Lecture Notes on Subject Reduction

15-814: Types and Programming Languages
Frank Pfenning

Lecture 5
September 17, 2019

1 Introduction

In the last lecture we proved some key aspect of a *representation theorem* for Booleans, namely that the closed normal forms type $\alpha \to (\alpha \to \alpha)$ are either $\mathit{true} = \lambda x.\, \lambda y.\, x$ or $\mathit{false} = \lambda x.\, \lambda y.\, y$. But how does this fit into the bigger picture? Recall that we wanted to relate arbitrary expressions of a certain type to Booleans. We had conjectured (L3.2)

Conjecture 1 (L3.2, Representation of Booleans, v2)
If $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *then* $e =_\beta$ *true or* $e =_\beta$ *false.*

But we want to relate this to computation. Fortunately, by the Church-Rosser Theorem, $e =_\beta e'$ for a normal form e' if and only if $e \longrightarrow^* e'$ (where $\longrightarrow^*$ is the reflexive and transitive closure of single-step reduction we have been mostly working with). So we recast this one more time, relating typing to computation and representation.

Conjecture 2 (Computation of Booleans)
If $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *then* $e \longrightarrow^*$ *true or* $e \longrightarrow^*$ *false.*

Since every well-typed expression has a normal form (Theorem L3.4, which we did not prove), the missing link in our reasoning chain is that typing is preserved under reduction: if we start with an expression e of type τ and we reduce it all the way to a normal form e', then e' will still have type τ. For the special case where $\tau = \alpha \to (\alpha \to \alpha)$ which means that any expression e of type τ that has a normal form represents a Boolean.

2 Type Inference

We explained how the rules for $\Gamma \vdash e \Leftarrow \tau$ and $\Gamma \vdash e \Rightarrow \tau$ describe an *algorithm* for type checking ($\Leftarrow$) and synthesis ($\Rightarrow$). Let's return to the original typing judgment $\Gamma \vdash e : \tau$ to see if it somehow embodies an algorithm.

Our example is self-application, $\lambda x.\, x\, x$. Recall that $\Omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$ does not have a normal form because it only reduces to itself. If we believe the claim that every simply-typed expression has a normal form, then Ω cannot have a type (and neither can Y). The component $\lambda x.\, x\, x$ is in normal form, so there is a chance it might by typable. But we cannot use our bidirectional type checking algorithm since we do not know which type to check it against. Instead, we use the fact that the typing rules are *syntax-directed*, which means we can construct the shape of the typing derivation without any thought.

$$\dfrac{\dfrac{}{x : \boxed{} \vdash x : \boxed{}}\,\textsf{var} \qquad \dfrac{}{x : \boxed{} \vdash x : \boxed{}}\,\textsf{var}}{\dfrac{x : \boxed{} \vdash x\, x : \boxed{}}{\cdot \vdash \lambda x.\, x\, x : \boxed{}}\,\textsf{lam}}\,\textsf{app}$$

We can now fill in the blanks with unknowns and collect some constraints. We write these unknowns as $\tau?$ to express "*some type* τ". For example, we know the final conclusion must have some type $\tau?$, and because it is inferred by the rule lam it must actually have the form $\tau? = \tau_1? \to \tau_2?$ for some $\tau_1?$ and $\tau_2?$.

$$\dfrac{\dfrac{}{x : \boxed{} \vdash x : \boxed{}}\,\textsf{var} \qquad \dfrac{}{x : \boxed{} \vdash x : \boxed{}}\,\textsf{var}}{\dfrac{x : \boxed{\tau_1?} \vdash x\, x : \boxed{\tau_2?}}{\cdot \vdash \lambda x.\, x\, x : \boxed{\tau?}}\,\textsf{lam}}\,\textsf{app}$$

where $\tau? = \tau_1 \to \tau_2$. Now we consider the app rule, which types $x\, x$. The function part of this (the first x) must have type $\tau_3? \to \tau_2?$, and the argument

(the second x) must have type $\tau_3?$ for some $\tau_3?$. Filling this in:

$$\dfrac{\dfrac{}{x : \boxed{\tau_1?} \vdash x : \boxed{\tau_3? \to \tau_2?}}\,\textsf{var} \qquad \dfrac{}{x : \boxed{\tau_1?} \vdash x : \boxed{\tau_3?}}\,\textsf{var}}{\dfrac{x : \boxed{\tau_1?} \vdash x\,x : \boxed{\tau_2?}}{\cdot \vdash \lambda x.\, x\,x : \boxed{\tau?}}\,\textsf{lam}}\,\textsf{app}$$

where $\tau? = \tau_1? \to \tau_2?$. Now we analyze the final two var rules, which tell us that $\tau_1? = \tau_3? \to \tau_2?$ and also $\tau_1? = \tau_3?$. Collecting all the constraints on the variables, any solution to the following equations will give us a valid typing derivation:

$$\begin{array}{lcl} \tau? & = & \tau_1? \to \tau_2? \\ \tau_1? & = & \tau_3? \to \tau_2? \\ \tau_1? & = & \tau_3? \end{array}$$

But there cannot be a solution! Substituting the third equation into the second one we deduce

$$\tau_3? = \tau_3? \to \tau_2?$$

which does not have a solution. No matter which type we try to substitute for $\tau_3?$, the right-hand side is always strictly larger than the left-hand side and therefore the two cannot be equal.

We then conclude that $\lambda x.\, x\,x$ cannot have a simple type.

As an example of an expression that *can* be typed, consider the identity function, $\lambda x.\, x$.

$$\dfrac{\dfrac{}{x : \boxed{} \vdash x : \boxed{}}\,\textsf{var}}{\cdot \vdash \lambda x.\, x : \boxed{}}\,\textsf{lam}$$

Filling in variables and collecting constraints:

$$\dfrac{\dfrac{}{x : \boxed{\tau_1?} \vdash x : \boxed{\tau_2?}}\,\textsf{var}}{\cdot \vdash \lambda x.\, x : \boxed{\tau?}}\,\textsf{lam}$$

where

$$\begin{array}{lcl} \tau? & = & \tau_1? \to \tau_2? \\ \tau_1? & = & \tau_2? \end{array}$$

This can easily be solved and we get

$$\tau? = \tau_2? \to \tau_2?$$

for any arbitrary $\tau_2?$. The "most general" solution is to use a variable α for $\tau_2?$, because we know that we can obtain another typing derivation from a given one with α by substituting any type τ for α.

The general form of the process we have illustrated here is *type inference*. It construct that skeleton of a typing derivation, collects constraints on the unknown types, and then solves them to find a most general solution. The algorithm for finding a most general solution (or reporting there is none) is called *unification*.

A great property of type inference is that the programmer does not have to supply any types. A drawback is that the collected equations are "global", that is, they come from the whole typing derivation. This means it may be difficult to pinpoint the source of a type error, and if you have done any significant functional programming you probably had to contend with this issue. Bidirectional type-checking on the other hand only matches types against each other *locally*, so type errors are quite specific. The drawback is that some types have to be supplied by the programmer. The biggest advantage of the bidirectional algorithm is that it is robust under language extensions, while type inference fairly quickly becomes either undecidable or impractical. We will reconsider this going forward, when our language becomes richer.

3 Subject Reduction

Now we return to the main topic of this lecture, namely subject reduction. Recall our characterization of reduction from Lecture 3, written here as a collection of inference rules.

$$\frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}\ \mathsf{red/lam} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app_1} \qquad \frac{e_2 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1\, e_2'}\ \mathsf{red/app_2}$$

$$\frac{}{(\lambda x.\, e_1)\, e_2 \longrightarrow [e_2/x]e_1}\ \mathsf{beta}$$

And, for reference, here are the typing rules.

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1.\, e_2 : \tau_1 \to \tau_2}\ \mathsf{lam} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathsf{var}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ \mathsf{app}$$

Theorem 3 (Subject Reduction)
If $\Gamma \vdash e : \tau$ *and* $e \longrightarrow e'$ *then* $\Gamma \vdash e' : \tau$.

Proof: In this theorem statement we are given derivations for two judgments: $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$. Most likely, the proof will proceed by rule induction on one of these and by inversion on the other. The typing judgment is syntax-directed and therefore amenable to reasoning by inversion, so we try rule induction over the reduction judgment.

By rule induction on the derivation of $e \longrightarrow e'$.

Case:

$$\frac{e_1 \longrightarrow e_1'}{\lambda x.\, e_1 \longrightarrow \lambda x.\, e_1'}\ \mathsf{red/lam}$$

where $e = \lambda x.\, e_1'$.

$\Gamma \vdash \lambda x.\, e_1 : \tau$	Assumption
$\Gamma, x : \tau_2 \vdash e_1 : \tau_1$ and $\tau = \tau_2 \to \tau_1$ for some τ_1 and τ_2	By inversion
$\Gamma, x : \tau_2 \vdash e_1' : \tau_1$	By induction hypothesis
$\Gamma \vdash \lambda x.\, e_1' : \tau_2 \to \tau_1$	By rule lam

Case:

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app_1}$$

where $e = e_1\, e_2$. We start again by restating what we know in this case and then apply inversion.

$\Gamma \vdash e_1\, e_2 : \tau$	Assumption
$\Gamma \vdash e_1 : \tau_2 \to \tau$ and	
$\Gamma \vdash e_2 : \tau_2$ for some τ_2	By inversion

At this point we have a type for e_1 and a reduction for e_1, so we can apply the induction hypothesis.

$\Gamma \vdash e_1' : \tau_2 \to \tau$ By ind.hyp.

Now we can just apply the typing rule for application. Intuitively, in the typing for $e_1\, e_2$ we have replaced e_1 by e_1', which is okay since e_1' has the type of e_1.

$\Gamma \vdash e_1'\, e_2 : \tau$ By rule lam

Case:

$$\frac{e_2 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app}_2$$

where $e = e_1\, e_2$. This proceeds completely analogous to the previous case.

Case:

$$\frac{}{(\lambda x.\, e_1)\, e_2 \longrightarrow [e_2/x]e_1}\ \beta$$

where $e = (\lambda x.\, e_1)\, e_2$. In this case we apply inversion twice, since the structure of e is two levels deep.

$\Gamma \vdash (\lambda x.\, e_1)\, e_2 : \tau$	Assumption
$\Gamma \vdash \lambda x.\, e_1 : \tau_2 \to \tau$	
and $\Gamma \vdash e_2 : \tau_2$ for some τ_2	By inversion
$\Gamma, x : \tau_2 \vdash e_1 : \tau$	By inversion

At this point we are truly stuck, because there is no obvious way to complete the proof.

To Show: $\Gamma \vdash [e_2/x]e_1 : \tau$

Fortunately, the gap that presents itself is exactly the content of the *substitution property*, stated below. The forward reference here is acceptable, since the proof of the substitution property does not depend on subject reduction.

$\Gamma \vdash [e_2/x]e_1 : \tau$ By the *substitution property* (Theorem 4)

□

Theorem 4 (Substitution Property)
If $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$ *then* $\Gamma \vdash [e/x]e' : \tau'$

Proof sketch: By rule induction on the deduction of $\Gamma, x : \tau \vdash e' : \tau'$. Intuitively, in this deduction we can use $x : \tau$ only at the leaves, and there to conclude $x : \tau$. Now we replace this leaf with the given derivation of $\Gamma \vdash e : \tau$ which concludes $e : \tau$. Luckily, $[e/x]x = e$, so this is the correct judgment.

There is only a small hiccup: when we introduce a different variable $x_1 : \tau_1$ into the context in the lam rule, the contexts of the two assumptions no longer match. But we can apply *weakening*, that is, adjoin the unused hypothesis $x_1 : \tau_1$ to every judgment in the deduction of $\Gamma \vdash e : \tau$. After that, we can apply the induction hypothesis. □

We recommend you write out the cases of the substitution property in the style of our other proofs, just to make sure you understand the details.

The substitution property is so critical that we may elevate it to an intrinsic property of the turnstile ($\vdash$). Whenever we write $\Gamma \vdash J$ for any judgment J we imply that a substitution property for the judgments in Γ must hold. This is an example of a *hypothetical* and *generic* judgment [ML83]. We may return to this point in a future lecture, especially if the property appears to be in jeopardy at some point. It is worth remembering that, while we may not want to prove an explicit substitution property, we still need to make sure that the judgments we define are hypothetical/generic judgments.

4 Taking Stock

Where do we stand at this point in our quest for a representation theorems for Booleans? We have the following:

Reduction and Normal Forms

(i) If $e \not\longrightarrow$ then e *normal*.

(ii) If e *normal* the $e \not\longrightarrow$.

Typing and Normal Forms (Exercise L4.1.2-3)

(i) If $\Gamma \vdash e \Leftarrow \tau$ then $\Gamma \vdash e : \tau$ and e *normal*.

(ii) If $\Gamma \vdash e : \tau$ and e *normal* then $\Gamma \vdash e \Leftarrow \tau$

Representation of Booleans in Normal Form (L4.2)
If $\cdot \vdash e \Leftarrow \alpha \to (\alpha \to \alpha)$ then either $e = \mathit{true} = \lambda x.\, \lambda y.\, x$ or $e = \mathit{false} = \lambda x.\, \lambda y.\, y$.

Subject Reduction (L5.3)
If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$ we have $\Gamma \vdash e' : \tau$.

We did not prove normalization (also called *termination*) or confluence (also called the Church-Rosser property).

Normalization
If $\Gamma \vdash e : \tau$ then $e \longrightarrow^* e'$ for some e' with e' *nf*.

Confluence
If $e \longrightarrow^* e_1$ and $e \longrightarrow^* e_2$ then there exists an e' such that $e_1 \longrightarrow^* e'$ and $e_2 \longrightarrow e'$.

We could replay the whole development for the representation of natural numbers instead of Booleans, with some additional complications, but we will forego this in favor of tackling more realistic programming languages.

References

[ML83] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Notes for three lectures given in Siena, Italy. Published in *Nordic Journal of Philosophical Logic*, 1(1):11-60, 1996, April 1983.