` is unnecessary. It is introduced into the program as an artifact of the sequence of `runFoo` with `runBar`. We can solve this problem using an *expression reduction*, as described in Chapter V. The reduction directive

```
@reduce copyFromGPU;copyToGPU => T
```

instructs Twig to search for the expression `copyFromGPU;copyToGPU` within `main`, and to replace it with `T`, the identity rule, wherever it occurs. In this context `T` serves essentially as a no-op – it does not generate any code. After the reduction is performed, the expanded version of `main` has had the extra copies removed and replaced by `T`:

```
main = copyToGPU;
       kernelFoo;
       T;
       kerneBar;
       copyFromGPU
```

Based on this expression, Twig will generate the following code:

```
float *twig_gen_fun(float *in) {
  float *tmp01,*tmp02,*tmp03,*tmp04,*tmp05;
  tmp01 = in;
  cudaMalloc((void **)&tmp02,SIZE);
  cudaMemcpy(tmp02,tmp01,SIZE,cudaMemcpyHostToDevice);
  foo <<<N_BLOCKS,BLOCK_SIZE>>> (tmp02,N);
  tmp03 = tmp02;
```

```
  bar <<<N_BLOCKS,BLOCK_SIZE>>> (tmp03,N);
  tmp04 = tmp03;
  tmp05 = (float *)malloc(SIZE * sizeof(float));
  cudaMemcpy(tmp05,tmp04,SIZE,cudaMemcpyDeviceToHost);
  return tmp05;
}
```

This code omits the redundant copy, and is therefore more efficient. Although this example is simple, it demonstrates the power of user-defined expression reductions. The reduction rule given here could be paired with the `copyToGPU` and `copyFromGPU` rules in a module intended for consumption by domain programmers, allowing them to perform GPU operations without worrying about the design of the rules. Sophisticated users, however, could add their own rules or even application-specific reductions, enabling very powerful and customizable code generation based on domain-specific logic.

Integrating Twig in Practice

The preceding example demonstrates how Twig can be used to generate a block of CUDA code. On its own, of course, the block is useless – it must be integrated into a larger program. At the moment, Twig uses a fairly rudimentary process to accomplish this. We describe that process in this section.

Twig is used to generate a block of code which is then incorporated into a larger program that calls it using the language's file inclusion facility. First, the programmer provides the Twig program along with an input type to the Twig tool. Twig will evaluate the program on the input type, producing (if successful) an output type along with a block of code in the target language that will transform

the input type to the output type. Our Twig implementation will optionally wrap the block of code inside a function that takes values having the input type(s) as arguments, and returns a value having the output type. The block of code may be redirected to a file. A main program may then include the generated function, and invoke it within its own functions.

The example above generates a C/CUDA function called `twig_gen_fun`, which takes a floating point array and produces another floating point array. We use Twig to redirect this code to a file called `twig.cu`. The main CUDA program resides in a file called `main.cu`, and contains the following code:

```
#include <stdio.h>
#include <cuda.h>

const int N = 10;
const size_t SIZE = N * sizeof(float);
const int BLOCK_SIZE = 4;
const int N_BLOCKS = N / BLOCK_SIZE + (N % BLOCK_SIZE == 0 ? 0:1);

__global__ void foo(float *a, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx<N) {
    a[idx] = a[idx] * a[idx];
  }
}

__global__ void bar(float *a, int N) {
```

```
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx<N) {
    a[idx] = a[idx] + 1;
  }
}

#include "twig.cu"

int main(void) {
  float *input, *result;
  input = (float *)malloc(SIZE);
  for(int i=0; i < N; i++) {
    input[i] = (float)i;
  }
  result = twig_gen_fun(input);
  for(int i=0; i < N; i++) {
    printf("%d -> %f\n", i, result[i]);
  }
  return 0;
}
```

Notice how we include `twig.cu` in the main file. The include occurs only `after` we have defined the various constructs (such as the `foo` and `bar` kernels, and constants like `BLOCK_SIZE`, that are needed within the generated code. Compiling `main.cu` with the regular CUDA compiler (`nvcc`) will suffice to produce an executable program.

CHAPTER VIII

FUTURE WORK

There are a number of potential avenues for future research with Twig.

Implementation

We presented our current implementation of Twig, called `twigc`, in Chapter VI. We are quite happy with the overall design of the interpreter, and in particular with how it we have embedded the core semantics in Haskell. There are still many improvements and additional features we would like to incorporate.

Term-to-type Specification

In order to generate C code, Twig needs to be configured with a mapping from terms representing C types to the appropriate syntax with which to declare variables of that type. For example, we have terms such as `ptr(int)` which, in the generated C code, must be declared like so:

```
int *x;
```

Automatically generating casts in C presents a similar problem. In our current implementation, `twigc` must be parameterized with a list of pairs, matching terms to the C syntax of the type they represent. So, users must provide a list like this one:

```
int = int
float = float
ptr(int) = int *
```

```
ptr(float) = float *

mystruct = struct MyStruct
```

Providing this list quickly becomes tedious if there are lots of user-defined types, or many types with recursive structure, or both. There are a few possible solutions to this problem. One idea is to add the ability for `twigc` to parse some structure in the list, like so

```
int = int

float = float

ptr(X) = X *
```

In this scheme, `X` is a variable that can be matched against the appropriate term, and then substituted with the appropriate syntax where necessary. This would make structured types easier to specify. Another idea would be to provide a built-in mapping for the basic C types, and then have Twig automatically decide on a term structure for user-defined types. For example, each `struct` in a header file could be mapped to a term based on the structure name, e.g. `struct Foo` would be mapped to the term `foo`. This would eliminate the need for users to provide the term-to-type mapping, but would reduce flexibility. In particular, users could not provide their own term structure, which is sometimes useful (see the example in Section 7.2).

Expression Reductions

As we mentioned in Chapter V, our current implementation of expression reductions is ad hoc. We export the Twig program and reduction rules as strings, parse and process them in the Maude term rewriting system, and then import the

result back into Twig. In addition to making Twig dependent on an external tool, this approach is potentially quite inefficient for large programs. Instead, it would be both beneficial and interesting to use Twig's own language to rewrite its expression tree.

In principle, rewriting Twig expressions with Twig itself is easy to do. Twig is based on System S, a core language for term rewriting. It is simple to specify reduction rules – they are simply primitive rules in Twig. To apply the rules, we can exploit Twig's ability to encode very general rewriting strategies. For example, to perform a top-down rewrite, we could use an expression such as

```
try(s) = s | T
repeat(s) = #fix(x, try(s ; x))
topdown(s) = #fix(x, s | #one(x))
reduce(s) = repeat(topdown(s))
```

The expression `reduce` takes an expression argument `s` and applies it to one term in a tree, searching in a top-down fashion. It then repeats the procedure until no rewrites are able to be applied, and returns the transformed tree. Alternatively, we could specify a similar bottom-up strategy, or a mix of the two.

Note that the strategies would require parameterized rules, covered in Section 8.2, which are not currently implemented in `twigc`.

Theory

The formal theory underlying Twig could be extended in a number of ways.

Environments and Conditional Rules

System S has different "levels" of semantics for term rewriting, corresponding to different levels of complexity and different features. A feature, which Twig does not currently incorporate into its own semantics, are the *environment operators* of System S. They environment operators are called *match* and *build* – they separate the operation of primitive rules into two distinct phases. In the first phase, the term is matched against an input pattern, and its variables are bound in an environment. This environment is then passed along via expressions and possibly extended. The environment is used to either build a new term from its bound values and an output pattern, or else used to test whether its bound values meet some condition. Essentially, this feature would allow Twig to incorporate "conditional" rules, i.e. rules which only match if certain other rules have matched before, as indicated by the environment.

This would be a relatively straightforward addition to the semantics, since it would follow System S fairly closely, and would not require extensive modification to accommodate our extended code generation semantics.

Parameterized Rules

In System S, expressions are first-order. That is, they may be parameterized by other expressions. In Twig, parameterized expressions might look like this:

```
try(s) = s | T
```

This statement creates a rule called `try` which is parameterized by an expression `s`. To invoke `try`, an expression must be passed as its argument, like so:

```
foo = [foo -> bar] <<< ... >>>
```

```
main = try(foo)
```

Here, `foo` is a primitive rule, but it could be any Twig expression.

Parameterized expressions would facilitate reuse and modularity. They turn out to be somewhat tricky to implement, since we must ensure that circular references are not introduced. We can solve this problem by stipulating that expressions must be defined before they are referenced, and that expressions may not reference themselves.

Code Generation

Our current model for code generation is intentionally abstract, in the sense that it does not assume very much about the languages it might be used to generate. The abstraction is a feature – it allows Twig to be used to generate different languages without altering its core semantics. However, there are certain features that, in our experience, would make code generation more convenient.

First, we often write primitive rules that allocate memory. It would be nice if the code generation model allowed for "closing blocks," i.e. a way to call some tear down code at the end of the generated code's lifetime. Clearly, this is a difficult problem in general, since it may not always be clear when objects should be deallocated or files closed. Still, one could imagine rules like the following being useful:

```
alloc_array = [int -> array(int)] <<<
  $out = malloc($in * sizeof(int));
>>><<<
```

```
  free($out);
>>>
```

In this (invented) syntax, Twig would automatically insert the second block wherever it determines that the variable generated for `$out` goes out of scope. One challenge with supporting such a syntax would be incorporating it into the abstract model.

Higher-order Rules

In many cases, Twig could generate much more complex kinds of code if we could parameterize primitive rules with expressions. Essentially, this would make rules higher-order. This idea is somewhat different than the parameterized expressions presented in Section 8.2, which would make expressions first-order.

Higher-order rules would allow Twig to generate code that takes context into account. For example, the following rule (using a made-up syntax), would transform an array of elements of type X to an array of elements of type Y, given another transformation named s that transforms X to Y.

```
map = [array(X) -> array(Y) | s : X -> Y] <<<
  $out = malloc(N * sizeof($Y));
  for(int i = 0; i < N; i++) {
    $out[i] = $s($in[i]);
  }
>>>
```

The example implements the familiar *map* function, common in functional programming languages.

Expression Reductions

Currently, expression reductions are not formalized with the same degree of rigor as the rest of Twig's semantics. In particular, we rely on our implementation to decide how Twig rewrites expressions. These details could matter in practice. For example consider the directives:

```
@reduce foo => bar
@reduce foo;foo => baz
```

Clearly, these reductions are ambiguous in the absence of a formal description of how they will be applied. This is because the rules are not confluent.

Twig will currently send both rules, in the order given, to Maude. At that point we defer to Maude's implementation. Currently Twig side-steps this problem by simply insisting that expression reductions should be confluent, and that the result is undefined if they are not. However, if the system was formalized, or if we allowed users to specify rewriting strategies, we could allow for non-confluent rules.

In addition, we think it would be quite useful to allow for variables in expression reductions, allowing users to write directives such as

```
@reduce foo;X;bar => foo
```

or

```
@reduce X;foo;Y => X;Y
```

This would greatly increase the flexibility and reusability of reduction directives.

CHAPTER IX

CONCLUSION

We have presented Twig, a language for writing composable typemaps. As illustrated in our evaluation, we have found that Twig is an interesting and useful system for creating configurable domain-specific languages. These languages are based on Twig's core semantics, but incorporate primitive rules and expression reduction directives that take the domain's structure into account.

Summary of Results

Twig is based on Systems S, a core language for term rewriting. We have extended System S and equipped it with semantics that allow it to generate code. Our model for code generation is abstracted from any particular language, allowing Twig to be used to generate a variety of target languages.

Twig uses terms to represent types in the target language. A Twig expression is a program that transforms one term into another. According to how the term transformation takes place, code is generated to transform values from and to target language types corresponding to the terms. The simplest expressions are primitive rules, which include blocks of code to be generated in a designated target language. Primitive rules can be combined into more complex expressions using Twig's operators and built-in expressions.

Twig allows users to customize the meaning of expressions by introducing their own expression reduction rules. Expression reductions are applied by Twig before the program is evaluated, and can cause a given Twig expression to be

rewritten into another. This can be useful for introducing optimizations and other automated reasoning that take advantage of domain-specific program structure.

We demonstrated how Twig can be used for multi-language programming as well as GPU programming. In both these contexts, we introduce domain-specific expression reductions that potentially improve the efficiency of the final program. We also compared a Twig program with a similar program written in SWIG, a language with a similar notion of typemapping, and found that, due to its operators and abstract code generation model, Twig's language allows greater flexibility, modularity, and reusability than SWIG's traditional, monolithic typemaps.

Advantages

As we discussed in Chapter I, the primary goal of our work was to design a language that allowed high-level, domain-specific logic to co-exist with low-level, detail-oriented code, with each informing the structure of the other, in a single program specification. Twig represents a step in this direction.

The main advantage conferred by Twig is that its simple, high-level semantics restrict the kinds of programs that can be written. While this advantage is also a limitation (see below), it does make the structure of Twig programs easy to reason about. We exploited this capability in our multi-language and GPU programming examples, using expression reduction directives to recognize simple but useful patterns in the code, and then rewriting those patterns into more efficient ones. The equivalent C or Python code would be very difficult to automatically rewrite in a similar fashion, since the patterns are obscured by the details of declaring variables, allocating memory, and so on.

Another advantage of Twig is its ability to mix high- and low-level code in its primitive rules, and that these rules are the only place where such mixing occurs. A corollary of this observation is that, once a set of primitive rules are properly designed, relatively unsophisticated users can make use of them by using them as "building blocks," that is, using a provided set of primitive rules by combining them with Twig's operators and built-in expressions only. This suggests a possible model of usage for Twig: a domain expert first designs and crafts a set of primitive rules and expression reduction directives that capture the required functionality for a domain, and then less sophisticated users can write programs without having to worry about writing any low-level code. If the expression reductions are well-designed, they may not even have to worry about optimization for common cases.

Limitations

Above, we observed that the simple, somewhat sparse structure of Twig's language makes it easy to reason about and to automatically rewrite. The dual of this observation is that writing large or complex programs in such a basic language can be challenging. This may limit Twig's applicability to relatively simple jobs, such as small typemaps for multi-language or multi-target programs. Writing larger programs cries out for more versatile abstractions such as the λ-calculus or traditional imperative control structures, but we think that it is precisely these facilities that make traditional programs so difficult to reason about. This tension may be inherent to programming language design.

In addition, our current model code generation does not adequately address the kinds of real-world problems faced by multi-language systems, such as resource management or error handling, discussed in Section 2.1. We believe it will be

challenging to add these features while retaining the highly abstract nature of the model. Again, intuitively there seems to be a fundamental tension here, between the expressivity and fidelity of a code generation model on the one hand, and its ability to be adapted to many different languages on the other.

Twig in Context

Twig represents a significant contribution to the state-of-the-art for typemapping languages. We believe that Twig's language, or something similar, should be adopted by tools that require programmers to write small programs to that transform data across different domains. In particular, the flexibility afforded by Twig's operators and its abstract approach to code generation recommend it for this purpose over current, monolithic approaches.

Twig is also an interesting experiment that moves towards achieving the goals we presented in the introduction. That is, Twig allows programmers to mix high- and low-level code in a way that retains the former's ease of development and reasoning, as well as the latter's flexibility and power. Our future research will determine whether we can resolve the outstanding issues with code generation and general usability that we find currently hamper Twig as a language for general program development.

APPENDIX

PROOFS

In this appendix we present proofs of some properties of Twig's operators and expressions.

Associative Operators

In this appendix we present proofs that sequence and choice are associative. This allows us to write expressions such as $f;g;h$ or $f|g|h$ without ambiguity.

Sequence

We want to show that $(f;g);h = f;(g;h)$. To do this, we have to prove two cases:

$$t \xrightarrow{(f;g);h} (t', m) \Leftrightarrow t \xrightarrow{f;(g;h)} (t', m) \tag{A.1}$$

and also

$$t \xrightarrow{(f;g);h} \perp \Leftrightarrow t \xrightarrow{f;(g;h)} \perp \tag{A.2}$$

Case #1:

We have to show

$$t \xrightarrow{(f;g);h} (t', m) \Rightarrow t \xrightarrow{f;(g;h)} (t', m) \tag{A.3}$$

So we assume the premise:

$$t \xrightarrow{(f;g);h} (t', m) \tag{A.4}$$

By (4.3):

$$t \xrightarrow{f;g} (t'', m_{fg}) \qquad t'' \xrightarrow{h} (t', m_h) \tag{A.5}$$

and let $m = m_{fg} + m_h$. By (4.3):

$$t \xrightarrow{f} (t''', m_f) \qquad (t''' \xrightarrow{g} (t'', m_g) \tag{A.6}$$

and let $m_{fg} = m_f + m_g$. By (4.3):

$$t \xrightarrow{f} (t''', m_f) \qquad t''' \xrightarrow{g;h} (t', m_g + m_h) \tag{A.7}$$

By (4.3):

$$t \xrightarrow{f;(g;h)} (t', m_f + m_g + m_h) \tag{A.8}$$

The reverse case of the implication proceeds almost identically.

Case #2:

We want to prove

$$t \xrightarrow{(f;g);h} \perp \Rightarrow t \xrightarrow{f;(g;h)} \perp \tag{A.9}$$

So we assume the premise:

$$t \xrightarrow{(f;g);h} \perp \tag{A.10}$$

and now there are three possible sub-cases.

Sub-case #1:

By (4.4):

$$t \xrightarrow{f;g} \bot \tag{A.11}$$

By (4.4):

$$t \xrightarrow{f} \bot \tag{A.12}$$

By (4.4):

$$t \xrightarrow{f;(g;h)} \bot \tag{A.13}$$

Sub-case #2:

By (4.4):

$$t \xrightarrow{f;g} \bot \tag{A.14}$$

By (4.5):

$$t \xrightarrow{f} (t', m_f) \qquad t' \xrightarrow{g} \bot \tag{A.15}$$

By (4.4):

$$t' \xrightarrow{g;h} \bot \tag{A.16}$$

By (4.3):

$$t \xrightarrow{f;(g;h)} \bot \tag{A.17}$$

Sub-case #3:

By (4.5):

$$t \xrightarrow{f;g} (t', m_{fg} \qquad t' \xrightarrow{h} \bot \tag{A.18}$$

By (4.3):

$$t \xrightarrow{f} (t'', m_f) \qquad t'' \xrightarrow{g} (t', m_g) \tag{A.19}$$

where $m_{fg} = m_f + m_g$. By (4.4):

$$t'' \xrightarrow{g;h} \bot \tag{A.20}$$

By (4.3):

$$t \xrightarrow{f;(g;h)} \bot \tag{A.21}$$

The reverse case of the implication proceeds almost identically.

Choice

We want to show that $(f|g)|h = f|(g|h)$. To do this, we have to prove two cases:

$$t \xrightarrow{(f|g)|h} (t', m) \Leftrightarrow t \xrightarrow{f|(g|h)} (t', m) \tag{A.22}$$

and also

$$t \xrightarrow{(f|g)|h} \bot \Leftrightarrow t \xrightarrow{f|(g|h)} \bot \tag{A.23}$$

Case #1:

We assume the premise:

$$t \xrightarrow{(f|g)|h} (t', m) \tag{A.24}$$

There are three sub-cases:

Sub-case #1:

By (4.6)

$$t \xrightarrow{f|g} (t', m) \tag{A.25}$$

By (4.6)

$$t \xrightarrow{f} (t', m) \tag{A.26}$$

By (4.6)

$$t \xrightarrow{f|(g|h)} (t', m) \tag{A.27}$$

Sub-case #2:

By (4.6)

$$t \xrightarrow{f|g} (t', m) \tag{A.28}$$

By (4.7)

$$t \xrightarrow{f} \bot \qquad t \xrightarrow{g} (t', m) \tag{A.29}$$

By (4.6)

$$t \xrightarrow{f} \bot \qquad t \xrightarrow{g|h} (t', m) \tag{A.30}$$

By (4.7)

$$t \xrightarrow{f|(g|h)} (t', m) \tag{A.31}$$

Sub-case #3:

By (4.6)

$$t \xrightarrow{f|g} \bot \qquad t \xrightarrow{h} (t', m) \tag{A.32}$$

By (4.8)

$$t \xrightarrow{f} \bot \qquad t \xrightarrow{g} \bot \qquad t \xrightarrow{h} (t', m) \tag{A.33}$$

By (4.6)

$$t \xrightarrow{f} \bot \qquad t \xrightarrow{g|h} (t', m) \tag{A.34}$$

By (4.7)

$$t \xrightarrow{f|(g|h)} (t', m) \tag{A.35}$$

Case #2:

We assume the premise:

$$t \xrightarrow{(f|g)|h} \bot \tag{A.36}$$

By (4.8):

$$t \xrightarrow{f|g} \bot \qquad t \xrightarrow{h} \bot \tag{A.37}$$

By (4.8):

$$t \xrightarrow{f} \bot \qquad t \xrightarrow{g} \bot \qquad t \xrightarrow{h} \bot \tag{A.38}$$

By (4.8):

$$t \xrightarrow{f} \bot \qquad t \xrightarrow{g|h} \bot \tag{A.39}$$

By (4.8):

$$t \xrightarrow{f|(g|h)} \bot \tag{A.40}$$

Expression Identities

In this section we prove the identity relationships between expressions. We exploit these relationships in expression reductions, as described above.

Distribution of sequence over choice

First, we show that sequence distributes over choice from the left, but not from the right.

<u>From the left</u>

First we will show that sequence distributes over choice from the left. We must prove that

$$\forall f, g, h \in S \ : \ f;(g|h) = (f;g)|(f;h) \tag{A.41}$$

There are two cases to be proved. First,

$$t \xrightarrow{f;(g|h)} t' \implies t \xrightarrow{(f;g)|(f;h)} t' \tag{A.42}$$

and second,

$$t \xrightarrow{f;(g|h)} \perp \implies t \xrightarrow{(f;g)|(f;h)} \perp \tag{A.43}$$

To prove the first case, we assume the premise

$$t \xrightarrow{f;(g|h)} t' \tag{A.44}$$

working backwards by (4.3) we can conclude

$$t \xrightarrow{f} t'' \qquad t'' \xrightarrow{g|h} t' \tag{A.45}$$

There are two possible sub-cases from $t'' \xrightarrow{g|h} t'$. From (4.6) we could conclude

$$t'' \xrightarrow{g} t' \tag{A.46}$$

and since we have established $t \xrightarrow{f} t''$ we have

$$t \xrightarrow{f} t'' \qquad t'' \xrightarrow{g} t' \tag{A.47}$$

and thus, by (4.3), we have

$$t \xrightarrow{f;g} t' \tag{A.48}$$

and by (4.6) we have

$$t \xrightarrow{(f;g)|(f;h)} t' \tag{A.49}$$

and this sub-case is complete. The next sub-case from $t'' \xrightarrow{g|h} t'$ by (4.7) is that

$$t'' \xrightarrow{g} \bot \qquad t'' \xrightarrow{h} t' \tag{A.50}$$

By (4.4):

$$t \xrightarrow{f;g} \bot \tag{A.51}$$

By (4.3):

$$t \xrightarrow{f;h} t' \tag{A.52}$$

By (4.7):

$$t \xrightarrow{(f;g)|(f;h)} t' \tag{A.53}$$

And this sub-case is complete. Next we must show

$$t \xrightarrow{f;(g|h)} \bot \implies t \xrightarrow{(f;g)|(f;h)} \bot \tag{A.54}$$

We assume the premise

$$t \xrightarrow{f;(g|h)} \bot \tag{A.55}$$

Case #1:

By (4.4):

$$t \xrightarrow{f} \bot \tag{A.56}$$

By (4.4):

$$t \xrightarrow{f;g} \bot \qquad t \xrightarrow{f;h} \bot \tag{A.57}$$

By (4.8):

$$t \xrightarrow{(f;g)|(f;h)} \bot \tag{A.58}$$

Case #2:

By (4.5):

$$t \xrightarrow{f} t' \qquad t' \xrightarrow{g|h} \bot \tag{A.59}$$

By (4.8):

$$t \xrightarrow{f} t' \qquad t' \xrightarrow{g} \bot \qquad t' \xrightarrow{h} \bot \tag{A.60}$$

By (4.5):

$$t \xrightarrow{f;g} \bot \qquad t \xrightarrow{f;h} \bot \tag{A.61}$$

By (4.8):

$$t \xrightarrow{(f;g)|(f;h)} \bot \tag{A.62}$$

From the right

It turns out that although sequence distributes over choice from the left, it does not do so from the right.

$$(f|g); h \not\Longrightarrow (f; h)|(g; h) \tag{A.63}$$

We can easily show this with a counterexample. Assume we have three expressions

$$t_1 \xrightarrow{f} t_2 \qquad t_1 \xrightarrow{g} t_3 \qquad t_3 \xrightarrow{h} t_4 \tag{A.64}$$

Now we can see that

$$t_1 \xrightarrow{(f|g);h} \bot \tag{A.65}$$

but

$$t_1 \xrightarrow{(f;h)|(g;h)} t_4 \tag{A.66}$$

So

$$t \xrightarrow{(f|g);h} \bot \not\Longrightarrow t \xrightarrow{(f;h)|(g;h)} \bot \tag{A.67}$$

which would be required to show (A.63).

Distribution of congruence over sequence

Here, we want to show that congruence distributes over sequence, i.e., that

$$\{r_1, \ldots, r_n\}; \{s_1, \ldots, s_n\} \implies \{r_1; s_1, \ldots, r_n; s_n\} \tag{A.68}$$

Case #1:

$$(t_1, \ldots, t_n) \xrightarrow{\{r_1,\ldots,r_n\};\{s_1,\ldots,s_n\}} (t'_1, \ldots, t'_n) \implies (t_1, \ldots, t_n) \xrightarrow{\{r_1;s_1,\ldots,r_n;s_n\}} (t'_1, \ldots, t'_n) \tag{A.69}$$

By (4.3):

$$(t_1, \ldots, t_n) \xrightarrow{\{r_1,\ldots,r_n\}} (t''_1, \ldots, t''_n) \qquad (t''_1, \ldots, t''_n) \xrightarrow{\{s_1,\ldots,s_n\}} (t'_1, \ldots, t'_n) \tag{A.70}$$

By (4.15):

$$t_1 \xrightarrow{r_1} t''_1 \cdots t_n \xrightarrow{r_n} t''_n \qquad t''_1 \xrightarrow{s_1} t'_1 \cdots t''_n \xrightarrow{s_n} t'_n \tag{A.71}$$

By (4.3):

$$t_1 \xrightarrow{r_1;s_1} t'_1 \ \cdots \ t_n \xrightarrow{r_n;s_n} t'_n \tag{A.72}$$

By (4.15):

$$\{r_1; s_1, \ldots, r_n; s_n\} \tag{A.73}$$

Case #2:

$$(t_1, \ldots, t_n) \xrightarrow{\{r_1,\ldots,r_n\};\{s_1,\ldots,s_n\}} \bot \implies (t_1, \ldots, t_n) \xrightarrow{\{r_1;s_1,\ldots,r_n;s_n\}} \bot \tag{A.74}$$

Assume the premise:

$$(t_1, \ldots, t_n) \xrightarrow{\{r_1,\ldots,r_n\};\{s_1,\ldots,s_n\}} \bot \tag{A.75}$$

Sub-case #1 by (4.4):

$$(t_1, \ldots, t_n) \xrightarrow{\{r_1,\ldots,r_n\}} \bot \tag{A.76}$$

By (4.16):

$$t_i \xrightarrow{r_i} \bot \text{ where } 1 \leq i \leq n \tag{A.77}$$

By (4.4):

$$t_i \xrightarrow{r_i;s_i} \bot \tag{A.78}$$

By (4.16):

$$t_i \xrightarrow{\{\ldots, r_i; s_i, \ldots\}} \bot \tag{A.79}$$

Sub-case #2 by (4.5):

$$(t_1, \ldots, t_n) \xrightarrow{\{r_1, \ldots, r_n\}} (t'_1, \ldots, t'_n) \qquad (t'_1, \ldots, t'_n) \xrightarrow{\{s_1, \ldots, s_n\}} \bot \tag{A.80}$$

By (4.15):

$$t_1 \xrightarrow{r_1} t'_1 \ \cdots \ t_n \xrightarrow{r_n} t'_n \tag{A.81}$$

By (4.16):

$$t'_i \xrightarrow{s_i} \bot \text{ where } 1 \leq i \leq n \tag{A.82}$$

By (4.5):

$$t_i \xrightarrow{r_i; s_i} \bot \tag{A.83}$$

By (4.16):

$$(\ldots, t_i, \ldots) \xrightarrow{\{\ldots, r_i; s_i, \ldots\}} \bot \tag{A.84}$$

REFERENCES CITED

[1] COM: Component Object Model Technologies. http://www.microsoft.com/com/default.mspx.

[2] The DICOM standard. http://medical.nema.org/.

[3] MEX-files Guide. http://www.mathworks.com/support/tech-notes/1600/1605.html.

[4] Python v2.6.4 documentation: ctypes – A foreign function library for Python. http://docs.python.org/library/ctypes.html.

[5] Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[6] Swig documentation: Typemaps. http://www.swig.org/Doc1.3/Typemaps.html.

[7] WHIRL intermediate language specification. http://www.open64.net/documentation/.

[8] XML-RPC Specification. http://www.xmlrpc.com/spec.

[9] Standard ECMA-335: Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/Ecma-335.htm, June 2006.

[10] Jansson C library for working with JSON data. http://www.digip.org/jansson/, Apr. 2012.

[11] JSON. http://www.json.org/, Apr. 2012.

[12] Python/C API Reference Manual. http://docs.python.org/c-api/, Jan. 2012.

[13] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 11(4):491–516, Oct. 1989.

[14] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management 2004*, pages 423–424, 2004.

[15] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for web services version 1.1. http://www.ibm.com/developerworks/library/specification/ws-bpel/, May 2003.

[16] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[17] R. C. Armstrong, D. Gannon, A. Geist, K. Keahey, S. R. Kohn, L. C. McInnes, S. R. Parker, and B. A. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 13–23, 1999.

[18] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[19] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[20] D. J. Barrett, A. Kaplan, and J. C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *In ACM SIGSOFT'96, Fourth Symposium on the Foundations of Software Engineering*, pages 147–155, 1996.

[21] D. M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996*, July 1996.

[22] N. Benton and A. Kennedy. Interlanguage working without tears: blending SML with Java. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 126–137, New York, NY, USA, 1999. ACM.

[23] P. V. Biron and A. Malhotra. XML schema part 2: Datatypes second edition. W3C recommendation, World Wide Web Consortium, October 2004.

[24] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[25] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C natively. In *Electronic Notes in Theoretical Computer Science*, volume 59, 2001.

[26] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Managing the evolution of dataflows with VisTrails. In *22nd International Conference on Data Engineering Workshops (ICDEW'06)*, pages 71–75, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[27] M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. P. Jones, A. Reid, M. Wallace, and M. Weber. The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 report. `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`, 2003.

[28] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[29] G. Coulouris, J. Dolllimore, and T. Kindberg. *Distributed Systems: Concepts and Design*, chapter Sun RPC, pages 138–144. International Computer Science. Addison Wesley, 2 edition, 1994.

[30] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI. *Internet Computing, IEEE*, 6(2):86–93, March/April 2002.

[31] L. DeMichiel and M. Keith. JSR-000220 Enterprise JavaBeans 3.0 Final Release Specification. http://java.sun.com/products/ejb/docs.html, May 2006.

[32] T. Fahringer, S. Pllana, and A. Villazon. A-GWL: Abstract Grid Workflow Language. In *4th International Conference on Computational Science (ICCS 2004)*, Lecture Notes in Computer Science, pages 42–49. Springer Berlin / Heidelberg, June 2004.

[33] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with AGWL: an abstract grid workflow language. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 676–685, Washington, DC, USA, 2005. IEEE Computer Society.

[34] K. Fisher, R. Pucella, and J. Reppy. A framework for interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1):3–19, September 2001.

[35] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, June 2002.

[36] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, Aug. 2001.

[37] T. Goodale. *Workflows for e-Science*, chapter Expressing Workflow in the Cactus Framework, pages 416–427. Springer London, 2007.

[38] T. Goodale, G. Allen, G. Lanfermann, J. Masso, E. Seidel, and J. Shalf. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing - VECPAR '2002, 5th International Conference*. Springer, 2003.

[39] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Prentice Hall PTR, second edition, June 2000.

[40] M. Grechanik, D. Batory, and D. E. Perry. Design of large-scale polylingual systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 357–366, Washington, DC, USA, 2004. IEEE Computer Society.

[41] J. Hamilton. Language integration in the Common Language Runtime. *ACM SIGPLAN Notices*, 38(2):19–28, 2003.

[42] P. B. Hansen. *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[43] R. Hayes and R. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering*, 13:1254–1264, 1987.

[44] M. Hirzel and R. Grimm. Jeannie: granting Java native interface developers their wishes. In *Proceedings of the 2007 OOPSLA conference*, volume 42, pages 19–38, New York, NY, USA, 2007. ACM.

[45] G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[46] J. Hopkins. Component primer. *Communications of the ACM*, 43(10):27–30, October 2000.

[47] P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[48] L. Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical report, AT&T Bell Laboratories, January 1996.

[49] G. C. Hulette, M. J. Sottile, R. Armstrong, and B. Allan. OnRamp: enabling a new component-based development paradigm. In *Proceedings of the 2009 Workshop on Component-Based High Performance Computing*, CBHPC '09, pages 1–10, New York, NY, USA, 2009. ACM.

[50] G. C. Hulette, M. J. Sottile, and A. D. Malony. WOOL: A workflow programming language. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 71–78, Washington, DC, USA, 2008. IEEE Computer Society.

[51] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng. Overview of the Ptolemy project. Technical report, University of California Berkeley, 2003.

[52] B. Janssen and M. Spreitzer. ILU: Inter-language unification via object modules. In *Workshop on Multi-language Object Models*, Portland, OR, August 1994.

[53] S. P. Jones, T. Nordin, and A. Reid. Green card: a foreign language interface for Haskell. In *ACM SIGPLAN Haskell Workshop (in conjuction with ICFP97)*, February 1997.

[54] A. Kaplan, J. Ridgway, and J. Wileden. Why IDLs are not ideal. *Software Specification and Design, International Workshop on*, 0:2, 1998.

[55] G. K. Kloss. Automatic C library wrapping – Ctypes from the trenches. *The Python Papers*, 3(3), 2008.

[56] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processes*, Portsmouth, VA, March 2001.

[57] S. Krishnan, P. Wagstrom, and G. von Laszewski. GSFL: A workflow framework for grid services, 2002.

[58] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[59] F. Leymann. Web Services Flow Language (WSFL 1.0). Technical report, IBM, May 2001.

[60] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, June 2002.

[61] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: an optimizing, portable OpenMP compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2006.

[62] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Prentice Hall, 2nd edition, April 1999.

[63] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, F. Juelich, R. Rivenburgh, C. Rasmussen, and B. Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.

[64] P. L. M. Clavel, S. Eker and J. Meseguer. Principles of maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[65] S. Macrakis. From Uncol to ANDF: Progress in standard intermediate languages. White paper, Open Software Foundation, 1993.

[66] S. Majithia, M. Shields, I. Taylor, and I. Wang. Triana: A graphical web service composition and execution toolkit. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, volume 0, pages 514–522, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[67] M. Murphy. NVIDIA's experience with Open64. *Open64 Workshop at CGO '08*, 2008.

[68] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, Mar. 2008.

[69] O. Nierstrasz and T. D. Meijler. Research directions in software composition. *ACM Computing Surveys (CSUR)*, 27(2):262–264, 1995.

[70] D. Notkin, A. P. Black, E. D. Lazowska, H. M. Levy, J. Sanislo, and J. Zahorjan. Interconnecting heterogeneous computer systems. *Commun. ACM*, 31(3):258–273, 1988.

[71] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

[72] D. Panda, R. Rahman, and D. Lane. *EJB 3 in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[73] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimization technique in GHC. In *Proceedings of the 2001 Haskell Workshop*, pages 203–233, Sept. 2001.

[74] C. E. Rasmussen, K. A. Lindlan, B. Mohr, J. Striegnitz, and F. Jlich. CHASM: Static Analysis and Automatic Code Generation for Improved Fortran 90 and C++ Interoperability. In *In Proceedings of the Los Alamos Computer Science Symposium 2001 (LACSI'01*, 2001.

[75] J. Reid. The new features of Fortran 2000. *SIGPLAN Fortran Forum*, 21(2):1–31, 2002.

[76] J. Reid. The new features of Fortran 2003. *SIGPLAN Fortran Forum*, 26(1):10–33, 2007.

[77] J. Reppy and C. Song. Application-specific foreign-interface generation. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 49–58, New York, NY, USA, 2006. ACM.

[78] J. Reppy and C. Song. Application-specific foreign-interface generation. In *Proceedings of the Fifth International Conference on Generative Programming and Component Engineering*, pages 49–58, Oct. 2006.

[79] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, Palo Alto, CA, April 2007.

[80] B. A. Smolinski, S. Kohn, N. Elliott, and N. Dykman. *Computing in Object-Oriented Parallel Environments*, volume 1732, chapter Language Interoperability for High-Performance Parallel Scientific Components, pages 61–71. Springer Berlin / Heidelberg, 1999.

[81] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Commun. ACM*, 25(4):246–260, 1982.

[82] R. Srinivasan. XDR: External Data Representation Standard. RFC 1832 (Draft Standard), 1995. Obsoleted by RFC 4506.

[83] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 24(3):68–79, 1990.

[84] S. Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, February 1997.

[85] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[86] E. Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57:109 – 143, 2001. WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming.

[87] E. Visser and Z. el Abidine Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422 – 441, 1998. International Workshop on Rewriting Logic and its Applications.

[88] E. Visser and Z. el Abidine Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422–441, Jan 1998.

[89] J. Waldo. Remote procedure calls and Java remote method invocation. *Concurrency, IEEE*, 6(3):5–7, Jul-Sep 1998.

[90] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[91] J. Yu and R. Buyya. A novel architecture for realizing grid workflow using tuple spaces. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 119–128, Washington, DC, USA, 2004. IEEE Computer Society.

[92] A. M. Zaremski and J. M. Wing. Signature matching: a key to reuse. In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, pages 182–190, New York, NY, USA, 1993. ACM.