When we added kinding last time, part of the motivation was to complement the *functions from types to terms* that we had in System F with *functions from types to types*. Of course, all of these languages have plain functions from terms to terms. So it's natural to wonder what would happen if we added *functions from values to types*.

The result is a language with "compile-time" types that can depend on "run-time" values. While this arrangement might seem paradoxical, it is a very powerful way to express the correctness of programs; and via the propositions-as-types principle, it also serves as the foundation for a modern crop of interactive theorem provers. The language feature is called *dependent types* (i.e., types *depend* on terms).

Prominent dependently-typed languages include Coq, Nuprl, Agda, Lean, F*, and Idris. Some of these languages, like Coq and Nuprl, are more oriented toward proving things using propositions as types, and others, like F* and Idris, are more oriented toward writing "normal programs" with strong correctness guarantees.

1 Typing Lists with Lengths

Dependent types can help avoid out-of-bounds errors by encoding the lengths of arrays as part of their type. Consider a plain recursive IList type that represents a list of any length. Using type operators, we might use a general type List that can be instantiated as List int, so its kind would be type ⇒ type. But let's use a fixed element type for now. With dependent types, however, we can make IList a type constructor that takes a natural number as an argument, so IList n is a list of length n. This way, IList has the kind nat ⇒ type.

Here are the *non*-dependent types of some values from a simple library for working with lists:

$$\begin{array}{ll} \mathsf{nil} & : \mathsf{IList} \\ \mathsf{cons} & : \mathsf{int} \to \mathsf{IList} \to \mathsf{IList} \\ \mathsf{hd} & : \mathsf{IList} \to \mathsf{int} \\ \mathsf{tl} & : \mathsf{IList} \to \mathsf{IList} \\ \mathsf{isnil} & : \mathsf{IList} \to \mathsf{bool} \end{array}$$

In the dependently typed version of this little list library, we need to provide a length argument everywhere we use the IList constructor. For example, we'll want to give nil a type that indicates that it's an empty list:

$$\mathsf{nil} \ : \ \mathsf{IList}\ 0$$

The functions cons, hd, and tl need to accept lists of *any* length as arguments. To make this work, we'll turn each type into a value-to-type function. Each type will take a natural number n as an argument and produce a type that uses n somewhere in its definition.

These value-to-type functions will be written using a λ-like variable binding syntax: a type of the form $\Pi x : \tau_1 . \tau_2$ takes a value x of type τ_1 and produces the type τ_2, which may reference x. For example, $\Pi n : \mathsf{nat}. \ldots$ is a dependent type abstraction that takes a natural number as an argument, as all our list-function types will.

Here is the dependent type for the cons function. It takes an integer and a list of length n and produces a list of length $n + 1$:

$$\mathsf{cons} \ : \ \Pi n : \mathsf{nat}.\, \mathsf{int} \to (\mathsf{IList}\ n) \to (\mathsf{IList}\ (\mathsf{succ}\ n))$$

This type asks the client code for a number n and then requires that they pass in a value of type IList n. Then, it uses our successor function on natural numbers, succ, to produce the result type.

The hd and tl are a little different. In a non-dependently-typed version, they would break at run time if we passed them the empty list, nil. Now, though, we want the type to ensure that they can only take a list of length at least 1. To do this, we'll again make the types functions of a natural number argument—which may be any natural number, including 0. Then, we'll require the input list to have length succ n. The types are:

$$\begin{aligned} \mathsf{hd} &: \Pi n : \mathsf{nat}.\, (\mathsf{IList}\ (\mathsf{succ}\ n)) \to \mathsf{int} \\ \mathsf{tl} &: \Pi n : \mathsf{nat}.\, (\mathsf{IList}\ (\mathsf{succ}\ n)) \to (\mathsf{IList}\ n) \end{aligned}$$

The last function in our library, isnil, is no longer necessary. We don't need to check the length of lists at run time—i.e., using a term-to-term function—because it's always available in the list's type.

2 Using and Checking Dependent Types

Using this dependently-typed list library can feel restrictive because you need to annotate every list with a static length, but it is also very powerful: the type of your program describes the "shape" of its interactions with lists. For example, here's how you would write a function that adds together the first two elements in a list:

$$\begin{aligned} &\lambda n : \mathsf{nat}.\, \lambda l : \mathsf{IList}\ (\mathsf{succ}\ (\mathsf{succ}\ n)). \\ &\quad (\mathsf{hd}\ (\mathsf{succ}\ n)\ l) + \\ &\quad (\mathsf{hd}\ n\ (\mathsf{tl}\ (\mathsf{succ}\ n)\ l)) \end{aligned}$$

This function first takes a natural number n, and then it takes a list of length $n+2$—i.e., it requires that the list must be of length 2 or more. Then, the function uses n in its calls to hd and tl as "evidence" that the operations will succeed. The expression tl (succ n) l, according to the library signature above, has the type IList (succ n), indicating that the resulting list still has at least one element in it. The type of hd requires that the program provide n again as an argument to get the head of that list. There is no well-typed expression tl m_1 (tl m_2 (tl m_3 l)) with any three numbers $m_{1\ldots3}$ that can get the tail of the input list three times.

Our example function's type indicates that it only works on lists of length 2 or more. The types for more complicated functions can also say things about the kinds of lists they produce. For example, we might concoct sorting or reversing functions on lists, which should return lists of the same length as the input, so they should have this type:

$$\Pi n : \mathsf{nat}.\, (\mathsf{IList}\ n) \to (\mathsf{IList}\ n)$$

To check that the functions sort or reverse have this type, you have to prove that they don't alter the length of their input lists. In fact, you can even imagine using more complex types to require that sort produces a sorted list: i.e., that every list's head is less than or equal to all of the elements in its tail. This means that checking the types in a dependently typed language can require proving arbitrary correctness theorems.

Imbuing the type system with the power of proofs is clearly a powerful tool for writing correct programs, but it comes at the cost of decidability. Because it entails proving arbitrary theorems, checking the types of expressions cannot be done automatically in dependently typed languages. Therefore, these languages tend to have features for helping you define strategies—called *tactics*—for searching for proof terms.

Recall that our little program above passed around a number n as "evidence" that the operations hd and tl were safe to perform. That number never actually affected the operation of hd and tl; it just worked as a constraint on the program to prevent it from doing anything bad at run time. This pattern, where functions take an extra parameter that has nothing to do do with run-time operation but is required to prove the program correct, is essential in dependently typed programming. To actually run these programs, you usually use a process called *extraction* to erase computationally irrelevant terms and translate into a language without dependent types—often, OCaml.

3 Propositions as Dependent Types

Dependent types are particularly useful through the Curry–Howard lens, where terms are seen as proofs of the theorems encoded by their types. For example, say you want to provide an indexing function on lists, called get. It would be nice to ensure, as with hd and tl, that the function doesn't go out of bounds when accessing the list. So if get takes a list of type IList n and an index m, its type should somehow ensure that $m < n$. There's no obvious way to use succ to enforce this requirement as we did in the types of hd and tl.

Instead, we want to require a *proof* that $m < n$ as an argument to the get function. Then, the get function can use this proof to construct other proofs in its body to ensure that its successive tl calls are safe. The idea is to define a new type Less m n that is inhabited if and only if $m < n$. A term of this type serves as a proof of the inequality. With this in hand, we can write the type of get:

$$\mathsf{get} \ : \ \Pi m : \mathsf{nat}.\, \Pi n : \mathsf{nat}.\, (\mathsf{Less}\ m\ n) \to (\mathsf{IList}\ n) \to \mathsf{int}$$

This function takes m and n as arguments as well as a proof that $m < n$ before doing its work on a list of length n.

But how do we define Less itself? The idea in dependently typed languages is to let you create opaque values of the types you declare. This lets you "call into existence" the elementary building blocks of proofs that you will use to build up larger theorems. For example, we might invent a value called Adjacent that serves as a proof that any number m is less than $m + 1$:

$$\mathsf{Adjacent} \ : \ \Pi m : \mathsf{nat}.\, \mathsf{Less}\ m\ (\mathsf{succ}\ m)$$

Adjacent has the kind nat ⇒ type. If you remember that types are propositions, then you can also think of it as a function from numbers to proofs. The value Adjacent 4 serves as a proof that $4 < 5$, i.e., it has the type Less 4 5.

To finish off a working definition of Less, we'll want an inductive case. Let's call it Inductive, uncreatively:

$$\mathsf{Inductive} \ : \ \Pi m : \mathsf{nat}.\, \Pi n : \mathsf{nat}.\, (\mathsf{Less}\ m\ n) \to (\mathsf{Less}\ m\ (\mathsf{succ}\ n))$$

This signature says that, given m and n and a proof that $m < n$, you can also construct a proof that $m < n + 1$.

Given these definitions, you can now write this term:

$$\mathsf{Inductive}\ 5\ 7\ (\mathsf{Inductive}\ 5\ 6\ (\mathsf{Adjacent}\ 5))$$

which has the type Less 5 8. In other words, it is a proof that $5 < 8$.

The dependently typed languages that focus on "real programming," like Idris, use this terms-as-proofs correspondence to guarantee that computational functions like get cannot go wrong. Other languages, like Coq, Nuprl, and Lean, focus on proving theorems instead, and this capability is their *raison d'être*. They offer special features to help you search for inhabitants of complicated types—i.e., to search for proofs of theorems.

4 The Successes of Dependent Types

Theorem-proving systems based on dependent types have seen an explosion of interest over the last decade or so. They have been used to produce provably correct implementations of frighteningly complex software. For example, CompCert [1] is a C compiler that proves that any assembly code it produces has the same semantics as the input C program. Verdi [3] has proven the correctness of a variety of distributed systems,

including their fault tolerance properties. Notably, the Verdi authors have a mechanized a proof of the Raft [2] consensus algorithm. These are examples of a broad category of work: the programming languages and systems conferences in the last few years have been filled with papers that use dependently typed languages to prove ever more sophisticated systems formally correct.

References

[1] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[2] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014.

[3] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Programming Language Design and Implementation (PLDI)*, 2015.