Lecture Notes on
From λ-Calculus to Programming Languages

15-814: Types and Programming Languages
Frank Pfenning

Lecture 5
September 18, 2018

1 Introduction

The λ-calculus is exceedingly elegant and minimal, but there are a number of problems if you want to think it of as the basis for an actual programming language. Here are some thoughts discussed in class.

Too abstract. Generally speaking, abstraction is good in the sense that it is an important role of functions (abstracting away from a particular special computation) or modules (abstracting away from a particular implementation). "Too abstract" would mean that we cannot express algorithms or ideas in code because the high level of abstraction prevents us from doing so. This is a legitimate concern for the λ-calculus. For example, what we observe as the result of a computation is only the normal form of an expression, but we might want to express some programs that interact with the world or modify a store. And, yes, the representation of data like natural numbers as functions has problems. While all recursive *functions* on natural numbers can be represented, not all *algorithms* can. 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.

Observability of functions. Since reduction results in normal form, to interpret the result 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.

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. In fact, the type system so far is very restrictive. Consider the conditional $\mathit{if} = \lambda b.\, b$, where we typed Booleans as $\alpha \to (\alpha \to \alpha)$. We would like to be able to type $\mathit{if}\ b\ e_1\ e_2$ for a Boolean e and expressions e_1 and e_2 of some type τ. Inspection of the typing rules will tell you that $e_1 : \alpha$ and $e_2 : \alpha$, but what if we want to type $\mathit{if}\ b\ \mathit{zero}\ (\mathit{succ}\ \mathit{zero})$ which returns $\overline{0}$ when b is true and $\overline{1}$ if b is false? Recall here that $\overline{n} : \beta \to (\beta \to \beta) \to \beta$ which is different from α. Can we then "instantiate" α with $\beta \to (\beta \to \beta) \to \beta$? It is possible to recover from this mess, but it is not easy.

In this lecture we focus on the first two points: 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.

2 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 $e\ \mathit{val}$. 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.

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 $e \mapsto e'$ or $e\ \mathit{val}$.

Values. If $e\ \mathit{val}$ then there is no e' such that $e \mapsto e'$.

Determinacy. 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.

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

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

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{ap}_1 \qquad \frac{}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \beta$$

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.

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{ap}_1 \qquad \frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto v_1\, e_2'}\ \mathsf{ap}_2$$

$$\frac{v_2\ \mathit{val}}{(\lambda x.\, e_1)\, v_2 \mapsto [v_2/x]e_1}\ \beta_{\mathit{val}}$$

We achieve determinacy 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.

In lecture, we proceeded with the call-by-name rules because there are fewer of them. But there are good logical reasons why functions should be call-by-value, so in these notes we'll use the call-by-value rules instead.

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

3 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:

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

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.

$$\frac{}{\mathsf{true}\ \mathit{val}} \qquad \frac{}{\mathsf{false}\ \mathit{val}}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{if}\ e_1\ e_2\ e_3 \mapsto \mathsf{if}\ e_1'\ e_2\ e_3}\ \mathsf{if}_1$$

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

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.

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

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.

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

In lecture, a student made the excellent suggestion that we could instead type it as

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_3 : \tau_3}{\Gamma \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau_2 \vee \tau_3}$$

saying that the result must be either of type τ_2 or τ_3. Something like this is indeed possible using so-called *union types*, but it turns out they are quite complex. For example, what can we do safely with the result of the conditional if all we know is that the result is either bool or $\mathsf{bool} \to \mathsf{bool}$? We will make a few more remarks on this in the next lecture.

4 Type Preservation

Now we should revisit the most important theorems about the programming language we define, namely preservation and progress. These two together constitute what we call *type safety*. Since these theorems are of such pervasive importance, we will prove them in great detail. Generally speaking, the proof decomposes along the types present in the language because we carefully designed the rules so that this is the case. For example, we added *if* $e_1\ e_2\ e_3$ as a language primitive instead of as *if* a function of three arguments. Doing the latter would significantly complicate the reasoning.

We already know that the rules should satisfy the substitution property (Theorem L4.4). We can easily check the new cases in the proof because substitution remains compositional. For example, $[e'/x](\mathsf{if}\ \ e_1\ \ e_2\ \ e_3) = \mathsf{if}\ ([e'/x]e_1)\ ([e'/x]e_2)\ ([e'/x]e_3)$.

Property 1 (Substitution)
If $\Gamma \vdash e' : \tau'$ *and* $\Gamma, x : \tau' \vdash e : \tau$ *then* $\Gamma \vdash [e'/x]e : \tau$.

On to preservation.

Theorem 2 (Type Preservation)
If $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \tau$.

Proof: By rule induction on the deduction of $e \mapsto e'$.

Case:

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \mathsf{ap}_1$$

where $e = e_1\ e_2$ and $e' = e_1'\ e_2$.

$\cdot \vdash e_1\ e_2 : \tau$	Assumption
$\cdot \vdash e_1 : \tau_2 \to \tau$ and $\cdot \vdash e_2 : \tau_2$ for some τ_2	By inversion
$\cdot \vdash e_1' : \tau_2 \to \tau$	By ind.hyp.
$\cdot \vdash e_1'\ e_2 : \tau$	By rule app

Case:

$$\frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto v_1\, e_2'}\ \mathsf{ap}_2$$

where $e = v_1\, e_2$ and $e' = v_1\, e_2'$. As in the previous case, we proceed by inversion on typing.

$\cdot \vdash v_1\, e_2 : \tau$	Assumption
$\cdot \vdash v_1 : \tau_2 \to \tau$ and $\cdot \vdash e_2 : \tau_2$ for some τ_2	By inversion
$\cdot \vdash e_2' : \tau_2$	By ind.hyp.
$\cdot \vdash v_1\, e_2' : \tau$	By rule app

Case:

$$\frac{v_2\ \mathit{val}}{(\lambda x.\, e_1)\, v_2 \mapsto [v_2/x]e_1}\ \beta_{\mathit{val}}$$

where $e = (\lambda x.\, e_1)\, v_2$ and $e' = [v_2/x]e_1$. Again, we apply inversion on the typing of e, this time twice. Then we have enough pieces to apply the *substitution property* (Theorem 1).

$\cdot \vdash (\lambda x.\, e_1)\, v_2 : \tau$	Assumption
$\cdot \vdash \lambda x.\, e_1 : \tau_2 \to \tau$ and $\cdot \vdash v_2 : \tau_2$ for some τ_2	By inversion
$x : \tau_2 \vdash e_1 : \tau$	By inversion
$\cdot \vdash [v_2/x]e_1 : \tau$	By the *substitution property* (Theorem 1)

Case:

$$\frac{e_1 \mapsto e_1'}{\mathsf{if}\ e_1\ e_2\ e_3 \mapsto \mathsf{if}\ e_1'\ e_2\ e_3}\ \mathsf{if}_1$$

where $e = \mathsf{if}\ e_1\ e_2\ e_3$ and $e' = \mathsf{if}\ e_1'\ e_2\ e_3$. As might be expected by now, we apply inversion to the typing of e, followed by the induction hypothesis on the type of e_1, followed by re-application of the typing rule for if.

$\cdot \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau$	Assumption
$\cdot \vdash e_1 : \mathsf{bool}$ and $\cdot \vdash e_2 : \tau$ and $\cdot \vdash e_3 : \tau$	By inversion
$\cdot \vdash e_1' : \mathsf{bool}$	By ind.hyp.
$\cdot \vdash \mathsf{if}\ e_1'\ e_2\ e_3 : \tau$	By rule

Case:

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

where $e = \mathsf{if}\ \mathsf{true}\ e_2\ e_3$ and $e' = e_2$. This time, we don't have an induction hypothesis since this rule has no premise, but fortunately one step of inversion suffices.

$\cdot \vdash \mathsf{if}\ \mathsf{true}\ e_2\ e_3 : \tau$	Assumption
$\cdot \vdash \mathsf{true} : \mathsf{bool}$ and $\cdot \vdash e_2 : \tau$ and $\cdot \vdash e_3 : \tau$	By inversion
$\cdot \vdash e' : \tau$	Since $e' = e_2$.

Case: Rule $\mathsf{if/false}$ is analogous to the previous case.

□

5 Progress

To complete the lecture, we would like to prove progress: ever closed, well-typed expression is either already a value or can take a step. First, it is easy to see that the assumptions here are necessary. For example, the ill-typed expression $\mathsf{if}\ (\lambda x.\, x)\ \mathsf{false}\ \mathsf{true}$ cannot take a step since the subject $(\lambda x.\, x)$ is a value but the whole expression is not and cannot take a step. Similarly, the expression $\mathsf{if}\ b\ \mathsf{false}\ \mathsf{true}$ is well-typed in the context with $b : \mathsf{bool}$, but it cannot take a step nor is it a value.

Theorem 3 (Progress)
If $\cdot \vdash e : \tau$ then either $e \mapsto e'$ for some e' or e val.

Proof: There are not many candidates for this proof. We have e and we have a typing for e. From that scant information we need obtain evidence that e can step or is a value. So we try the rule induction on $\cdot \vdash e : \tau$.

Case:

$$\frac{x_1 : \tau_1 \vdash e_2 : \tau_2}{\cdot \vdash \lambda x_1.\, e_2 : \tau_1 \to \tau_2}$$

where $e = \lambda x_1.\, e_2$. Then we have

$\lambda x_1.\, e_2$ *val*	By rule $\mathsf{val/lam}$

It is fortunate we don't need the induction hypothesis, because it cannot be applied! That's because the context of the premise is not empty, which is easy to miss. So be careful!

Case:

$$\frac{\cdot \vdash e_1 : \tau_2 \to \tau \quad \cdot \vdash e_2 : \tau_2}{\cdot \vdash e_1\, e_2 : \tau}$$

where $e = e_1\, e_2$. At this point we apply the induction hypothesis to e_1. If it reduces, so does $e = e_1\, e_2$. If it is a value, then we apply the induction hypothesis to e_2. If is reduces, so does $e_1\, e_2$. If not, we have a β_{val} redex. In more detail:

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 *val*	By ind.hyp.
$e_1 \mapsto e_1'$	Subcase
$e = e_1\, e_2 \mapsto e_1'\, e_2$ by rule ap_1	
e_1 *val*	Subcase
Either $e_2 \mapsto e_2'$ for some e_2' or e_2 *val*	By ind.hyp.
$e_2 \mapsto e_2'$	Sub^{2}case
$e_1\, e_2 \mapsto e_1\, e_2'$	By rule ap_2 since e_1 *val*
e_2 *val*	Sub^{2}case
$e_1 = \lambda x.\, e_1'$ and $x : \tau_2 \vdash e_1' : \tau$	By "inversion"

We pause here to consider this last step. We know that $\cdot \vdash e_1 : \tau_2 \to \tau$ and e_1 *val*. By considering all cases for how both of these judgments can be true at the same time, we see that e_1 must be a λ-abstraction. This is often summarized in a *canonical forms lemma* which we didn't discuss in lecture, but state after this proof. Finishing this sub^{2}case:

$e = (\lambda x\, e_1')\, e_2 \mapsto [e_2/x]e_1'$	By rule β_{val} since e_2 *val*

Case:

$$\frac{}{\cdot \vdash \mathsf{true} : \mathsf{bool}}$$

where $e = \mathsf{true}$. Then $e = \mathsf{true}$ *val* by rule.

Case: Typing of false. As for true.

Case:

$$\frac{\cdot \vdash e_1 : \mathsf{bool} \quad \cdot \vdash e_2 : \tau \quad \cdot \vdash e_3 : \tau}{\cdot \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau}$$

where $e = \mathit{if}\ e_1\ e_2\ e_3$.

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 *val* — By ind.hyp.

$e_1 \mapsto e_1'$ — Subcase
$e = \mathsf{if}\ e_1\ e_2\ e_3 \mapsto \mathsf{if}\ e_1'\ e_2\ e_3$ — By rule if_1

e_1 *val* — Subcase
$e_1 = \mathsf{true}$ or $e_1 = \mathsf{false}$
By considering all cases for $\cdot \vdash e_1 : \mathsf{bool}$ and e_1 *val*

$e_1 = \mathsf{true}$ — Sub^{2}case
$e = \mathsf{if}\ \mathsf{true}\ e_2\ e_3 \mapsto e_2$ — By rule

$e_1 = \mathsf{false}$ — Sub^{2}case
$e = \mathsf{if}\ \mathsf{false}\ e_2\ e_3 \mapsto e_3$ — By rule

□

This completes the proof. The complex inversion steps can be summarized in the *canonical forms lemma* that analyzes the shape of well-typed values. It is a form of the representation theorem for Booleans we proved in an earlier lecture for the simply-typed λ-calculus.

Lemma 4 (Canonical Forms)

(i) If $\cdot \vdash v : \tau_1 \to \tau_2$ and v val then $v = \lambda x_1.\, e_2$ for some x_1 and e_2.

(ii) If $\cdot \vdash v : \mathsf{bool}$ and v val then $v = \mathsf{true}$ or $v = \mathsf{false}$.

Proof: For each part, analyzing all the possible cases for the value and typing judgments. □

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.