Lecture Notes on Data Representation

15-814: Types and Programming Languages
Frank Pfenning

Lecture 9
Tuesday, October 2, 2018

1 Introduction

In this lecture we'll see our type system in action. In particular we will see how types enable and guide data representation. We first look at a traditional problem (representing numbers in binary form) then at a less traditional one (representing the untyped λ-calculus). Before that, we'll review recursive types and their properties, since they play a central role in what follows.

2 Natural Numbers, Revisited

Recall that we were thinking of natural numbers as the type

$$\mathit{nat} = 1 + (1 + (1 + \ldots))$$

which doesn't seem directly implementable. Instead, we noticed that under the approach we have

$$\mathit{nat} \text{ ``='' } 1 + \mathit{nat}$$

where the notion of equality between these two types was a bit murky. So we devised an explicit construction $\rho\,\alpha.\,\tau$ to form a recursive type of this nature.

$$\mathit{nat} = \rho\,\alpha.\,1 + \alpha$$

The *constructor* for elements of recursive types is fold, while unfold *destructs* elements.

$$\frac{\Gamma \vdash e : [\rho\,\alpha.\,\tau/\alpha]\tau}{\Gamma \vdash \mathsf{fold}\, e : \rho\,\alpha.\,\tau} \qquad \frac{\Gamma \vdash e : \rho\,\alpha.\,\tau}{\Gamma \vdash \mathsf{unfold}\, e : [\rho\,\alpha.\,\tau/\alpha]\tau}$$

This "unfolding" of the recursion seems like a strange operation, and it is. For example, for all other data constructors the components have a smaller type than the constructed expression, but that's not the case here because $[\rho\,\alpha.\,\tau/\alpha]\tau$ is in general a larger type than $\rho\,\alpha.\,\tau$. To get more intuition, let's look at the special case of these rules for natural numbers. We exploit the definition of *nat* in order to avoid explicitly use of the ρ binder and substitution.

$$[\rho\,\alpha.\,1 + \alpha/\alpha](1 + \alpha) = 1 + \mathit{nat}$$

With this shortcut, the specialized rules are

$$\frac{\Gamma \vdash e : 1 + \mathit{nat}}{\Gamma \vdash \mathsf{fold}\, e : \mathit{nat}} \qquad \frac{\Gamma \vdash e : \mathit{nat}}{\Gamma \vdash \mathsf{unfold}\, e : 1 + \mathit{nat}}$$

When recursive types are given names (which is usually the case), this technique makes it much easier to see how the fold and unfold operations actually work.

The funky equality from the beginning of the lecture is actually an *isomorphism*, that is,

$$\mathit{nat} \cong 1 + \mathit{nat}$$

In fact, the functions going back and forth are exactly fold and unfold.

$$\mathit{nat} \underset{\text{unfold}}{\overset{\text{fold}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\cong}}}} 1 + \mathit{nat}$$

We can (re)write simple programs. As we did in lecture, you should write these programs following the structure of the type; here we just show the final code.

$$\begin{array}{lcl} \mathit{zero} : \mathit{nat} & = & \mathsf{fold}\,(l \cdot \langle\,\rangle) \\ \mathit{succ} : \mathit{nat} \to \mathit{nat} & = & \lambda n.\, \mathsf{fold}\,(r \cdot n) \end{array}$$

In order to check the isomorphism, we need to show that the functions compose to the identity in both directions. That is:

(i) For every value $v : 1 + \mathit{nat}$, $\mathsf{unfold}\,(\mathsf{fold}\, v) = v$, and

(ii) for every value v : *nat*, $\mathsf{fold}\,(\mathsf{unfold}\,v) = v$

Before we can prove this, we should write down the definition of values and the operational semantics. The constructor is fold, and we had decided to make it eager, that is

$$\frac{e\ \mathsf{val}}{\mathsf{fold}\,e\ \mathsf{val}}$$

The destructor is unfold, so it acts on a value of the expect form, namely a fold.

$$\frac{v\ \mathsf{val}}{\mathsf{unfold}\,(\mathsf{fold}\,v) \mapsto v}$$

Finally, we have congruence rules: for the constructor because it is eager, and for the destructor because we need to reduce the argument until it exposes the constructor.

$$\frac{e \mapsto e'}{\mathsf{fold}\,e \mapsto \mathsf{fold}\,e'} \qquad \frac{e \mapsto e'}{\mathsf{unfold}\,e \mapsto \mathsf{unfold}\,e'}$$

Back to our putative isomorphism. The first direction is almost trivial, since we can directly step.

(i) $\mathsf{unfold}\,(\mathsf{fold}\,v) \mapsto v$ since v val.

The second part is slightly more complex

(ii) We want to show that $\mathsf{fold}\,(\mathsf{unfold}\,v) = v$ for any value v : *nat*. The left-hand side does not appear to reduce, because fold is the constructor. However, because v : *nat* is a value we know it must have the form $\mathsf{fold}\ v'$ for a value v' (by the canonical forms theorem, see below) and then we reason:

$$\begin{array}{lll} & \mathsf{fold}\,(\mathsf{unfold}\,v) & \\ = & \mathsf{fold}\,(\mathsf{unfold}\,(\mathsf{fold}\ v')) & \text{since } v = \mathsf{fold}\,v' \\ \mapsto & \mathsf{fold}\,v' & \text{by computation rule} \\ = & v & \text{since } v = \mathsf{fold}\,v' \end{array}$$

Before stating the canonical form theorem it is worth realizing that properties (i) and (ii) actually do not depend on the particular recursive type *nat* but hold for any recursive type $\rho\,\alpha.\,\tau$. This means that we have in general

$$\rho\,\alpha.\,\tau \underset{\mathsf{unfold}}{\overset{\mathsf{fold}}{\cong}} [\rho\,\alpha.\,\tau/\alpha]\rho$$

This is why we call types in this form *isorecursive*. There is a different form called *equirecursive* which attempts to get by without explicit fold and unfold constructs. Programs become more succinct, but type-checking easily becomes undecidable or impractical, depending on the details of the language. We therefore take the more explicit isorecursive approach here.

Theorem 1 (Canonical forms for recursive types)
If $\cdot \vdash v : \rho\,\alpha.\,\tau$ *and* v *val then* $v = \mathsf{fold}\ v'$ *for some* v' *val.*

Proof: By case analysis of values and typing rules. □

3 Representing Binary Numbers

Natural numbers in unary form are an elegant foundational representation, but the size of the representation of n is linear in n. We can do much better if we have a *binary* representation with two bits. A binary number then is a finite string of bits, satisfying something like

$$\mathit{bin} \cong \mathit{bin} + \mathit{bin} + 1$$

where the first summand represents a bit 0, the second a bit 1, and the last the empty string of bits. Code is easier to write if we use the n-ary form of the sum where each alternative is explicitly labeled.

$$\mathit{bin} \cong (\mathsf{b0} : \mathit{bin}) + (\mathsf{b1} : \mathit{bin}) + (\epsilon : 1)$$

Here we have used the labels b0 (for a 0 bit), b1 (for a 1 bit), and ϵ (for the empty bit string).

Now it is convenient (but not necessary) to represent 0 by the empty bit string.

$$\mathit{bzero} : \mathit{bin} \quad = \quad \mathsf{fold}\,(\epsilon \cdot \langle\,\rangle)$$

We can also construct larger numbers from smaller ones by adding a bit at the end. For the purposes of writing programs, it is most convenient to represent numbers in "little endian" form, that is, the least significant bit comes first. The two constructors then either double the number n to $2n$ (if we add bit 0) or $2n + 1$ if we add bit 1.

$$\begin{array}{rcl} \mathit{dbl0} : \mathit{bin} \to \mathit{bin} & = & \lambda x.\,\mathsf{fold}\,(\mathsf{b0} \cdot x) \\ \mathit{dbl1} : \mathit{bin} \to \mathit{bin} & = & \lambda x.\,\mathsf{fold}\,(\mathsf{b1} \cdot x) \end{array}$$

As a sample program that must analyze the structure of numbers in binary form, consider a function to increment a number. In order to analyze the

argument of type *bin* we must first unfold its represenation to a sum and then case over the possible summands. There are three possibilities, so our code so far has the form

$$
\begin{array}{l}
\mathit{inc} : \mathit{bin} \to \mathit{bin} = \\
\quad \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\, x) \\
\qquad\qquad \{\ \mathsf{b0} \cdot y \Rightarrow \ldots \\
\qquad\qquad \mid \mathsf{b1} \cdot y \Rightarrow \ldots \\
\qquad\qquad \mid \epsilon \cdot y \Rightarrow \ldots\ \}
\end{array}
$$

In each branch, the missing code should have type *bin*. In the case of $\mathsf{b0} \cdot y$ we just need to flip the lowest bit from b0 to b1 and keep the rest of the bit string the same.

$$
\begin{array}{l}
\mathit{inc} : \mathit{bin} \to \mathit{bin} = \\
\quad \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\, x) \\
\qquad\qquad \{\ \mathsf{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b1} \cdot y) \\
\qquad\qquad \mid \mathsf{b1} \cdot y \Rightarrow \ldots \\
\qquad\qquad \mid \epsilon \cdot y \Rightarrow \ldots\ \}
\end{array}
$$

In the second branch, we need to flip b1 to b0, and we also need to implement the "carry", which means that we have to *increment* the remaining higher-order bits.

$$
\begin{array}{l}
\mathit{inc} : \mathit{bin} \to \mathit{bin} = \\
\quad \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\, x) \\
\qquad\qquad \{\ \mathsf{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b1} \cdot y) \\
\qquad\qquad \mid \mathsf{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b0} \cdot (\mathit{inc}\ y)) \\
\qquad\qquad \mid \epsilon \cdot y \Rightarrow \ldots\ \}
\end{array}
$$

Finally, in the last case we need to return the representation of the number 1, because $\mathsf{fold}\ (\epsilon \cdot \langle\,\rangle)$ represents 0. We obtain it from the the representation of 0 (which we called *bzero*) by adding a bit 1.

$$
\begin{array}{l}
\mathit{inc} : \mathit{bin} \to \mathit{bin} = \\
\quad \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\, x) \\
\qquad\qquad \{\ \mathsf{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b1} \cdot y) \\
\qquad\qquad \mid \mathsf{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b0} \cdot (\mathit{inc}\ y)) \\
\qquad\qquad \mid \epsilon \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b1} \cdot \mathit{bzero})\ \}
\end{array}
$$

In the last branch, $y : 1$ and it is unused. As suggest in lecture, we could have written instead

$$
\mid \epsilon \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b1} \cdot \mathsf{fold}\ (\epsilon \cdot y))
$$

In this program we largely reduced the operations back to fold and explicitly labeled sums, but we could have also used the *dbl0* and *dbl1* functions.

At this point we have seen all the pieces we need to implement addition, multiplication, subtraction, etc. on the numbers in binary form.

4 Representing the Untyped λ-Calculus

Recall that in the pure, untyped lambda calculus we only have three forms of expression: λ-abstraction $\lambda x.\, e$, application $e_1\, e_2$ and variables x. A completely straightforward representation would be given by the following recursive type:

$$\begin{array}{lcl} \mathit{var} & \cong & \mathit{nat} \\ \mathit{exp} & \cong & (\mathsf{lam} : \mathit{var} \otimes \mathit{exp}) + (\mathsf{app} : \mathit{exp} \otimes \mathit{exp}) + (\mathsf{v} : \mathit{var}) \end{array}$$

Here we have chosen variables to be represented by natural numbers because we need unboundedly many different ones.

This representation is fine, but it turns out to be somewhat awkward to work with. One issue is that we have already said that $\lambda x.\, x$ and $\lambda y.\, y$ should be indistinguishable, but in the representation above they are (for example, x might be the number 35 and y the number 36.

In order to solve this problem, de Bruijn [dB72] developed a representation where we cannot distinguish these two terms. It is based on the idea that a variable occurrence should be a *pointer* back to the place where it is bound. A convenient representation for such a pointer is a natural number that indicates how many binders we have to traverse upwards to reach the appropriate λ-abstraction. For example:

$$\begin{array}{lcl} \lambda x.\, x & \sim & \lambda.0 \\ \lambda y.\, y & \sim & \lambda.0 \\ \lambda x.\, \lambda y.\, x & \sim & \lambda.\lambda.1 \\ \lambda x.\, \lambda y.\, y & \sim & \lambda.\lambda.0 \end{array}$$

For free variables, we have to assume they are ordered in some context and the variables refers to them, counting from right to left. For example:

$$\begin{array}{lcl} y, z \vdash \lambda x.\, x & \sim & \lambda.0 \\ y, z \vdash \lambda x.\, y & \sim & \lambda.2 \\ y, z \vdash \lambda x.\, z & \sim & \lambda.1 \end{array}$$

One strange effect of this representation (which we did not mention in lecture) is that in de Bruijn notation, the same variable may occur with

different numbers in an expression. For example

$$\lambda x.\,(\lambda y.\,x\,y)\,x \quad \sim \quad \lambda.(\lambda.1\,0)\,0$$

The first occurrence of x becomes 1 because it is located under another binder (that for y), while the second occurrence of x becomes 0 because it is not in the scope of the binder on y.

There are some clever algorithms for implementing operations such as substitution on this representation. However, we will move on to an even cooler representation.

5 A Shallow Embedding of the Untyped λ-Calculus

The standard representations we have seen so far are sometimes called *deep embeddings*: objects we are trying to represent simply become "lifeless" data. Any operation on them (as would usually be expected) has to be implemented explicitly and separately.

A *shallow embedding* tries to exploit the features present in the host language (here: our statically typed functional language) as directly as possible. In shallow embeddings mostly we represent only the constructors (or values) and try to implement the destructors. In the case of the untyped λ-calculus, the only constructor is a λ-abstraction so a shallow embedding would postulate

$$\mathsf{E} \underset{\text{unfold}}{\overset{\text{fold}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\cong}}}} \mathsf{E} \to \mathsf{E}$$

At first it seems implausible that a type E would be isomorphic to its own function space, but surprisingly we can make it work! In the different context of *denotational semantics* this isomorphism was first solved by Dana Scott [Sco70]. Let's work out the representation function $\ulcorner e \urcorner$ where e is an expression in the untyped λ-calculus. We start with some examples.

$$\ulcorner \lambda x.\,x \urcorner = \underbrace{\ldots}_{:\,\mathsf{E}}$$

We want the representation to be of type E. Since the left-hand side represents a λ-expression, it should be the result of a fold. A fold requires an argument of type $\mathsf{E} \to \mathsf{E}$

$$\ulcorner \lambda x.\,x \urcorner = \mathsf{fold}\ \underbrace{\ldots}_{:\,\mathsf{E}\to\mathsf{E}}$$

That should be a λ-expression in the host language, which binds a variable x of type E. The body of the expression is again of E.

$$\ulcorner \lambda x.\, x \urcorner = \mathsf{fold}\ (\lambda x.\ \underbrace{\ldots}_{:\,\mathsf{E}})$$

Because we want to represent the identity function, we finish with

$$\ulcorner \lambda x.\, x \urcorner = \mathsf{fold}\ (\lambda x.\, x)$$

The following two examples work similarly:

$$\ulcorner \lambda x.\, \lambda y.\, x \urcorner = \mathsf{fold}\ (\lambda x.\, \mathsf{fold}\ (\lambda y.\, x))$$
$$\ulcorner \lambda x.\, \lambda y.\, y \urcorner = \mathsf{fold}\ (\lambda x.\, \mathsf{fold}\ (\lambda y.\, y))$$

The first hurdle arises when we try to represent application. Let's consider something that might be difficult, namely self-application.

$$\omega = \lambda x.\, x\, x$$

Note that this expression itself cannot in the host language. If there were a typing derivation, it would have to look as follows for some τ, σ, and τ':

$$\dfrac{\dfrac{x : \tau \vdash x : \tau' \to \sigma \quad x : \tau \vdash x : \tau'}{x : \tau \vdash x\, x : \sigma}}{\cdot \vdash \lambda x.\, x\, x : \tau \to \sigma}$$

To complete the derivations, we would have to have simultaneously

$$\tau = \tau' \to \sigma \quad \text{and} \quad \tau = \tau'$$

and there is no solution, because

$$\tau' = \tau' \to \sigma$$

has no solution. Therefore, ω cannot be typed in the simply-typed λ-calculus, even though it is a perfectly honorable untyped term. The key now is the following general table of representations

$$\begin{array}{lcl} \ulcorner \lambda x.\, e \urcorner & = & \mathsf{fold}\,(\lambda x.\, \ulcorner e \urcorner) \\ \ulcorner x \urcorner & = & x \\ \ulcorner e_1\, e_2 \urcorner & = & \underbrace{(\mathsf{unfold}\, \ulcorner e_1 \urcorner)}_{:\,\mathsf{E} \to \mathsf{E}}\ \underbrace{\ulcorner e_2 \urcorner}_{:\,\mathsf{E}} \end{array}$$

To summarize, λ-abstraction becomes a fold, application becomes an unfold, and a variable is represented by a corresponding variable with (for convenience) the same name.

To get back to self-application, we obtain

$$\ulcorner\omega\urcorner = \ulcorner\lambda x.\, x\, x\urcorner = \mathsf{fold}\,(\lambda x.\,(\mathsf{unfold}\,x)\,x) : \mathsf{E}$$

Recall that $\Omega = \omega\,\omega = (\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$ has no normal form in the untyped λ-calculus in the sense that it only reduces to itself. We would expect the representation to diverge as well. Let's check:

$$\begin{array}{lll}
 & \ulcorner\omega\,\omega\urcorner & \\
= & (\mathsf{unfold}\,\ulcorner\omega\urcorner)\,\ulcorner\omega\urcorner & \\
= & (\mathsf{unfold}\,(\mathsf{fold}\,(\lambda x.\,(\mathsf{unfold}\,x)\,x)))\,\ulcorner\omega\urcorner & \\
\mapsto & (\lambda x.\,(\mathsf{unfold}\,x)\,x)\,\ulcorner\omega\urcorner & \text{since } (\lambda x.\,(\mathsf{unfold}\,x)\,x)\ \textit{val} \\
\mapsto & (\mathsf{unfold}\,\ulcorner\omega\urcorner)\,\ulcorner\omega\urcorner & \text{since } \ulcorner\omega\urcorner\ \textit{val} \\
= & \ulcorner\omega\,\omega\urcorner &
\end{array}$$

We can see that the representation of Ω also steps to itself, but now in two steps instead of one. That's because the fold/unfold reduction requires one additional step.

We haven't proved this, but without a fixed point constructor for programs ($\mathsf{fix}\,x.\,e$) and without recursive types, every expression in our language reduces to a value. This example demonstrates that this is no longer true in the presence of recursive types. Note that we did not need the fixed point constructor—just the single recursive type $\mathsf{E} = \rho\,\alpha.\,\alpha \to \alpha$ was sufficient.

6 Untyped is Unityped

In the previous section we have seen that there is a compositional embedding of the untyped λ-calculus in our simply-typed language with recursive types. This demonstrates that we don't lose any expressive power by moving to a typed language, as long as we are prepared to accept recursive types. In fact, the whole untyped language is mapped to a *single type* in our host language, so we summarize this by saying that

The untyped λ-calculus is unityped.

It is important to see that the typed language is in fact a *generalization of the untyped language* rather than the other way around. By using fold and

unfold we can still express all untyped programs. In the next lecture we will explore this a little bit further to talk about *dynamic typing* and that the observation made in this lecture generalizes to richer settings.

Beyond typing there is one more difference between the untyped λ-calculus and our typed representation that we should not lose sight of. The meaning of an untyped λ-expression is given by its *normal form*, which means we can reduce any subexpression including under λ-abstractions. On the other hand, in the functional host language we do not evaluate under λ-abstractions or lazy pairs. For example, $\lambda z.\, \Omega$ has no normal form, but its representation $\ulcorner \lambda x.\, \Omega \urcorner = \mathsf{fold}\, (\lambda x.\, \ulcorner \Omega \urcorner)$ is a value. So we have to be careful when reasoning about the *operational behavior* of the embedding, which is true for all shallow embeddings.

References

[dB72] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[Sco70] Dana S. Scott. Outline of a mathematical theory of computation. Technical Monograph PRG-2, Oxford University Computing Laboratory, Oxford, England, November 1970.