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

\[(Exp, Exp, Exp)Exp\]

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:

\[
\text{let}(1, x,x + 2) \text{ for let } x = 1 \text{ in } x + 2 \text{ end}
\]

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.

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:
2.1 Weakening and Induction Hypothesis with ABTs

Weakening and Induction Hypothesis with ABTs 2 STATICS

Typ \( \tau \ ::= \) 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 \( \tau \) using the typing context \( \Gamma \):

\[ \Gamma \vdash e : \tau \]

Example of our language’s typing judgment definition:

\[ \Gamma, x : \tau_1 \vdash x : \tau_1 \]
\[ \Gamma \vdash \text{num}[n] : \text{num} \]
\[ \Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num} \]
\[ \Gamma \vdash \text{plus}(e_1;e_2) : \text{num} \]
\[ \Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num} \]
\[ \Gamma \vdash \text{times}(e_1;e_2) : \text{num} \]
\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \]
\[ \Gamma \vdash \text{let}(e_1;x.e_2) : \tau_2 \]

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 \( \tau_1 \), if substituting \( e_1 \) for \( x \) with type \( \tau_1 \) in \( e_2 \) would give it type \( \tau_2 \), then let \( x = e_1 \) in \( e_2 \) has type \( \tau_2 \).

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

For all \( \Gamma \) and \( e \), there is at most one \( \tau \) such that \( \Gamma \vdash e : \tau \).

2.1 Weakening and Induction Hypothesis with ABTs

There are two structural properties that we would like show:

- **Substitution Lemma** says we can substitute a variable in a term with another term of same type without breaking typing of original term:

  \[ \text{If } \Gamma, x : \tau \vdash e' : \tau' \text{ and } \Gamma \vdash e : \tau \text{ then } \Gamma \vdash [e/x]e' : \tau'. \]

- **Weakening Lemma** says if we successfully typed a term under some context, adding more judgments into the context won’t break our previous typing:

  \[ \text{If } \Gamma \vdash e : \tau \text{ then } \Gamma, x : \tau' \vdash e : \tau \text{ given } x \text{ not already in } \Gamma. \]

Both properties can be proved from the typing judgments. They show that our typing judgments are “well-behaved”. Here we outline a proof of the weakening lemma in order to illustrate a recurring technicality. This proof is more lengthy because we deliberately included some “comments” to help you check your intuition.
2.1 Weakening and Induction Hypothesis with ABTs

Proof

We want to prove the Weakening Lemma stated above, proceed by induction on the derivation of typing judgment \( \Gamma \vdash e : \tau \):

All other cases, with the exception of let, are straightforward. They are left as an exercise.

The case of let works as follows:

\[
\begin{align*}
\Gamma &\vdash e_1 : \tau_1 & \Gamma, y : \tau_1 &\vdash e_2 : \tau_2 \\
\text{CASE:} & & &
\hline
\Gamma &\vdash \text{let}(e_1; y.e_2) : \tau_2
\end{align*}
\]

- Induction allows us to assume \( e \) takes the form of \( \text{let}(e_1; y.e_2) \) and \( \tau \) being \( \tau_2 \).
- Want to show: If \( \Gamma \vdash \text{let}(e_1; y.e_2) : \tau_2 \) then \( \Gamma, x : \tau' \vdash \text{let}(e_1; y.e_2) : \tau_2 \) given \( x \) not already in \( \Gamma \).
- Assume premises of the rule: \( \Gamma \vdash e_1 : \tau_1 \) and \( \Gamma, y : \tau_1 \vdash e_2 : \tau_2 \)
- Specialize Induction Hypothesis to the premises:
  - If \( \Gamma \vdash e_1 : \tau_1 \) then \( \Gamma, x : \tau' \vdash e_1 : \tau_1 \) given \( x \) not already in \( \Gamma \).
  - (!) If \( \Gamma, y : \tau_1 \vdash e_2 : \tau_2 \) then \( \Gamma, y : \tau_1, x : \tau' \vdash e_2 : \tau \) given \( x \) not already in \( \Gamma \).
- Now the premise we have allows us to immediately to apply IHs to obtain:
  \( \Gamma, y : \tau_1 \vdash e_2 : \tau_2 \) and \( \Gamma, y : \tau_1, x : \tau' \vdash e_2 : \tau \).

Since \( \Gamma \) is unordered, we can rearrange the second judgment \( \Gamma, x : \tau', y : \tau_1 \vdash e_2 : \tau \)
- Typing judgment of let allows us to derive the desired result:

\[
\begin{align*}
\hline
\Gamma, x : \tau' &\vdash e_1 : \tau_1 & \Gamma, x : \tau', y : \tau_1 &\vdash e_2 : \tau_2 \\
\Gamma, x : \tau' &\vdash \text{let}(e_1; y.e_2) : \tau_2
\end{align*}
\]

This proof has a problem: it only works if \( y \) and \( x \) are different variables! In the unfortunate case where the \( x \) is selected to be the same variable as \( y \), the context \( \Gamma, y : \tau_1, x : \tau' \) won’t make any sense: a variable can appear at most once in \( \Gamma \). The same problem has come up in the proof of \( \equiv_a \)’s transitivity.

To resolve the situation, we will admit the “renaming principal”: the judgment \( \Gamma \vdash \text{let}(e_1; y.e_2) : \tau_2 \) shows there exists a choice of the bound variable \( y \) that derives it the derivation, since the name of bounded variables are non-material, we should get to change that choice as long as the new choice is fresh. In other words, we get to rename the variable under binder in the premise!

- Pick a fresh \( y' \). Our second premise becomes \( \Gamma, y' : \tau_1 \vdash [y'/y]e_2 : \tau_2 \)
- Now we want to state the IH. The renaming principal also applies to the IH. The induction hypothesis should work for every fresh new choice of the bounded variable. Essentially we have an infinite number of IHs (one for each choice of the bounded variable). In this case, the IH is:

  If \( \Gamma, y' : \tau_1 \vdash e_2 : \tau_2 \) then \( \Gamma, y' : \tau_1, x : \tau' \vdash e_2 : \tau \) given \( x \) not already in \( \Gamma \).

We are back on track. The rest of the proof are just the same. However, the conclusion we draw would look like:

\[
\begin{align*}
\hline
\Gamma, x : \tau' &\vdash e_1 : \tau_1 & \Gamma, x : \tau', y' : \tau_1 &\vdash e_2 : \tau_2 \\
\Gamma, x : \tau' &\vdash \text{let}(e_1; y'.e_2) : \tau_2
\end{align*}
\]
It’s fine. Remember that name of the bounded variable is not material, it α-varies.

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 val states that e is a value. The judgment e \rightarrow e' states that expression e steps to expression e'.

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

- \text{num}[n] \text{ val}
  - n_1 + n_2 = n
  - \text{plus}(\text{num}[n_1]; \text{num}[n_2]) \rightarrow \text{num}[n]
  - e_1 \rightarrow e'_1
  - \text{plus}(e_1; e_2) \rightarrow \text{plus}(e'_1; e_2)
  - e_1 \text{ val} \quad e_2 \rightarrow e'_2
  - \text{plus}(e_1; e_2) \rightarrow \text{plus}(e'_1; e'_2)

- n_1 \times n_2 = n
  - \text{times}(\text{num}[n_1]; \text{num}[n_2]) \rightarrow \text{num}[n]
  - e_1 \rightarrow e'_1
  - \text{times}(e_1; e_2) \rightarrow \text{times}(e'_1; e_2)
  - e_1 \text{ val} \quad e_2 \rightarrow e'_2
  - \text{times}(e_1; e_2) \rightarrow \text{times}(e'_1; e'_2)

- \text{let}(e_1; x. e_2) \rightarrow \text{let}(e'_1; x. e_2)
  - e_1 \rightarrow e'_1
  - \text{let}(e_1; x. e_2) \rightarrow [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 $\tau$ 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 $\Gamma \vdash e : \text{num}$, then $e = \text{num}[n]$ 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 $e \rightsquigarrow e'$.

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

If $e \rightsquigarrow e'$ and $\cdot \vdash e : \tau$, 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 $P(e)$ be the property that either $e$ val or there exists an $e'$ such that $e \rightsquigarrow e'$.

To prove that if $\cdot \vdash e : \tau$, then either $e$ val, or there exists $e'$ such that $e \rightsquigarrow e'$, it is sufficient to prove the following:

- $P(\text{num}[n])$
- If $P(e_1)$, $P(e_2)$, $\cdot \vdash e_1 : \text{num}$, and $\cdot \vdash e_2 : \text{num}$, then $P(\text{plus}(e_1; e_2))$
- If $P(e_1)$, $P(e_2)$, $\cdot \vdash e_1 : \text{num}$, and $\cdot \vdash e_2 : \text{num}$, then $P(\text{times}(e_1; e_2))$
- If $P(e_1)$, $\cdot \vdash e_1 : \tau_1$, and $x : \tau_1 \vdash e_2 : \tau_2$, then $P(\text{let}(e_1; x.e_2))$

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 $\Gamma \vdash \text{plus}(e_1; e_2) : \tau$ then $\tau = \text{num}$, $\Gamma \vdash e_1 : \text{num}$, and $\Gamma \vdash e_2 : \text{num}$.
   - If $\Gamma \vdash \text{times}(e_1; e_2) : \tau$ then $\tau = \text{num}$, $\Gamma \vdash e_1 : \text{num}$, and $\Gamma \vdash e_2 : \text{num}$.
   - If $\Gamma \vdash \text{let}(e_1; x.e_2) : \tau$ then $\Gamma \vdash e_1 : \tau_1$ s.t. $\Gamma, x : \tau_1 \vdash e_2 : \tau$.

Inversion lemmas can be easily proved by induction on the typing judgment. Essentially there is one and only one typing rule matches the form of the given typing judgment. All other cases are vacuous. It is possible to “inline” appeals to inversion lemmas with an extra induction on typing rules. However, as you will see in the following proof, it is so commonly used that it make sense to abstract it out as a separate lemma.

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 $\vdash e : \tau$, then $\vdash e' : \tau$. Proceed by rule induction on judgment $e \mapsto e'$.

- **Case**: If $n_1 + n_2 = n$, then $\mathcal{P}(\text{plus}(\text{num}[n_1]; \text{num}[n_2]), \text{num}[n])$
  
  I.H.: $n_1 + n_2 = n$
  
  Assume $\vdash \text{num}[n_1]; \text{num}[n_2] : \tau$

  WTS: $\vdash \text{num}[n] : \tau$

  $\vdash \text{plus}(\text{num}[n_1]; \text{num}[n_2]) : \text{num}$ [Inversion on Typing]

  $\vdash \text{num}[n] : \text{num}$ [Typing rule for num]

  $\vdash \text{num}[n] : \tau$ [Take $\tau = \text{num}$]

- **Case**: If $\mathcal{P}(e_1, e'_1)$ and $e_1 \mapsto e'_1$, then $\mathcal{P}(\text{plus}(e_1; e_2), \text{plus}(e'_1; e_2))$
  
  I.H.: $\mathcal{P}(e_1, e'_1)$ and $e_1 \mapsto e'_1$
  
  Assume $\vdash \text{plus}(e_1; e_2) : \tau$

  WTS: $\vdash \text{plus}(e'_1; e_2) : \tau$

  $\vdash \text{plus}(e_1; e_2) : \text{num}$ [Inversion on Typing]

  $\vdash e_1 : \text{num}$ [Inversion on Typing]

  $\vdash e'_1 : \text{num}$ [I.H.]

  $\vdash e_2 : \text{num}$ [Inversion on Typing]

  $\vdash \text{plus}(e'_1; e_2) : \text{num}$ [Typing rule for plus]

  $\vdash \text{plus}(e'_1; e_2) : \tau$ [Take $\tau = \text{num}$]

- **Case**: If $\mathcal{P}(e_2, e'_2)$ and $e_1 \text{ val}$ and $e_2 \mapsto e'_2$, then $\mathcal{P}(\text{plus}(e_1; e_2), \text{plus}(e_1; e'_2))$
  
  I.H.: $\mathcal{P}(e_2, e'_2)$ and $e_1 \text{ val}$ and $e_2 \mapsto e'_2$
  
  Assume $\vdash \text{plus}(e_1; e_2) : \tau$

  WTS: $\vdash \text{plus}(e_1; e'_2) : \tau$

  $\vdash \text{plus}(e_1; e_2) : \text{num}$ [Inversion on Typing]

  $\vdash e_2 : \text{num}$ [Inversion on Typing]

  $\vdash e'_2 : \text{num}$ [I.H.]

  $\vdash e_1 : \text{num}$ [Inversion on Typing]

  $\vdash \text{plus}(e_1; e'_2) : \text{num}$ [Typing rule for plus]

  $\vdash \text{plus}(e_1; e'_2) : \tau$ [Take $\tau = \text{num}$]
Case: If $n_1 \times n_2 = n$, then $P(\text{times}(\text{num}[n_1]; \text{num}[n_2]), \text{num}[n])$

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

Case: If $P(e_1, e'_1)$ and $e_1 \mapsto e'_1$, then $P(\text{times}(e_1; e_2), \text{times}(e'_1; e_2))$

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

Case: If $P(e_2, e'_2)$ and $e_1 \text{ val}$ and $e_2 \mapsto e'_2$, then $P(\text{times}(e_1; e_2), \text{times}(e_1; e'_2))$

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

Case: If $P(e_1, e'_1)$ and $e_1 \mapsto e'_1$, then $P(\text{let}(e_1; x.e_2), \text{let}(e'_1; x.e_2))$

I.H.: $P(e_1, e'_1)$ and $e_1 \mapsto e'_1$
Assume $\cdot \vdash \text{let}(e_1; x.e_2) : \tau$

WTS: $\cdot \vdash \text{let}(e'_1; x.e_2) : \tau$

\[\vdash e_1 : \tau_1 \text{ s.t. } x : \tau_1 \vdash e_2 : \tau \quad \text{[Inversion on Typing]}\]
\[\vdash e'_1 : \tau_1 \text{ s.t. } x : \tau_1 \vdash e_2 : \tau \quad \text{[I.H.]}\]
\[\vdash \text{let}(e'_1; x.e_2) : \tau \quad \text{[Typing rule for let]}\]

Case: If $e_1 \text{ val}$, then $P(\text{let}(e_1; x.e_2), [e_1/x]e_2)$

I.H.: $e_1 \text{ val}$
Assume $\cdot \vdash \text{let}(e_1; x.e_2) : \tau$

WTS: $\cdot \vdash [e_1/x]e_2 : \tau$

\[\vdash e_1 : \tau_1 \text{ s.t. } x : \tau_1 \vdash e_2 : \tau \quad \text{[Inversion on Typing]}\]
\[\vdash [e_1/x]e_2 : \tau \quad \text{[Substitution Lemma]}\]

This completes the proof of preservation for the example language.

5 Appendix: The Y Combinator

To see the idea behind expressing general recursion in Untyped Lambda Calculus (ULC), we'll use the example of a simple function that loops forever. In Standard ML, we'd write this function as

```ml
fun f x = f x
```

Note that we need to refer to $f$ in the body of $f$. One way we can achieve this in a language without built-in recursion is to pass the function to itself as its first argument. So we would have

```ml
val f' = fn f => fn x => f f x
val f = f' f'
```

This achieves general recursion, but if you typed it into SML/NJ’s REPL, it would not compile. This is because in a language like SML, this self-referential expression is not well-typed. This is evident from the fact that the $f'$ function immediately takes an argument of the same type as itself. The corresponding type must be infinite. However, in ULC, this does not matter. We can define

$$\lambda f. \lambda x \ldots (f(f) \ldots$$
as we please.

However, writing this by hand is cumbersome, and so we create a function that performs this passing-function-to-itself operation for us. This is known as a fixed-point combinator. For example, the well-known $\text{Y}$ combinator performs this operation:

$$\text{Y} \triangleq \lambda F. (\lambda f. F(f f)) (\lambda f. F(f f))$$

With $\text{Y}$, we can define $f$ in a more natural way instead (this time in lambda calculus):

$$f = \text{Y}(\lambda f. \lambda x. f x)$$

Why is this equivalent to the previous definition? Notice what happens when we $\beta$-convert this expression a bit, substituting $(\lambda f. \lambda x. f x)$ for $F$.

$$\begin{align*}
(\lambda f. (\lambda f. \lambda x. f x)(f f))(\lambda f. (\lambda f. \lambda x. f x)(f f)) &= \beta \\
(\lambda f. (\lambda x. f f x))(\lambda f. (\lambda x. f f x))
\end{align*}$$

What is $\lambda f. (\lambda x. f f x)$? It’s $f'$! When making a recursive call, the function passes itself into itself as its first argument. Now, to create $f$, all we need to do is apply $f'$ to itself, which is exactly what the above expression does.

This particular fixed-point combinator was discovered by Haskell Curry, and has the following property:

$$\text{Y} f = f(\text{Y} f) = f(f(\text{Y} f)) = \ldots$$

If we give $\text{Y}$ a self-referential function $f$, it produces an output which is equivalent to its own infinite iteration under $f$. Mathematically, this is known as a fixed point of $f$, an input which is identical to its corresponding output. This construct allows us to create general recursive expressions. Notice especially how easy it is to introduce divergent (non-terminating) computation through this combinator. With $\text{Y}$, we can easily turn self-referential functions into recursive ones. An added advantage is the ease of defining $f$. Whereas before we had to apply the self-reference explicitly as in $f(f)$, this is no longer necessary with the $\text{Y}$ combinator; we may just write $f$.

The $\text{Y}$ combinator is not meant as a particularly practical method of writing recursive functions, nor is ULC particularly practical as a programming language. However, it is a theoretically powerful construct that encodes recursion directly into the lambda calculus.