Chapter 26

Typing Control

26.1 Conditionals

Let's expand our language with a conditional construct. We can use `if0` like before, but for generality it's going to be more convenient to have a proper conditional and a language of predicates. The type judgment for the conditional must have the following form:

$$\frac{\Gamma \vdash c : ??? \qquad \Gamma \vdash t : ??? \qquad \Gamma \vdash e : ???}{\Gamma \vdash \{\texttt{if}\ \ c\ \ t\ \ e\} : ???}$$

where c is the conditional, t the "then"-expression, and e the "else"-expression.

Let's begin with the type for c. What should it be? In a language like Scheme we permit any value, but in a stricter, typed language, we might demand that the expression always evaluate to a boolean. (After all, if the point is to detect errors sooner, then it does us no good to be overly lax in our type rules.) However, we don't yet have such a type in our type language, so we must first extend that language:

```
<type> ::= number
         | boolean
         | (<type> -> <type>)
```

Armed with the new type, we can now ascribe a type to the conditional expression:

$$\frac{\Gamma \vdash c : \texttt{boolean} \qquad \Gamma \vdash t : ??? \qquad \Gamma \vdash e : ???}{\Gamma \vdash \{\texttt{if}\ \ c\ \ t\ \ e\} : ???}$$

Now what of the other two, and of the result of the expression? One option is, naturally, to allow both arms of the conditional to have whatever types the programmer wants:

$$\frac{\Gamma \vdash c : \texttt{boolean} \qquad \Gamma \vdash t : \tau_1 \qquad \Gamma \vdash e : \tau_2}{\Gamma \vdash \{\texttt{if}\ \ c\ \ t\ \ e\} : ???}$$

By using two distinct type variables, we do not demand any conformity between the actual types of the arms. By permitting this flexibility, however, we encounter two problems. The first is that it isn't clear what

type to ascribe to the expression overall.[1] Second, it reduces our ability to trap program errors. Consider a program like this:

```
{+ 3
   {if {is-zero mystery}
       5
       {fun {x} x}}}
```

Because we know nothing about `mystery`, we must conservatively conclude that it *might* be non-zero, which means eventually we are going to see a type error that we only catch at run-time. But why permit the programmer to write such a program at all? We might as well prevent it from ever executing. Therefore, we use the following rule to type conditionals:

$$\frac{\Gamma \vdash c : \texttt{boolean} \qquad \Gamma \vdash t : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \{\texttt{if } c \ t \ e\} : \tau}$$

Notice that by forcing the two arms to have the same type, we can assign that common type to the entire expression, so the type system does not need to know which branch was chosen on a given execution: the type remains the same.

Having added conditionals and the type `boolean` isn't very useful yet, because we haven't yet introduced predicates to use in the test position of the conditional. Indeed, we can easily see that this is true becuase we have not yet written a function type with `boolean` on the right-hand side of the arrow. You can, however, easily imagine adding procedures such as `is-zero`, with type `number -> boolean`.

26.2 Recursion

Now that we have conditionals, if we can also implement recursion, we would have a Turing-complete language (that could, for instance instance, with a little more arithmetic support, enable writing factorial). So the next major piece of the puzzle is typing recursion.

Given the language TFAE (typed FAE), can we write a recursive program? Let's try to write an infinite loop. Our first attempt might be this FAE program

```
{with {f {fun {i}
           {f i}}}
  {f 10}}
```

which, expanded out, becomes

```
{{fun {f}
   {f 10}}
 {fun {i}
   {f i}}}
```

[1] It's tempting to create a new kind of type, a *union* type, so that the type of the expression is $\tau_1 \cup \tau_2$. This has far-reaching consequences, however, including a significant reduction in type-based guarantee of program reliability.

When we place type annotations on this program, we get

```
{{fun {f : (num -> num)} : num
   {f 10}}
 {fun {i : num} : num
   {f i}}}
```

These last two steps don't matter, of course. This program doesn't result in an infinite loop, because the `f` in the body of the function isn't bound, so after the first iteration, the program halts with an error.

As an aside, this error is easier to see in the typed program: when the type checker tries to check the type of the annotated program, it finds no type for `f` on the last line. Therefore, it would halt with a type error, preventing this erroneous program from ever executing.[2]

Okay, that didn't work, but we knew about that problem: we saw it in Section 9 when introducing recursion. At the time, we asked you to consider whether it was possible to write a recursive function without an explicit recursion construct, and Section 22 shows that it is indeed possible. The essence of the solution presented there is to use *self-application*:

```
{with {omega {fun {x}
                {x x}}}
  {omega omega}}
```

How does this work? Simply substituting `omega` with the function, we get

```
{{fun {x} {x x}}
 {fun {x} {x x}}}
```

Substituting again, we get

```
{{fun {x} {x x}}
 {fun {x} {x x}}}
```

and so on. In other words, this program executes forever. It is conventional to call the function ω (lower-case Greek "omega"), and the entire expression Ω (upper-case Greek "omega").[3]

Okay, so Ω seems to be our ticket. This is clearly an infinite loop in FAE. All we need to do is convert it to TFAE, which is simply a matter of annotating all procedures. Since there's only one, ω, this should be especially easy.

To annotate ω, we must provide a type for the argument and one for the result. Let's call the argument type, namely the type of `x`, τ_a and that of the result τ_r, so that $\omega : \tau_a \rightarrow \tau_r$. The body of ω is `{x x}`. From this, we can conclude that τ_a must be a function (arrow) type, since we use `x` in the function position of an application. That is, τ_a has the form $\tau_1 \rightarrow \tau_2$, for some τ_1 and τ_2 yet to be determined.

[2] In this particular case, of course, a simpler check would prevent the erroneous program from starting to execute, namely checking to ensure there are no free variables.

[3] Strictly speaking, it seems anachronistic to refer to the lower and upper "case" for the Greek alphabet, since the language predates moveable type in the West by nearly two millennia.

What can we say about τ_1 and τ_2? τ_1 must be whatever type x's argument has. Since x's argument is itself x, τ_1 must be the same as the type of x. We just said that x has type τ_a. This immediately implies that

$$\tau_a = \tau_1 \rightarrow \tau_2 = \tau_a \rightarrow \tau_2$$

In other words,

$$\tau_a = \tau_a \rightarrow \tau_2$$

What type can we write that satisfies this equation? In fact, no types in our type language can satisify it, because this type is recursive without a base case. Any type we try to write will end up being infinitely long. Since we cannot write an infinitely long type (recall that we're trying to annotate ω, so if the type is infinitely long, we'd never get around to finishing the text of the program), it follows by contradiction[4] that ω and Ω cannot be typed in our type system, and therefore their corresponding programs are not programs in TFAE. (We are being rather lax here—what we've provided is informal reasoning, not a proof—but such a proof does exist.)

26.3 Termination

We concluded our exploration of the type of Ω by saying that the annotation on the argument of ω must be infinitely long. A curious reader ought to ask, is there any connection between the boundlessness of the type and the fact that we're trying to perform a non-terminating computation? Or is it mere coincidence?

TFAE, which is a first cousin of a language you'll sometimes see referred to as the *simply-typed* lambda calculus,[5] enjoys a rather interesting property: it is said to be *strongly normalizing*. This intimidating term says of a programming language that no matter what program you write in the language, it will *always terminate*!

To understand why this property holds, think about our type language. The only way to create compound types is through the function constructor. But every time we apply a function, we discharge one function constructor: that is, we "erase an arrow". Therefore, after a finite number of function invocations, the computation must "run out of arrows".[6] Because only function applications can keep a computation running, the computation is forced to terminate.

This is a *very* informal argument for why this property holds—it is cetainly far from a proof (though, again, a formal proof of this property does exist). However, it does help us see why we must inevitably have bumped into an infinitely long type while trying to annotate the infinite loop.

What good is a language without infinite loops? There are in fact lots of programs that we would like to ensure will *not* run forever. These include:

- inner-loops of real-time systems
- program linkers

[4] We implicitly assumed it would be possible to annotate ω and explored what that type annotation would be. The contradiction is that no such annotation is possible.

[5] Why "simply"? You'll see what other options there are next week.

[6] Oddly, this never happens to mythological heroes.

- packet filters in network stacks
- client-side Web scripts
- network routers
- device (such as photocopier) initialization
- configuration files (such as Makefiles)

and so on. That's what makes the simply-typed lambda calculus so wonderful: instead of pondering and testing endlessly (so to speak), we get mathematical certitude that, with a correct implementation of the type checker, no infinite loops can sneak past us. In fact, the module system of the SML programming language is effectively an implementation of the simply-typed lambda calculus, thereby guaranteeing that no matter how complex a linking specification we write, the linking phase of the compiler will always terminate.

Exercise 26.3.1 *We've been told that the Halting Problem is undecidable. Yet here we have a language accompanied by a theorem that proves that all programs will terminate. In particular, then, the Halting Problem is not only very decidable, it's actually quite simple: In response to the question "Does this program halt", the answer is* always *"Yes!" Reconcile.*

Exercise 26.3.2 *While the simply-typed lambda calculus is fun to discuss, it may not be the most pliant programming language, even as the target of a compiler (much less something programmers write explicitly). Partly this is because it doesn't quite focus on the right problem. To a Web browsing user, for instance, what matters is whether a downloaded program runs* immediately*; five minutes isn't really distinguishable from non-termination.*

Consequently, a better variant of the lambda calculus might be one whose types reflect resources*, such as time and space. The "type" checker would then ask the user running the program for resource bounds, then determine whether the program can actually execute within the provided resources. Can you design and implement such a language? Can you write useful programs in it?*

26.4 Typed Recursive Programming

Strong normalization says we must provide an explicit recursion construct. To do this, we'll simply reintroduce our `rec` construct to define the language *TRCFAE*. The BNF for the language is

```
<TRCFAE> ::= ...
           | {rec {<id>  : <type> <TRCFAE>} <TRCFAE>}
```

with the same type language. Note that the `rec` construct needs an explicit type annotation also.

What is the type judgment for `rec`? It must be of the form

$$\frac{???}{\Gamma \vdash \{\texttt{rec}\ \{i\ :\ \tau_i\ v\}\ b\} : \tau}$$

since we want to conclude something about the entire term. What goes in the antecedent? We can determine this more easily by realizing that a `rec` is a bit like an immediate function application. So just as with functions, we're going to have *assumptions* and *guarantees*—just both in the same rule.

We want to assume that τ_i is a legal annotation, and use that to check the body; but we also want to guarantee that τ_i is a legal annotation. Let's do them in that order. The former is relatively easy:

$$\frac{\Gamma[i \leftarrow \tau_i] \vdash b : \tau \qquad ???}{\Gamma \vdash \{\texttt{rec}\ \ \{i\ :\ \ \tau_i\ \ v\}\ \ b\} : \tau}$$

Now let's hazard a guess about the form of the latter:

$$\frac{\Gamma[i \leftarrow \tau_i] \vdash b : \tau \qquad \Gamma \vdash v : \tau}{\Gamma \vdash \{\texttt{rec}\ \ \{i\ :\ \ \tau_i\ \ v\}\ \ b\} : \tau}$$

But what is the structure of the term named by v? Surely it has references to the identifier named by i in it, but i is almost certainly not bound in Γ (and even if it is, it's not bound to the value we want for i). Therefore, we'll have to extend Γ with a binding for i—not surprising, if you think about the scope of i in a `rec` term—to check v also:

$$\frac{\Gamma[i \leftarrow \tau_i] \vdash b : \tau \qquad \Gamma[i \leftarrow \tau_i] \vdash v : \tau}{\Gamma \vdash \{\texttt{rec}\ \ \{i\ :\ \ \tau_i\ \ v\}\ \ b\} : \tau}$$

Is that right? Do we want v to have type τ, the type of the entire expression? Not quite: we want it to have the type we promised it would have, namely τ_i:

$$\frac{\Gamma[i \leftarrow \tau_i] \vdash b : \tau \qquad \Gamma[i \leftarrow \tau_i] \vdash v : \tau_i}{\Gamma \vdash \{\texttt{rec}\ \ \{i\ :\ \ \tau_i\ \ v\}\ \ b\} : \tau}$$

Now we can understand how the typing of recursion works. We extend the environment not once, but twice. The extension to type b is the one that *initiates* the recursion; the extension to type v is the one that *sustains* it. Both extensions are therefore necessary. And because a type checker doesn't actually run the program, it doesn't need an infinite number of arrows. When type checking is done and execution begins, the run-time system does, in some sense, need "an infinite quiver of arrows", but we've already seen how to implement that in Section 10.

Exercise 26.4.1 *Define the* BNF *entry and generate a type judgment for* `with` *in the typed language.*

Exercise 26.4.2 *Typing recursion looks simple, but it's actually worth studying in detail. Take a simple example such as* Ω *and work through the rules:*

- *Write* Ω *with type annotations so it passes the type checker. Draw the type judgment tree to make sure you understand why this version of* Ω *types.*
- *Does the expression named by v in* `rec` *have to be a procedure? Do the typing rules for* `rec` *depend on this?*