

Type Systems for Programming Languages (15-814)

Lecture Notes, Fall 2006, Week 2

Fritz Obermeyer

Sept 19–21, 2006

(this document has hyperlinks)

1 Learning SML

Standard ML is a *functional language*, in contrast to *imperative languages* such as Java and C. In contrast to the earlier functional programming language Lisp, SML is typed. Functional languages operate by modifying a single expression (the program), whereas imperative languages operate by performing a list of instructions.

Most of what you'll need to know about SML is in Bob Harper's book *Programming in Standard ML*, and William Lovas' examples.

Here are a few tips to keep in mind:

- To wrap the SML interpreter in readline (for better command-line editing, history, and completion), use `ledit sml` (implemented in OCaml, a flavor of ML), or `cle sml` (implemented in C), or use `rlwrap`.
- The MLton compiler has better error-messages than SML of New Jersey.
- Use the `~` sign for negative numbers, as in `~ 3 + 4 = 1`.
- The `;` sign has two meanings:
 - the top-level declaration terminator, as in `“val x = 5;”`, and
 - the sequence separator, as in `“fun f () = (print “hello ”; print “world!\n”)”`
- You can put compiling commands in a `.cm` file, typically `sources.cm`, and then in SML, `CM.make “sources.cm”`.
- There's a lot of handy stuff in the standard basis library.

2 Binding and Scope via ABTs

Consider operators θ with arity $\sigma(t_1, \dots, t_k)$, where each t_i is the number of variables bound in the i^{th} argument. We will often write informally

<code>let x be e_1 in e_2</code>	<code>$\lambda x.e$</code>	<code>$e_1 e_2$</code>
<code>for let $(e_1, x.e_2)$</code>	<code>$\lambda(x.e_1)$</code>	<code>ap (e_1, e_2)</code>
with signatures $(0, 1)$	(1)	$(0, 0)$

2.1 α -equivalence

We will consider ABTs modulo name changes of bound variables. Following Alonzo Church, this is called α -equivalence.

Definition 2.1. α -equivalence of abts, denoted $a_1 =_\alpha a_2 \text{ abt}$, is the equivalence relation define inductively by the main rule

$$\frac{x\#y \quad x\#b \quad a =_\alpha [x \leftrightarrow y]b}{x.a =_\alpha y.b \text{ abt}}$$

and the simpler rules for respect for equivalence relations and respect for abstraction and term formation, as in the previous notes.

2.2 Structural induction modulo α -equivalence

To show $\forall n.a \text{ abt}^n \implies P_n(a)$, it is sufficient to show

1. P holds for variables: $P_0(x)$;
2. P holds for symbols: if $P_{n_1}(a_1), \dots, P_{n_k}(a_k)$, then also $P_0(\theta(a_1, \dots, a_k))$;
3. P holds for bindings: if for some/any fresh x , $P_n(a)$, then also $P_{n+1}(x.a)$.

Note that the quantifier in (3) is neither \forall nor \exists , but rather quantifies over fresh *names*; here ‘some’ is equivalent to ‘any’ since if $P_n(a)$ holds for some fresh x , it works for any fresh variable.

2.3 Substitution laws

$$\frac{}{[a/x]b = c \text{ abt}^n} \quad \frac{}{[a/x]x = a \text{ abt}^0} \quad \frac{y\#x}{[a/x]y = y \text{ abt}^0} \quad \frac{[a/x]a_1 = b_1 \text{ abt} \quad \dots \quad [a/x]a_k = b_k \text{ abt}}{[a/x]\theta(a_1, \dots, a_k) = \theta(b_1, \dots, b_k) \text{ abt}}$$

$$\frac{x\#y \quad y\#a \quad [a/x]b = c \text{ abt}^n}{[a/x]y.b = y.c \text{ abt}^{n+1}}$$

Note that the $y\#a$ is required to avoid capture, i.e. to ensure that substitution and abstraction commute. The following example illustrates this necessity.

Example 2.2. Let $a = y$, $b = x$ in the rule above. Then

$$[a/x]y.b = [y/x]y.x = y.x$$

but

$$y.[a/x]b = y.[y/x]x = y.y =_\alpha x.x$$

3 Un(i)typed λ -Calculus

(Note on naming: ‘untyped’ it is perhaps a misnomer, since the untyped calculus can be viewed as a calculus with a single trivial all-encompassing type; whence our preferred ‘untyped’.)

Informally we define the grammar of lambda-terms as

$$t ::= x \mid \lambda x.t \mid tt$$

which we might consider more formally as

Definition 3.1. the *untyped* λ -calculus is the language with a pair of symbols

$$\lambda : \text{arity}(1) \qquad \text{ap} : \text{arity}(0, 0)$$

The set of all lambda-terms is denoted Λ .

The notion computation is entirely encapsulated as an equivalence relation, also named by Church, β -equivalence,

Definition 3.2. β -equivalence of λ -terms is defined by: the β -axiom

$$\overline{(\lambda x.t)u =_{\beta} [u/x]t}$$

the rules of an equivalence relation

$$\frac{}{a =_{\beta} a} \qquad \frac{a =_{\beta} b}{b =_{\beta} a} \qquad \frac{a =_{\beta} b \quad b =_{\beta} c}{a =_{\beta} c}$$

and respect for abstraction and application

$$\frac{a =_{\beta} b}{x.a =_{\beta} x.b} \qquad \frac{a =_{\beta} b}{ac =_{\beta} bc} \qquad \frac{a =_{\beta} b}{ca =_{\beta} cb}$$

This rule alone is sufficient to ensure Turing-completeness of the untyped lambda-calculus.

3.1 Hacking λ

Booleans are specified by the interface

introduction: `tr`, `fa`

elimination: `if e then e1 else e2`,

specification: satisfying the two rules

$$\begin{aligned} \text{if } \text{tr} \text{ then } e_1 \text{ else } e_2 &=_{\beta} e_1 \\ \text{if } \text{fa} \text{ then } e_1 \text{ else } e_2 &=_{\beta} e_2 \end{aligned}$$

Note that this specification might be strengthened by additional rules such as

$$\text{if } b \text{ then } e \text{ else } e =_{\beta} e$$

; however that is beyond the scope of this course.

We may choose to encode the booleans by assuming

$$\llbracket \text{if } e \text{ then } e_1 \text{ else } e_2 \rrbracket := \llbracket e \rrbracket \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket$$

which has unique solution for truth values:

$$\begin{aligned} \text{tr} &:= \lambda x. \lambda y. x \\ \text{fa} &:= \lambda x. \lambda y. y \end{aligned}$$

Now consider the logical operations, say, `and` specified by

$$\begin{array}{ll} x, y & \mapsto \text{and } xy \\ \text{tr}, \text{tr} & \mapsto \text{tr} \\ \text{tr}, \text{fa} & \mapsto \text{fa} \\ \text{fa}, \text{tr} & \mapsto \text{fa} \\ \text{fa}, \text{fa} & \mapsto \text{fa} \end{array}$$

This has a solution, e.g.,

$$\text{and} := \lambda x. \lambda y. x(y \text{ tr } \text{fa}) \text{fa}$$

which using the if-then-else interpretation we read “if x is true then (if y then true else false) else false”. But for any booleans y , we have $y \text{ tr } \text{fa} = y$, so why not simplify and define

$$\text{and}' := \lambda x. \lambda y. xy \text{fa}$$

Are and and and' β -equivalent? Additionally, we know that conjunction is commutative, i.e., we have $xy = \text{and } yx$ for all booleans x, y . This suggests yet a third definition of and :

$$\text{and}'' := \lambda x. \lambda y. yx \text{fa}$$

Are and and and'' β -equivalent? We will eventually prove that all three are β -inequivalent; for now we simply assume the terms are distinct. The moral of this example is that in this untyped/unityped setting, things don't behave as we might expect: we intend terms to exhibit some behavior on a certain domain, but without a way to restrict them to this domain, they may behave in any manner outside of the domain.