Recitation 2:
Binding, Semantics, and Safety

15-312: Foundations of Programming Languages

Charles Yuan, Jeanne Luning Prak

January 24, 2018

1 Abstract Binding Trees

The abstract syntax trees we saw previously contained the concept of variables, but not the ability to give variables meaning. To do that, we need the concept of **binding**, which is provided by the **abstract binding tree**, or **abt**. Like ast's, abt's are characterized by variables and operators, but operators have notion of binding.

In an abt, every operator may specify a number of variables that are bound within the **scope** of the operator. Operators accordingly have arities that reflect the number and sorts of variables they bind. For example, an operator `let` of arity

$$(\mathsf{Exp}, \mathsf{Exp}.\mathsf{Exp})\mathsf{Exp}$$

takes a first abt argument that has no bound variables of sort Exp and a second argument with one bound variable of sort Exp and which is also of sort Exp. The notation $x.e$ is used to indicate a **binder**, or an expression coupled with a named bound variable. We might use such an operator to represent a `let`-expression in ML:

$$\texttt{let}(1, x.x + 2) \text{ for } \texttt{let } x = 1 \texttt{ in } x + 2 \texttt{ end}$$

Inside an operator that binds some variable x, x is considered **bound**, and a variable that is not inside a binding of its own name is considered **free**. The distinction between bound and free variables is significant when we consider the semantics of substitution.

Binders themselves are not valid abt's but for convenience we often use notation that pretends they are.

1.1 Substitution

Substitution in a system without bindings involves simply replacing instances of the specified variable with instances of the substitute in a structural manner:

$$[e/x]A(x, y, B(z, x)) \hookrightarrow A(e, y, B(z, e))$$

In a system with bindings, we now limit ourselves to only substituting for free instances. Bound variables are ignored in substitution until they are later unbound:

$$[e/x]A(x, y, x.B(z, x)) \hookrightarrow A(e, y, x.B(z, x))$$

However, this rule is not enough. Consider substitution on this ML expression:

$$[x/y]\,\texttt{fn}\ x \Rightarrow y \hookrightarrow \texttt{fn}\ x \Rightarrow x$$

Here, y was free and we attempt to substitute, but directly substituting has allowed us to turn a constant function into the identity function, which is absurd. This situation is known as unintentional **capture** of the variable y by the binder. We must use a more restrictive rule when it comes to binders:

$$[e'/x]y.e \text{ substitutes } e' \text{ for } x \text{ in } e \text{ only if } x \neq y \text{ and } y \text{ is not free in } e'$$

The requirement that y not be free in e' is known as **freshness** of y. We may solve the freshness requirement in two ways: one is through **α-conversion**, and another is the use of **de Bruijn indices**, which you will see on Assignment 1.

1.2 α-Equivalence

α-equivalence is an equivalence relation on abt's that allows free exchange of the choice of bound variable inside a binder. Namely:

$$x.e \cong_\alpha y.([y/x]e)$$

Using α-equivalence, we may convert an expression in which some variable is not fresh to an equivalent expression in which it is fresh, and proceed with substitution.

In this course we will often speak of expressions that are α-equivalent as equal. Such conversions will often be done implicitly.

2 Statics

The **statics** of a language is the system of rules that govern the meaning of the language at expression-level *before* the expression is evaluated (updated) according to a different set of rules known as the dynamics. It usually consists of **typing judgments** that determine whether an expression is well-formed.

Given a language, we introduce two sorts, Typ and Exp, corresponding to the types and expressions of the language respectively. When parsing of the program completes, we will have an abt that can be decomposed into subtrees falling into these sorts.

Example of the syntax of a language:

Typ	τ	::=	`num`	number
Exp	e	::=	x	variable
			`num`$[n]$	literal
			`plus`$(e_1; e_2)$	addition
			`times`$(e_1; e_2)$	multiplication
			`let`$(e_1; x.e_2)$	definition

We then define a judgment, **typing**, which relates an expression e with its type τ using the **typing context** Γ:

$$\Gamma \vdash e : \tau$$

Example of our language's typing judgment definition:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{num}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1; e_2) : \texttt{num}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{times}(e_1; e_2) : \texttt{num}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1; x.e_2) : \tau_2}$$

The rules state the following:

- Variables are given types by the context.
- Numeric literals are of type `num`.
- Sums and products of two expressions of type `num` are of type `num`.
- Given that e_1 has some type τ_1, if substituting e_1 for x with type τ_1 in e_2 would give it type τ_2, then `let` $x = e_1$ `in` e_2 has type τ_2.

A **unicity of typing** theorem ensures the consistency of a type system. It says:

For all Γ and e, there is at most one τ such that $\Gamma \vdash e : \tau$.

3 Dynamics

The **dynamics** of a language describe how expressions evaluate. In this class we mainly focus on **structural dynamics**, which is given by a system of **transitions** from one expression to another, until final states called **values** are reached.

The judgment $e\ \mathsf{val}$ states that e is a value. The judgment $e \mapsto e'$ states that expression e steps to expression e'.

Example of our language's value and step judgment definitions:

$$\frac{}{\texttt{num}[n]\ \mathsf{val}}$$

$$\frac{n_1 + n_2 = n}{\texttt{plus}(\texttt{num}[n_1]; \texttt{num}[n_2]) \mapsto \texttt{num}[n]}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{plus}(e_1; e_2) \mapsto \texttt{plus}(e_1'; e_2)}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{plus}(e_1; e_2) \mapsto \texttt{plus}(e_1; e_2')}$$

$$\frac{n_1 \times n_2 = n}{\texttt{times}(\texttt{num}[n_1]; \texttt{num}[n_2]) \mapsto \texttt{num}[n]}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{times}(e_1; e_2) \mapsto \texttt{times}(e_1'; e_2)}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{times}(e_1; e_2) \mapsto \texttt{times}(e_1; e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{let}(e_1; x.e_2) \mapsto \mathtt{let}(e_1'; x.e_2)}$$

$$\frac{e_1 \ \mathsf{val}}{\mathtt{let}(e_1; x.e_2) \mapsto [e_1/x]e_2}$$

The rules state the following:

- Numeric literals are values.
- Sums and products evaluate their first argument, then their second, then evaluate to the arithmetic result. Note that this constitutes an **eager, left-to-right** dynamics, meaning that arguments are evaluated immediately from left to right. Most languages we study in this course will share this property.
- Let-expressions evaluate their substitute argument, then perform substitution to yield a new expression. This is eager as before, though there is discussion in Section 5.2 of PFPL as to how it might be lazy instead.

Note what the dynamics does not do: account for free variables, or likewise ill-typed expressions. The intention is for the statics check to have already occurred, and for dynamics to purely describe evaluation. Here we also did not account for runtime errors, but you will do so in Assignment 1.

A **canonical forms lemma** says that if we know the type of an expression, we already know what the values of the expression look like. It looks like this:

If e val, then [for each type τ in the language,]

if $\Gamma \vdash e : \tau$ then $e = V$ [where V is the form of the value].

Example of our language's canonical forms lemma:

If e val, then if $\Gamma \vdash e : \mathtt{num}$, then $e = \mathtt{num}[n]$ for some n.

Not all languages have canonical forms, as it depends on the definition of values.

There are two other useful notions for a dynamics: finality, which says that a well-typed expression is either a value or can step (never both); and determinacy, which says that an expression always steps to a unique expression if it can step at all. Finality is usually assumed to hold in this course, and so is determinacy (until much later in the course!).

4 Type Safety

A **progress** theorem guarantees our language does not get stuck during execution:

If $\cdot \vdash e : \tau$, then either e val, or there exists e' such that $e \mapsto e'$.

A **preservation** theorem guarantees our language never violates the type of an expression:

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

Notice that the progress and preservation theorems only apply to typing judgments where the context is empty. This means that they only applies to *closed terms*: there can be no free variables in e.

Together, progress and preservation constitute the central property of a language: **type safety**.

Progress and preservation can be proven about a language using rule induction.

4.1 Progress Proof

A proof of progress proceeds by rule induction on the typing derivation. We can specialize the principle of rule induction for a proof of progress on the example language like so:

> Let $\mathcal{P}(e)$ be the property that either $e\ \mathsf{val}$ or there exists an e' such that $e \mapsto e'$.
>
> To prove that if $\cdot \vdash e : \tau$, then either $e\ \mathsf{val}$, or there exists e' such that $e \mapsto e'$, it is sufficient to prove the following:
>
> - $\mathcal{P}(\mathtt{num}[n])$
> - If $\mathcal{P}(e_1)$, $\mathcal{P}(e_2)$, $\cdot \vdash e_1 : \mathtt{num}$, and $\cdot \vdash e_2 : \mathtt{num}$, then $\mathcal{P}(\mathtt{plus}(e_1; e_2))$
> - If $\mathcal{P}(e_1)$, $\mathcal{P}(e_2)$, $\cdot \vdash e_1 : \mathtt{num}$, and $\cdot \vdash e_2 : \mathtt{num}$, then $\mathcal{P}(\mathtt{times}(e_1; e_2))$
> - If $\mathcal{P}(e_1)$, $\cdot \vdash e_1 : \tau_1$, and $x : \tau_1 \vdash e_2 : \tau_2$, then $\mathcal{P}(\mathtt{let}(e_1; x.e_2))$

The proof itself is left as an exercise.

4.2 Preservation Proof

A proof of preservation proceeds by rule induction on the transition judgment, because it hinges on examining all possible transitions from a given expression. A proof of preservation for the example language is given below.

In the below proof, we make use of the following lemmas:

1. Inversion on Typing:
   - If $\Gamma \vdash \mathtt{plus}(e_1; e_2) : \tau$ then $\tau = \mathtt{num}$, $\Gamma \vdash e_1 : \mathtt{num}$, and $\Gamma \vdash e_2 : \mathtt{num}$.
   - If $\Gamma \vdash \mathtt{times}(e_1; e_2) : \tau$ then $\tau = \mathtt{num}$, $\Gamma \vdash e_1 : \mathtt{num}$, and $\Gamma \vdash e_2 : \mathtt{num}$.
   - If $\Gamma \vdash \mathtt{let}(e_1; x.e_2) : \tau$ then $\Gamma \vdash e_1 : \tau_1$ s.t. $\Gamma, x : \tau_1 \vdash e_2 : \tau$.
2. Substitution Lemma: If $\Gamma, x : \tau \vdash e' : \tau'$ and $\Gamma \vdash e : \tau$ then $\Gamma \vdash [e/x]e' : \tau'$.

PROOF:

> Let $\mathcal{P}(e, e')$ be the property that if $\cdot \vdash e : \tau$, then $\cdot \vdash e' : \tau$. Proceed by rule induction on judgment $e \mapsto e'$.
>
> - CASE: If $n_1 + n_2 = n$, then $\mathcal{P}(\mathtt{plus}(\mathtt{num}[n_1]; \mathtt{num}[n_2]), \mathtt{num}[n])$
>
>   I.H.: $n_1 + n_2 = n$
>   Assume $\cdot \vdash \mathtt{plus}(\mathtt{num}[n_1]; \mathtt{num}[n_2]) : \tau$
>   WTS: $\cdot \vdash \mathtt{num}[n] : \tau$
>
> | | |
> |---|---|
> | $\cdot \vdash \mathtt{plus}(\mathtt{num}[n_1]; \mathtt{num}[n_2]) : \mathtt{num}$ | [Inversion on Typing] |
> | $\cdot \vdash \mathtt{num}[n] : \mathtt{num}$ | [Typing rule for $\mathtt{num}$] |
> | $\cdot \vdash \mathtt{num}[n] : \tau$ | [Take $\tau = \mathtt{num}$] |
>
> - CASE: If $\mathcal{P}(e_1, e_1')$ and $e_1 \mapsto e_1'$, then $\mathcal{P}(\mathtt{plus}(e_1; e_2), \mathtt{plus}(e_1'; e_2))$
>
>   I.H.: $\mathcal{P}(e_1, e_1')$ and $e_1 \mapsto e_1'$
>   Assume $\cdot \vdash \mathtt{plus}(e_1; e_2) : \tau$

WTS: $\cdot \vdash \texttt{plus}(e_1'; e_2) : \tau$

$\cdot \vdash \texttt{plus}(e_1; e_2) : \texttt{num}$	[Inversion on Typing]
$\cdot \vdash e_1 : \texttt{num}$	[Inversion on Typing]
$\cdot \vdash e_1' : \texttt{num}$	[I.H.]
$\cdot \vdash e_2 : \texttt{num}$	[Inversion on Typing]
$\cdot \vdash \texttt{plus}(e_1'; e_2) : \texttt{num}$	[Typing rule for `plus`]
$\cdot \vdash \texttt{plus}(e_1'; e_2) : \tau$	[Take $\tau = \texttt{num}$]

- CASE: If $\mathcal{P}(e_2, e_2')$ and e_1 val and $e_2 \mapsto e_2'$, then $\mathcal{P}(\texttt{plus}(e_1; e_2), \texttt{plus}(e_1; e_2'))$

  I.H.: $\mathcal{P}(e_2, e_2')$ and e_1 val and $e_2 \mapsto e_2'$
  Assume $\cdot \vdash \texttt{plus}(e_1; e_2) : \tau$
  WTS: $\cdot \vdash \texttt{plus}(e_1; e_2') : \tau$

$\cdot \vdash \texttt{plus}(e_1; e_2) : \texttt{num}$	[Inversion on Typing]
$\cdot \vdash e_2 : \texttt{num}$	[Inversion on Typing]
$\cdot \vdash e_2' : \texttt{num}$	[I.H.]
$\cdot \vdash e_1 : \texttt{num}$	[Inversion on Typing]
$\cdot \vdash \texttt{plus}(e_1; e_2') : \texttt{num}$	[Typing rule for `plus`]
$\cdot \vdash \texttt{plus}(e_1; e_2') : \tau$	[Take $\tau = \texttt{num}$]

- CASE: If $n_1 \times n_2 = n$, then $\mathcal{P}(\texttt{times}(\texttt{num}[n_1]; \texttt{num}[n_2]), \texttt{num}[n])$

  [Identical to the proof for `plus`. Left as exercise.]

- CASE: If $\mathcal{P}(e_1, e_1')$ and $e_1 \mapsto e_1'$, then $\mathcal{P}(\texttt{times}(e_1; e_2), \texttt{times}(e_1'; e_2))$

  [Identical to the proof for `plus`. Left as exercise.]

- CASE: If $\mathcal{P}(e_2, e_2')$ and e_1 val and $e_2 \mapsto e_2'$, then $\mathcal{P}(\texttt{times}(e_1; e_2), \texttt{times}(e_1; e_2'))$

  [Identical to the proof for `plus`. Left as exercise.]

- CASE: If $\mathcal{P}(e_1, e_1')$ and $e_1 \mapsto e_1'$, then $\mathcal{P}(\texttt{let}(e_1; x.e_2), \texttt{let}(e_1'; x.e_2))$

  I.H.: $\mathcal{P}(e_1, e_1')$ and $e_1 \mapsto e_1'$
  Assume $\cdot \vdash \texttt{let}(e_1; x.e_2) : \tau$
  WTS: $\cdot \vdash \texttt{let}(e_1'; x.e_2) : \tau$

$\cdot \vdash e_1 : \tau_1$ s.t. $x : \tau_1 \vdash e_2 : \tau$	[Inversion on Typing]
$\cdot \vdash e_1' : \tau_1$ s.t. $x : \tau_1 \vdash e_2 : \tau$	[I.H.]
$\cdot \vdash \texttt{let}(e_1'; x.e_2) : \tau$	[Typing rule for `let`]

- CASE: If e_1 val, then $\mathcal{P}(\texttt{let}(e_1; x.e_2), [e_1/x]e_2)$

  I.H.: e_1 val
  Assume $\cdot \vdash \texttt{let}(e_1; x.e_2) : \tau$
  WTS: $\cdot \vdash [e_1/x]e_2 : \tau$

$\cdot \vdash e_1 : \tau_1$ s.t. $x : \tau_1 \vdash e_2 : \tau$	[Inversion on Typing]
$\cdot \vdash [e_1/x]e_2 : \tau$	[Substitution Lemma]

This completes the proof of preservation for the example language.

■