Lecture Notes on
From λ-Calculus to Programming Languages

15-814: Types and Programming Languages
Frank Pfenning

Lecture 7
Tuesday, September 21, 2021

1 Introduction

First, we will briefly talk about the dynamic of polymorphism (which abstracts over types and applies functions to types), and then exercise polymorphism a little to generalizing iteration from natural numbers to richer types, using trees as an example.

Then we take the a big step from a pure λ-calculus to real programming languages by changing our attitude on data: we would like to represent them directly instead of indirectly as functions, for several reasons explained in Section 4.

2 Dynamics of Polymorphism*

The material from this section was not covered in lecture, but is included here, anticipating its introduction later in the course.

We already gave the typing rules for parametric polymorphism in the previous lecture, but we did not yet update the rules for computation or normal and neutral terms. A key observation is that the structure of the types in our little language is such that we should be able to just add new rules without touching the old ones in any way. This form of modularity also carries over to the proofs of the key properties we would like the system to have: they decompose into cases along the lines of the type constructs we have.

First, reduction:

$$\frac{}{(\Lambda\alpha.\, e)\,[\tau] \longrightarrow [\tau/\alpha]e}\ \textsf{red/tpbeta}$$

$$\frac{e \longrightarrow e'}{\Lambda\alpha.\, e \longrightarrow \Lambda\alpha.\, e'}\ \textsf{red/tplam} \qquad \frac{e \longrightarrow e'}{e\,[\tau] \longrightarrow e'\,[\tau]}\ \textsf{red/tpapp}_1$$

There is no $\textsf{red/tpapp}_2$ rule since we do not reduce types themselves.

In this definition we use substitution $[\tau/\alpha]e$, which is defined in the expected way, possibly renaming type variables bound by $\Lambda\beta.\,\sigma$ or $\forall\beta.\,sigma$ that may occur in e so as to avoid capturing any type variables free in τ.

There are also two new rules for normal and neutral terms, retaining all the others.

$$\frac{e\ \textit{normal}}{\Lambda\alpha.\, e\ \textit{normal}}\ \textsf{norm/lam} \qquad \frac{e\ \textit{neutral}}{e\,[\tau]\ \textit{neutral}}\ \textsf{neut/app}$$

The key theorems are *preservation* and *progress*, establishing a connection between types, reduction, and normal forms.

Preservation. If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : \tau$

Progress. If $\Gamma \vdash e : \tau$ then either $e \longrightarrow e'$ for some e' or e *normal*.

Finality of Normal Forms. There is no $\Gamma \vdash e : \tau$ such that $e \longrightarrow e'$ for some e' and e *normal*.

3 Generalizing Iteration

It may be helpful to think of iteration on natural numbers to arise from they way they are constructed

$$\begin{array}{lcl} \textsf{zero} & : & \textit{nat} \\ \textsf{succ} & : & \textit{nat} \rightarrow \textit{nat} \end{array}$$

Namely, if we imagine a term

$$\textsf{succ}\,(\textsf{succ}\ \ldots\ (\textsf{succ}\,\textsf{zero})) : \textit{nat}$$

then we *replace* the constructor by appropriate functions and constants (using g for succ and c for zero

$$g\,(g\ \ldots,(g\,c))$$

Now we should work out the types of g and c. Clearly, $g : \tau \to \tau$ for any type τ and $c : \tau$. We can obtain these types from the type of zero and succ by replacing *nat* with τ. So, if we want to see $n : \mathit{nat}$ as an *iterator* then

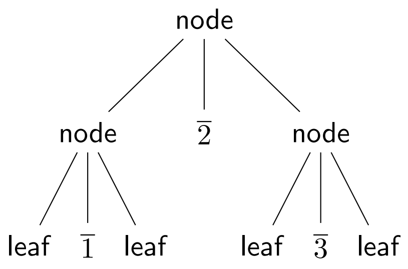
$$\mathit{nat} = \forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$$

where the first argument is the result type τ following by a function $g : \tau \to \tau$ and a constant $c : \tau$.

Let's follow the same recipe for trees of natural numbers. They are generated from

$$\begin{array}{lcl} \mathsf{node} & : & \mathit{tree} \to \mathit{nat} \to \mathit{tree} \to \mathit{tree} \\ \mathsf{leaf} & : & \mathit{tree} \end{array}$$

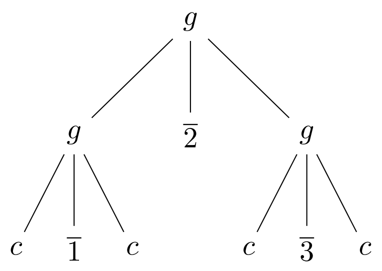
In this representation, leaves carry no information and every interior node has a left subtree, a natural number, and right subtree. For example, the tree



would be constructed with

$$\mathsf{node}\ (\mathsf{node}\ \mathsf{leaf}\ \overline{1}\ \mathsf{leaf})\ \overline{2}\ (\mathsf{node}\ \mathsf{leaf}\ \overline{3}\ \mathsf{leaf})$$

To see the form of an iterator we replace the constructors node and leaf with functions g and a constant c, respectively, which would give use the tree



This time, we see that we should have

$$\begin{array}{lcl} g & : & \tau \rightarrow \mathit{nat} \rightarrow \tau \rightarrow \tau \\ c & : & \tau \end{array}$$

for an arbitrary type τ. Once again, this was obtained from replacing the type *tree* in the types of node and leaf with an arbitrary type. We can express this as a polymorphic type as:

$$\mathit{tree} = \forall\alpha.\, (\alpha \rightarrow \mathit{nat} \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$$

As an example, to sum up the elements of the tree we would define

$$\begin{array}{lcl} \mathit{sum} & : & \mathit{tree} \rightarrow \mathit{nat} \\ \mathit{sum} & = & \lambda t.\, t\ [\mathit{nat}]\ (\lambda x.\, \lambda n.\, \lambda y.\, \mathit{plus}\ x\ (\mathit{plus}\ n\ y))\ \mathit{zero} \end{array}$$

First, we pass to t the result type *nat*, then a function g expecting the sum of the left subtree as x, then n as the value stored in the node, and then the sum of the right subtree as y. The function g then just has to add these three numbers to obtain the sum of a tree. Since the leaf does not contain any number, its value is 0 (the neutral element of addition).

The definition of the tree constructors themselves follow the structure of the type. The easy case first:

$$\begin{array}{lcl} \mathsf{leaf} & : & \mathit{tree} \\ \mathsf{leaf} & = & \Lambda\alpha.\, \lambda n.\, \lambda l.\, l \end{array}$$

For the node constructor we have the parameter n at the head of the term, and we just have to remember to match the types by applying the representations of the left and right subtrees to all parameters (including the type α).

$$\begin{array}{lcl} \mathsf{node} & : & \mathit{tree} \rightarrow \mathit{nat} \rightarrow \mathit{tree} \rightarrow \mathit{tree} \\ \mathsf{node} & = & \lambda t_1.\, \lambda x.\, \lambda t_2.\, \Lambda\alpha.\, \lambda n.\, \lambda l.\, n\ (t_1\ [\alpha]\ n\ l)\ x\ (t_2\ [\alpha]\ n\ l) \end{array}$$

We did not live-code this in lecture, but below is the code for trees in LAMBDA, which should come after the code for natural number from the last lecture. You can find this code online at tree.poly.

```
type tree = !a. (a -> nat -> a -> a) -> a -> a

decl leaf : tree
decl node : tree -> nat -> tree -> tree

defn leaf = /\a. \n. \l. l
defn node = \t1. \x. \t2. /\a. \n. \l. n (t1 [a] n l) x (t2 [a] n l)

decl sum : tree -> nat
defn sum = \t. t [nat] (\s1. \x. \s2. plus s1 (plus x s2)) zero

norm t123 = node (node leaf _1 leaf) _2 (node leaf _3 leaf)
norm s6 = sum t123
conv s6 = _6
```

Listing 1: Trees of natural numbers in LAMBDA

Other data types have similar encodings. For example, we can think of the natural numbers in binary form (type *bin*) as generated from the constructors

$\mathsf{b0}$	:	$bin \rightarrow bin$	bit 0
$\mathsf{b1}$	:	$bin \rightarrow bin$	bit 1
e	:	bin	empty bit string

where the least significant bit comes first ("little endian"). Here are some examples:

$$\begin{array}{rcl} \overline{(0)_2} & = & \mathsf{e} \\ \overline{(1)_2} & = & \mathsf{b1}\,\mathsf{e} \\ \overline{(2)_2} & = & \mathsf{b0}\,(\mathsf{b1}\,\mathsf{e}) \\ \overline{(6)_2} & = & \mathsf{b0}\,(\mathsf{b1}\,(\mathsf{b1}\,\mathsf{e})) \end{array}$$

To obtain the representation in the *untyped* λ-calculus *abstract* over all the constructors, so, for example

$$\overline{(6)_2} \quad = \quad \lambda \mathsf{b0}.\, \lambda \mathsf{b1}.\, \lambda \mathsf{e}.\, \mathsf{b0}\,(\mathsf{b1}\,(\mathsf{b1}\,\mathsf{e}))$$

To assign a type and represented a number by its iterator, we have to replace *bin* with α and quantify over all possible instances. That is

$$bin \quad = \quad \forall \alpha.\, \underbrace{(\alpha \rightarrow \alpha)}_{\mathsf{b0}} \rightarrow \underbrace{(\alpha \rightarrow \alpha)}_{\mathsf{b1}} \rightarrow \underbrace{\alpha}_{\mathsf{e}} \rightarrow \alpha$$

Iteration in this case now plugs in functions g_0 for $\mathsf{b0}$, g_1 for $\mathsf{b1}$ and c for e. You may try to define some interesting functions over this representation.

4 Evaluation versus Reduction

The λ-calculus is exceedingly elegant and minimal, a study of functions in the purest possible form. We find versions of it in most, if not all modern programming languages because the abstractions provided by functions are a central structuring mechanism for software. On the other hand, there are some problem with the data-as-functions representation technique of which we have seen Booleans, natural numbers, and trees. Here are a few notes:

Generality of typing. The untyped λ-calculus can express fixed points (and therefore all partial recursive functions on its representation of natural numbers) but the same is not true for Church's simply-typed λ-calculus or even the polymorphic λ-calculus where all well-typed expressions have a normal form. Types, however, are needed to understand and classify data representations and the functions defined over them. Fortunately, this can be fixed by introducing *recursive types*, so this is not a deeper obstacle to representing data as functions.

Expressiveness. While all *computable functions* on the natural numbers can be represented in the sense of correctly modeling their input/output behavior, some natural *algorithms* are difficult or impossible to express. For example, under some reasonable assumptions the minimum function on numbers n and k has complexity $O(\max(n, k))$ [CF98], which is surprisingly slow, and our predecessor function took $O(n)$ steps. Other representations are possible, but they either complicate typing or inflate the size of the representations.

Observability of functions. Since reduction results in normal forms, to interpret the outcome of a computation we need to be able to inspect the structure of functions. But generally we like to compile functions and think of them only as something opaque: we can probe it by applying it to arguments, but its structure should be hidden from us. This is a serious and major concern about the pure λ-calculus where all data are expressed as functions.

In the remainder of this lecture we focus on the last point: rather than representing all data as functions, we add data to the language directly, with new types and new primitives. At the same time we make the structure of functions *unobservable* so that implementation can compile them to machine code, optimize them, and manipulate them in other ways. Functions become more *extensional* in nature, characterized via their input/output behavior rather than distinguishing functions that have different internal structure.

5 Revising the Dynamics of Functions

The *statics*, that is, the typing rules for functions, do not change, but the way we compute does. We have to change our notion of reduction as well as that of normal forms. Because the difference to the λ-calculus is significant, we call the result of computation *values* and define them with the judgment e *value*. Also, we write $e \mapsto e'$ for a single step of computation. For now, we want this step relation to be *deterministic*, that is, we want to arrange the rules so that every expression either steps in a unique way or is a value. We'll call this property *sequentiality*, since it means execution is sequential rather than parallel or concurrent. Furthermore, since we do not reduce underneath λ-abstractions, we only evaluate expressions that are *closed*, that is, have *no free variables*.

When we are done, we should then check the following properties.

Preservation. If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$.

Progress. For every expression $\cdot \vdash e : \tau$ either $e \mapsto e'$ for some e' or e *value*.

Finality of Values. There is no $\cdot \vdash e : \tau$ such that $e \mapsto e'$ for some e' and e *value*.

Sequentiality. If $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$.

Devising a set of rules is usually the key activity in programming language design. Proving the required theorems is just a way of checking one's work rather than a primary activity. First, one-step computation. We suggest you carefully compare these rules to those in Lecture 4 where reduction could take place in arbitrary position of an expression.

$$\frac{}{\lambda x.\, e \; \mathit{value}}\ \mathsf{val/lam}$$

Note that e here is unconstrained and need not be a value.

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app_1} \qquad \frac{}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \mathsf{beta}$$

These two rules together constitute a strategy called *call-by-name*. There are good practical as well as foundational reasons to use *call-by-value* instead, which we obtain with the following three alternative rules.

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \textsf{step/app}_1 \qquad \frac{e_1\ \mathit{value} \quad e_2 \mapsto e_2'}{e_1\, e_2 \mapsto e_1\, e_2'}\ \textsf{step/app}_2$$

$$\frac{e_2\ \mathit{value}}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \textsf{step/app/lam}$$

We achieve sequentiality by requiring certain subexpressions to be values. Consequently, computation first reduces the function part of an application, then the argument, and then performs a (restricted form) of β-reduction.

There are a lot of spurious arguments about whether a language should support call-by-value or call-by-name. This turns out to be a false dichotomy and only historically in opposition.

We could now check our desired theorems, but we wait until we have introduced the Booleans as a new primitive type.

6 Booleans as a Primitive Type

Most, if not all, programming languages support Booleans. There are two values, true and false, and usually a conditional expression if e_1 then e_2 else e_3. From these we can define other operations such as conjunction or disjunction. Using, as before, α for type variables and x for expression variables, our language then becomes:

$$\begin{array}{llcl} \text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha.\, \tau \mid \textsf{bool} \\ \text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \mid \Lambda \alpha.\, e \mid e\, [\tau] \\ & & \mid & \textsf{true} \mid \textsf{false} \mid \textsf{if}\ e_1\ e_2\ e_3 \end{array}$$

The additional rules seem straightforward: true and false are values, and a conditional computes by first reducing the condition to true or false and then selecting the correct branch.

$$\frac{}{\textsf{true}\ \mathit{value}} \qquad \frac{}{\textsf{false}\ \mathit{value}}$$

$$\frac{e_1 \mapsto e_1'}{\textsf{if}\ e_1\ e_2\ e_3 \mapsto \textsf{if}\ e_1'\ e_2\ e_3}\ \textsf{step/if}$$

$$\frac{}{\textsf{if true}\ e_2\ e_3 \mapsto e_2}\ \textsf{step/if/true} \qquad \frac{}{\textsf{if false}\ e_2\ e_3 \mapsto e_3}\ \textsf{step/if/false}$$

Note that we do not evaluate the branches of a conditional until we know whether the condition is true or false.

How do we type the new expressions? true and false are obvious.

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ \mathsf{tp/true} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \mathsf{tp/false}$$

The conditional is more interesting. We know its subject e_1 should be of type bool, but what about the branches and the result? We want type preservation to hold and we cannot tell before the program is executed whether the subject of conditional will be true or false. Therefore we postulate that both branches have the same general type τ and that the conditional has the same type.

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau}\ \mathsf{tp/if}$$

Exercises

Exercise 1 For each of the following statements, provide either a proof or counterexample. Consider only the case for e in the untyped λ-calculus from earlier in the course, and recall that a closed expression is one with no free variables.

(i) For closed e, if e *value* then e *normal*.

(ii) For closed e, if e *normal* then e *value*.

Exercise 2 Show the *new cases* in the proof of preservation and progress arising from parametric polymorphism.

(i) (Preservation) If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : \tau$

(ii) (Progress) If $\Gamma \vdash e : \tau$ then either $e \longrightarrow e'$ for some e' or e *normal*

(iii) (Finality of Normal Forms) There is no $\Gamma \vdash e : \tau$ such that $e \longrightarrow e'$ for some e' and e *normal*.

Explicitly state any additional substitution properties you need (in addition to Theorem L5.6), but you do not need to prove them.

Exercise 3 An alternative form of binary tree given in Section 3 is one where all information is stored in the leaves and none in the nodes. Let's call such a tree a *shrub*.

(i) Give the types for shrub constructors.

(ii) Give the construction of a shrub containing the numbers 1, 2, and 3.

(iii) Give the polymorphic definition of the type *shrub*, assuming it is represented by its own iterator.

(iv) Write a function $sumup$ to sum the elements of a shrub.

(v) Write a function $mirror$ that returns the mirror image of a given tree, reflected about a vertical line down from the root.

Exercise 4 We say two types τ and σ are *isomorphic* (written $\tau \cong \sigma$) if there are two functions $\mathit{forth} : \tau \to \sigma$ and $\mathit{back} : \sigma \to \tau$ such that they compose to the identity in both directions, that is, $\lambda x.\, \mathit{back}\ (\mathit{forth}\ x))$ is equal to $\lambda x.\, x$ and $\lambda y.\, \mathit{forth}\ (\mathit{back}\ y)$ is equal to $\lambda y.\, y$.

Consider the two types

$$\begin{array}{rcl} \mathit{nat} & = & \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha \\ \mathit{tan} & = & \forall \alpha.\, \alpha \to (\alpha \to \alpha) \to \alpha \end{array}$$

(i) Provide functions $\mathit{forth} : \mathit{nat} \to \mathit{tan}$ and $\mathit{back} : \mathit{tan} \to \mathit{nat}$ that, intuitively, should witness the isomorphism between *nat* and *tan*.

(ii) Compute the normal forms of the two function compositions. You may recruit the help of the LAMBDA implementation for this purpose.

(iii) Are the two function compositions β-equal to the identity? If yes, you are done. If not, can you see a sense under which they would be considered equal, either by changing your two functions or be defining a suitably justified notion of equality?

Exercise 5 Prove sequentiality: If $\cdot \vdash e : \tau$, $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$.

References

[CF98] Loïc Colson and Daniel Fredholm. System T, call-by-value, and the minimum problem. *Theoretical Computer Science*, 206(1–2):301–315, 1998.