1 Typing the Infinite Loop

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

\[
\{ \text{with } \{ f \text{ (fun } \{ i \}
\text{ (f } i)) \}
\{ f \text{ 10)} \}
\]

which, expanded out, becomes

\[
\{ \text{fun } \{ f \}
\text{ (f 10))}
\{ \text{fun } \{ i \}
\text{ (f } i))\}
\]

When we place type annotations on this program, we get

\[
\{ \text{fun } \{ f : (\text{num } \rightarrow \text{num}) \} : \text{num}
\{ f \text{ 10))}
\{ \text{fun } \{ i : \text{num} \} : \text{num}
\{ f \text{ } i))\}
\]

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.\(^1\)

Okay, that didn’t work, but we knew about that problem: we saw it before 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. We have since seen that this is possible from the lambda calculus lecture (using the \( Y \) combinator\(^2\)). Without going into the full glory of \( Y \), we can write an infinite loop by exploiting its central idea, namely self-application:

\[
\{ \text{with } \{ \omega \text{ (fun } \{ x \}
\text{ (x } x)) \}
\{ \omega \text{ } \omega)\}
\]

How does this work? Simply substituting \( \omega \) with the function, we get

\[
\{ \text{fun } \{ x \} \{ x } x))\}
\{ \text{fun } \{ x \} \{ x } x))\}
\]

Substituting again, we get

\(^1\)In this particular case, however, a simpler check would prevent the erroneous program from starting to execute, namely checking to ensure there are no free variables. Since the identifier is free in the last line, the program will halt with an unbound identifier error if execution ever reaches that code fragment.

\(^2\)A combinator is simply a procedure that has no free variables. Why would it have such a name?
and so on. In other words, this program executes forever! It is conventional to call the function \( \omega \) (lower-case omega), and the entire expression \( \Omega \) (upper-case omega). \(^3\)

Okay, so \( \Omega \) seems to be our ticket. This is clearly an infinite loop in \( \text{FWAE} \). All we need to do is convert it to \( \text{TFWAE} \), which is simply a matter of annotating all procedures. Since there’s only one, \( \omega \), this should be especially easy.

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

What can we say about \( \tau_1 \) and \( \tau_2 \)? \( \tau_1 \) must be whatever type \( x \)’s argument has. Since \( x \)’s argument is itself \( x \), \( \tau_1 \) must be the same as the type of \( x \). We just said that \( x \) has type \( \tau_a \). 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 satisfy 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 \( \omega \), 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 \( \omega \) and \( \Omega \) cannot be typed in our type system, and therefore their corresponding programs are not programs in \( \text{TFWAE} \). (We are being rather lax here—what we’ve provided is informal reasoning, not a proof—but such a proof does exist.)

## 2 Termination

We concluded our exploration of the type of \( \Omega \) by saying that the annotation on the argument of \( \omega \) 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? In fact, it’s not—\( \text{TFWAE} \) is a rather strange language!

\( \text{TFWAE} \), 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!\(^6\)

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 certainly far from a proof (though, again, formal proofs of this property do 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. This formal property also tells us that, no matter how valid our reasoning about the particulars of typing \( \Omega \), in general, it is impossible. We will need a different language if we must write infinite loops.

What good is a language without infinite loops?!? Well, think how often you encountered a program that truly does (or, rather, is intended to) run forever . . . That’s what we thought. (You should actually be able to come up with plenty of answers: operating systems, for instance!) In contrast, there are lots of programs that we would like to ensure will not run forever. These include:

\[^3\]Strictly speaking, it’s anachronistic to refer to the lower and upper “case” for the Greek alphabet, since it predates moveable type in the West by two millennia. I’m not sure why Greek has two cases, anyway.

\[^4\]We implicitly assumed it would be possible to annotate \( \omega \) 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 seems to never happen to the heroes of Indian and other ancient mythologies whose warriors fought with bows.
- real-time systems
- program linkers
- packet filters in network stacks
- client-side Web scripts
- network routers
- photocopier (and other) device initialization
- configuration files (such as Makefiles)

and so on. That’s what makes the simply-typed lambda calculus so neat: instead of pondering and testing endlessly (no pun intended), we get mathematical certitude that, with a correct implementation of the type checker, no infinite loops can sneak past us.

Of course, programming in this language can be a bit unwieldy. Nevertheless, it may still be useful to think of it as an “object code”, namely as a target for compilers of higher-level languages. Many people now use C, which was intended for humans, as a mere back-end for their compilers (where C takes the place of assembly language, being more portable and offering slightly better features such as register allocation). Likewise, in many domains, it may make sense to offer a better surface syntax, but use the simply-typed lambda calculus as a back-end target for a compiler. Indeed, next week we will see how effective a translator of this form can be, up to the point of inferring the type annotations!

**Puzzle**

- 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 (a loud and affirmative) “Yes!” What gives? Did the folks who wrote down the Halting Problem miss something? (Think this through carefully. You should be able to offer a very precise answer.)

- 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. If it doesn’t, then sometimes whether it runs after five minutes or five days, or even loops forever, scarcely matters, as the user’s attention has flitted to some other topic.

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?

### 3 Typed Recursive Programming

The last time we introduced recursion as an explicit language construct, it was because our limited imaginations did not figure out Y on their own. To compensate, we added a recursion construct. We later discovered that Y would have accomplished the same effect (conceptually, even if not in terms of efficiency, say). This time, however, there is no lurking, secret way of reclaiming recursion: strong normalization says we really are sunk. So we’ll have to add recursion again, explicitly, this time because we really don’t have a choice.

To do this, we’ll simply reintroduce our `rec` construct to define the language TRFAE. The BNF for the language is

```plaintext
<TRFAE> ::= <num>
    | (+ <TRFAE> <TRFAE>)
    | {fun {<id> : <type>} : <type> <TRFAE>}
    | {<TRFAE> <TRFAE>}
    | {rec {<id> : <type> <TRFAE>} <TRFAE>}
```

3
where

\[
\texttt{<type> ::= number} \\
| (\texttt{<type> -> <type>})
\]

Note that the \texttt{rec} construct now needs an explicit type annotation also.

What is the type judgment for \texttt{rec}? It must be of the form

\[
???
\]

\[
\Gamma \vdash \texttt{rec \{ i : \tau_i \} \ b} : \tau
\]

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 \texttt{rec} is a bit like an immediate function application—after all, it’s a refinement of \texttt{with}. So just as with functions, we’re going to have \textit{assumptions} and \textit{guarantees}, just both in the same rule.

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

\[
\Gamma[i\leftarrow\tau_i] \vdash b : \tau
\]

\[
\Gamma \vdash \texttt{rec \{ i : \tau_i \} \ b} : \tau
\]

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

\[
\Gamma[i\leftarrow\tau_i] \vdash b : \tau \quad \Gamma \vdash \tau
\]

\[
\Gamma \vdash \texttt{rec \{ i : \tau_i \} \ b} : \tau
\]

But what 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 \(\Gamma\) (and even if it is, it’s not bound to the value we want for \(i\)). Therefore, we’ll have to extend \(\Gamma\) with a binding for \(i\)—not surprising, if you think about the scope of \(i\) in a \texttt{rec} term—to check \(v\) also:

\[
\Gamma[i\leftarrow\tau_i] \vdash b : \tau \quad \Gamma[i\leftarrow\tau_i] \vdash v : \tau
\]

\[
\Gamma \vdash \texttt{rec \{ i : \tau_i \} \ b} : \tau
\]

Is that right? Do we want \(v\) to have type \(\tau\), the type of the entire expression? Not quite: we want it to have the type we promised it would have, namely \(\tau_i\):

\[
\Gamma[i\leftarrow\tau_i] \vdash b : \tau \quad \Gamma[i\leftarrow\tau_i] \vdash v : \tau_i
\]

\[
\Gamma \vdash \texttt{rec \{ i : \tau_i \} \ b} : \tau_i
\]

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 \textit{initiates} the recursion; the extension to type \(v\) is the one that \textit{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 using two different representations of environments!

\section*{Exercises}

Typing recursion looks deceptively simple, but it’s actually worth studying in a bit of detail. Take a simple example such as \(\Omega\) and work through the rules:

- Does the expression named by \(v\) have to be a procedure? Do the typing rules for \texttt{rec} depend on this? And if not, can you write a judgment for \texttt{rec} that forces \(v\) to syntactically be a procedure?
- First, write \(\Omega\) with type annotations so it passes the type checker.
- Second, trace through the type rules to make sure you understand why this version of \(\Omega\) types. Draw the \textit{tree} of judgments applied and discharged. This tree, like botanical ones but unlike other ones in computer science, actually grows upward; its leaves are \textit{axioms}, or rules with no antecedents (so they are always true). The axioms in our type system are the judgments for identifiers and numbers.
- Define the BNF entry and generate a type judgment for \texttt{with} in the typed language.