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's approach to typemapping, language design, and comparison to Swig.

1. INTRODUCTION

Twig is a new language for writing *typemaps* – programs that transform data from one type to another, while preserving (as much as possible) the underlying value of the data. Typemaps have proven useful in many kinds of programming and especially automated code generation, where we require a transformation to pass a single nominal value across a pair of mismatched types that we know to be interchangeable in some way. The best-known example of this problem is found in multi-language programming. For example, a programmer may wish to pass a Python integer to a C function, where a C int is expected. If we have a typemap that performs the transformation from Python integers to C integers, then an automated tool can generate the conversion code automatically, and expose the C function in Python via a generated wrapper.

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

First, Twig's typemaps are composable, i.e., new typemaps may be constructed by combining old ones. Thus, complex typemap transformations may be built from simpler ones. Our notion of typemap composition is based on the formalisms used in Fig[9] and System S[10], but we extend and refine that work in some key ways.

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

Finally, Twig includes a facility for *reducing* typemaps by exploiting identity relationships among typemap expressions. Some reductions are based on a universally-applicable algebra of typemaps, while others are domain-specific and must be described by the user. We have shown in prior work that typemap reduction can be used to optimize certain transformations. Reductions are covered in our previous work, and we will not address them further here.

In this paper, we will describe Twig's formal language structure, and then show how this structure allows us to express complex typemaps more concisely than with traditional tools. First, we review existing approaches to typemaps and related problems. Second, we present the semantics for Twig's code generation model, and then the semantics for the typemap language itself. Third, we present a typemap example in Swig, and show how the same problem can be solved more concisely and clearly in Twig. Finally, we conclude with ideas for future work.

2. RELATED WORK

There are many tools which incorporate some notion of typemaps which are in widespread use today. The idea originated in Swig [3], a system for generating foreign function interfaces from C header files and some user-specified directives. Typemaps in Swig are robust, and do support user customization. However, the semantics of Swig's typemaps are ad-hoc and not especially flexible. Also, they are specialized to generate C code.

FIG [9] introduced the notion of application-specific typemaps, and is quite similar to our own work in both its spirit and the actual semantics of its typemaps. Unlike Twig, however, FIG generates code for Moby [7]. Moby is, in some ways, 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 Moby-specific rules within FIG. Moby is not nearly as ubiquitous as C, however, and therefore not a very practical target language for many people.

There are many other tools that provide some kind of typemap language or facility in the service of their intended function, particularly foreign-function interface generators such as Charon [6], NLFFIGen [4], etc. We feel that Twig could complement many of these sytems quite well, providing a foundational semantics for their typemap languages, while providing the abilty to generate C code.

Our abstract code generation model was based in part on our own previous work on a language called Wool [8].

3. CODE GENERATION

One of our goals for Twig was that it be able to generate code in many different target languages. To that end, Twig's semantics rely on an abstract, language-independent model for code generation with a small number of basic operations. To incorporate a new target language, it suffices to implement only these operations. There is no need to modify the core Twig interpreter, which assumes nothing beyond the abstract model.

Our simplified code generation model is used in formulating Twig's core semantics, described in Section 4. It is also helpful in clarifying the precise operations which Twig supports, without getting bogged down in the (potentially rather complicated) details of outputting code for a particular target programming language.

In Section 3.1, we describe the abstract code generation model, apart from any specific target programming language, including the formal semantics. Then, in Section 3.2 we show how we have used this model in order to generate C code.

3.1 Abstract Code Generation

In Twig, a unit of code to be generated is called a *block*. Intuitively, a block represents some code in the target language, which accepts inputs and produces outputs. We call the set of blocks M, and we say that for all $x \in M$ we have the functions

$$\texttt{in} : M \to \mathbb{Z}+$$
$$\texttt{out} : M \to \mathbb{Z}+$$

That is, `in` and `out` will map a block to the number of its inputs and outputs, respectively. Note that either or both may be zero for a given x.

3.1.1 Sequential Composition

The first of Twig's elementary block operations is called *sequential composition*, which we represent as addition ($+$). Intuitively, sequencing represents connecting two blocks "vertically," by feeding the outputs of the first block to the inputs of the second. Formally, we define the subset $x + y$ by

$$\frac{x \in M \quad y \in M \quad \texttt{out}(x) = \texttt{in}(y)}{x + y \in M}$$

$$\texttt{in}(x + y) \equiv \texttt{in}(x)$$
$$\texttt{out}(x + y) \equiv \texttt{out}(y)$$

Note that the condition that the number of outputs on the left side be equal to the number of inputs on the right implies that M is not closed under sequential composition. The result of sequencing blocks where the inputs and outputs are mismatched is undefined. We take care in the design of the rest of Twig's semantics to avoid this case.

3.1.2 Parallel Composition

The second elementary operator is called *parallel composition*. We represent this operation with the multiplication operator ($\times$). Intuitively, parallel composition fuses two blocks "horizontally," where each block executes independently of one another, but they appear as a single block with combined inputs and outputs. Formally, we define $\times$ as

$$\frac{x \in M \quad y \in M}{x \times y \in M}$$

and we define

$$\texttt{in}(x \times y) \equiv \texttt{in}(x) + \texttt{in}(y)$$
$$\texttt{out}(x \times y) \equiv \texttt{out}(x) + \texttt{out}(y)$$

Note that M is closed under parallel composition.

3.1.3 Special Blocks

In the abstract model, we single out a set of special elements in M called *permutation* elements. These elements are intended to represent the primitive operation of taking m inputs and rearranging them into n outputs, possibly in a different order, and possibly duplicating or dropping elements. The intended interpretation is that the actual values are unchanged by the operation. Formally, we name the block representing the permutation of m inputs to n outputs $\Pi_m(i_1, \ldots, i_n)$, where $i_1, \ldots, i_n \in \{i | 1 \leq i \leq m\}$.

An interesting subset of the permutation elements are *identity* permutations. 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 element as I_1. In fact, there are an unlimited number of identity transformations, which take n inputs to n outputs, unchanged and without reordering. These are referred to as I_n, where $1 \leq n$, and $I_n = \Pi_n(1, 2, \ldots, n)$. Clearly, we can infer that

$$\texttt{in}(I_n) = \texttt{out}(I_n) = n$$

Since the elements I_n are intended to represent identity operations, we assign them a special meaning in the composition semantics. Namely, we define

$$\frac{\texttt{out}(x) = n}{x + I_n \to x} \qquad \frac{\texttt{in}(x) = n}{I_n + x \to x}$$

That is, I_n acts as both a left- and right-identity when it is combined in sequence with a block x (with the appropriate number of inputs or outputs). We sometimes use I_n as a kind of "no-op."

It is worth noting one further identity, namely that

Note that this implementation does not allocate or free memory, or otherwise perform resource management as part of the permutation operations. This implies that the generated code will follow C's semantics for passing by value versus reference. The user must keep these rules in mind when designing and using C typemaps in Twig.

4. TWIG

Twig is based on a core semantics called *System S* [10], originally designed as a core language for term rewriting systems [2]. In Twig, we use the operators of System S to combine primitive *rules* into more complex transformations on types. These transformations are then applied to a given type, which performs the transformation. We extend the semantics of System S in order to have the evaluation of the transformation generate code as a side effect. In this way, domain specific code can be generated depending on the input types. Our semantics are inspired by and are quite similar to those in Fig [9], but we have extended the rules to accomodate our code generation model. In this section we describe the semantics of Twig's language.

4.1 Values

Values in Twig can be any valid *term*. Terms are tree structured data with labeled internal nodes. Examples of terms include simple values like `int` and `float`, as well as compound types like `ptr(int)`, which represents a pointer to an integer. More complicated terms may involve multiple child terms, and may be nested to any depth. For example, `struct(int,float,struct(ptr(char)))` may 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 in Twig and types in the target language is considered to be a configuration option, controllable by the user and customizable to the domain. Furthermore, the mapping need not be injective, i.e., multiple values in Twig may map to the same type in C. For example, you might use distinct Twig values `string` and `ptr(char)`, but map both to a `char` pointer in C.

Twig has just one special kind of term: tuples. Tuples may have any length, and the elements of the tuple are represented as the sub-terms of a term with a special constructor: `tuple`. Twig's concrete syntax also equates the absense 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 `tuple` constructor syntax is more verbose, so we typically will eschew it in practice; it is convenient, however, for our presentation of the formal semantics below.

4.1.1 *Tuple "size" versus "width"*

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

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

is a valid tuple with a nesting depth of two. In some of our semantics, we need to provide a *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 the size of its "flattened" version – 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

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

and so its width would be three.

4.2 Expressions

Programs in Twig take the form of *rule expressions*. A rule expression s acts upon a term t in some given set of valid terms T. The result of applying s to t is always either "failure," which we denote $\perp$, or else a pair (t', m), where t' is another term and m is a *block* in M, as described in Section 3. Formally, we say that s is a function

$$s : T \to (T \times M) \cup \perp$$

In the rest of this section we explore the ways in which we can construct rule expressions.

4.3 Primitive Rules

The simplest Twig expression is called a *primitive rule*. Primitive rules describe a single transformation step. Since Twig's terms represent types, a primitive rule in Twig describes how to transform an instance of one type into an instance of another.

For example, in C it is easy to convert an integer value to floating point, and we can write this rule in Twig as follows:

```
[int -> float]
```

The term to the left of the arrow is the *input pattern*, and the term to the right is the *output pattern*. In this example, the rule says that if the input to the expression *matches* `int`, then the output of the expression will be the value `float` (along with a code block, discussed below). If the input term does not match `int` then the output will be $\perp$.

Rules can also have *variables* in place of terms or sub-terms. For example the rule

```
[ptr(X) -> X]
```

transforms any 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 then substituted for the variable where it appears on the left. Variables may stand in place of terms only, not constructors; e.g., rules such as `[X(int) -> X]` are invalid.

We must now make clear what we mean when we say that the input term *matches* the input pattern. In effect, the input term is *unified*[2] with the input pattern. In fact, the algorithm used 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 the unification is successful, we say that the term *matched*. The bound variables (if any) are saved in order to construct the output term via substitution.

4.3.1 Blocks

Primitive rules are associated with have a block. The block is constructed based on some target language-dependent text, which appears immediately after the rule definition and is surrounded by triple-angle brances, like so:

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

Block construction depends on the target language; in this case, we have provided a block of escaped C code (as described in Section 3.2). The target language must be specified as a parameter to the Twig runtime (in our current implementation, as a command line option). This means that the text between the <<< and >>> is interpreted based on the chosen block implementation. A target langauge specific procedure is passed the text, along with the input and output term(s). It can then construct the block however it sees fit. In our C block implementation, for example, variable names are generated and the terms are used to determine the appropriate types for declaring those variables in a prelude section of the code. Other target languages might choose very different block construction mechanisms.

It is important to understand that Twig does not check the block code in any way (although the block implementation could), and that the code generation procedure is strictly syntactic. This scheme is similar to that used by SWIG [3].

4.4 Operators

Rules expressions can be combined using Twig's operators. In the formal semantics, we let t range over terms, m range over blocks and s_i range over rule expressions (i.e., a primitive rule, or another expression built with operators).

A primitive rule s transforms t to t' with an associated block m:

$$t \xrightarrow{s} (t', m)$$

if the application of rule s to value t succeeds. If no code is given for the rule, then m is the "no-op" element, e. If the application of s to t fails, e.g., if t does not match the pattern in s, then we say

$$t \xrightarrow{s} \bot$$

Note that no block is emitted in this case.

The first and arguably most useful of Twig's combinators is the *sequence* operator, which chains the application of two rules together, sending the output of the first to the input of the second, and failing if either rule fails (see Figure 4). With this operator, individual rules can be composed into multi-step transformations. Note how, upon success, the blocks are themselves composed in sequence.

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

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

Figure 4: Semantics for sequence operator

Another important operator is *left-biased choice*. This operator will attempt to apply the first rule expression to the input, and if it succeeds then its output is the result (see Figure 5). If it fails (i.e., results in $\bot$), then it attempts to apply the second rule instead. This operator allows different paths to be taken through the rules, and different code to be generated, depending on the input type being passed.

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

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

Figure 5: Semantics for left-biased choice

Figure 6 gives the formal semantics for Twig's other operators. Identity (`T`) will always succeed, returning its input and an identity block, failure (`F`) will always return $\bot$. Test (`?`) succeeds only if its single parameter expression succeeds, and returns the original term. Negation $\neg$ succeeds only if its parameter expression fails, returning the original term.

Twig also provides some special operators for tuples. The semantics in Figure 7 apply to all of the following tuple operators, unless otherwise indicated. These semantics state that a tuple operator will fail if the input term is not a tuple, or if the rule tries to reference a tuple element that is out of bounds.

One important tuple operator is *congruence*, which applies a tuple of expressions to the elements of a tuple term, pairwise, and returns a tuple of results, or else failure in case any rule application fails. Upon success, the returned block is the parallel composition of the individual result blocks. The semantics for congruence are shown in Figure 8.

explain notation.

$$I_n \equiv (I_1)^n$$

That is, I_n is equivalent to the n-way parallel composition of I_1.

When n is implied from the context, we will sometimes write I for I_n. For example, we use $x + I$ as a shorthand to refer to $x + I_n$ where n is understood to be $\texttt{out}(x)$.

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 just another special case of the permutation block, so we can define

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

for the fanout block with n outputs.

is it what the user write in the rule? it is not generated

It is worth noting 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. To be used as we intended, the code generation implementation should conform to the intuitive interpretation of blocks and their composition.

3.2 Generating C

Now we will show how the model described above can be adapted to generate C code.

You will notice that in the abstract model, we provide no way to construct "primitive" blocks, i.e., blocks that contain actual code. The only primitive blocks defined are the special permutation blocks, which by definition do nothing to the values. So, the first thing we need is a way to construct a primitive block from an arbitrary chunk of C code.

A primitive C block is just a string of C code with some annotations to indicate the inputs and outputs. In fact, our implementation makes no attempt to parse the C language *per se* – it simply accepts the C code as plain text.

is the user responsability to write correct!

We represent the inputs and outputs of each block as specially generated variables in C. To make use of the inputs and outputs, a block uses the escaped values `$in1`, `$in2`, etc. to represent the first, second, and so on values. Analogously, `$out1`, `$out2`, etc. represent the outputs. For the common case where a block has just one input and/or output, we allow `$in` as a synonym for `$in1`, and `$out` for `$out1`. When the code is rendered, these placeholders will be replaced with generated unique variable names. For example, the text

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

represents a primitive C block with one input and one output. Figure 1 shows a visual representation of two primitive blocks of C code.

To accomplish 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 2 for an example.

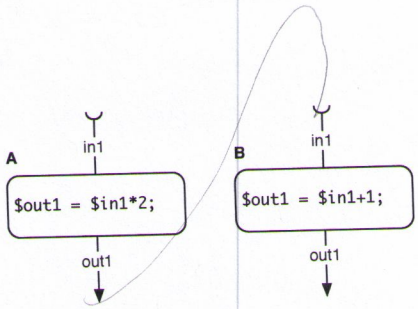

Figure 1: Two basic blocks, A and B. Inputs are on top, outputs on the bottom.

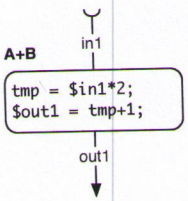

Figure 2: Two blocks from Figure 1 composed sequentially. The variable "tmp" is created, and renaming performed, so that the output of block A would flow to the input of block B.

Parallel composition for C is accomplished similarly; Twig generates independently-named variables for the inputs and outputs of the two blocks, and then concatenates the text. An example is shown in Figure 3.

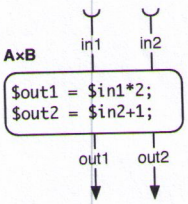

Figure 3: Two blocks from Figure 1 composed in parallel. Renaming is performed such that the composed block has two inputs and two outputs.

Finally, to implement the special permutation and identity blocks, it suffices to perform the appropriate bookkeeping on the variable names. The identity block, I_1, requires no generated code at all – the variable name on the input is simply used for the output as well. More complex permutations work similarly – Twig will rearrage the generated variable names, but no actual code needs to be generated.

leave the code generation out.
Describe the block together with the rule.

$$\frac{}{t \xrightarrow{\mathrm{T}} (t, I)} \qquad \frac{}{t \xrightarrow{\mathrm{F}} \bot}$$

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

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

Figure 6: Semantics for basic operators

$$\frac{f \neq \texttt{tuple}}{f(\ldots) \xrightarrow{s} \bot} \qquad \frac{i > n}{\texttt{tuple}(t_1, \ldots, t_n) \xrightarrow{s(i)} \bot}$$

Figure 7: Common semantics for tuple operators

The family of *branch* operators on tuples applies a single rule to one, all, or some of the tuple's elements, depending on the variation. In Figure 9 we present the semantics for the case where the rule applies to a single element: the first element, from left to right, for which the rule application does not fail. The other elements of the tuple are passed through wherein the branch operation fails if any single rule fails. Note the branch operator's use of a fanout block, defined in Section 3.1.3. We elide the semantics of the other branch operators for lack of space.

Next, we have the *projection* operator which extracts a single indexed element from a tuple, and its close relative, the *path* operator, which applies a rule to a single indexed tuple element, leaving the other elements untouched. The semantics are given in Figure 10 and Figure 11.

Finally, the *permutation* operator allows arbitrary permutation of a tuple's elements, including duplicating or dropping elements. The semantics are given in Figure 12. We provide a slightly specialized semantics when the permutation takes just one input – in this case, we equate a 1-tuples and terms, which allows the permutation operator to duplicate single terms. This case is given in Figure 13.

Building on this special case, we also provide a "fan out" operator. We define this operator in terms of the one input permutation operator, shown in Figure 14.

4.5 Expression names

As a convenience, Twig allows rules and rule expressions to be named, like so:

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

This assigns a primitive rule to the name `intToFloat`. Names must begin with lower-case letters, and can only be used once. Now that name can be used in place of the rule itself

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

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

Figure 8: Semantics for congruence operator

in expressions, like this:

```
pairOfIntsToFloats = (intToFloat,intToFloat)
```

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.

5. IMPLEMENTATION

Our implementation of Twig is written in Haskell. Twig expects as input a `.twig` file containing a list of named rule expressions along with a `main` rule expression, as described in Section 4.5. It also expects an initial value (i.e. a term, representing a C type), which will be used as the input to the main rule expression.

Twig must also be configured with a mapping from terms to C types. Currently, this mapping is provided with a simple key/value text file, but we are working on a more flexible alternative.

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 C code. If desired, this code block may be redirected to a separate file and included in a C program using the `#include` directive.

We are currently examining ways in which this process might be more easily incorporated into a typical C programmer's workflow.

5.1 Code Generation

Our current implementation of this model supports the generation of C code, and adds some extra features to support that language. These features include such details 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 looking into the possibility of incorporating these and other features into the language-neutral model, but for the moment they are specific to C.

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 choice combinator, a single typemap expression may be used to generate multiple variations

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

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

Figure 9: Semantics for branch-one operator

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

Figure 10: Semantics for projection operator

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

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

Figure 11: Semantics for path operator

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. Unlike Swig, Twig can

4. Type variables: allows polymorphic typemaps, e.g., `[ptr(A) -> A]`.

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

We will 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 structure comes in both a `float` and `double` variety, and we need to be able to convert both. As a final twist, the Python code is expecting a Cartesian, not polar, coordinate system, 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 `floats` to `doubles`, and use a congruence to lift it to a conversion on tuples. We sequence this cast after `unpackf` so that that rule will produce `doubles` instead of `floats`. 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 `doubles`.

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

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

Next, we need to 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 `doubles`, 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}
```

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

Next, we must convert from the C types to Python. We use Python's C interface API [1], which allows us to work with Python values in C (the same system used by Swig).

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

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

Figure 12: Semantics for permutation operator

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

Figure 14: Semantics for fan operator

```
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 stuff a pair of `pyfloats` into a Python tuple object (no longer 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;
  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;
}
```

[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.

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

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

Figure 13: Semantics for permute 1 operator

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);
```