PROGRAMMING LANGUAGES AS THEOREM VERIFIERS

ANDREW FOOTE

One of the most interesting ideas in modern logic is the *Curry-Howard correspondence*. This is the informal observation that, from a certain perspective, mathematical proofs and computer programs are the same thing.

To be more exact: there is a correspondence between mathematical *formulas* and data *types* in computer programs. In particular, the correspondence is not with data *values*. This is something that puzzled me a little when learning about this—why couldn't it be with data values? But I think I can now express why it has to be with types: a mathematical formula expresses that something is the case, and a data type, when attached to a data value, can likewise be thought of as expressing that something is the case for that data value. A data value, on the other hand, is just a thing; it makes no sense to say that it *expresses* something. (Perhaps one could say that data values correspond to the *truth values* of formulas, as opposed to the formulas themselves.)

In addition, there is a correspondence between rules of inference and operations which combine programs. Just as mathematical proofs can be thought of as being built up from smaller proofs whose combination is justified by rules of inference, so computer programs can be thought of as being built up from smaller programs using various operations (different words may be used depending on the programming language—"subroutine", "function", etc.—but the most general way to describe how programs are built up is to say that they are built up from smaller programs).

As an illustration, let's have a look at how the Hilbert system which I talked about in the last post can be described in terms of programs, types and operations rather than proofs, formulas and rules of inference. A Hilbert system described in such a way becomes a system of *combinatory logic*.

The defining characteristic of Hilbert system is that they have few rules of inference and rely on axioms for their expressive power. Accordingly, combinatory logic systems have few ways of combining programs and rely on atomic programs known as *combinators* for their expressive power. (Note that the programs in combinatory logic are simple ones, which just transform inputs into outputs without altering any state. As a result, they can just as easily be thought of as mathematical functions—thinking of combinatory logic as a programming language is just one way of thinking of it, albeit one which comes quite naturally.)

Although combinatory logic can be presented formally, it exists embedded in any programming language with a reasonably powerful type system, such as Haskell. Doing combinatory logic in a practical programming language helps it sink in that there's a real correspondence there. So, here is a complete proof of the reflexivity of the conditional connective in a restricted fragment of Haskell:

```
s x y z = x z (y z)
k x y = x
```

```
i :: a -> a
i = s k k

main :: IO ()
main = return ()
```

The first two lines in this program define the `k` and `s` combinators, which correspond to the two axiom schemes in a minimal Hilbert system. These are implemented in Haskell as polymorphic functions, with instantiations of the functions with specific types corresponding to the individual axioms permitted by the schemes. Note the type signatures (I didn't write them out explicitly in the program, since Haskell can infer them automatically):

```
k :: a -> b -> a
s :: (a -> b -> c) -> (a -> b) -> a -> c
```

The signatures are just the same as the axioms of the minimal Hilbert system, except that the function type-forming operator `->` is used in place of the conditional connective, and the operands are thought of as type variables, not formula variables.

The third line, the type signature of `i`, is where we state the theorem that we're trying to prove—in this particular case, it's $p \to p$ for an arbitrary formula p. Replacing the formula variable p with a type variable a and the conditional connective with the `->` operator, we get the type signature `a -> a`. So in combinatory logic, the theorem amounts to the assertion that there is a function that returns values of the same type as its argument.

The proof itself is the definition of the function. This definition, by itself, determines the type of the function; the type signature declared in the third line only comes into play because the Haskell compiler will check that it is compatible with the actual type determined, and if it isn't, it will refuse to compile the program. Thus, the theorem is proven if and only if the program successfully compiles.

In the proof of $p \to p$ in a Hilbert system, the first step is to assert that

$$(p \to (p \to p) \to p) \to (p \to p \to p) \to p \to p$$

is an axiom, because it is an instance of the axiom scheme $(p \to q \to r) \to (p \to q) \to p \to r$. In Haskell, we could make this assertion explicit by defining a new function with the type signature `(a -> (a -> a) -> a) -> (a -> a -> a) -> a -> a`, but making the definition of this function just delegate to `s`:

```
s1 :: (a -> (a -> a) -> a) -> (a -> a -> a) -> a -> a
s1 = s
```

This would make the compiler check that the polymorphic function `s` can indeed be instantiated with this type. It isn't absolutely necessary, though; we can just use `s` itself in place of `s1`, without requiring the compiler to verify the intermediate step. In fact, in the program I wrote above I have not asked the compiler to verify any of the intermediate steps, only the final type signature of `a -> a`, because the program is nice and concise that way.

The next step in the proof of $p \to p$, in its most detailed version, would be to assert that $p \to (p \to p) \to p$ is a valid instance of the axiom scheme $p \to q \to p$. This can be done in Haskell in much the same way, by defining a function `k1` which is the same as `k` but has a more specific type signature.

The next step after that is to apply the *modus ponens* rule of inference to these two axioms, proving the formula $(p \to p \to p) \to p \to p$. What is the counterpart

of *modus ponens* in combinatory logic? It's simply function application, because if `x` is a function of type `a -> b` and `y` is a value of type `a`, then type of the value `x y` returned by `x`, given `y` as its argument, is `b`. So the Haskell way of expressing this step is as follows:

```
i0 :: (a -> a -> a) -> a -> a
i0 = s1 k1
```

(Note that `s1` and `k1` can be replaced with `s` and `k` here, if they were not defined earlier.)

The final step is to apply *modus ponens* to the formula $p \to p \to p) \to p \to p$ (the one we just proved) and the axiom $p \to p \to p$. This amounts to applying the `i0` function to `k` (of course we could also define a `k2` function with the specific type signature `a -> a -> a`, and use it in place of `k`, as with the other two axioms used). If we compress the whole proof into one function, we get the `i` function in the program above:

```
i :: a -> a
i = s k k
```

And if you're wondering about the last two lines in the original code snippet, they're just required in order to make the program a valid Haskell program that will compile.

There's a lot more to say about this subject, such as the role of intuitionistic vs. classical logic, and the relationship between quantifiers and dependent types . . . however, I still don't understand these deeper aspects of the subject very well, so I won't say any more in this post.