An Introduction to the Lambda Calculus

David Walker
Department of Computer Science
Princeton University

February 7, 2002

Foundations of Language Based Security

- Prof: David Walker
- email: dpw@cs.princeton.edu
- phone: 258-7654
- web site: HTTP://www.cs.princeton.edu/ dpw/spring02/

This Lecture

- The Simply Typed Lambda Calculus
  - Syntax
  - Operational semantics
  - Typing rules

The Simply Typed Lambda Calculus

Let's start with the syntax...

- The abstract syntax is the set of things you can say in the language. These things might not make any sense, but they can be said.

- For the simply typed lambda calculus, we actually need to define two sets, the set of *terms* and the set of *types* that classify the terms.

- We will start out by assuming the existence of a set of variables $\mathcal{V}$ and a set of constants $\mathcal{C}$

- We will let x range over variables and c range over constants.

- We can specify the set of types and terms using BNF notation.

$$\begin{array}{lcll} \text{types} & \tau & ::= & int \mid \tau_1 \to \tau_2 \\ \text{terms} & e & ::= & x \mid c \mid \lambda x : \tau . e \mid e_1 e_2 \end{array}$$

Translation from Bob Harper's book

- If you are familiar with Andrew's (Bob Harper's) notation just do the following conversion:
- $\lambda x : \tau . e \equiv$ `fun _ (`$x{:}\tau$`) :` τ' `is` e `end`
- $e_1\ e_2 \equiv$ `apply` (e_1, e_2)
- The simply-typed lambda calculus is MinML without recursion (at least for now...).
- More generally, when people talk about the "lambda calculus" they can mean pretty much any small formal language based around the notion of higher-order functions. There is no mystery here.

Syntax Alternative

- BNF is convenient shorthand for specifying the syntax of a language.
- Another way to specify the syntax of a language is to define a collection of judgments that specify which objects are in the language
- $\tau \;\in\; \mathcal{T}$
- $e \;\in\; \mathcal{E}$

$$\frac{}{\mathit{int} \;\in\; \mathcal{T}}\;\mathit{intF}$$

$$\frac{\tau_1 \;\in\; \mathcal{T} \qquad \tau_2 \;\in\; \mathcal{T}}{\tau_1 \to \tau_2 \;\in\; \mathcal{T}}\;{\to}F$$

- The set of types is the *smallest* set you can get by starting with rule intF and successively applying rule ${\to}F$.
- For example:

$$\frac{}{\mathit{int} \;\in\; \mathcal{T}}$$

$$\frac{\dfrac{}{\mathit{int} \;\in\; \mathcal{T}} \qquad \dfrac{}{\mathit{int} \;\in\; \mathcal{T}}}{\mathit{int} \to \mathit{int} \;\in\; \mathcal{T}}$$

$$\frac{\dfrac{\dfrac{}{\mathit{int} \;\in\; \mathcal{T}} \qquad \dfrac{}{\mathit{int} \;\in\; \mathcal{T}}}{\mathit{int} \to \mathit{int} \;\in\; \mathcal{T}} \qquad \dfrac{}{\mathit{int} \;\in\; \mathcal{T}}}{(\mathit{int} \to \mathit{int}) \to \mathit{int} \;\in\; \mathcal{T}}$$

The Simply Typed Lambda Calculus – Syntax

- Of course, we can do the same thing for terms.

$$\frac{x \ \in \ \mathcal{V}}{x \ \in \ \mathcal{E}} \ (xF)$$

$$\frac{c \ \in \ \mathcal{C}}{c \ \in \ \mathcal{E}} \ (cF)$$

$$\frac{x \ \in \ \mathcal{V} \quad \tau \ \in \ \mathcal{T} \quad e \ \in \ \mathcal{E}}{\lambda x : \tau . e \ \in \ \mathcal{E}} \ (\lambda F)$$

$$\frac{e_1 \ \in \ \mathcal{E} \quad e_2 \ \in \ \mathcal{E}}{e_1 e_2 \ \in \ \mathcal{E}} \ (\mathtt{app}F)$$

Variables and Scope

- The expression `let` $x = e_1$ `in` e_2 is an abbreviation for $(\lambda x{:}\tau.e_2)e_1$.
- Example program: `let` $id = \lambda x{:}int.x$ `in` $id\ 3$
- Chances are, the programmer named the function "*id*" for a reason. However, when we want to type check the program, compile it or understand what it means (semantically), the name "*id*" is irrelevant.
- Moreover, if the programmer wants to change the name of the function, it shouldn't behave differently (provided all uses of the name are changed consistently).
- In other words,

  $$\texttt{let}\ id = \lambda x{:}int.x\ \texttt{in}\ id\ 3$$

  is just the same as

  $$\texttt{let}\ foo = \lambda y{:}int.y\ \texttt{in}\ foo\ 3$$

- Bound variables:
  - The x in $\lambda x{:}\tau.e$ is *bound* and its *scope* is e.
- As discussed above, we normally treat terms that only differ in the names of their bound variables as the same.

- Another way of saying this is: “we identify terms up to α-conversion of bound variables.
- We call the variables that are not bound, *free*. Which variables are free in the following?

$$\texttt{let}\ id = \lambda x{:}int \to int.x\ y\ \texttt{in}\ id\ z$$

- We call an expression *closed* if it contains no free variables.

Substitution

- The main operation in the simply-typed lambda calculus is *substitution*.
- We write $e[e'/x]$ for the term that results in replacing all free occurrences of x with e' (being careful to α-convert e to avoid capturing any free variables in e)
- If I wanted to be really careful then I would define substitution the formally by induction on the structure of e (See Harper for details)

Semantics

- The semantics of a language usually comes in two parts:
  - The *operational* (or *dynamic*) semantics specifies how to execute a program and what sort of errors can occur while running a program.
  - The *static* semantics specifies which programs make sense.

Operational Semantics

- There are a number of different formulations of the operational semantics of the simply-typed lambda calculus. Here is one.
- The operational semantics is a relation between terms, which we write as $e \longmapsto e'$.
- Our operational semantics will be *call-by-value*
- The values:

$$\text{(values)} \quad v \quad ::= \quad x \mid c \mid \lambda x{:}\tau.e$$

- We define the relation inductively as follows.

$$\frac{}{(\lambda x{:}\tau.e)\ v \longmapsto e[v/x]}\ (\beta \longmapsto)$$

$$\frac{e_1 \longmapsto e_1'}{e_1 e_2 \longmapsto e_1' e_2}\ (cong1 \longmapsto)$$

$$\frac{e_2 \longmapsto e_2'}{v_1 e_2 \longmapsto v_1 e_2'}\ (cong2 \longmapsto)$$

- If we have some constants with function type in our language then they must have operational rules associated with them:

$$\frac{v_1, v_2 \text{ are integers} \quad k = v_1 + v_2}{(+)\ v_1\ v_2 \longmapsto k}$$

- To extend the relation to a multi-step relation $e \longmapsto^* e'$ we have the following rules:

$$\frac{}{e \longmapsto^* e}\ (reflex \longmapsto^*)$$

$$\frac{e \longmapsto e'' \quad e'' \longmapsto^* e'}{e \longmapsto^* e'}\ (trans \longmapsto^*)$$

- How would we change it so it was call-by-name?

Properties of the Dynamic Semantics

Lemma 1 *For every closed expression e, there is at most one e' such that $e \longmapsto e'$.*

Proof:

?

$\square$

Lemma 2 *For every closed expression e, there is at most one v such that $e \longmapsto^* v$.*

Proof:

?

$\square$

Errors

- Whenever a transition in the operational semantics is undefined, we will consider it an error. It is the job of the type system to rule out these sorts of errors.

- For example:

  $$(3)\ (17) \qquad \text{The number 3 is used as a function.}$$

- Expressions that are not values and do not step to any other expression are called *stuck*.

Static Semantics

- To define the static semantics of our language, we need to introduce a typing judgment for terms: $\Gamma \vdash e : \tau$
- Γ is a finite map from variables to types.
  - By definition, a variable x can only occur once in the domain of Γ.
  - We use the notation $\Gamma, x : \tau$ to extend Γ with $x : \tau$. This doesn't make sense if x is already in Γ, so we will have to be careful.
  - Sometimes, I like to specify the syntax of typing environments:

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

- We will also assume some signature Σ that maps constants to their types.
- Technically, typing judgments should look like: $\Gamma \vdash_\Sigma e : \tau$ but since Σ will never change, we omit it from the judgment.

Typing Rules

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Sigma(c) = \tau}{\Gamma \vdash c : \tau}$$

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

- Finally, the rule for lambda abstraction. To preserve the invariant that typing contexts Γ contain no repeated variables, we can implicitly α-convert bound variables before entering them into the context.

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \; (x \notin \mathsf{Dom}(\Gamma))$$

Properties of the Static Semantics

For the purposes of reasoning about the static semantics, we will fix the collection of constants to be the integers and addition. Both sorts of constants have the appropriate typing rules. We will assume that addition is a total function from integers to integers.

$$\text{(constants)} \quad c \quad ::= \quad 0 \mid 1 \mid 2 \mid \cdots \mid i \mid -1 \mid -2 \mid \cdots \mid -i \mid +$$

The first lemma provides a convenient property of the typing context. It is a form of *structural rule.* There are other structural rules we could consider proving, but this is all we need for now.

Lemma 3 (Weakening) *If* $\Gamma \vdash e : \tau$ *then* $\Gamma, x{:}\tau' \vdash e : \tau$.

The next important lemma explains what we can learn about a typing derivation, given the shape of the expression being typed.

Lemma 4 (Inversion)

- *If* $\Gamma \vdash x : \tau$ *then* $\Gamma(x) = \tau$.
- *If* $\Gamma \vdash i : \tau$ *then* $\tau = \mathit{int}$.
- *If* $\Gamma \vdash \lambda x{:}\tau_1.e : \tau$ *then* $\tau = \tau_1 \to \tau_2$ *and* $\Gamma, x{:}\tau \vdash e : \tau_2$.
- *If* $\Gamma \vdash e_1 e_2 : \tau$ *then* $\Gamma \vdash e_1 : \tau_1 \to \tau$ *and* $\Gamma \vdash e_2 : \tau_1$.

Properties of Static Semantics

An analogous lemma tells us what we can learn about a value given the shape of its type.

Lemma 5 (Canonical Forms) *Suppose* $\cdot \vdash v : \tau$.

- *If* $\tau = int$ *then* $v = i$.
- *If* $\tau = \tau_1 \rightarrow \tau_2$ *then* $v = \lambda x{:}\tau.e$.

Finally, the ubiquitous substitution principal.

Lemma 6 (Substitution) *If* $\Gamma, x{:}\tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash e[e'/x]$.

Type Safety

Type safety describes the relationship between the static and dynamic semantics of a programming language. It boils down to two key properties.

- **Preservation** If $\cdot \vdash e : \tau$ and $e \longmapsto e'$ then $\cdot \vdash e' : \tau$.
- **Progress** If $\cdot \vdash e : \tau$ then either e is a value or there exists a e' such that $e \longmapsto e'$.

Proof:

Preservation: By induction on the rules defining $e \longmapsto e'$.
Progress: By induction on the typing rules.

$\square$

Theorem 1 (Type Safety) *If $\cdot \vdash e : \tau$ and $e \longmapsto^* e'$ then e' is not stuck.*

Proof:

?

$\square$

Homework

- Make sure you know all of the stuff we did today cold.
- Review Harper, chapters 5-8.
- Next time, we will study some new proof techniques for the lambda calculus called logical relations and show that all well-typed lambda terms actually terminate. This is a very deep and powerful result. Go types!