Lecture Notes on
From λ-Calculus to Programming Languages

15-814: Types and Programming Languages
Frank Pfenning

Lecture 5
Tue Sep 9, 2025

1 Introduction

We start this lecture with a general representation of pairs in the polymorphic λ-calculus. This means we can represent not only pairs of natural numbers as at the end of the last lecture, but general pairs. This is a crucial step in showing how primitive recursion can be represented in the λ-calculus.

The we start the transition 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 next.

2 Polymorphic Pairs

Pairs can have components of arbitrary type. We write *prod* α β as the type of pairs whose first component has type α and second component has type β. Technically, this is not part of the polymorphic λ-calculus because *prod* : *type* $\to$ *type* $\to$ *type* is a function from types to types. This is available in the F^ω, but we don't generalize this far but think of it as a notation at the metalevel.

Recall that in the untyped setting we have

$$\mathit{pair} = \lambda x.\, \lambda y.\, \lambda k.\, k\, x\, y$$

We are aiming for:

$$\begin{array}{l}\mathit{pair} : \forall\alpha.\, \forall\beta.\, \alpha \to \beta \to \mathit{pair}\ \alpha\ \beta \\ \mathit{fst} : \forall\alpha.\, \forall\beta.\, \mathit{pair}\ \alpha\ \beta \to \alpha \\ \mathit{snd} : \forall\alpha.\, \forall\beta.\, \mathit{pair}\ \alpha\ \beta \to \beta\end{array}$$

We can start mechanically on the definition of *pair*.

$$\mathit{pair} = \Lambda\alpha.\, \Lambda\beta.\, \lambda x{:}\alpha.\ \lambda y{:}\beta.\ \boxed{}$$

Somehow, we need to give a type to $\lambda k.\, k\, x$ from the untyped definition. We see:

$$k : \alpha \to \beta \to \boxed{}$$

In the case of the first projection, the result type needs to be α and in the case of the second projections, the result type need to be β. Therefore, we generalize the result type to be polymorphic.

$$k : \alpha \to \beta \to \gamma.$$

What, then, is the type *pair* α β? It requires a k of the above type, and the returns a result of type γ. That is:

$$\textit{prod}\ \alpha\ \beta = \forall\gamma.\ (\alpha \to \beta \to \gamma) \to \gamma$$

$$\textit{pair} = \Lambda\alpha.\ \Lambda\beta.\ \lambda x{:}\alpha.\ \lambda y{:}\beta.\ \Lambda\gamma.\ (\alpha \to \beta \to \gamma) \to \gamma$$

Now we can fill in the definitions of the projections.

$$\textit{fst} : \forall\alpha.\ \forall\beta.\ \textit{pair}\ \alpha\ \beta \to \alpha$$
$$\textit{fst} = \Lambda\alpha.\ \Lambda\beta.\ \lambda p{:}\textit{pair}\ \alpha\ \beta.\ p\ [\alpha]\ (\lambda x{:}\alpha.\ \lambda y{:}\beta.\ x)$$

$$\textit{snd} : \forall\alpha.\ \forall\beta.\ \textit{pair}\ \alpha\ \beta \to \beta$$
$$\textit{snd} = \Lambda\alpha.\ \Lambda\beta.\ \lambda p{:}\textit{pair}\ \alpha\ \beta.\ p\ [\beta]\ (\lambda x{:}\alpha.\ \lambda y{:}\beta.\ y)$$

Note how γ in the type of *prod* α β is instantiated to α in the first case and β in the second.

In these examples we have written in the types for λ-abstraction, which is the official syntax. In practice, much of this information is redundant and may be omitted in the LAMBDA implementation.

3 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 $O(\max(n, k))$ [Colson and Fredholm, 1998], 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.

4 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 3 where reduction could take place in arbitrary position of an expression.

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

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.

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

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

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.

5 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 $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$. 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:

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

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.

$$\frac{}{\mathsf{true}\ \textit{value}} \qquad \frac{}{\mathsf{false}\ \textit{value}}$$

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

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

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 An form of binary tree given is one where all information is stored in the leaves and none in the nodes. Let's call such a tree a *shrub*. In this exercise we give a representation of shrubs in the polymorphic λ-calculus.

(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 2 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 3 Prove sequentiality: If $\cdot \vdash e : \tau$, $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$.

References

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