Composing Typemaps in Twig

Geoffrey C. Hulette
University of Oregon
Eugene, OR
ghulette@cs.uoregon.edu

Matthew Sottile
Galois, Inc.
Portland, OR
mjsottile@computer.org

Allen D. Malony
University of Oregon
Eugene, OR
malony@cs.uoregon.edu

ABSTRACT

Twig is a language for writing *typemaps*, programs which transform the type of a value while preserving its underlying meaning. Typemaps are typically used by tools that generate code, such as multi-language wrapper generators, to automatically convert types as needed. Twig builds on existing typemap tools in a few key ways. Twig's typemaps are composable so that complex transformations may be built from simpler ones. In addition, Twig incorporates an abstract, formal model of code generation, allowing it to output code for different target languages. We describe Twig's formal semantics and show how the language allows us to concisely express typemaps. Then, we demonstrate Twig's utility by building an example typemap.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.12 [**Software Engineering**]: Interoperability

General Terms

Languages

Keywords

Type mapping, Foreign function interface

1. INTRODUCTION

Twig is a language for writing *typemaps* – programs that transform data from one type to another while preserving, as much as possible, the underlying meaning of the data. Typemaps have proven useful in many kinds of programming and especially automated code generation. The best-known application of typemaps has been for multi-language programming. In this domain, for example, a programmer may wish to pass an integer from a Python program across a foreign function interface to a C function, where a C `int` is expected. A typemap can be used to describe the generic transformation from Python integers to C integers, enabling a tool such as SWIG to generate the conversion code automatically. The C function is exposed to Python via the generated wrapper.

There are a number of existing languages for typemaps and tools which generate code from them. Twig builds on existing typemap tools in several ways.

First, Twig's typemaps are composable, i.e., complex typemap transformations may be constructed by combining simpler ones. Our typemap semantics are based on those found in Fig[9] and System S[10], but we extend and refine those systems.

Second, Twig incorporates a robust, formal model of code generation in its semantics. This allows Twig to generate code based on typemaps for different target languages.

Finally, Twig includes a facility for *reducing* typemaps by exploiting identity relationships among typemap expressions. Some reductions are based on a formal algebra of typemaps, while others are domain-specific and provided by the user. In forthcoming work we show how user-supplied typemap reductions can be used to optimize certain transformations. We will not cover reductions further in this paper.

In this paper, we will describe Twig's formal language structure, and then show how this structure allows us to concisely express complex typemaps. First, we review existing approaches to typemaps and related problems. Second, we will present Twig's semantics. Third, we walk through a typemap example in SWIG, and show how the typemaps can be expressed more concisely in Twig. We conclude with ideas for future work.

2. RELATED WORK

There are many tools which incorporate a notion of typemaps. The idea originated in SWIG [3], a tool used for generating foreign function interfaces from C header files. Typemaps in Swig are robust, and support user customization. However, the semantics of Swig's typemaps are ad-hoc, inflexible, and specialized to generate C code.

FIG [9] introduced the notion of application-specific typemaps, and is similar in spirit to our own work. Unlike Twig, FIG is specialized to generate code for Moby [7]. Moby is a convenient target language – its declarative structure and semantics are amenable to generation via System S [5]. Indeed, FIG takes advantage of this fact by providing rules specific to Moby. Moby is not nearly as ubiquitous as C, however, and therefore not a very practical target language for many people.

GPCE'12, September 26–27, 2012, Dresden, Germany.

There are other tools that utilize typemaps, particularly foreign-function interface generators such as Charon [6] or NLFFIGen [4]. Twig might complement these systems well, providing a foundational semantics for their typemaps along with the ability to generate code for a variety of target languages.

Our abstract code generation model is based in part on our own previous work on the Wool [8] language for workflow programming.

3. TWIG

Twig is based on *System S* [10], originally designed as a core language for term rewriting systems [2]. We 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 [9], but extended to incorporate our code generation model.

3.1 Values

Values in Twig are tree structured data with labeled internal nodes, called *ground terms*. We define ground terms t:

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

where c is a *constant*, and f is a *constructor* that builds terms from other terms. Constants and constructors can be any string of characters (except the special constructor `tuple`, see below). The meaning of particular terms is left abstract – they are defined by their use in the program's *rules*, described below. We denote the set of all terms T.

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

3.1.1 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 small complication arises because 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 occasionally 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.

3.2 Expressions

Twig *expressions* can be either *primitive rules* (Section 3.3) or else built from other expressions using operators (Section 3.4). An expression s maps 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 is a *block* of generated code in the set M (see Section 4), or else the special, distinguished value $\bot$. Formally, s is a function:

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

Following Fig's notation, we use $\bot$ to denote "failure." In particular, $\bot$ is used in the semantics for the operators described in Section 3.4.

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

$$v{=}e$$

where v is a valid expression name and e is an expression, as described below. A Twig program is a list of such name/expression assignments. There is a special expression name, `main`, which designates the top-level expression for the program.

3.3 Primitive Rules

The simplest Twig expressions are *primitive rules*, which describe a single transformation step. 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

$$[p_1 - > p_2] <<< m >>>$$

where p_1 is the *input pattern*, p_2 the *output pattern*, and m is a block of code (code blocks are explained in Section 4).

Input and output patterns are terms that can also contain *variables*.

Informally, the primitive rule above transforms t to t' with an associated block of code m if and only if

1. t successfully *matches* the input pattern p_1, binding terms to variables in an *environment* ϵ and
2. t' is successfully *built*, by substituting the bound values in ϵ into the variables of p_2;

and otherwise transforms it to $\perp$.

When a term is *matched*, we mean that Twig attempts to match it with the input pattern [2]. Twig's matching algorithm is similar to, but simpler than, term unification. Twig does not need full unification since there is no equational theory and the input term may not contain variables (i.e., must be a ground term). If the match is successful, variables are bound to their corresponding terms in an environment. The environment is then used to construct the output term by substitution over the output pattern. See [2, 10] for a formal discussion of the algorithm.

Consider the following example of a 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.

3.4 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 (see Figure 1). 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 4.1.1).

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 (see Figure 2). If it fails, it attempts to apply the second rule instead. This operator allows different code to be generated depending on the input type.

Figure 3 and Figure 4 give the semantics for Twig's other basic operators. Identity (`T`) will always succeed, returning its input and an identity block, failure (`F`) will always return $\perp$. Test (`?`) takes a single expression as a parameter and succeeds only if its argument succeeds, returning the original term. Negation ($\neg$) also takes a single expression argument, and succeeds only if its argument fails, returning the original term.

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

$$\frac{t \xrightarrow{s_1} \perp}{t \xrightarrow{s_1;s_2} \perp} \qquad \frac{t \xrightarrow{s_1} (t', m) \quad t' \xrightarrow{s_2} \perp}{t \xrightarrow{s_1;s_2} \perp}$$

Figure 1: Semantics for sequence operator

$$\frac{t \xrightarrow{s_1} (t', m_1)}{t \xrightarrow{s_1|s_2} (t', m_1)} \qquad \frac{t \xrightarrow{s_1} \perp \quad t \xrightarrow{s_2} (t', m_2)}{t \xrightarrow{s_1|s_2} (t', m_2)}$$

$$\frac{t \xrightarrow{s_1} \perp \quad t \xrightarrow{s_2} \perp}{t \xrightarrow{s_1|s_2} \perp}$$

Figure 2: Semantics for left-biased choice

$$\frac{}{t \xrightarrow{\mathtt{T}} (t, I)} \qquad \frac{}{t \xrightarrow{\mathtt{F}} \perp}$$

Figure 3: Semantics for Success (`T`) and Failure (`F`)

Twig also provides some operators especially for tuples. For each of the following tuple operators there are a few additional rules that we have elided. Informally, these rules state that any tuple operator will fail (i.e., return $\perp$) if the input is not a tuple, or if the expression references an element that is outside the tuple bounds.

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 4.1.2) of the individual result blocks. The semantics for congruence are shown in Figure 5.

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 given in Figure 6.

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 given in Figure 7.

The branch operator `#some` applies its parameter s to at least one element of a tuple, and fails in case s fails for all the elements. It will apply s to any element for which it succeeds, and leave the remaining elements unchanged.

$$\frac{t \xrightarrow{s} (t', m)}{t \xrightarrow{?s} (t, I)} \qquad \frac{t \xrightarrow{s} \bot}{t \xrightarrow{?s} \bot}$$

$$\frac{t \xrightarrow{s} (t', m)}{t \xrightarrow{\neg s} \bot} \qquad \frac{t \xrightarrow{s} \bot}{t \xrightarrow{\neg s} (t, I)}$$

Figure 4: Semantics for Test (?) and Negation (¬)

The *projection* operator extracts a single indexed element from a tuple (see Figure 9). Similarly, the *path* operator applies a rule to a single indexed tuple element, leaving the other elements unchanged (see Figure 10).

Finally, the *permutation* operator allows arbitrary permutation of a tuple's elements, including duplicating or dropping elements. The semantics are given in Figure 11. We treat permute_1, where the permutation expects a single input, specially – in this case, we treat non- tuple input terms as tuples of length one. This allows the permutation operator to be used to replicate single terms. The semantics for this special case are given in Figure 12.

As a convenience, we also provide a "fan out" operator which is defined as a permute_1 followed by a congruence. The definition is given in Figure 13.

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 $\#\texttt{fix}_x(s)$, that is, x appearing within s, is essentially a recursive call to the expression $\#\texttt{fix}_x(s)$.

4. 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.

We use the code generation model in describing Twig's semantics (see Section 3). 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 4.1, we describe the code generation model abstractly, and apart from any specific target language. Then, in Section 4.2, we show how the model can be specialized to generate C code.

4.1 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 to the number of its inputs and outputs, respectively.

4.1.1 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.

4.1.2 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," i.e., each block executes independently of the other, 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

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

4.1.3 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. Permutations may also "drop" an element by not wiring its input to any output, and "duplicate" elements by wiring an input to more than one output. The exact meaning of dropping or duplicating values depends on the implementation.

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. In addition, for any natural number n, there exists an identity transformation taking 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, $\text{in}(I_n) = \text{out}(I_n) = n$.

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

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

Figure 5: Semantics for congruence operator

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

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

Figure 6: Semantics for branch `#one` operator

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

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

Another important special block is the *fanout* block, which takes a single input and "copies" it to n outputs. We denote this block F_n. This turns out to be another special case of the permutation block, and we can define

$$F_n = \Pi_1(1, \ldots, n)$$

to be the fanout block with n outputs.

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 nonsensical implementations. The code generation implementation should conform to the intuitive interpretation of blocks and their composition.

4.2 Generating C

Now we show how 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 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 15 shows a visual representation of two primitive blocks of C code.

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. See Figure 16 for an example.

Parallel composition for C is implemented similarly; Twig generates uniquely-named variables for the inputs and outputs of the two blocks, and then concatenates the text. An example is shown in Figure 17.

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.

5. IMPLEMENTATION

We wrote our implementation of Twig, called `twigc`, in Haskell. The `twigc` tool expects two input files. The first file must contain a list of named expressions, including a `main` expression, as described in Section 3.2. The second file contains a ground term (representing a C type), used as input to the main expression.

Our version of `twigc` must be configured with a mapping from terms to C types. The user provides this mapping with a simple key/value text file.

If the input value can be successfully rewritten using the main rule expression provided, then Twig will output the

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

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

Figure 7: Semantics for branch `#all` operator

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

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

Figure 8: Semantics for branch `#some` operator

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

Figure 9: Semantics for projection operator

rewritten term along with the generated block of C code. This code block may be redirected to a separate file and included in a C program using the `#include` directive.

5.1 Code Generation

Our implementation supports the generation of C code, and adds some extra features to support that language. These features includes mundane details such as managing type declarations, support for parameterized blocks, and for "closing" blocks, which are generated as variables go out of scope and are intended to be used to free resources. We are working on incorporating these features into the abstract model.

6. EVALUATION

Twig has several advantages over typemap facilities such as those found in SWIG.

1. Sequencing: simple typemaps can be composed in sequence to produce more complex transformations.

2. Choice: With the choice combinator, a single typemap expression may be used to generate multiple variations of a transformation, depending on the input type.

3. Tuples: 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.

4. Type variables: Twig allows for rules that abstract over types, e.g., `[ptr(A) -> A]`.

5. Target language flexibility: Twig can be extended to generate target languages other than C.

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;
>>>
```

$$\frac{t_i \xrightarrow{s} (t_i', m_i)}{\texttt{tuple}(\ldots, t_i, \ldots) \xrightarrow{\#i(s)} (\texttt{tuple}(\ldots, t_i', \ldots), I \times \ldots \times m_i \times \ldots \times I)}$$

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

Figure 10: Semantics for path operator

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

$$\begin{aligned} w &= \sum_{j=1}^{n} \text{width}(t_i) \\ b_i &= \begin{cases} 0 & \text{if } i = 1 \\ \sum_{j=1}^{i-1} \text{width}(t_i) & \text{if } i > 1 \end{cases} \\ y_i &= b_i + 1, \ldots, b_i + \text{width}(t_i) \end{aligned}$$

Figure 11: Semantics for permutation operator

$$\frac{}{t \xrightarrow{\#\texttt{permute}_1(1,\overset{n}{\ldots},1)} (\texttt{tuple}(t, \overset{n}{\ldots}, t), \Pi_1(y, \overset{n}{\ldots}, y))}$$

$$y = 1, \ldots, \text{width}(t)$$

Figure 12: Semantics for $\#\texttt{permute}_1$ operator

$$\#\texttt{fan}(n) \equiv \#\texttt{permute}_1(\underbrace{1, \ldots, 1}_{\text{n}})$$

Figure 13: Semantics for `#fan` operator

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

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

Figure 14: Semantics for `#fix` operator

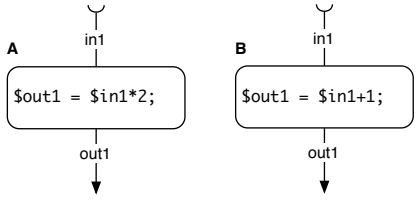

Figure 15: Two basic blocks, A and B. Inputs are on top, outputs on the bottom.

Here, a `polard` is a term representing the `PolarD` struct defined above. 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}
```

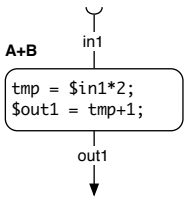

Figure 16: Two blocks from Figure 15 composed sequentially. The variable "tmp" is created, and renaming performed, so that the output of block A would flow to the input of block B.

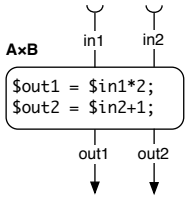

Figure 17: Two blocks from Figure 15 composed in parallel. Renaming is performed such that the composed block has two inputs and two outputs.

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

Next, we define rules to convert from C types to Python. We use Python's C interface API [1], 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`, and define it like so:

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

We can invoke Twig with the `convert` 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;
  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;
}
```

6.1 Twig versus SWIG

It is interesting to contrast Twig's implementation of this typemap 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 `doubles` 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.

[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.

6.2 Twig versus Fig

Twig also improves on Fig in a number of ways. Fig is intimately tied to its target language, Moby, whereas Twig can be extended to generate a variety of mainstream languages. Notably, Twig is able to generate C, which is currently the *de facto* language for interoperability. In addition, Twig supports type variables, allowing Twig programs to express polymorphic primitive rules.

7. FUTURE WORK

Twig is limited in its handling of *container* types, such as arrays, lists, or trees. In order to properly handle this kind of data, Twig needs to be extended with higher-order primitive rules, i.e., rules which take expressions as parameters. We are currently working on a syntax and formal semantics for rules such as:

```
[array(X) -> array(Y) | X -> Y] <<< ... >>>
```

This should be read as a rule which transforms an array of any type X to an array of another type Y, given a rule to transform a single element of type X to type Y.

Other improvements to Twig may include extending `twigc` to support other target languages, such as Java or Python. This would take advantage of our abstract code generation model.

8. CONCLUSION

We have presented Twig, a language for typemaps that may serve as a more flexible alternative to the languages found in tools such as SWIG. We have demonstrated how Twig typemaps are created and composed, and shown how our language incorporates useful features such as tuples, polymorphic rules with variables, and runtime choice based on the input type.

Twig includes a flexible model for code generation. While our current implementation is focused on generating C, other languages, such as Java or Python, could be generated instead.

9. ACKNOWLEDGEMENTS

This work was supported in part by the Department of Energy Office of Science, Advanced Scientific Computing Research.

10. REFERENCES

[1] Python/C API Reference Manual. `http://docs.python.org/c-api/`, Jan. 2012.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[3] D. M. Beazley. Automated scientific software scripting with SWIG. *Future Gener. Comput. Syst.*, 19:599–609, July 2003.

[4] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C natively. *Electronic Notes in Theoretical Computer Science*, 59(1):36–52, 2001.

[5] K. Fisher, R. Pucella, and J. Reppy. Data-level interoperability. In *Electronic Notes in Theoretical Computer Science*, 2001.

[6] K. Fisher, R. Pucella, and J. Reppy. A framework for interoperability. In *Proceedings of the First International Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01)*, volume 59 of *Electronic Notes in Theoretical Computer Science*, Sept. 2001.

[7] K. Fisher and J. Reppy. The design of a class mechanism for Moby. In *Proceedings of the SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 37–49, New York, NY, May 1999. ACM.

[8] G. Hulette, M. Sottile, and A. Malony. Wool: A workflow programming language. *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, Dec 2008.

[9] 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.

[10] E. Visser and Z. el Abidine Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422–441, Jan 1998.