MinML: A MiniMaL Functional Language

Robert Harper
Carnegie Mellon University
Spring, 1999

Overview

- Introduce **static** and **dynamic** semantics.
  - Static semantics = combinatorial strictures on abstract syntax
  - Dynamic semantics = language-level description of evaluation
- Together these constitute a **formal definition** of a language.

Overview

- Introduce a **language-based** model of computation.
  - Semantics is "intrinsic" to the language (no reference to implementation).
- Contrast with **machine-based** models of computation.
  - Semantics defined by mapping onto an abstract machine.

Overview

- This leads to a careful analysis of **type safety**.
  - What is a "safe" language?
  - Which languages are "safe"? Which are not?
- We will give a precise formulation and proof of safety for MinML.

MinML, the E. Coli of PL's

- We'll study a tiny fragment of ML, called **MinML**.
  - Based on Plotkin's PCF language.
  - Two base types (`int` and `bool`).
  - Partial function types.
- Powerful enough to define all computable functions on integers.

MinML Abstract Syntax

- We'll be cavalier about the **concrete**, **surface**, **syntax**.
  - (But see assignment 1 for full details!)
- However, we'll be very precise about the **abstract**, or **deep**, **syntax**.
  - First-order: ignore spelling, layout, etc.
  - Higher-order: consider binding structure.

MinML Types

- Abstract syntax of **types** is defined by these rules:

$$\frac{}{\texttt{int}} \qquad \frac{}{\texttt{bool}} \qquad \frac{\tau_1 \quad \tau_2}{\tau_1 \rightarrow \tau_2}$$

Abstract Syntax of Expr's

$$\frac{}{x} \quad \frac{}{n} \quad \frac{}{\texttt{true}} \quad \frac{}{\texttt{false}}$$

$$\frac{e_1 \quad \ldots \quad e_n}{o(e_1,\ldots,e_n)} \qquad \frac{e \quad e_1 \quad e_2}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2}$$

Abstract Syntax of Expr's

$$\frac{e}{\texttt{fun } f(x:\tau_1):\tau_2 \texttt{ is } e \texttt{ end}}$$

$$\frac{e_1 \quad e_2}{\texttt{apply}(e_1,e_2)}$$

Abstract Syntax in ML

- MinML Types

```
datatype typ =
  INT | BOOL | ARROW of typ * typ
```

- MinML Expressions

```
datatype exp =
  Var of string | Int of int | Var of string |
  Bool of bool | If of exp * exp * exp |
  Primop of primop * exp list |
  Apply of exp * exp |
  Fun of string * string * typ * typ * exp
```


Abstract Syntax

- This defines the **first-order** abstract syntax of MinML.
  - Expressions as "trees".
  - Abstracts away surface details.
- Now we consider **higher-order** abstract syntax.
  - Captures binding and scope.

Freedom and Bondage

- Variables are introduced by **binders**.
  - "fun" is the only binder in MinML.
- **Distinct** binders introduce **distinct** variables.
  - Even if they have the same spelling!
  - Variable names are immaterial; they are used only to resolve binding structure.

Scope Resolution

- A variable is bound by the **innermost** binder that introduces it.
  - fun f(x:int):int is f(x-1) end
  - fun f(x:int):int->int is
    fun g(y:int):int is x+y
- Nested binders **puncture** scope.
  - fun f(x:int):int->int is
    fun g(x:int):int is f(x-1)

Variables as Pronouns

- The **names** of bound variables don't matter, provided clashes are avoided.
  - fun f(x:int):int is f(x-1) end =
    fun g(y:int):int is g(y-1) end
  - fn (x:int) in fn (y:int) in x end <>
    fn (x:int) in fn (x:int) in x end
- We **identify** expressions that differ only in names of variables.

Structural Equivalence

- Formally, we define e ≡ e'iff e and e' differ **only** in the names of their bound variables.
  - For formal definition, see notes.
  - Eg, fn x in x+1 end ≡ fn y in y+1 end
- Officially, we work with **equivalence classes** of expressions modulo this relation.

Free Variables

- Program fragments contain **unbound**, or **free**, variables.
  - Complete programs are **closed**, i.e. have no free variables.
- FV(e) defined by induction on e.
  - FV(x) = {x};
  - FV(n) = FV(true) = FV(false) = {}
  - FV(o(e1,...,en)) = FV(e1) ∪ ... ∪ FV(en)

Free Variables

- FV(e), cont'd
  - FV(if e then e1 else e2 fi) =
    FV(e) ∪ FV(e1) ∪ FV(e2)
  - FV(apply(e1,e2)) = FV(e1) ∪ FV(e2)
  - FV(fun f(x:t):t' is e end) = FV(e) \ {f, x}
- (Check that this defines a function on expressions by structural induction on e.)

Closed Substitution

- For e **closed**, define [e/x]e' to be the result of **substituting** e for **free occurrences** of x in e'.
  - [3/x] (x+x) = 3+3
  - [3/y] fun f(x:int):int is f(y-1) end =
    fun f(x:int):int is f(3-1) end
  - [3/x] fun f(x:int):int is f(x-1) end =
    fun f(x:int):int is f(x-1) end (!)

Closed Substitution

- Define [e/x]e' by induction on e'.
  - [e/x] x = e;
  - [e/x] y = y (if y ≠ x);
  - [e/x] n = n;
  - [e/x] true = true, [e/x] false = false
  - [e/x] o(e1,…,en) = o ([e/x]e1, …, [e/x]en)
  - [e/x] apply(e1, e2) = apply([e/x]e1, [e/x]e2)
  - …

Closed Substitution

- [e/x]e', cont'd
  - [e/x] fun f(y:t):t' is e' end = fun f(y:t):t' is [e/x]e' end, if x ≠ f, x ≠ y
  - [e/x] fun f(y:t):t' is e' end = fun f(y:t):t' is e' end, if x=f or x=y
- Last two clauses avoid confusion of distinct variables that happen to have the same name.

Open Substitution

- What if e has free variables too?
  - Must avoid **capture** of free variables in e by binders in e'!
  - Eg, [x+1/y] fun f(x:int):int is x+y end ≠ fun f(x:int):int is x+x+1 end !!!
  - Rather, … = fun f(y:int):int is y+x+1 end
- We must **implicitly rename** bound variables during substitution!

Open Substitution

- Change the last two clauses thus:

  [e/x] fun f(y:t):t' is e' end = fun f(y:t):t' is e' end, if x=f or x=y

  [e/x] fun f(y:t):t' is e' end = fun f(y:t):t' is [e/x]e' end, if x ≠ f, x ≠ y, and f,y ∉ FV(e)
- By implicit renaming, the last condition can always be met!

Summary: Levels of Syntax

- Concrete syntax
  - layout, spelling, precedence conventions
- First-order abstract syntax
  - expose deep structure of expressions
- Higher-order abstract syntax
  - binding and scope
  - renaming of bound variables