Reducing and Removing Extraneous Typemaps

Geoff Hulette

July 14, 2010

1 Introduction

Typo is a programming language intended to assist in creating *typemaps* – programs that convert types in one language to those in another. In fact, in Typo, a typemap is a program itself, used to generate type-specific code in one or more target languages.

A typemap can be thought of a function, where the domain is the set of types in the source language, and the co-domain is the set of types in the target language plus an additional, special value $\perp$. The value $\perp$ is used to indicate that the given input type could not be mapped. Note that the domain and co-domain of a typemap may be the same – this corresponds to a transformation of a datatype within the single language.

Typo has two kinds of typemaps: primitive and composite. Primitive typemaps represent the basic conversions that are built into the target language interoperability system. Composite typemaps are created by composing typemaps (either primitives or other composites) together to form more complex rules. Some of the ways in which typemaps may be composed are discussed in the next section.

2 Notation

The domain and co-domain of a typemap are sets of possible types in a language. The particular language is usually represented by a single capital letter. Here, we use `C` for the C language, `F` for Fortran, and `S` for SIDL.

A typemap operates on terms, which represent instances of a type (i.e. any potential value having that type) in the target language. The output from a typemap is either another term or else $\perp$. We represent terms by the name of the type in the language, preceded by the letter code for the language. So, to represent a floating point number in C, we write `Cfloat`.

Terms may also be tuples of other terms, and these tuples may be nested. Typo does not distinguish between a 1-tuple and a non-tupled term. For example, (`Cint`, `Cfloat`) is a 2-tuple of a C integer and float. Tuples always exist in a single domain, so it is not permitted to mix languages in a single tuple.

Typemaps are named by lowercase identifiers. When necessary, the domain and co-domain of a typemap will be indicated as well. For example, $f : \texttt{C} \rightarrow \texttt{S}$ is a typemap named f, with C and SIDL as its domain and co-domain, respectively.

Typemaps are composed using a set of typemap operators. There are many operators in Typo; here we use just a few:

- $f; g$ represents a sequence, in this case the sequence f then g. If the result of f is $\perp$, g is ignored and the result is $\perp$.
- (f, g) represents a branch. In this case, the input term is fed to both f and g, the output is a 2-tuple with the results f in the first position, and the result of g in the second. Branching can easily be generalized to n-tuples.
- $\langle f, g \rangle$ represents a congruence. The input term must be a 2-tuple, and the result is a 2-tuple with f applied to the first element and g applied to the second. Congruence can easily be generalized to n-tuples.
- $f|g$ represents left choice. In this case, the input is fed to f first, and if the output is not $\perp$, it is the final result. If it is $\perp$, the input is then tried on g, and the final result is the output of g.

  Finally, $f = g; h$ represents a definition of f. In this case, f is assigned the sequence of g with h. Now, wherever f appears, we may substitute $g; h$. Once defined, the name f is considered bound and may not changed or be redefined.

3 Typemap Reductions

Determining the mapping of complex types between just two languages is a complicated task. For n languages, the number of required mappings (for a complete set) is $O(n^2)$ – too many mappings to code and maintain. A common solution is to define only the maps from each language to a common IDL, and back. Now for n languages you need just $2n$ mappings for the IDL, and to get from one language to another you use the IDL as an intermediate step. Thus, any language can be mapped to any other.

But, the IDL may introduce overhead, both in terms of performance and software complexity. It would be nice, therefore, if we could use the IDL mappings but "factor out" the middle, IDL-related portion of the code. If the mappings are encoded as Typo typemaps, we may be able to do exactly this. The idea is to *reduce* the typemaps by recognizing and removing redundant parts of the composite typemap rules. How we do this is a matter of ongoing research, but we outline some of the ideas below.

In some cases, typemaps may be *invertible*. For example, if we have a typemap $f : \texttt{C} \rightarrow \texttt{C}$ which accepts a `Cstruct` with fields `Cfloat x` and `Cfloat y`, and produces a 2-tuple of `Cfloat`s corresponding to two fields, then the inverse $f^{-1} : \texttt{C} \rightarrow \texttt{C}$ would be a typemap which accepts a 2-tuple of `Cfloat`s and produces the original `Cstruct`. Now, if we ever see a sub-sequence $f; f^{-1}$, we can safely replace it with an identity typemap. We call this replacement a *reduction*.

Other rules include such things as replacing a sequenced typemap $f : \texttt{A} \rightarrow \texttt{B}; g : \texttt{B} \rightarrow \texttt{C}$ with some single, equivalent $h : \texttt{A} \rightarrow \texttt{C}$, or combining sequences of rule congruences with a single congruence of sequences.

Note that at least some of these reduction rules are necessarily part of the domain typemaps, as they may be different for different languages. It may be possible in some cases to infer rules from other rules; this is another topic of ongoing research.

4 Example

Consider the example of mapping datatypes representing complex numbers. Assume we start with the following pre-defined typemaps:

- $csprim : \texttt{C} \rightarrow \texttt{S}$, a primitive typemap that accepts an `Cfloat` and produces an equivalent `Sfloat` (possibly among other simple primitive type mappings).
- $sfprim : \texttt{S} \rightarrow \texttt{F}$, like $csprim$, but maps `Sfloat`s to `Ffloat`s.
- $cfprim : \texttt{C} \rightarrow \texttt{F}$, like $csprim$ and $sfprim$, but maps `Cfloat`s to `Ffloat`s.
- $cdec : \texttt{C} \rightarrow \texttt{C}$, accepts a C structure called `Complex` having fields `Cfloat real` and `Cfloat imag`, and producing a 2-tuple $(\texttt{Cfloat}, \texttt{Cfloat})$ where the elements are the `real` and `imag` fields, respectively.
- $fcom : \texttt{F} \rightarrow \texttt{F}$, accepts a 2-tuple $(\texttt{Ffloat}, \texttt{Ffloat})$ representing the real and imaginary complex parts, and produces an `Fcomplex`.
- $scom : \texttt{S} \rightarrow \texttt{S}$, accepts a 2-tuple of `Sfloats` and produces an `Sfcomplex`.
- $sdec : \texttt{S} \rightarrow \texttt{S}$, accepts an `Sfcomplex` and produces a 2-tuple of `Sfloats`.

Now we can build a pair of typemaps. The first, $f : \texttt{C} \rightarrow \texttt{S}$ converts a C complex `struct` to a SIDL complex number, like so:

$$f = cdec; \langle csprim, csprim \rangle; scom$$

The second, $g : \texttt{S} \rightarrow \texttt{F}$, converts a SIDL complex number to a Fortran complex number:

$$g = sdec; \langle sfprim, sfprim \rangle; fcom$$

If we sequence f and g, we can create a new typemap $h : \texttt{C} \rightarrow \texttt{F}$:

$$h = f; g$$

or, by substituting f and g with their definitions:

$$h = cdec; \langle csprim, csprim \rangle; scom; sdec; \langle sfprim, sfprim \rangle; fcom$$

This sequence of typemaps will take us from the C complex structure to a Fortran complex type, but it will do so by way of SIDL. To eliminate the SIDL, we can reduce the typemap by introducing and applying some reduction rules.

First, we recognize that `scom` and `sdec` are inverse typemaps (i.e. $sdec = scom^{-1}$). Exactly how we recognize this is a matter of research; in the worst case, and certainly in this case in the absence of additional information, we would have to be told by the typemap programmer. In any event, we introduce a rule

$$\frac{x; y; y^{-1}; z}{x; z}$$

where x, y, and z range over typemaps. The rule basically says that wherever we see a typemap and its inverse in direct sequence, it can be eliminated. After considering this rule, $scom; sdec$ may be eliminated, and h reduced to

$$h = cdec; \langle csprim, csprim \rangle; \langle sfprim, sfprim \rangle; fcom$$

Now, we introduce another rule

$$\frac{\langle w, x \rangle; \langle y, z \rangle}{\langle w; y, x; z \ \rangle}$$

This allows us to infer that a sequence of congruences can be equivalently expressed as a congruence of sequences. We can use this to rewrite h once again:

$$h = cdec; \langle csprim; sfprim, csprim; sfprim \rangle; fcom$$

Finally, we introduce a rule declaring that converting a primitive `float` from C to SIDL, and then from SIDL to Fortran, is the same as converting it directly from C to Fortran:

$$\frac{csprim; sfprim}{cfprim}$$

Now h may be reduced one last time:

$$h = cdec; \langle cfprim, cfprim \rangle; fcom$$

Note that SIDL-related conversions are entirely removed – none of the typemaps have SIDL as either their domain or co-domain. As pointed out above, this could have a drastic positive effect on the speed of the conversion, as well as reducing the software infrastructure required to support it.

5 Conclusion

Typo, with the addition of reduction rules, represents a possible significant step towards reducing the code complexity of interlanguage programming. It provides a simple but powerful language for encoding complex type mappings, and also may be useful in circumventing slow, IDL-related code in some software contexts.