Typing Template Haskell: Soft Types

Ian Lynagh*
Programming Tools Group
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, OX1 3QD, UK
ian.lynagh@comlab.ox.ac.uk

ABSTRACT

In this paper we give the motivation for an improved type system for Template Haskell, a recent compile-time meta-programming extension for Haskell, and explain what properties we would like the type system to have and why. We then present a type system, based on *soft types*, for a simplified Template Haskell-like language that has these properties. We prove that it has principal types and describe an algorithm that implements type inference for this type system.

1. INTRODUCTION

Template Haskell [8] is a recent extension to Haskell [4] which has been implemented in GHC [5], the leading Haskell compiler. Template Haskell adds the ability to do compile-time meta-programming.

The bulk of the extension by volume is a set of datatypes, similar to those you would see in a compiler, representing the abstract syntax trees of expressions, types, patterns etc. However, to support compile-time meta-programming we also need some new syntax. The key pieces that GHC gives us are quasi-quotes and splices, akin to similar constructs of LISP [1].

Quasi-quotes, represented by $[\![\ldots]\!]$ brackets, allow you to turn concrete syntax into abstract syntax trees in the Template Haskell datatypes; for example, the result of $[\![1 + 2]\!]$ is an abstract syntax tree representing the expression $1 + 2$. The expression $[\![1 + 2]\!]$ is given the type Exp by the type-checker.

Splices, represented by a \$ followed by an expression, are the opposite—they turn abstract syntax trees into code, as if the code had been entered by hand by the programmer. The contents of a splice, *i.e.* the expression following the \$, must have type Exp. For example, $\$(\mathbf{let}\ x = [\![1 + 2]\!]\ \mathbf{in}\ x)$ is the same as just writing $1 + 2$, but in general you would expect more work to be done in the body of a splice.

It is also possible to splice expressions inside quasi-quotes; for example, $\backslash e_1\ e_2 \rightarrow [\![\$e_1 + \$e_2]\!]$ is the function that takes two abstract syntax trees and returns the abstract syntax tree with a + infix operator node at the root and the two inputs as the children. By contrast, $\backslash e_1\ e_2 \rightarrow [\![e_1 + e_2]\!]$ takes two *numbers* and returns the abstract syntax tree with three nodes, the + infix operator node at the root and the two numbers as chilren.

As we said earlier, these language features are similar to those of LISP. A major difference, however, is the fact that Template Haskell is type-safe: it is not possible to compile a program that generates ill-typed code. To achieve this property, the use of splicing and quotation is confined to compile-time, so that the type-correctness can be verified.

The language provides more than we have described here, and things are also complicated slightly by the use of a monad and a couple of side conditions, but this description will suffice for our purposes.

*Sponsored by a grant from Microsoft Research, Cambridge

$$
\begin{aligned}
c &= [\![\texttt{'C'}]\!] \\
s &= [\![\texttt{"Str"}]\!] \\
x &= [\![\$c \mathbin{+\!\!+} \$s]\!] \\
y &= \$x
\end{aligned}
$$

Figure 1: Example incorrect meta-program

2. OBJECTIVES AND CONTRIBUTIONS

Consider the code shown in Figure 1. The first line binds c to a quasi-quoted character and the second binds s to a quasi-quoted string. The third then binds x to the piece of code which represents the concatenation of the values represented by c and s. Finally y is bound to the value of represented by x.

However, $(+\!\!+)$ is meant for concatenating two lists, so has type $[\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$, whereas here we are just prepending an additional character 'C' onto the list of characters "Str" so we should have used the (:) operator. We would therefore expect our compiler to tell us we have used a value of type Char where a value of type $[\alpha]$ was expected.

We can think of the current Template Haskell implementation as working in four stages. First the contents of all quasi-quotes are typechecked; if there is a splice inside a quasi-quote then we treat it as having the type $\forall\alpha.\alpha$, as computation may be needed to know what the spliced in abstract syntax tree—and thus its type—is. The type we infer for the quasi-quote contents is discarded—this stage only checks that the quasi-quoted code is typeable[1].

All quasi-quoted expressions are given the type Exp before the second stage, which checks that the contents of all top-level splices (*i.e.* splices not inside quasi-quotes) have type Exp.

The third stage is to actually run the top-level splices, substituting the code generated into the module at the corresponding splice point, before the fourth stage in which the whole module, including the generated code, is checked with the normal Haskell typechecker to make sure the resulting module is type-correct. Compilation continues as normal from here on.

In our example, in the first stage we find that the contents of the quasi-quotes can be given types Char, String and $[\alpha]$ respectively, so we move on to the second stage. The body of the only top-level splice is x, which is bound to a quasi-quoted expression so has type Exp, so we pass this stage too. The splice is then run and y is bound to `'C'` ++ `"Str"`. It is only in the final stage, when the whole module is type-checked normally, that the error is detected.

There are two issues here. The first is that we have had to perform computation before we discovered that we had a type error. Here the amount of computation was trivial, but in general complete Haskell programs can be run to generate splices. Just as the static type checking of Haskell rejects bad programs without having to run them and see if they fail, so too we would like to be able to reject bad meta-programs without running them.

The second is issue is to do with the error we are currently given:

```
<no location info>:
    Couldn't match `[a]' against `Char'
      Expected type: [a]
      Inferred type: Char
    In the first argument of `(++)', namely 'C'
    In the definition of `y': y = $[splice]x
```

This is describing a type error in the generated code rather than the generating code, and is showing us the point where it is spliced in, the definition of y, rather than where the incorrect code was created, the definition of x. Although in this simple case it is easy to trace the error back, with real programs the generated output will be a lot larger, and the generating code may be spread across many modules. We would prefer to get a type error that tells us what is wrong with the generating program rather than the generated program, such as:

[1]It is also necessary to find out which way any operators in the quasi-quoted code should bind, and how tightly.

```
Line 3, character 10:
    Couldn't match `[a]' against `Char'
      Expected type: [a]
      Inferred type: Char
    In the first argument of `(++)', namely $c
    In the definition of `x': x = [| $c ++ $s |]
```

One possible way to achieve this would be to take the same approach as MetaML [9]. Rather than using the type Exp for quasi-quoted we would give it type Exp τ, where τ is the type of the code that it represents. In such a system we would have assigned the type Exp Char to c, the type Exp String to s and then failed to assign a type to x under standard typing rules.

However, there are two problems with such a type system. First, when we have the ability to manipulate the code not just with quasi-quotes and splices, but also with the data constructors of the abstract syntax tree datatypes, we would need a type system that is a radical departure from the familiar Hindley-Milner style type system of Haskell, and as such is probably still some time off. Work is being carried out in this direction [7], though.

Second, such a type system significantly restricts the set of programs that we can write. For example, with the current system it is possible to write a function akin to C's printf provided the format string is known statically. Code for the particular use of printf would then be spliced in at compile time, e.g.

$(*printf* `"%d %s"`) 5 `"foo"`

It is not possible to write this with a type system akin to that of MetaML for the same reason that it is not possible to write printf as a Haskell function. To see why, consider this massively simplified printf-a-like:

$$\backslash x \to \ \textbf{if}\ x == \texttt{"\%s"}\ \textbf{then}\ [\![\backslash s \to s]\!]\ \textbf{else}\ [\![x]\!]$$

Here we would want the first branch of the conditional to be given type Exp (String $\to$ String) and the second branch to be given type Exp String, but under a Hindley-Milner style type system this means the conditional as a whole cannot be typed. What we would like, then, is a type system in which different Exp τ types in different branches of conditionals are permitted.

To summarise, the properties we would like from our type system, which we will specify as an extension to Hindley-Milner, are:

- It must be a conservative extension of the Hindley-Milner type system, *i.e.* if you have a program that does not use any template features and a type that does not mention the template types then the type system must allow the program to have the type if and only if Hindley-Milner does.
- For each typeable expression there must be a principal type, and furthermore the principal type of expressions

e	$=$	n	Integer
	$\|$	true	True data constructor
	$\|$	false	False data constructor
	$\|$	v	Program variable
	$\|$	$e_1\ e_2$	Application
	$\|$	$\textbf{if}\ e1\ \textbf{then}\ e2\ \textbf{else}\ e_3$	Conditional
	$\|$	$\lambda v \rightarrow e$	Lambda abstraction
	$\|$	$\textbf{let}\ v = e_1\ \textbf{in}\ e_2$	Let binding
	$\|$	$[\![e]\!]$	Quasi-quoted expression
	$\|$	$\$e$	Splice

Figure 2: Simple template expression language

that do not use any template syntax must be the same as the principal type of the expression under Hindley-Milner.

- Whether an expression can be typed must be decidable. Furthermore there must be a type inference algorithm which, given an expression, returns its principal type if it is typeable and fails otherwise.
- Finally, the type system must accept all reasonable metaprograms.

Note that this last point does not mean that all metaprograms that are currently accepted should be accepted. For example, just as Haskell would reject the program:

$\textbf{let}\ s = \text{"Str"}\ \textbf{in}\ \textbf{if}\ \mathsf{True}\ \textbf{then}\ 1+2\ \textbf{else}\ 1+s$

we would like to reject:

$\$(\textbf{let}\ s = [\![\text{"Str"}]\!]\ \textbf{in}\ \textbf{if}\ \mathsf{True}\ \textbf{then}\ [\![1+2]\!]\ \textbf{else}\ [\![1+\$s]\!])$

but the current Template Haskell implementation would allow it.

In addition we have two desiderata of a less formal nature, namely:

- An implementation should be able to report errors at their origin.
- Errors should be reported as early as possible.

The main contribution of this paper is a type system and inference algorithm for a simplified Template Haskell that satisfy the above requirements.

3. TYPE SYSTEM

After investigating a number of possibilities, the best match that we have found for our requirements is a type system based on soft types [3]. In this section we present the type system we have devised.

3.1 Simple template expression language

In order to describe the type system we will use the simple expression language shown in Figure 2. The operational semantics for all but the last two cases are standard and evaluation order is unimportant; the only noteworthy point is that **let** bindings may be recursive. The final two cases are quasi-quotes and splices which behave as previously discussed. In examples we may additionally use character and string literals, tuples, lists etc. for reasons of clarity.

τ	$=$	Int	Integer type
	$\|$	Bool	Boolean type
	$\|$	$\tau_1 \rightarrow \tau_2$	Function type
	$\|$	α	Type variable (α, β, γ)
	$\|$	$\mathcal{E}\ \tau$	Expression type

Figure 3: Types for the simple template expression language

Figure 3 describes the types we will be assigning to expressions. The first 4 cases are again standard. We use $\mathcal{E}\ \tau$ to mean an abstract syntax tree representing a value of type τ, *i.e.* it is equivalent to the $\mathsf{Exp}\ \tau$ type of MetaML. We use a different name to make it clear that we are referring to the type in our type system. Again we may additionally use other ground types, tuple types, list types etc. in examples.

3.2 Prerequisites

To specify the type system we will make use of an environment which we denote Γ. This is an upward-growing stack of binders b representing bound variables. Each binder is a tuple of the form (λ, v, τ) or $(\textbf{let}, v, \tau)$. The former says that the variable v is λ-bound to τ, while the latter says that it is **let**-bound to τ. We use b, Γ to mean the stack Γ with the binder b pushed on top. We say $b \sqsubset \Gamma$, pronounced b is *active in* Γ, if b is the top-most binder on the stack that binds its variable.

Substitutions ϕ are maps from type variables to types. We extend them in the natural way to map types to types, using juxtaposition to mean application and $(\cdot)$ for composition, *i.e.* $(\phi_1 \cdot \phi_2)\tau = \phi_1(\phi_2\tau)$. We will use the $\{v_1 \mapsto \tau_1, v_2 \mapsto \tau_2\}$ notation to write substitutions with the obvious meaning. We say $\phi_1 \leq \phi_2$, pronounced ϕ_1 is *less general* than ϕ_2, if $\exists \phi . \phi_1 = \phi \cdot \phi_2$.

Given a pair of types τ_1 and τ_2, we say $\tau_1 \leq \tau_2$, pronounced τ_1 is *less general* than τ_2, if $\exists \phi . \tau_1 = \phi\tau_2$. We say $\tau_1 \leq_\Gamma \tau_2$ if $\exists \phi . \tau_1 = \phi\tau_2$ and all type variables in all λ-bound types in Γ are not in the domain of ϕ.

While $=$ is used to test whether two types are exactly identical, we define a variant $\stackrel{\not{\mathcal{E}}}{=}$ which ignores the $\mathcal{E}$ subtypes of the inputs when comparing them, e.g. $(\mathcal{E}\ \mathsf{Bool}, \mathsf{Char}) \neq (\mathcal{E}\ \mathsf{Int}, \mathsf{Char})$ but $(\mathcal{E}\ \mathsf{Bool}, \mathsf{Char}) \stackrel{\not{\mathcal{E}}}{=} (\mathcal{E}\ \mathsf{Int}, \mathsf{Char})$.

3.3 Antiunification

We say (τ, ϕ_1, ϕ_2) is an antiunification [6] of τ_1 and τ_2 if ϕ_1 and ϕ_2 are the least general substitutions such that $\tau_1 = \phi_1\tau$ and $\tau_2 = \phi_2\tau$. For any pair of types τ_1 and τ_2 there is always an antiunification as $(\alpha, \{\alpha \mapsto \tau_1\}, \{\alpha \mapsto \tau_2\})$, for a fresh α, always satisfies the requirements.

Given a pair of antiunifications $\omega = (\tau, \phi_1, \phi_2)$ and $\omega' = (\tau', \phi'_1, \phi'_2)$ of two types we say $\omega \leq \omega'$, pronounced ω is *less general than* ω', if $\tau \leq \tau'$. A simple proof by contradiction

$$\textit{num:}\ \frac{}{\Gamma \vdash n :: \mathsf{Int}}$$

$$\textit{true:}\ \frac{}{\Gamma \vdash \mathsf{true} :: \mathsf{Bool}} \qquad \textit{false:}\ \frac{}{\Gamma \vdash \mathsf{false} :: \mathsf{Bool}}$$

$$\textit{var:}\ \frac{(\_, v, \tau) \sqsubset \Gamma \qquad \tau' \leq_\Gamma \tau}{\Gamma \vdash v :: \tau'}$$

$$\textit{app:}\ \frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash e_1\ e_2 :: \tau_2}$$

$$\boxed{\textit{cond:}\ \frac{\begin{array}{ll} \Gamma \vdash g :: \mathsf{Bool} & \tau \in G(\tau_t, \tau_f) \\ \Gamma \vdash t :: \tau_t & \tau_t \overset{\not\mathcal{E}}{=} \tau \\ \Gamma \vdash f :: \tau_f & \tau_f \overset{\not\mathcal{E}}{=} \tau \end{array}}{\Gamma \vdash \mathbf{if}\ g\ \mathbf{then}\ t\ \mathbf{else}\ f :: \tau}}$$

$$\textit{lambda:}\ \frac{(\lambda, v, \tau_1), \Gamma \vdash e :: \tau_2}{\Gamma \vdash \lambda v \to e :: \tau_1 \to \tau_2}$$

$$\textit{let:}\ \frac{(\bar{\lambda}, v, \tau_1), \Gamma \vdash e_1 :: \tau_1 \qquad (\mathbf{let}, v, \tau_1), \Gamma \vdash e_2 :: \tau_2}{\Gamma \vdash \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 :: \tau_2}$$

$$\textit{quote:}\ \frac{\Gamma \vdash e :: \tau}{\Gamma \vdash [\![e]\!] :: \mathcal{E}\ \tau} \qquad \textit{splice:}\ \frac{\Gamma \vdash e :: \mathcal{E}\ \tau}{\Gamma \vdash \$e :: \tau}$$

Figure 4: Type inference rules

shows that there is a least general antiunification of any two types.

As an example to aid intuition, if we take the two types $(\mathsf{Int}, \mathsf{Int}, \mathsf{Int}, \mathsf{Bool})$ and $(\mathsf{Char}, \mathsf{Char}, \alpha, \mathsf{Bool})$ then the least general antiunification is (modulo naming of new variables) $((\beta, \beta, \gamma, \mathsf{Bool}), \{\beta \mapsto \mathsf{Int}, \gamma \mapsto \mathsf{Int}\}, \{\beta \mapsto \mathsf{Char}, \gamma \mapsto \alpha\})$.

4. TYPING RULES

We use judgements of the form $\Gamma \vdash e :: \tau$ to mean expression e can be given type τ in the environment Γ. The typing rules, shown in Figure 4, are predominantly routine. The use of $\bar{\lambda}$ rather than λ in the *let* rule will be explained later; for now $\bar{\lambda}$ and λ should be treated identically. In the *var* rule we use $\_$ to match any type of binder in the same manner as Haskell's pattern syntax.

The unfamiliar quote and splice rules are just as you would expect, lifting and unlifting the type into $\mathcal{E}$. It is the *cond* rule which is non-standard. Normally we would write the rule for conditionals as

$$\frac{\Gamma \vdash g :: \mathsf{Bool} \qquad \Gamma \vdash t :: \tau \qquad \Gamma \vdash f :: \tau}{\Gamma \vdash \mathbf{if}\ g\ \mathbf{then}\ t\ \mathbf{else}\ f :: \tau}$$

which is equivalent to

$$\frac{\begin{array}{lll} \Gamma \vdash g :: \mathsf{Bool} & \Gamma \vdash t :: \tau_t & \tau_t = \tau \\ \Gamma \vdash f :: \tau_f & & \tau_f = \tau \end{array}}{\Gamma \vdash \mathbf{if}\ g\ \mathbf{then}\ t\ \mathbf{else}\ f :: \tau}$$

If we compare this to our *cond* rule we see that the equality has been restricted to just compare the non-$\mathcal{E}$ parts of the type. Furthermore, there is an additional clause $\tau \in G(\tau_t, \tau_f)$. We will now show the motivation behind this clause and explain what it means.

Let us consider an example, again based on a cutdown printf:

$$\begin{array}{ll} \backslash x \to & \mathbf{case}\ x\ \mathbf{of} \\ & \texttt{"\%s"} \to [\![\backslash s \to s :: \mathsf{String}]\!] \\ & \texttt{"\%d"} \to [\![\backslash d \to \mathit{show}\ (d :: \mathsf{Int})]\!] \end{array}$$

The types of the two branches of the conditional are

$\mathsf{String} \to \mathcal{E}\ (\mathsf{String} \to \mathsf{String})$

and

$\mathsf{String} \to \mathcal{E}\ (\mathsf{Int} \to \mathsf{String})$.

When this is given a string argument and run at compile time to produce a piece of code we know that the generated piece of code will take a value of some type—although we are not sure which type—and return something of type String. We can represent this as the type[2]

$\mathsf{String} \to \mathcal{E}\ (\alpha \to \mathsf{String})$

This is the least general antiunification of the types of the branches of the conditional.

However, while using the least general antiunification would give us the strongest type that we knew was satisfied, it introduces issues for which we do not currently have a solution. These issues are discussed in Section 6. Instead we use a weaker variant in which, if the two branch types differ in the same way in different positions, then we do not use the same type variable in the result type, but rather introduce a fresh type variable for each position. For example, if the branches have types $(\mathsf{Int}, \mathsf{Int})$ and $(\mathsf{Bool}, \mathsf{Bool})$ then, rather than the least general antiunification type (α, α), we use (α, β). We still disallow matching parts of types to be replaced by type variables, though, so γ would not be permissible in this example. When we need to distinguish between the two we will call this form *weak least general antiunification* and the traditional form *strong least general antiunification*. To compute the weak least general antiunification of two types one walks over the two type abstract syntax trees in lockstep, copying nodes while the two input trees match and inserting a fresh type variable where the roots of a subtree differ.

However, the antiunified type is not what should appear in our typing rule. To see why, consider a use of this expression:

$$\$ \left(\left(\begin{array}{ll} \backslash x \to & \mathbf{case}\ x\ \mathbf{of} \\ & \texttt{"\%s"} \to [\![\backslash s \to s :: \mathsf{String}]\!] \\ & \texttt{"\%d"} \to [\![\backslash d \to \mathit{show}\ (d :: \mathsf{Int})]\!] \end{array} \right) \texttt{"\%d"} \right) 5$$

[2]Future revisions of the type system may introduce new features, such as union types, which might allow us to give the type of the conditional as

$\mathcal{E}\ ((\mathsf{String} \lor \mathsf{Int}) \to \mathsf{String})$.

For both simplicity and familiarity to the user we have chosen to use the softer type $\mathsf{String} \to \mathcal{E}\ (\alpha \to \mathsf{String})$.

Now suppose that we had assigned the conditional the type $\mathcal{E}\ (\alpha \to \mathsf{String})$. Then the root rule of the typing derivation tree would be of the form

$$app{:}\frac{\Gamma \vdash \$(\ldots) :: \alpha \to \mathsf{String} \qquad \Gamma \vdash 5 :: \mathsf{Int}}{\Gamma \vdash \$(\ldots)\ 5 :: \mathsf{String}}$$

But this does not fit the requirements of the *app* rule as α is not the same as Int. Thus rather than inserting type variables at points where the branch types differ we need to allow the insertion of an arbitrary type.

Thus the function G is similar to a function that returns the type component of the weak least general antiunification of two types, but rather than returning the type with type variables inserted it instead returns the infinite set of types with every possible combination of types substituted. This is different to applying a substitution after antiunifying the two types as it cannot affect type variables already in the input types.

Note that the equality constraints mean that it is only the $\mathcal{E}$ types that can be affected by the antiunification. As a special case, when no $\mathcal{E}$ types are involved the rule is identical to the traditional one.

We have only allowed softness to be introduced by conditional statements. To see why this decision makes sense, as an example consider what would happen if we allowed softness in applications. Suppose an application has type $\mathcal{E}\ \mathsf{Int} \to \mathcal{E}\ \mathsf{Int}$; to aid your intuition it might help to think about $\backslash x \to [\![\$x + 5]\!]$. If we were to apply something of type $\mathsf{Exp\ Char}$ to it then we would get a value that does not make sense. The reason it makes sense for conditionals is that, when our meta-program is run, only one of the branches will be chosen, so we will have code of the type of that branch.

4.1 Principal Types

The principal type property [2] says that, in an environment Γ, there is a "best" type for each typeable expression e; for Hindley-Milner based type systems that is normally a type τ such that e can be assigned type τ' if and only if $\tau' \leq \tau$.

Having principal types is important for a number of reasons; for example, principal types make it meaningful to ask the implementation what type an expression has and they make type inference and separate compilation possible. We therefore require that our type system has principal types. Furthermore, so as not to confuse the user, it is important that the principal type of an expression not making use of the Template extensions should be the same as its principal type in Haskell; the obvious choice, then, is the least general type, as described above.

However, in our type system as specified so far, not every typeable expression does have a principal type. To see why, let us consider the expression $\backslash x\ y \to \mathbf{if}\ \mathsf{True}\ \mathbf{then}\ x\ \mathbf{else}\ y$. This expression can be assigned the type $a \to a \to a$, just as we could if we were dealing with normal Haskell. Using the flexibility of the new *cond* rule, however, we could also assign the type $\mathcal{E}\ a \to \mathcal{E}\ b \to \mathcal{E}\ c$ to the expression. However, no type more general than both of these is acceptable. Furthermore there is no obvious alternative definition of principal types that would mean we could assign this expression a principal type.

In order to make the principal types property apply we impose a side condition on the type derivation trees that, informally, says that we can only assign $\mathcal{E}\ \tau$ types where they are actually needed to make the expression typeable. A more formal definition follows.

We first introduce a new type constructor $\bar{\mathcal{E}}$ which takes a single type argument in the same way as $\mathcal{E}$. There are no typing rules that mention $\bar{\mathcal{E}}$ explicitly.

Now suppose we have a type derivation tree for some expression. Everywhere where $\mathcal{E}$ appears we replace it with $\bar{\mathcal{E}}$. Then, while there is at least one such rule, we replace an $\bar{\mathcal{E}}$ that a typing rule requires to be an $\mathcal{E}$ with an $\mathcal{E}$; for example, if somewhere in our tree we had

$$app{:}\frac{\Gamma \vdash e_1 :: \mathcal{E}\ \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 :: \bar{\mathcal{E}}\ \tau_1}{\Gamma \vdash e_1\ e_2 :: \tau_2}$$

then we would have to change the $\bar{\mathcal{E}}$ to $\mathcal{E}$ due to the requirements of the *app* rule. Once there are no such cases left, if the type derivation tree has no $\bar{\mathcal{E}}$s then the typing is defined to be valid.

A simple inductive argument shows that $\mathcal{E}$ types now appear in exactly the same places in any two typing trees for any given expression. This means that we cannot have problems with some derivation trees using antiunification but others not, as we had in the earlier example.

This is enough to give us principal types. However, before we prove it in the next section, we have one last issue to take care of—it is not clear that type inference for the type system is tractable. Consider this expression:

$$\begin{array}{l}\mathbf{let}\ f = \backslash x\ y \to \mathbf{if}\ \mathsf{True}\ \mathbf{then}\ x\ \mathbf{else}\ y\\ \ \mathbf{in}\ f\ [\![\texttt{'C'}]\!]\ [\![\texttt{"Str"}]\!]\end{array}$$

In (normal) Haskell we would type the binding of f, generalise the variables in its type as appropriate and then type the body of the **let**. However, in this case we need to look at the use of f to see that we need to give the arguments $\mathcal{E}$ types, and hence that we need to perform antiunification in the branches of the conditional.

For a related problem, consider this expression:

$$\backslash x\ y \to \begin{pmatrix}\mathbf{let}\ a = \backslash z \to \mathbf{if}\ \mathsf{True}\ \mathbf{then}\ x\ \mathbf{else}\ z\\ \ \mathbf{in}\ \mathbf{if}\ \mathsf{True}\ \mathbf{then}\ y\ \mathbf{else}\ [\![\texttt{'C'}]\!] \qquad ,\\ \mathbf{let}\ b = \backslash z \to \mathbf{if}\ \mathsf{True}\ \mathbf{then}\ y\ \mathbf{else}\ z\\ \ \mathbf{in}\ \mathbf{if}\ \mathsf{True}\ \mathbf{then}\ x\ \mathbf{else}\ [\![\texttt{'C'}]\!]\end{pmatrix}$$

If a variable used in a **let**-binding is λ-bound outside of the binding, such as x and y here, and if we are allowed to make use of its type to decide whether to allow antiunification, then we run in to similar difficulties. In the above case it is not possible to work out a's type without looking at the body of the second **let** and vice-versa.

We thus add a second side-condition, similar to the first. Suppose there is a *let* rule somewhere in the type derivation tree for our expression. Then consider just the binding subtree of this rule. As before we replace every $\mathcal{E}$ with $\bar{\mathcal{E}}$, and then iteratively replace a $\bar{\mathcal{E}}$ that a typing rule requires to be $\mathcal{E}$ with an $\mathcal{E}$ until no such $\bar{\mathcal{E}}$ remains. If the type derivation (sub-)tree now doesn't meet the requirements of the typing rules then the original, complete, type derivation tree is defined to be invalid. This ensures that a let binding can be typed without knowledge of the context, as in Haskell.

4.2 Proof of principal types

Suppose we have an environment Γ, an expression e and two derivation trees Δ_1 and Δ_2 for e in Γ. The type inference rules are syntax-directed. Therefore there is never any choice as to which rule should be applied for any expression, so the structure of the two trees must be identical; the only things that can vary are the types. Furthermore, as the expressions appearing in the antecedents of the rules are always strictly smaller (using the obvious definition for size of expressions) than the expressions in the consequent, the derivation tree for a finite expression must be finite. If two trees are identical apart from alpha renaming of type variables then we consider them equal; for the purposes of the proof we need only consider non-equal trees. We also assume, without loss of generality, that the two trees use disjoint sets of type variables apart from those in the initial environment.

We define the *size* of a type to be the number of nodes in its abstract syntax tree minus the number of distinct type variables in it. We define the size of a derivation tree to be the sum of the sizes of all the types in it.

Trivially, the antiunification of two different types is smaller than one of the original types and no larger than the other. We extend antiunification to work on entire derivation trees in the natural way and define $T(\Delta_1, \Delta_2)$ to return the tree component of the strong least general antiunification of Δ_1 and Δ_2. Any type variables T inserts will be fresh.

Suppose that, for any typeable expression e, if $\Gamma \vdash e :: \tau_1$ and $\Gamma \vdash e :: \tau_2$ then $\Gamma \vdash e :: T(\tau_1, \tau_2)$. Then the set of typings for any typeable expression is a directed subset of the set of all types. As the size of a type has a lower bound of 0, there must be a set of alpha-equivalent smallest types in the set of typings of e, representing its most general type. Thus to show that we have principal types it will suffice to show that

$$(\Gamma \vdash e :: \tau_1 \wedge \Gamma \vdash e :: \tau_2) \Rightarrow \Gamma \vdash e :: T(\tau_1, \tau_2)$$

For the proof, we need some lemmata. These are shown below. When we use T in the lemmata we are referring to the result as produced by the antiunification of the complete type trees rather than a separate invocation.

Lemma 4.1

$(u_1 \leq t_1 \wedge u_2 \leq t_2) \Rightarrow T(u_1, u_2) \leq T(t_1, t_2)$

Proof

By the definition of $\leq$ we have $\exists \phi_x . u_1 = \phi_x t_1$. By the definition of T we have $\exists \phi_y . t_1 = \phi_y T(t_1, t_2)$. Thus if we let $\phi_1 = \phi_x . \phi_y$ then $u_1 = \phi_1 T(t_1, t_2)$. Symmetrically we can construct ϕ_2 such that $u_2 = \phi_2 T(t_1, t_2)$. Hence $(T(t_1, t_2), \phi_1, \phi_2)$ is an antiunification of u_1 and u_2. By the definition of T, $T(u_1, u_2) \leq T(t_1, t_2)$. ■

Lemma 4.2

$$(t \in G(t_1, t_2) \wedge u \in G(u_1, u_2)) \Rightarrow T(t, u) \in G(T(t_1, u_1), T(t_2, u_2))$$

Proof

By induction over the abstract syntax tree of $T(t, u)$.

Lemma 4.3

Two valid typings have $\mathcal{E}$ types in the same places in the abstract syntax trees of their types.

Proof

Suppose for the purpose of contradiction we have two valid typings which have $\mathcal{E}$ types in a different set of places. Take an instance nearest the root of the typing trees where there is an $\mathcal{E}$ type in only one tree. The need for there to be an $\mathcal{E}$ type here comes from one of 4 places:

- This node is a quasi-quote node and it is an $\mathcal{E}$ at the outer level in the consequent. Contradiction: The other tree must also have an $\mathcal{E}$ type here.
- This node is a splice node and it is an $\mathcal{E}$ at the outer level in the antecedent. Contradiction: The other tree must also have an $\mathcal{E}$ type here.
- This node is a variable node and the $\mathcal{E}$ type comes from its type in the environment. Contradiction: The node that inserted the variable into the environment must also have an $\mathcal{E}$ node in one tree only and is nearer the root than us.
- The need comes from one of our children nodes. In this case we can hide the current node and repeat the process on each of our children.

Due to the finite size of derivation trees we can show that no node can demand the $\mathcal{E}$ node by induction. ■

Theorem 4.1

$(\Gamma \vdash e :: \tau_1 \wedge \Gamma \vdash e :: \tau_2) \Rightarrow \Gamma \vdash e :: T(\tau_1, \tau_2)$

Proof

If Δ_1 is a type derivation tree for $\Gamma \vdash e :: \tau_1$ and Δ_2 is a type derivation tree for $\Gamma \vdash e :: \tau_2$ then $T(\Delta_1, \Delta_2)$ is a type derivation tree for $\Gamma \vdash e :: T(\tau_1, \tau_2)$ provided it is a derivation tree according to our typing rules. Therefore it will suffice to show that $T(\Delta_1, \Delta_2)$ is a type derivation tree given our type rules.

We will proceed by induction on the size of the derivation tree, omitting cases where the result is immediate given it is true for the antecedents. For each case we will show the root node in the two input trees followed by the root node in the resulting tree in a box.

$$\boxed{\begin{array}{l}
\bullet\ \text{HM-var:}\dfrac{(b, v, \tau_{x1}) \sqsubset \Gamma_1 \qquad \tau_{y1} \leq_{\Gamma_1} \tau_{x1}}{\Gamma_1 \vdash v :: \tau_{y1}} \\
\bullet\ \text{HM-var:}\dfrac{(b, v, \tau_{x2}) \sqsubset \Gamma_2 \qquad \tau_{y2} \leq_{\Gamma_2} \tau_{x2}}{\Gamma_2 \vdash v :: \tau_{y2}} \\
\bullet\ \text{HM-var:}\dfrac{(b, v, \tau_{xT}) \sqsubset \Gamma_T \qquad \tau_{yT} \leq_{\Gamma_T} \tau_{xT}}{\Gamma_T \vdash v :: \tau_{yT}}
\end{array}}$$

Trivially we have $\tau_{y1} \leq \tau_{x1}$ and $\tau_{y2} \leq \tau_{x2}$, so Lemma 4.1 tells us $\tau_{yT} \leq \tau_{xT}$. Now suppose the substitution needs to change a type variable α in τ_{xT} that is λ-bound in Γ_T. Then there must be corresponding types τ_1 and τ_2 that α was the antiunification of in the other trees. Furthermore, these types must be in the respective environments in the corresponding places, so any type variables in them are also λ-bound. Therefore they must be unchanged by the $\leq_{\Gamma_1}$ and $\leq_{\Gamma_2}$ relations so must appear in the same place in the type tree on the left hand side as the right hand side. Thus T will have used the same type variable in the corresponding place in τ_{yT}. Therefore it is not necessary to substitute for type variables λ-bound in Γ_T, so $\tau_{yT} \leq_{\Gamma_T} \tau_{xT}$ as required.

$$\boxed{\begin{array}{l}
\bullet\ \textit{cond:}\dfrac{\begin{array}{cc}\Gamma_1 \vdash g :: \mathsf{Bool} & \tau_1 = G(\tau_{t1}, \tau_{f1}) \\ \Gamma_1 \vdash t :: \tau_{t1} & \tau_{t1} \stackrel{\not\subseteq}{=} \tau_1 \\ \Gamma_1 \vdash f :: \tau_{f1} & \tau_{f1} \stackrel{\not\subseteq}{=} \tau_1\end{array}}{\Gamma_1 \vdash \textbf{if } g \textbf{ then } t \textbf{ else } f :: \tau_1} \\[2em]
\bullet\ \textit{cond:}\dfrac{\begin{array}{cc}\Gamma_2 \vdash g :: \mathsf{Bool} & \tau_2 = G(\tau_{t2}, \tau_{f2}) \\ \Gamma_2 \vdash t :: \tau_{t2} & \tau_{t2} \stackrel{\not\subseteq}{=} \tau_2 \\ \Gamma_2 \vdash f :: \tau_{f2} & \tau_{f2} \stackrel{\not\subseteq}{=} \tau_2\end{array}}{\Gamma_2 \vdash \textbf{if } g \textbf{ then } t \textbf{ else } f :: \tau_2} \\[2em]
\bullet\ \textit{cond:}\dfrac{\begin{array}{cc}\Gamma_T \vdash g :: \mathsf{Bool} & \tau_T = G(\tau_{tT}, \tau_{fT}) \\ \Gamma_T \vdash t :: \tau_{tT} & \tau_{tT} \stackrel{\not\subseteq}{=} \tau_T \\ \Gamma_T \vdash f :: \tau_{fT} & \tau_{fT} \stackrel{\not\subseteq}{=} \tau_T\end{array}}{\Gamma_T \vdash \textbf{if } g \textbf{ then } t \textbf{ else } f :: \tau_T}
\end{array}}$$

The three clauses in the left-hand column are just simple recursive cases. The top clause in the right-hand column is an instance of Lemma 4.2, and the other two clauses are instances of Lemma 4.3. ∎

5. ALGORITHM

We start off this section by giving some more definitions necessary to discuss the algorithm. Then, before giving the soft typing algorithm, we give a traditional Hindley-Milner algorithm in the style we will use so the differences are clear.

We introduce a new alternative in the definition of types; we now have a new sort of type variable which we call an *internal type variable* and write ι. These are introduced and removed during the type inference process and never remain in the final type. We also introduce a new form of substitution, from internal type variables to types, written σ. We define $U(\tau_1, \tau_2)$ to return the most general unifier (as a σ substitution) of τ_1 and τ_2 when it exists and fails otherwise.

5.1 Hindley-Milner Algorithm

The algorithm makes use of an environment Γ, as we used for the type system specification. We assume that Γ initially contains only **let**-bound variables.

We will also use a global stack of sets of constraints. Constraints are of the form $\mathscr{U}(\tau_1, \tau_2)$, meaning that τ_1 and τ_2 are to be unified, *i.e.* they are the same type. Initially the stack is empty.

The algorithm is defined in terms of a pair of functions, *gather* and *solve*, which have types (Env, Exp) $\to$ Type and Set Constraint $\to$ Substitution respectively. Given an expression e, the type of e is given by $\mathit{typecheck}(\Gamma, e)$, which is defined to be $\mathit{rename}(\mathit{gather}(\Gamma, \textbf{let}\ v = e\ \textbf{in}\ v))$ where Γ is the initial environment and v is fresh. The reason for wrapping the expression up in a **let** is so that constraint solving happens for the entire expression. We specify $\mathit{rename}(\tau)$ as returning $\sigma\tau$, where σ is chosen such that $\sigma\tau$ contains no internal type variables and $\sigma\iota = \sigma\iota' \Rightarrow \iota = \iota'$. We define $\mathit{gather}(\Gamma, e)$ by matching on e as follows:

n

- Return Int

true

- Return Bool

false

- Return Bool

v

- Return the type in the top binding for v in Γ, substituting a fresh internal type variable for each instance of a type variable

$e_1\ e_2$

- Let $\tau_1 = \mathit{gather}(\Gamma, e_1)$ and $\tau_2 = \mathit{gather}(\Gamma, e_2)$
- Take a fresh internal type variable ι
- Add the constraint $\mathscr{U}(\tau_1, \tau_2 \to \iota)$ to the top constraint set on the stack
- Return ι

if e_1 **then** e_2 **else** e_3

- Let $\tau_1 = \mathit{gather}(\Gamma, e_1)$, $\tau_2 = \mathit{gather}(\Gamma, e_2)$ and $\tau_3 = \mathit{gather}(\Gamma, e_3)$
- Add the constraints $\mathscr{U}(\tau_1, \mathsf{Bool})$ and $\mathscr{U}(\tau_2, \tau_3)$ to the top constraint set on the stack
- Return τ_2

$\lambda v \to e$

- Take a fresh internal type variable ι
- Let $\tau = \mathit{gather}(((\lambda, v, \iota), \Gamma), e)$

- Return $\iota \to \tau$

let $v = e_1$ **in** e_2

- Push a new constraint set onto the stack
- Take a fresh internal type variable ι
- Let $\tau_1 = \mathit{gather}(((\lambda, v, \iota), \Gamma), e_1)$
- Add the constraint $\mathscr{U}(\iota, \tau_1)$
- Let C be the top constraint set on the stack and pop it
- Let $\sigma = \mathit{solve}(C)$
- Apply σ to the entire constraint set stack and any environments and types being used in instances of *gather* in the call chain[3], including our Γ and τ_1.
- Substitute in τ_1 a fresh type variable for each instance of an internal type variable not in the type of a λ binding in Γ, yielding τ_1'
- Return $\mathit{gather}(((\mathbf{let}, v, \tau_1'), \Gamma), e_2)$

$[\![e]\!]$

- Let $\tau = \mathit{gather}(\Gamma, e)$
- Return $\mathcal{E}\ \tau$

$\$e$

- Take a fresh internal type variable ι
- Let $\tau = \mathit{gather}(\Gamma, e)$
- Add the constraint $\mathscr{U}(\mathcal{E}\ \iota, \tau)$
- Return ι

The *solve* function takes a set of constraints and returns the most general substitution σ such that $\sigma\tau_1 = \sigma\tau_2$ if and only if τ_1 and τ_2 are equal given the constraints in the set, or fails if no such substitution exists.

5.2 Loop Checking

Before giving the Soft Type algorithm, we first specify a loop checking procedure that is used by it.

Given a pair of types τ_1 and τ_2 we define a relation $\mathcal{R}_i(\iota_1, \iota_2)$, read "$\iota_2$ is immediately reachable from ι_1", to be $f(\tau_1, \tau_2)$ where

$$\begin{array}{lcl} f(\iota_1, \iota_2) & = & \{\} \\ f(\iota_1, \tau) & = & \{(\iota_1, \iota_2) \mid \iota_2 \in \mathbb{I}_{\not\mathcal{E}}(\tau)\} \\ f(\tau, \iota_2) & = & \{(\iota_1, \iota_2) \mid \iota_1 \in \mathbb{I}_{\not\mathcal{E}}(\tau)\} \\ f(\tau_{1a} \to \tau_{1b}, \tau_{2a} \to \tau_{2b}) & = & f(\tau_{1a}, \tau_{2a}) \cup f\tau_{1b}, \tau_{2b}) \\ f(\mathcal{E}\ \tau_1, \mathcal{E}\ \tau_2) & = & f(\tau_1, \tau_2) \end{array}$$

Given a pair of types τ_1 and τ_2 we define a relation $\mathcal{N}(\iota_1, \iota_2)$, read "$\iota_2$ is a neighbour of ι_1", to be $f(\tau_1, \tau_2)$ where

$$\begin{array}{lcl} f(\iota, \iota) & = & \{\} \\ f(\iota_1, \iota_2) & = & \{(\iota_1, \iota_2)\} \\ f(\iota_1, \tau) & = & \{\} \\ f(\tau, \iota_2) & = & \{\} \\ f(\tau_{1a} \to \tau_{1b}, \tau_{2a} \to \tau_{2b}) & = & f(\tau_{1a}, \tau_{2a}) \cup f\tau_{1b}, \tau_{2b}) \\ f(\mathcal{E}\ \tau_1, \mathcal{E}\ \tau_2) & = & f(\tau_1, \tau_2) \end{array}$$

[3]This could be described more formally but, as it will be removed in the new algorithm, we opt for simplicity instead.

Given a pair of types τ_1 and τ_2 we define a relation $\mathcal{R}(\iota_1, \iota_2)$, read "$\iota_2$ is reachable from ι_1", to be the smallest transitive relation such that $\mathcal{R}_i(\iota_1, \iota_2) \Rightarrow \mathcal{R}(\iota_1, \iota_2)$ and $\mathcal{R}(\iota_1, \iota_2) \wedge \mathcal{N}(\iota_2, \iota_3) \Rightarrow \mathcal{R}(\iota_1, \iota_3)$.

We then lift all three relations from pairs of types to sets of types in the natural way, and further lift it to sets of soft unification constraints, where the three type in the constraint are all used.

Then, given a constraint set C, if the reachable relation for the soft unification constraints in this set is such that $\exists \iota . \mathcal{R}(\iota, \iota)$ then the loop check fails. Otherwise it succeeds.

5.3 Soft Type Algorithm

We now give the changes to the algorithm needed to implement the soft type algorithm. In order to handle conditionals in the soft type system we now have three forms of constraints:

- $\mathscr{U}(\tau_1, \tau_2)$
- $\mathscr{U}(\tau, \mathscr{S}(\tau_1, \tau_2))$
- $\mathscr{U}(\tau, \mathscr{A}(\tau_1, \tau_2))$

The first form states that τ_1 and τ_2 are to be unified, as before. The second says that τ_1 and τ_2 are to be soft-unified—the form of unification used for branches of conditionals, which we can think of as unification until we are inside an $\mathcal{E}$ type at which point we switch to antiunification—and the result unified with τ. The third says that τ_1 and τ_2 are to be antiunified and the result unified with τ. What will happen is that conditionals will introduce constraints of the second form and, during the solving process, constraints of the third form will be generated when we enter $\mathcal{E}$ types while solving a soft unification constraint.

We need only change two clauses in the definition of *gather*, the conditional case and the let case. The changed lines are marked with ►:

if e_1 **then** e_2 **else** e_3

- Let $\tau_1 = \mathit{gather}(\Gamma, e_1)$, $\tau_2 = \mathit{gather}(\Gamma, e_2)$ and $\tau_3 = \mathit{gather}(\Gamma, e_3)$
- ► Add the constraints $\mathscr{U}(\tau_1, \mathsf{Bool})$ and $\mathscr{U}(\iota, \mathscr{S}(\tau_2, \tau_3))$ to the top constraint set on the stack
- ► Return ι

let $v = e_1$ **in** e_2

- Push a new constraint set onto the stack
- Take a fresh internal type variable ι
- Let $\tau_1 = \mathit{gather}(((\lambda, v, \iota), \Gamma), e_1)$
- Add the constraint$\mathscr{U}(\iota, \tau_1)$
- Let C be the top constraint set on the stack and pop it
- ► Add the constraints in C to the top constraint set on the stack

- Let $\sigma = solve(C)$
- ► Let $\Gamma' = \sigma\Gamma$ and $\tau_1' = \sigma\tau_1$
- Substitute in τ_1' a fresh type variable for each instance of an internal type variable not in the type of a λ binding in Γ', yielding τ_1''
- ► Return $gather(((\mathbf{let}, v, \tau_1''), \Gamma), e_2)$

First, in the conditional case, we need to change from normal unification to soft unification. Then, in the let case, rather than immediately applying the solution to the constraint set we got, we just add the constraints to the set below the one they were collected in. This is necessary so that if we have to type another let expression at the same level the information we got from solving this one cannot leak out and affect what antiunification is done. For the same reason the substitution we get from solving the constraints is not applied to the environments and types in calls to *gather* above us, and an unmodified environment is used to gather constraints in the **let** body.

We also need to change $solve(C)$ to cope with the new forms of constraints it is handling. We say a type is structured if it is not an internal type variable, and when we say "we demand that τ_1 and τ_2 have the same root structure" we do the following, dependent on the two types:

τ_1 is structured, τ_2 is structured
: If the root node of the two type trees are not identical then fail.

τ_1 is unstructured, τ_2 is structured
: Create a type τ whose type tree has the same root node as τ_2 and fresh internal type variables for each of its children. Add the constraint $\mathscr{U}(\tau_1, \tau)$ to the top constraint set on the stack. Return $\{\tau_1 \mapsto \tau\}$.

τ_1 is structured, τ_2 is unstructured
: Symmetric to the τ_1 is unstructured, τ_2 is structured case.

We will never demand that two unstructured types have the same root structure. We extend the operation to sets of types in the natural way.

We say that a constraint is a *unification* constraint if:

- It is of the form $\mathscr{U}(\tau_1, \tau_2)$.

We say that a constraint is *reducible* if:

- It is of the form $\mathscr{U}(\tau, \mathscr{S}(\tau_1, \tau_2))$ and any of τ, τ_1 and τ_2 are structured.
- It is of the form $\mathscr{U}(\tau, \mathscr{A}(\tau_1, \tau_2))$ and τ_1 and τ_2 are structured.

and *unifiable* if:

- It is of the form $\mathscr{U}(\tau, \mathscr{S}(\tau_1, \tau_2))$
- It is of the form $\mathscr{U}(\tau, \mathscr{A}(\tau_1, \tau_2))$ where $\tau_1 = \tau_2$

We solve the constraints as follows:

- Let $\sigma = \{\}$
- While there is a unification constraint in C:
  - Remove a unification constraint $\mathscr{U}(\tau_1, \tau_2)$ from C.
  - Let $\sigma' = U(\tau_1, \tau_2)$
  - Let $\sigma = \sigma' \cdot \sigma$
  - Apply σ' to C
- Loop check C
- While there is a reducible constraint in C:
  - Remove a reducible constraint c from C.
  - If c is of the form $\mathscr{U}(\tau, \mathscr{S}(\tau_1, \tau_2))$:
    * Let σ' be the result of demanding τ, τ_1 and τ_2 have the same root structure
    * Apply σ' to τ, τ_1 and τ_2
    * Let $\sigma = \sigma' \cdot \sigma$
    * If τ is of the form $\mathcal{E}\ \tau'$ then add a constraint of the form $\mathscr{U}(,\mathscr{A}(,))$ for the child type triple to C
    * Otherwise we add a a constraint of the form $\mathscr{U}(,\mathscr{S}(,))$ for each of the child type triples to C
  - If c is of the form $\mathscr{U}(\tau, \mathscr{A}(\tau_1, \tau_2))$ and τ_1 and τ_2 have the same root structure:
    * Let σ' be the result of demanding τ and τ_1 have the same root structure
    * Apply σ' to τ, τ_1 and τ_2
    * Let $\sigma = \sigma' \cdot \sigma$
    * Add a constraint of the form $\mathscr{U}(,\mathscr{S}(,))$ for each of the child type triples to C
  - If c is of the form $\mathscr{U}(\tau, \mathscr{A}(\tau_1, \tau_2))$ and τ_1 and τ_2 have different root structure:
    * Let $\sigma' = \{\}$
  - Apply σ' to C
- While there is a unifiable constraint in C:
  - Remove a unifiable constraint c from C.
  - If c is of the form $\mathscr{U}(\tau, \mathscr{S}(\tau_1, \tau_2))$:
    * Let $\sigma_1 = U(\tau, \tau_1)$
    * Let $\sigma_2 = U(\sigma_1\tau, \sigma_1\tau_2)$
    * Let $\sigma = \sigma_2 \cdot \sigma_1 \cdot \sigma$
    * Add the constraints $\mathscr{U}(\tau, \tau_1)$ and $\mathscr{U}(\tau, \tau_2)$ to the top constraint set on the stack.
  - If c is of the form $\mathscr{U}(\tau, \mathscr{A}(\tau', \tau'))$:
    * Let $\sigma' = U(\tau, \tau')$
    * Let $\sigma = \sigma' \cdot \sigma$
  - Apply σ' to C
- Return σ

When we say "a constraint of the form $\mathscr{U}(,\mathscr{S}(,))$ for each of the child type triples" we mean this:

Suppose that we are dealing with three types $\tau = (\tau_x, \tau_y)$, $\tau_1 = (\tau_{1x}, \tau_{1y})$ and $\tau_2 = (\tau_{2x}, \tau_{2y})$. Then there are two child type triples, namely $\{\tau_x, \tau_{1x}, \tau_{2x}\}$ and $\{\tau_y, \tau_{1y}, \tau_{2y}\}$. We would thus add the pair of constraints $\mathscr{U}(\tau_x, \mathscr{S}(\tau_{1x}, \tau_{2x}))$ and $\mathscr{U}(\tau_y, \mathscr{S}(\tau_{1y}, \tau_{2y}))$.

We are currently working on a formal proof that the algorithm is correct with respect to the specification. Its outline is as follows. For all but the constraint solving in the **let** case, the correctness of the algorithm is immediate given the typing rules. It can be shown that the algorithm is guaranteed to terminate given a finite constraint set, and that, upon termination, the algorithm has produced a valid solution of the constraint set. It is less intuitive that simply passing the constraints down the stack will give a consistent solution given the antiunification constraints, but this can also be shown to be the case. Finally, as each step of the algorithm leaves the types as general as is possible, we can show that when the algorithm terminates we have the most general solution.

6. STRONG ANTIUNIFICATION

When specifying the type system we allowed G to give different replacement types for identical pairs of differences in the input types. While that means we catch fewer errors, we asserted it was necessary. We now explain the reasons why.

In order to think about this we can imagine a map from pairs of types to types associated with each conditional. This tells us what type each pair of corresponding different types in the conditional branches should map to in the conditional's type.

We will show two problems that arise when strong antiunification is used. The first is to do with the principal types property. Suppose we have this expression, where all the variables are λ bound:

$(f\ (\textbf{if}\ \mathsf{True}\ \textbf{then}\ [\![(a,b)]\!]\ \textbf{else}\ [\![(5,5)]\!])$
$, f\ [\![(c,[\![5]\!])]\!]$
$, \textbf{if}\ \mathsf{True}\ \textbf{then}\ c\ \textbf{else}\ d)$

We can give a and b different types, giving a type assignment such as:

$a :: \mathsf{Char}$
$b :: \mathsf{Bool}$
$c :: \mathsf{String}$
$d :: \mathsf{String}$

and the map for the first conditional contains

$(\mathsf{Char}, \mathsf{Int}) \mapsto \mathsf{String}$
$(\mathsf{Bool}, \mathsf{Int}) \mapsto \mathcal{E}\ \mathsf{Int}$

However, if we make a and b the same then c is required to have the same type as $[\![5]\!]$, giving a type assignment like

$a :: \mathsf{Bool}$
$b :: \mathsf{Bool}$
$d :: \mathcal{E}\ \mathsf{Int}$
$d :: \mathcal{E}\ \alpha$

and the map

$(\mathsf{Bool}, \mathsf{Int}) \mapsto \mathcal{E}\ \mathsf{Int}$

However, as we saw earlier, having $\mathcal{E}$ types in different places in different types for an expression means we do not get principal types, as is the case here, most clearly seen in the third component of the tuple. Before we used the valid typing side condition to eliminate this possibility, but the antiunification maps allow it to return.

The second problem is an implementation issue. Although it may be that a different form of algorithm is needed for the stronger system, we show what goes wrong with our algorithm anyway as it hints towards a possible solution to the principal types problem.

Suppose we kept the algorithm largely as it is now, but modified our constraints to also carry around a map, with the same map being shared by all constraints stemming from a single conditional in the expression; *i.e.* our constraints would look like this:

- $\mathscr{U}(\tau_1, \tau_2)$
- $\mathscr{U}(\tau, \mathscr{S}(\tau_1, \tau_2, \mu))$
- $\mathscr{U}(\tau, \mathscr{A}(\tau_1, \tau_2, \mu))$

Now suppose that during a run of the algorithm we have a map μ containing $(v, \mathsf{Int}) \mapsto x$ and $(w, \mathsf{Int}) \mapsto \mathcal{E}\ \mathsf{Int}$, that the constraint set contains no reducible constraints but there is a constraint $\mathscr{U}(y, \mathscr{S}(v, w, \mu))$. Then we assume that there are no $\mathcal{E}$ types within v, w and y and unify them. However, if v and w are unified then the two mappings in the map have the same left hand side, so x and $\mathcal{E}\ \mathsf{Int}$ must be the same type. Therefore we need to add the constraint $\mathscr{U}(x, \mathcal{E}\ \mathsf{Int})$. This in turn may mean that constraints that we previously thought were unifiable are now reducible; furthermore, we may have already unified a constraint that at the time looked unifiable, but which we now know think should have been antiunified. It is possible to construct expressions such that there is no order in which the constraints can be solved without finding you should have antiunified a constraint which you have already unified.

The key point here is that the obvious thing for the algorithm to do in this case is to ignore the additional structure that may appear in constraints and to continue unifying unifiable constraints regardless. The problem is that it is not clear how to translate this back into the type system specification, and thus how to explain what is going on to users. If this could be done then intuitively it seems likely that the additional conditions would restore the principal types property.

7. CONCLUSIONS AND FURTHER WORK

We have presented a type system, and a type inference algorithm for the type system, that meet our desiderata for a type system for Template Haskell. As we said earlier, we are currently working on a formal proof of correctness of the algorithm with respect to the specification. We have also written a prototype implementation for it. However, the type system currently only covers the language of this paper; we have yet to investigate expanding it to cover full Haskell, plus the other GHC extensions.

By converting the example in Figure 1 to an equivalent program in our language we have confirmed that our prototype does find the error statically, and have also checked it on a number of other small examples. We have not yet tried it on any real meta-programs.

As discussed in the previous section, we would like to be able to use strong antiunification for conditionals rather than the weak form we are currently using. However, we are not currently sure how to specify the strengthened type system.

There is also potential for the type system to be strengthened by introducing union types. The type of a conditional **if** True **then** $[\![\texttt{'C'}]\!]$ **else** $[\![\texttt{"Str"}]\!]$ might be $\mathcal{E}\ (\mathsf{Char} \vee \mathsf{String})$ under such a type system, rather than our $\mathcal{E}\ \alpha$, so a larger set of errors could potentially be detected.

Finally, there are some cases where we do not infer the best type possible due to the hiding of type information when typing **let** bindings. For example, in:

$$
\begin{array}{ll}
\backslash x \to & \mathbf{let}\ y = \mathbf{if}\ \mathsf{True}\ \mathbf{then}\ x\ \mathbf{else}\ 5\ \ \mathbf{in} \\
 & \mathbf{let}\ z = \mathbf{if}\ \mathsf{True}\ \mathbf{then}\ [\![x]\!]\ \mathbf{else}\ [\![5]\!]\ \ \mathbf{in} \\
 & z
\end{array}
$$

we infer the type $\mathsf{Int} \to \mathcal{E}\ a$ whereas looking at the code we can tell it has type $\mathsf{Int} \to \mathcal{E}\ \mathsf{Int}$. If such cases turn out to be common in practice then it would be worth investigating whether the type system can be strengthened to find the better types in these cases.

8. ACKNOWLEDGEMENTS

This research was funded by a grant from Microsoft Research Cambridge. We would like to thank Oege de Moor and Andres Loeh for their valuable comments on a draft of this paper.

9. REFERENCES

[1] A. Bawden. Quasiquotation in lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999.

[2] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982.

[3] M. Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, Aug. 1992.

[4] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[5] S. Peyton Jones and S. Marlow et al. The Glasgow Haskell Compiler.
URL: `http://www.haskell.org/ghc/`.

[6] G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence 5*, pages 153–164, 1969.

[7] T. Sheard and E. Pasalic. Meta-programming with built-in type equality. 2004.

[8] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

[9] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.