Higher Order Abstract Syntax and λ-Calculus

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

Abstract Syntax for Variables

Recall the `let` construct we added in the last lecture. We shall extend our AST and parsing relation to include a definition for `let` and variables.

Let Syntax

$$\frac{x\ \textbf{Ident}}{x\ \textbf{Atom} \longleftrightarrow (\texttt{Var}\ x)\ \textbf{AST}}$$

$$\frac{x\ \textbf{Ident} \qquad e_1\ \textbf{SExp} \longleftrightarrow a_1\ \textbf{AST} \qquad e_2\ \textbf{SExp} \longleftrightarrow a_2\ \textbf{AST}}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}\ \textbf{Atom} \longleftrightarrow (\texttt{Let}\ x\ a_1\ a_2)\ \textbf{AST}}$$

First Order Abstract Syntax

Consider the following two pieces of abstract syntax:

```
(Let "x" (Num 5) (Plus (Num 4) (Var "x")))
```

```
(Let "y" (Num 5) (Plus (Num 4) (Var "y")))
```

This demonstrates some problems with our abstract syntax approach.

1. Substitution capture is a problem.
2. α-equivalent expressions are not equal. Determining if an expression is α-equivalent requires us to search for a consistent α-renaming of variables.
3. No distinction is made between binding and usage occurrences of variables. This means that we must define substitution by hand on each type of expression we introduce.
4. Scoping errors cannot be easily detected — malformed syntax is easy to write.

de Bruijn Indices

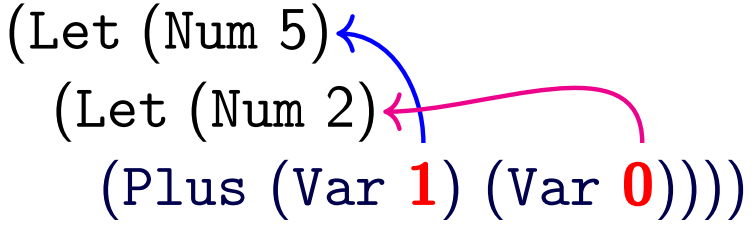
One popular approach to address the first issue is *de Bruijn indices*.

Key Idea

1. Remove all identifiers from binding expressions like `Let`.
2. Replace the identifier in a `Var` with a number indicating how many binders we must skip in order to find the binder for that variable.

```
(Let "a" (Num 5)
 (Let "y" (Num 2)
  (Plus (Var "a") (Var "y"))))
```

```
(Let (Num 5)
  (Let (Num 2)
    (Plus (Var 1) (Var 0))))
```



Debruijnification

Algorithm

Given a piece of *first order abstract syntax* with explicit variable names, we can convert to de Bruijn indices by keeping a *stack* of variable names, pushing onto the stack at each `Let` and popping after the variable goes out of scope. When a usage occurrence is encountered, replace the variable name with its first position in the stack (starting at the top of the stack).

This approach naturally handles shadowing. It's also possible, but harder, to have de Bruijn indices going in the other direction (from the bottom of the stack, upwards).

de Bruijn Substitution

Substitution is now capture avoiding by definition.

$$
\begin{aligned}
(\texttt{Num}\ i)[n := t] &= (\texttt{Num}\ i) \\
(\texttt{Plus}\ a\ b)[n := t] &= (\texttt{Plus}\ a[n := t]\ b[n := t]) \\
(\texttt{Times}\ a\ b)[n := t] &= (\texttt{Times}\ a[n := t]\ b[n := t]) \\
(\texttt{Var}\ m)[n := t] &= \begin{cases} t & \text{if } n = m \\ (\texttt{Var}\ (m-1)) & \text{if } m > n \\ (\texttt{Var}\ m) & \text{otherwise} \end{cases} \\
(\texttt{Let}\ e_1\ e_2)[n := t] &= (\texttt{Let}\ e_1[n := t]\ e_2[n+1 := t_{\uparrow 0}])
\end{aligned}
$$

Where $e_{\uparrow n}$ is an *up-shifting* operation defined as follows:

$$
\begin{aligned}
(\texttt{Num}\ i)_{\uparrow n} &= (\texttt{Num}\ i) \\
(\texttt{Plus}\ a\ b)_{\uparrow n} &= (\texttt{Plus}\ a_{\uparrow n}\ b_{\uparrow n}) \\
(\texttt{Times}\ a\ b)_{\uparrow n} &= (\texttt{Times}\ a_{\uparrow n}\ b_{\uparrow n}) \\
(\texttt{Var}\ m)_{\uparrow n} &= \begin{cases} (\texttt{Var}\ (m+1)) & \text{if } m \geq n \\ (\texttt{Var}\ m) & \text{otherwise} \end{cases} \\
(\texttt{Let}\ e_1\ e_2)_{\uparrow n} &= \left(\texttt{Let}\ e_{1\uparrow n}\ e_{2\uparrow n+1}\right)
\end{aligned}
$$

Examining de Bruijn indices

How do de Bruijn indices stack up against our explicit names in terms of the problems we identified?

1. Substitution capture solved.
2. α-equivalent expressions are now equal.
3. We still must define substitution machinery by hand for each type of expression.
4. It is still possible to make malformed syntax – indices that overflow the stack, for example.

Two out of four isn't bad, but can we do better by changing the term language?

Higher Order Terms

We shall change our term language to include built-in notions of variables and binding.

$$
\begin{array}{rcll}
t & ::= & \texttt{Symbol} & \textit{(symbols)} \\
 & | & x & \textit{(variables)} \\
 & | & t_1\ t_2 & \textit{(application)} \\
 & | & x.\ t & \textit{(binding or abstraction)}
\end{array}
$$

As in Haskell, we shall say that application is left-associative, so

$$(\texttt{Plus (Num 3) (Num 4)}) = ((\texttt{Plus (Num 3)}) \texttt{ (Num 4)})$$

Now the binding and usage occurrences of variables are distinguished from regular symbols in our term language. Let's see what this lets us do...

Representing Let

$$\frac{a_1\ \textbf{AST} \qquad a_2\ \textbf{AST}}{(\texttt{Let}\ a_1\ (x.\ a_2))\ \textbf{AST}}$$

We no longer need a rule for variables, because they're baked into the structure of terms.

How would we represent this AST in Haskell?

$$\begin{array}{rcl} \textbf{data}\ \mathit{AST} & = & \text{Num}\ \mathit{Int} \\ & | & \text{Plus}\ \mathit{AST}\ \mathit{AST} \\ & | & \text{Times}\ \mathit{AST}\ \mathit{AST} \\ & | & \text{Let}\ \mathit{AST}\ ???(\mathit{AST} \to \mathit{AST}) \end{array}$$

So `let` $x = 3$ `in` $x + 2$ `end` becomes, in Haskell:

$$(\texttt{Let}\ (\texttt{Num}\ 3)\ (\lambda x \to \texttt{Plus}\ x\ (\texttt{Num}\ 2))$$

Substitution

We can now define substitution across all terms in the meta-logic:

$$
\begin{aligned}
\texttt{Symbol}[x := e] &= \texttt{Symbol} \\
y[x := e] &= \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
(t_1\ t_2)[x := e] &= t_1[x := e]\ t_2[x := e] \\
(y.\ t)[x := e] &= \begin{cases} (y.\ t) & \text{if } x = y \\ (y.\ t[x := e]) & \text{if } y \notin \mathrm{FV}(e) \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}
$$

Where $\mathrm{FV}(\cdot)$ is the set of all free variables in a term:

$$
\begin{aligned}
\mathrm{FV}(\texttt{Symbol}) &= \emptyset \\
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(t_1\ t_2) &= \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2) \\
\mathrm{FV}(x.\ t) &= \mathrm{FV}(t) \setminus \{x\}
\end{aligned}
$$

Cheating Outrageously

Substitution capture is still a problem in the substitution we just defined but it is not our problem. Because substitution is defined in the meta-language, it's the job of the implementors of the meta-language (if any) to deal with issues about capture.

- When Haskell is our meta-language, it's the job of the GHC developers to sort out capture.
- When we are doing proofs in our meta-logic, there is no implementation, so we can just say that we assume α-equivalent terms to be equal, and therefore assume that variables are always renamed to avoid capture.

So, we have solved the problem by making it someone else's problem. **Outrageous cheating!**

Evaluating All Approaches

	HOAS		FOAS	
	Proofs	**Haskell**	**Strings**	**de Bruijn**
Capture	Cheat	Cheat	Problem	Solved
α-equivalence	Cheat	Cheat	Problem	Solved
Generic subst.	Solved	Solved	Problem	Problem
Malformed syntax	Cheat	Cheat	Problem	Problem

- In embedded languages and in proofs, HOAS is very common.
- In conventional language implementations and machine-checked formalisations, de Bruijn indices are more popular.
- In your assignments, strings will be used 

λ-Calculus

The term language we just defined is almost a full featured programming language.
Just change the syntax slightly:

$$
\begin{array}{rcll}
t & ::= & \texttt{Symbol} & \textit{(symbols)} \\
 & | & x & \textit{(variables)} \\
 & | & t_1\ t_2 & \textit{(application)} \\
 & | & \lambda x.\ t & \textit{(}\lambda\textit{-abstraction)}
\end{array}
$$

There is just one rule to evaluate terms, called β-*reduction*:

$$(\lambda x.\ t)\ u \quad \mapsto_\beta \quad t[x := u]$$

Just as in Haskell, $(\lambda x.\ t)$ denotes a function that, given an argument for x, will return t.

β-reduction

β-reduction is a *congruence*:

$$\frac{}{(\lambda x.\ t)\ u \mapsto_\beta t[x := u]}$$

$$\frac{t \mapsto_\beta t'}{s\ t \mapsto_\beta s\ t'} \qquad \frac{s \mapsto_\beta s'}{s\ t \mapsto_\beta s'\ t} \qquad \frac{t \mapsto_\beta t'}{\lambda x.\ t \mapsto_\beta \lambda x.\ t'}$$

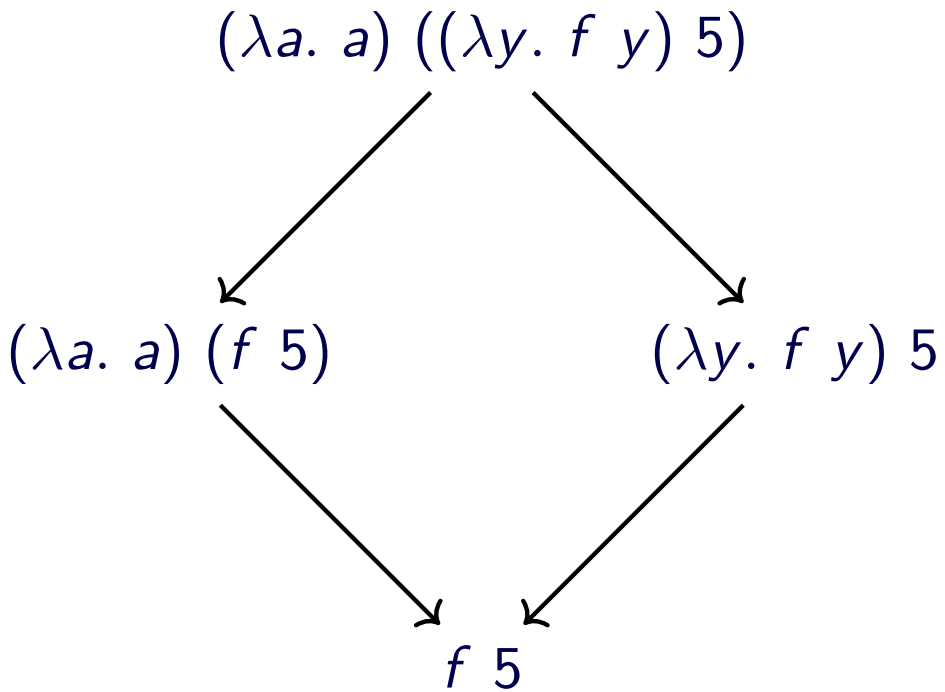
This means we can pick any reducible subexpression (called a *redex*) and perform β-reduction.

Example:

$$\begin{aligned}(\lambda x.\ \lambda y.\ f\ (y\ x))\ 5\ (\lambda x.\ x) &\mapsto_\beta (\lambda y.\ f\ (y\ 5))\ (\lambda x.\ x)\\ &\mapsto_\beta f\ ((\lambda x.\ x)\ 5)\\ &\mapsto_\beta f\ 5\end{aligned}$$

Confluence

Supposing we arrive via one reduction path to an expression that cannot be reduced further (called a *normal form*), then any other reduction path will result in the same normal form.



Equivalence

Confluence means we can define another notion of *equivalence*, which equates more than α-equivalence. Two terms are $\alpha\beta$-*equivalent*, written $s \equiv_{\alpha\beta} t$ if they β-reduce to α-equivalent normal forms.

η

There is also another equation that cannot be proven from β-equivalence alone, called η-conversion:

$$(\lambda x.\ f\ x) \mapsto_\eta f$$

Adding this reduction to the system preserves confluence and uniqueness of normal forms, so we have a notion of $\alpha\beta\eta$-*equivalence* also.

Normal Forms

Does every term in λ-calculus have a normal form?

$$(\lambda x.\ x\ x)(\lambda x.\ x\ x)$$

Try to β-reduce this! (the answer is that it doesn't have a normal form)

Why learn this stuff?

- λ-calculus is a *Turing-complete* programming language. We'll explain how to use it in the extension lecture.
- λ-calculus is the foundation for every functional programming language and some non-functional ones.
- λ-calculus is the foundation of *Higher Order Logic* and *Type Theory*, the two main foundations used for mathematics in interactive proof assistants.
- λ-calculus is the smallest example of a usable programming language, so it's good for teaching about programming languages.