Computer Science 15-312
MinML, A MiniMaL Functional Language
Part II: Type System

Robert Harper

Spring, 2000

1 Introduction

In this note we consider the *static semantics*, or *typing rules*, for MinML. The type system of MinML is given by an inductive definition consisting of a set of rules for deriving *typing judgements*.

2 Type System

Not all expressions in MinML are well-formed. For example, the expression `if` 3 `then` 1 `else` 0 `fi` is not well-formed because 3 is an integer, whereas the conditional test expects a boolean. In other words, this expression is *ill-typed* because the expected constraint is not met. Expressions which do satisfy these constraints are said to be *well-typed*.

Typing is clearly context-sensitive. The expression $x + 3$ may or may not be well-typed, according to the type we assume for the variable x. That is, it depends on the surrounding context whether this sub-expression is well-typed or not.

The definition of well-typed expressions is given by a three-place *typing relation*, or *typing judgement*, written $\Gamma \vdash e : \tau$, where Γ is a partial function with finite domain mapping variables to types, and $\mathrm{FV}(e) \subseteq \mathrm{dom}(\Gamma)$. This relation may be read as "the expression e has type τ, under the assumption that its free variables have the types given by Γ." The function Γ may be thought of as a "symbol table" recording the types of the free variables of the expression e; the type τ is the type of e under the assumption that its free variables have the types assigned by Γ. When e is closed (has no free variables), we write simply $e : \tau$ instead of the more unwieldy $\emptyset \vdash e : \tau$.

We write $\Gamma(x)$ for the unique type τ (if any) assigned to x by Γ. The function $\Gamma[x{:}\tau]$, where $x \notin \mathrm{dom}(\Gamma)$, is defined by the following equation

$$\Gamma[x{:}\tau](y) = \begin{cases} \tau & \textit{if } y = x \\ \Gamma(y) & \textit{otherwise} \end{cases}$$

The typing relation is inductively defined by the following rules:

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ [\mathbf{VarTyp}] \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\ [\mathbf{NumTyp}]$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ [\mathbf{TrueTyp}] \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ [\mathbf{FalseTyp}]$$

In the rule for variables, it is understood that if $\Gamma(x)$ is undefined, then no type for x is derivable from assumptions Γ.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{+}(e_1, e_2) : \texttt{int}}\ [\mathbf{PlusTyp}]$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{*}(e_1, e_2) : \texttt{int}}\ [\mathbf{TimesTyp}]$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{-}(e_1, e_2) : \texttt{int}}\ [\mathbf{MinusTyp}]$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{=}(e_1, e_2) : \texttt{bool}}\ [\mathbf{EqualsTyp}]$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{<}(e_1, e_2) : \texttt{bool}}\ [\mathbf{LessTyp}]$$

The typing rules for the arithmetic and boolean primitive operators are as expected.

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau}\ [\mathbf{IfTyp}]$$

Notice that the "then" and the "else" clauses must have the same type!

$$\frac{\Gamma[f{:}\tau_1 {\to} \tau_2][x{:}\tau_1] \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\, f\ \texttt{(}x \texttt{:} \tau_1\texttt{)} \texttt{:} \tau_2\, \texttt{is}\, e\, \texttt{end} : \tau_1 {\to} \tau_2}\ [\mathbf{FunTyp}]$$

Here we require that the variables f and x be chosen (by suitable renaming of the function expression) so that $\{\, f, x \,\} \cap \mathrm{dom}(\Gamma) = \emptyset$.

$$\frac{\Gamma \vdash e_1 : \tau_2 {\to} \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply(}e_1, e_2\texttt{)} : \tau}\ [\mathbf{AppTyp}]$$

This completes the typing rules for MinML.

3 Properties of Typing

It is useful at this stage to catalogue some properties of the typing relation. We will make use of the principle of *induction on typing derivations*, or *induction on the typing rules*.

A key observation about the typing rules is that there is exactly one rule for each form of expression — that is, there is one rule for the each of the boolean constants, one rule for functions, *etc.*. The typing relation is therefore said to be *syntax-directed*; the form of the expression determines the typing rule to be applied. While this may seem inevitable at this stage, we will later encounter type systems for which this is not the case.

Exercise 1
Prove by induction on the structure of e that for every e and every Γ there exists at most one τ such that $\Gamma \vdash e : \tau$. Hint: *use rule induction for the rules defining the abstract syntax of expressions. The proof in each case is straightforward; what is important about this exercise is that you know how to present the proof clearly and rigorously using the machinery we've introduced.*

A simple — but important — consequence of syntax-directedness are the following *inversion principles* for typing. The typing rules define *sufficient* conditions for typing. For example, to show that $\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau$, it suffices to show that $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$, and $\Gamma \vdash e_2 : \tau$, because of rule [**IfTyp**]. Since there is exactly one typing rule for each expression, the typing rules also express *necessary* conditions for typing. For example, if $\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau$, then $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. That is, we can "invert" each rule to obtain a necessary typing condition. This is the content of the following theorem.

Theorem 2 (Inversion)

1. *If $\Gamma \vdash x : \tau$, then $\Gamma(x) = \tau$.*
2. *If $\Gamma \vdash n : \tau$, then $\tau = \texttt{int}$.*
3. *If $\Gamma \vdash \texttt{true} : \tau$, then $\tau = \texttt{bool}$, and similarly for $\texttt{false}$.*
4. *If $\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau$, then $\Gamma \vdash e : \texttt{bool}$, $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$.*
5. *If $\Gamma \vdash \texttt{fun}\, f\ \texttt{(}x{:}\tau_1\texttt{)}{:}\tau_2\ \texttt{is}\, e\, \texttt{end} : \tau$, then $\tau = \tau_1 {\to} \tau_2$ and then $\Gamma[f{:}\tau_1{\to}\tau_2][x{:}\tau_1] \vdash e : \tau_2$.*

6. *If* $\Gamma \vdash$ `apply(`e_1, e_2`)` $: \tau$, *then there exists* τ_2 *such that* $\Gamma \vdash e_1 : \tau_2 \rightarrow \tau$ *and* $\Gamma \vdash e_2 : \tau_2$.

Proof: Each case is proved by induction on typing. In each case exactly one rule applies, from which the result is obvious. ■

Exercise 3
Can you think of a type system for a variant of MinML in which inversion fails? What form would such a type system have to take? Hint: *think about overloading arithmetic operations.*

Lemma 4

1. *Typing is not affected by "junk" in the symbol table. If* $\Gamma \vdash e : \tau$ *and* $\Gamma' \supseteq \Gamma$, *then* $\Gamma' \vdash e : \tau$.

2. *Substitution for a variable with type* τ *by an expression of the same type doesn't affect typing. If* $\Gamma[x{:}\tau] \vdash e' : \tau'$, *and* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

Proof:

1. By induction on the typing rules. For example, consider the typing rule for applications. Inductively we may assume that if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e_1 : \tau_2 \rightarrow \tau$ and if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e_2 : \tau_2$. Consequently, if $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash$ `apply(`e_1, e_2`)` $: \tau$, as required. The other cases follow a similar pattern.

2. By induction on the derivation of the typing $\Gamma[x{:}\tau] \vdash e' : \tau'$. We will consider several rules to illustrate the idea.

   [VarTyp] We have that e' is a variable, say y, and $\tau' = \Gamma[x{:}\tau](y)$. If $y \neq x$, then $[e/x]y = y$ and $\Gamma[x{:}\tau](y) = \Gamma(y)$, hence $\Gamma \vdash y : \Gamma(y)$, as required. If $x = y$, then $\tau' = \Gamma[x{:}\tau](x) = \tau$, and $[e/x]x = e$. By assumption $\Gamma \vdash e : \tau$, as required.

   [FunTyp] We have that $e' = \texttt{fun}\, f\ (y{:}\tau_1){:}\tau_2\ \texttt{is}\ e_2\ \texttt{end}$ and $\tau' = \tau_1 \rightarrow \tau_2$. We may assume that f and y are chosen so that

$$\{ f, y \} \cap (\mathrm{FV}(e) \cup \{ x \} \cup \mathrm{dom}(\Gamma)) = \emptyset.$$

By definition of substitution,

$$[e/x]e' = \texttt{fun}\, f\ \texttt{(}y\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\ \texttt{is}\ [e/x]e_2\ \texttt{end}.$$

Applying the inductive hypothesis to the premise of the rule [**FunTyp**],

$$\Gamma[x{:}\tau][f{:}\tau_1{\rightarrow}\tau_2][y{:}\tau_1] \vdash e_2 : \tau_2,$$

it follows that

$$\Gamma[f{:}\tau_1{\rightarrow}\tau_2][y{:}\tau_1] \vdash [e/x]e_2 : \tau_2.$$

Hence

$$\Gamma \vdash \texttt{fun}\, f\ \texttt{(}y\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\ \texttt{is}\ [e/x]e_2\ \texttt{end} : \tau_1{\rightarrow}\tau_2,$$

as required.

■

4 Summary

This completes the definition of the *static semantics*, or *typing rules*, for MinML. The main points are these:

1. Typing is context-sensitive. The type (if any) of an expression with free variables in it is sensitive to the types we assume for those variables.

2. The typing relation is given by an inductive definition. This licenses proofs by induction on the typing rules.

3. The typing relation is syntax-directed: there is one rule for each form of expression.