Computer Science 15-312
MinML, A MiniMaL Functional Language
Part I: Abstract Syntax

Robert Harper

Spring, 2000

1 Introduction

The language MinML will serve as the object of study for much of the first half of the course. MinML is a call-by-value, effect-free language with two base types, `int` and `bool`, and (partial) function types $\tau_1 \rightarrow \tau_2$. It is thus a call-by-value variant of Plotkin's PCF language, and an extremely stripped-down version of the ML language.

2 First-Order Abstract Syntax

The abstract syntax of MinML is divided into three categories, *types*, *expressions*, and *programs*.

The set of types is the least set closed under the following rules:

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

We let ρ, σ, and τ range over the set of types.

The set of expressions is the least set closed under the following rules:

$$\frac{n\ \mathsf{num}}{n\ \mathsf{exp}}$$

$$\frac{}{\texttt{true}\ \mathsf{exp}} \qquad \frac{}{\texttt{false}\ \mathsf{exp}}$$

$$\frac{x\ \mathsf{var}}{x\ \mathsf{exp}}$$

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

$$\frac{f\ \mathsf{var}\quad x\ \mathsf{var}\quad e\ \mathsf{exp}}{\texttt{fun}\,f\ (x\texttt{:}\tau_1)\texttt{:}\tau_2\ \texttt{is}\ e\ \texttt{end}\ \mathsf{exp}}\qquad\frac{e_1\ \mathsf{exp}\quad e_2\ \mathsf{exp}}{\texttt{apply(}e_1,e_2\texttt{)}\ \mathsf{exp}}$$

We use the variable e to stand for a general expression.

We do not specify precisely the sets of numbers or variables. We generally write x, y, *etc.* for variables, and we write numbers in ordinary decimal notation. The set of primitive operators is inductively defined by these rules:

$$\frac{}{\texttt{+}\ \mathsf{op}}\qquad\frac{}{\texttt{*}\ \mathsf{op}}\qquad\frac{}{\texttt{-}\ \mathsf{op}}$$

$$\frac{}{\texttt{=}\ \mathsf{op}}\qquad\frac{}{\texttt{<}\ \mathsf{op}}$$

It is important, in general, to distinguish programs from expression. In the present case programs are just (closed) expressions, which is expressed by the following rule:

$$\frac{e\ \mathsf{exp}}{e\ \mathsf{prog}}$$

That is, a program is nothing more (and nothing less) than an expression. We let p range over the set of programs.

If you prefer grammar notation for specifying abstract syntax, here is an equivalent grammar for the abstract syntax of MinML:

Variables	x	$::=$	$\ldots$
Numerals	n	$::=$	$\ldots$
Operators	o	$::=$	$\texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<}$
Types	τ	$::=$	$\texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2$
Expressions	e	$::=$	$x \mid n \mid o(e_1, \ldots, e_n) \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi}$
		$\mid$	$\texttt{fun}\,f\ (x\texttt{:}\tau_1)\texttt{:}\tau_2\ \texttt{is}\ e\ \texttt{end} \mid \texttt{apply(}e_1, e_2\texttt{)}$
Programs	p	$::=$	e

Here, as before, we leave the exact definition of variables, numerals, and operators unspecified.

We will make use of informal syntactic conventions that it is hoped are clear from context. Officially we should define a concrete syntax for the language, and a mapping (parser) from the concrete to the abstract syntax. But this would take us too far afield for the present purposes. Here are a few notational conventions:

1. The function space type constructor is right-associative, so that the type $\texttt{int}\to\texttt{int}\to\texttt{int}$ parses as $\texttt{int}\to(\texttt{int}\to\texttt{int})$, rather than as the type $(\texttt{int}\to\texttt{int})\to\texttt{int}$.

2. We will often write expressions such as e_1+e_2 for +(e_1, e_2), presuming + to be a primitive operation, and employing the obvious infix syntax.

3. We will often write e_1(e_2), instead of apply(e_1, e_2), to denote function application.

3 Higher-Order Abstract Syntax

The rules of the *binding* and *scope* of variables are of fundamental importance. While these rules may seem to be a trivial, or at any rate niggling, technical detail, they are in fact of paramount importance. Many languages[1] fail to properly define and maintain sensible binding and scope conventions, with the result that code written in those languages is difficult to read and impossible to maintain.

Every variable has a *binder*, which introduces it, and a *scope*, which is its range of significance. Outside of its scope, the variable has no meaning, nor any particular identity. There is only one binder in MinML, the function expression $\mathtt{fun}\, f\ \mathtt{(}x\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\ \mathtt{is}\ e\ \mathtt{end}$. The variables f and x are *bound* by the function with *scope* the expression e. Nested binders may introduce "holes" in the scope of a variable — a variable occurrence in an expression is bound by the nearest enclosing binder for that variable, if there is one, in which case the variable is said to be *bound* by that binder. If no enclosing binder governs that variable, it is said to be *free* in the expression. It may be helpful to think of bound variables as "generalized pronouns" — they are used to refer to an antecedent binding occurrence. (Unlike natural languages, however, we are free to introduce new pronouns as we see fit.)

We may define the set of *free variables* of an expression e, $\mathrm{FV}(e)$, as follows:

$$
\begin{array}{rcl}
\mathrm{FV}(x) & = & \{\, x \,\} \\
\mathrm{FV}(n) & = & \emptyset \\
\mathrm{FV}(\mathtt{true}) & = & \emptyset \\
\mathrm{FV}(\mathtt{false}) & = & \emptyset \\
\mathrm{FV}(o(e_1, \ldots, e_n)) & = & \mathrm{FV}(e_1) \cup \cdots \cup \mathrm{FV}(e_n) \\
\mathrm{FV}(\mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi}) & = & \mathrm{FV}(e) \cup \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(\mathtt{fun}\, f\ \mathtt{(}x\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\ \mathtt{is}\ e\ \mathtt{end}) & = & \mathrm{FV}(e) \backslash \{\, f, x \,\} \\
\mathrm{FV}(\mathtt{apply}(e_1, e_2)) & = & \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2)
\end{array}
$$

We say that the variable x is *free* in the expression e iff $x \in \mathrm{FV}(e)$. An expression e is *closed* iff $\mathrm{FV}(e) = \emptyset$; that is, a closed expression has no free variables.

[1]Including the markup language I'm using to write these notes, TeX.

4 Substitution

Capture-avoiding substitution of an expression e for free occurrences of a variable x in another expression e', written $[e/x]e'$, is (partially) defined as follows:

$$\begin{array}{rcll}
[e/x]x & = & e & \\
[e/x]n & = & n & \\
[e/x]\texttt{true} & = & \texttt{true} & \\
[e/x]\texttt{false} & = & \texttt{false} & \\
[e/x]o(e_1,\ldots,e_n) & = & o([e/x]e_1,\ldots,[e/x]e_n) & \\
[e/x]\texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} & = & \texttt{if}\,[e/x]e\,\texttt{then}\,[e/x]e_1\,\texttt{else}\,[e/x]e_2\,\texttt{fi} & \\
[e/x]\texttt{fun}\,f\,\texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e''\,\texttt{end} & = & \texttt{fun}\,f\,\texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e''\,\texttt{end} & \textit{if } x=f \textit{ or } x=y \\
[e/x]\texttt{fun}\,f\,\texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,e''\,\texttt{end} & = & \texttt{fun}\,f\,\texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\,\texttt{is}\,[e/x]e''\,\texttt{end} & \textit{if } \{\,f,y\,\}\cap(\mathrm{FV}(e)\cup\{\,x\,\})=\emptyset \\
[e/x]\texttt{apply}\texttt{(}e_1,e_2\texttt{)} & = & \texttt{apply}\texttt{(}[e/x]e_1,[e/x]e_2\texttt{)} &
\end{array}$$

Simultaneous capture-avoiding substitution, written $[e_1,\ldots,e_n/x_1,\ldots,x_n]e$, is defined in an analogous manner.

Capture-avoiding substitution is *undefined* if the condition in the penultimate equation is not met! In this case free occurrences of f or y in e would be *captured* by the binder for f and y, thereby erroneously changing the meanings of the "pronouns". This means, for example, that

$$[x/y]\texttt{fun}\,f\,\texttt{(}x\texttt{:int)}\texttt{:int}\,\texttt{is}\,x+y\,\texttt{end}$$

is *undefined*, rather than equal to

$$\texttt{fun}\,f\,\texttt{(}x\texttt{:int)}\texttt{:int}\,\texttt{is}\,x+x\,\texttt{end},$$

wherein capture of x has occurred.

The possibility of capture during substitution can always be avoided by *renaming of bound variables.* The meaning of an expression should not be affected by the choice of variable names; whether we call a variable x, *he*, *she*, *it*, or *baloney_sandwich* should be immaterial, as long as we avoid the clashes (giving two *different* variables the *same* name). Thus

$$\texttt{fun}\,f\,\texttt{(}x\texttt{:int)}\texttt{:int}\,\texttt{is}\,\texttt{apply(}f,y+x+1\texttt{)}\,\texttt{end}$$

is precisely the same expression as

$$\texttt{fun}\,g\,\texttt{(}z\texttt{:int)}\texttt{:int}\,\texttt{is}\,\texttt{apply(}g,y+z+1\texttt{)}\,\texttt{end}$$

but is different from the expression

$$\texttt{fun}\,g\,\texttt{(}y\texttt{:int)}\texttt{:int}\,\texttt{is}\,\texttt{apply(}g,y+y+1\texttt{)}\,\texttt{end}.$$

To express this indifference to the names of bound variables, we introduce an equivalence relation between expressions, called*structural equivalence*, and written $e_1 \equiv e_2$, that is defined to be the smallest binary relation closed under the following rules. The first rule is called the *renaming* axiom.

$$\frac{}{\texttt{fun}\, f\ (x\,\texttt{:}\,\tau_1)\,\texttt{:}\,\tau_2\ \texttt{is}\ e\ \texttt{end} \equiv \texttt{fun}\, f'\ (x'\,\texttt{:}\,\tau_1)\,\texttt{:}\,\tau_2\ \texttt{is}\ [f', x'/f, x]e\ \texttt{end}}$$

(where f' and x' are chosen in any way such that substitution is defined). The remaining rules specify that structural equivalence is reflexive, symmetric, and transitive, and that we may perform renaming anywhere in an expression. These rules are sometimes called *congruence* rules.

$$\frac{}{e \equiv e} \qquad \frac{e_2 \equiv e_1}{e_1 \equiv e_2} \qquad \frac{e_1 \equiv e_2 \quad e_2 \equiv e_3}{e_1 \equiv e_3}$$

$$\frac{e_1 \equiv e_1' \quad \ldots \quad e_n \equiv e_n'}{o(e_1, \ldots, e_n) \equiv o(e_1', \ldots, e_n')}$$

$$\frac{e \equiv e' \quad e_1 \equiv e_1' \quad e_2 \equiv e_2'}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi} \equiv \texttt{if}\ e'\ \texttt{then}\ e_1'\ \texttt{else}\ e_2'\ \texttt{fi}}$$

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

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

Any binary relation closed under the congruence rules above is said to be a *congruence* relation. Structural equivalence is the *least congruence* closed under the renaming axiom.

In order to express the irrelevance of the names of bound variables, we implicitly identity expressions that are structurally equivalent. That is, in what follows, we never work directly with expressions, but rather with *equivalence classes* of expressions, modulo structural equivalence. When choosing a representative of a structural equivalence class, we do so in accordance with the *variable convention*, according to which the names of the bound variables are chosen to be distinct from one another and distinct from the name of any free variable in the context of use.

One consequence of this convention is that substitution is *never* undefined when applied to expressions taken modulo structural equivalence. The reason is that if the variable conditions on substitution that prevent capture were not met, we could always choose another representative so that they are met. This means, for example, that

$$[x/y]\texttt{fun}\, f\ (x\,\texttt{:}\,\texttt{int})\,\texttt{:}\,\texttt{int}\ \texttt{is}\ x + y\ \texttt{end}$$

is equivalent to

$$\texttt{fun}\, f\ (z\texttt{:int})\texttt{:int is}\ z + x\ \texttt{end}$$

(for any variable $z \neq x$), but *not* to

$$\texttt{fun}\, f\ (x\texttt{:int})\texttt{:int is}\ x + x\ \texttt{end}$$

(wherein captured has occurred).

5 Summary

This completes the presentation of the abstract syntax of MinML. The key points are these

1. Inductive definitions of the syntactic categories.

2. Definition of binding and scope.

3. Definition of capture-avoiding substitution.

4. Structural equivalence to identity expressions differing only in the choice of names of bound variable.