MinML - a MiniMaL Functional Language

Robert Harper

1999 (extracted & edited by A. Appel, 2006)

Types. \( \tau := \text{int} | \text{bool} | \tau_1 \rightarrow \tau_2 \)

Expressions.

\( e := x | n | \text{true} | \text{false} | o(e_1, \ldots, e_n) | \text{if} e_1 \text{ then } e_2 \text{ else } e_3 | \text{fun} \ f (x: \tau_1) : \tau_2 \text{ is }\text{end} | \text{apply} (e_1, e_2) \)

Here \( x \) ranges over a countably infinite set of variables, \( n \) ranges over the integers, and \( o \) ranges over a set of primitive operations.

Free variables.

\[
\begin{align*}
\text{FV}(x) &= \{x\} \\
\text{FV}(n) &= \emptyset \\
\text{FV}(\text{true}) &= \emptyset \\
\text{FV}(\text{false}) &= \emptyset \\
\text{FV}(o(e_1, \ldots, e_n)) &= \text{FV}(e_1) \cup \cdots \cup \text{FV}(e_n) \\
\text{FV}(\text{if} e_1 \text{ then } e_2 \text{ else } e_3) &= \text{FV}(e) \cup \text{FV}(e_1) \cup \text{FV}(e_2) \\
\text{FV}(\text{fun} f (x: \tau_1) : \tau_2 \text{ is }\text{end}) &= \text{FV}(e) \setminus \{f, x\} \\
\text{FV}(\text{apply} (e_1, e_2)) &= \text{FV}(e_1) \cup \text{FV}(e_2)
\end{align*}
\]

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

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

\[
\begin{align*}
[e/x]x &= e \\
[e/x]n &= n \\
[e/x]\text{true} &= \text{true} \\
[e/x]\text{false} &= \text{false} \\
[e/x]o(e_1, \ldots, e_n) &= o([e/x]e_1, \ldots, [e/x]e_n) \\
[e/x]\text{if} e_1 \text{ then } e_2 \text{ else } e_3 &= \text{if} [e/x]e_1 \text{ then } [e/x]e_2 \text{ else } [e/x]e_3 \\
[e/x]\text{fun} f (y: \tau_1) : \tau_2 \text{ is }\text{end} &= \text{fun} f (y: \tau_1) : \tau_2 \text{ is }\text{end} \quad \text{if } f = x \text{ or } x = y \\
[e/x]\text{fun} f (y: \tau_1) : \tau_2 \text{ is }\text{end} &= \text{fun} f (y: \tau_1) : \tau_2 \text{ is }\text{end} \quad \text{if } \{f, y\} \cap (\text{FV}(e) \cup \{x\}) = \emptyset \\
[e/x]\text{apply} (e_1, e_2) &= \text{apply} ([e/x]e_1, [e/x]e_2)
\end{align*}
\]

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

\[
[x/y]\text{fun} f (x: \text{int}) : \text{int} \text{ is } \text{end}
\]

is undefined, rather than equal to

\[
\text{fun} f (x: \text{int}) : \text{int} \text{ is } \text{end},
\]

wherein capture of \( x \) has occurred.

The possibility of capture during substitution can always be avoided by renaming of bound variables.
Single-step evaluation.

This is really a rule schema; it specifies one rule for each primitive operation \( o \). For example, if \( o \) is the addition operation, then this rule would be written as \( m_1 + n_2 \mapsto n \) where \( n = m_1 + n_2 \). This is itself a rule schema, standing for the infinite collection of rules

\[
\begin{align*}
0 + 0 & \mapsto 0 \\
0 + 1 & \mapsto 1 \\
1 + 0 & \mapsto 1 \\
1 + 1 & \mapsto 2 \\
\end{align*}
\]

\[
\text{iftrue} \ e_1 \, \text{else} \ e_2 \, \text{fi} \mapsto e_1 \\
\text{iffalse} \ e_1 \, \text{else} \ e_2 \, \text{fi} \mapsto e_2
\]

\[
\text{CallFun} \quad (v = \text{fun} \ (x : \tau_1) : \tau_2 \, \text{is} \, \text{end} \, \Rightarrow \text{apply}(v, v_1) \mapsto [v, v_1/f, x]e)
\]

\[
\begin{align*}
\text{OpArg} & \quad e \mapsto e' \\
\text{IfTest} & \quad e \mapsto e' \\
\text{AppFun} & \quad \text{apply}(e_1, e_2) \mapsto \text{apply}(e'_1, e_2) \\
\text{AppArg} & \quad e_1 \mapsto e'_1 \\
\text{NumTyp} \quad e_2 \mapsto e'_2
\end{align*}
\]

Multistep evaluation.

\[
\text{e} \mapsto^* \text{e}' \\
\text{e}' \mapsto^* \text{e}''
\]

Environments. \( \Gamma[x : \tau] (y) = \begin{cases} 
\tau & \text{if } y = x \\
\Gamma(y) & \text{otherwise}
\end{cases} \)

Typing judgment.

\[
\begin{align*}
\Gamma & = \tau & \text{VarTyp} \\
\Gamma & \vdash n : \text{int} & \text{NumTyp} \\
\Gamma & \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma & \vdash e_n : \tau_n & \text{OpTyp} \\
\Gamma & \vdash o(e_1, \ldots, e_n) : \tau & \text{We state this typing rule for each primitive operation \( o \) taking \( n \) arguments of type \( \tau_1, \ldots, \tau_n \), respectively, and yielding a value of type \( \tau \).}
\Gamma & \vdash e : \text{bool} & \text{IfTyp} \\
\Gamma & \vdash e_1 : \tau & \Gamma & \vdash e_2 : \tau \\
\Gamma & \vdash \text{ifthen} e_1 \, \text{else} \, e_2 \, \text{fi} : \tau & \text{AppTyp}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash \text{fun} \ (x : \tau_1) : \tau_2 \, \text{is} \, \text{end} : \tau_1 \rightarrow \tau_2 & \text{FunTyp} \\
\Gamma & \vdash \text{apply} (e_1, e_2) : \tau & \text{AppTyp}
\end{align*}
\]

Notation. When \( e \) is closed, we can write \( e : \tau \) instead of \( \emptyset \vdash e : \tau \).

**Theorem: Inversion**

1. If \( \Gamma \vdash x : \tau \), then \( \Gamma(x) = \tau \).
2. If \( \Gamma \vdash n : \tau \), then \( \tau = \text{int} \).
3. If \( \Gamma \vdash \text{true} : \tau \), then \( \tau = \text{bool} \), and similarly for \( \text{false} \).
4. If \( \Gamma \vdash \text{ifthen} e_1 \, \text{else} \, e_2 \, \text{fi} : \tau \), then \( \Gamma \vdash e : \text{bool} \) and \( \Gamma \vdash e_1 : \tau \) and \( \Gamma \vdash e_2 : \tau \).
5. If \( \Gamma \vdash \text{fun} \ (x : \tau_1) : \tau_2 \, \text{is} \, \text{end} : \tau_1 \rightarrow \tau_2 \), then \( \Gamma[f : \tau_1 \rightarrow \tau_2][x : \tau_1] \vdash e : \tau_2 \).
6. If \( \Gamma \vdash \text{apply} (e_1, e_2) : \tau \), then there exists \( \tau_2 \) such that \( \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \) and \( \Gamma \vdash e_2 : \tau_2 \).

**Proof.** Each case is by induction on typing. In each case exactly one rule applies, from which the result is obvious. \( \blacksquare \)
Lemma: Environment extension. Typing is not affected by “junk” in the symbol table. If $\Gamma \vdash e : \tau$ and $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e : \tau$.

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

Lemma: Deterministic multistep evaluation. Theorem: Preservation. If $\Gamma \vdash e : \tau$, then $\Gamma \vdash e' : \tau$.

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

Lemma: Typing after substitution. Substitution for a variable with type $\tau$ by an expression of the same type doesn’t affect typing. If $\Gamma \vdash e : \tau$, then $\Gamma \vdash [e/x]e' : \tau$.

Proof. By induction on the derivation of the typing $\Gamma \vdash e : \tau$. We will consider several rules to illustrate the idea.

FunTyp We have that $e'$ is a variable, say $y$, and $\tau' = \Gamma[y]$. If $y \neq x$, then $[e/x]y = y$ and $\Gamma[y]$ is the same type, as required. If $y = x$, then $\tau' = \Gamma[y]$ is the same type, as required. By assumption $\Gamma \vdash e : \tau$, as required.

FunTyp We have that $e' = \text{fun} f (y: \tau_1) : \tau_2 \text{ is } e_2 \text{ end}$ and $\tau' = \tau_1 \to \tau_2$. We may assume that $f$ and $y$ are chosen so that $$\{f, y\} \cap (\text{FV} (e) \cup \{x\} \cup \text{dom} (\Gamma)) = \emptyset.$$ By definition of substitution, $[e/x]e' = \text{fun} f (y: \tau_1) : \tau_2 \text{ is } e_2 \text{ end}$. Applying the inductive hypothesis to the premise of the rule FunTyp, $\Gamma[y] [f : \tau_1 \to \tau_2] [y : \tau_1] \vdash e_2 : \tau_2$, it follows that $\Gamma[y] [f : \tau_1 \to \tau_2] [y : \tau_1] \vdash [e/x]e_2 : \tau_2$. Hence

$$\Gamma \vdash \text{fun} f (y: \tau_1) : \tau_2 \text{ is } e_2 \text{ end} : \tau_1 \to \tau_2,$$

as required. ■

Lemma: Deterministic single-step evaluation. For every closed expression $e$, there exists at most one $e'$ such that $e \mapsto e'$. In other words, the relation $\mapsto$ is a partial function.

Proof. By induction on the structure of $e$. We leave the proof as an exercise to the reader. Be sure to consider all rules that apply to a given expression $e$!

Lemma: Deterministic multistep evaluation. For every closed expression $e$, there exists at most one value $v$ such that $e \mapsto v$.

Proof. Follows immediately from the preceding lemma, with the observation that there is no transition from a value.

Lemma: Renaming bound variables. If $e_1 \mapsto e_2$ and $e_1 \equiv e'_1$, then there exists $e'_2$ such that $e'_1 \mapsto e'_2$ and $e'_2 \equiv e_2$.

Proof. By induction on the rules defining one-step evaluation.

Theorem: Preservation. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

Proof. Note that we are proving not only that $e'$ is well-typed, but that it has the same type as $e$. The proof is by induction on the rules defining one-step evaluation. We will consider each rule in turn.

OpVals Here $e = o(v_1, \ldots, v_n)$ and $e' = v$ is the result of executing operation $o$ on arguments $v_1, \ldots, v_n$. By our assumptions about the primitive operations, if $e : \tau$, then $v : \tau$.

IfTrue Here $e = \text{if true then } e_1 \text{ else } e_2 \text{ fi}$ and $e' = e_1$. Since $e : \tau$, by inversion $e_1 : \tau$, as required.

IfFalse Here $e = \text{if false then } e_1 \text{ else } e_2 \text{ fi}$ and $e' = e_2$. Since $e : \tau$, by inversion $e_2 : \tau$, as required.

CallFun Here $e = \text{apply} (v_1, v_2)$, where $v_1 = \text{fun} f (x : \tau_2) : \tau_1 \text{ is } e_2 \text{ end}$, and $e' = [v_1, v_2 / f, x] e_2$. By inversion applied to $e$, we have $v_1 : \tau_2 \to \tau$ and $v_2 : \tau_2$. By inversion applied to $v_1$, we have $[f : \tau_2 \to \tau] [x : \tau_2] \vdash e_2 : \tau$. Therefore, by substitution we have $[v_1, v_2 / f, x] e_2 : \tau$, as required.

OpArg Here $e = o(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n)$, $e' = o(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)$, and $e_i \mapsto e'_i$. Suppose that the $i$th argument of $o$ is of type $\tau_i$. By inversion, $v_1 : \tau_1, \ldots, v_{i-1} : \tau_{i-1}, e_i : \tau_i, \ldots, e_n : \tau_n$. By inductive hypothesis $e'_i : \tau_i$, and hence $e' : \tau$, as required.
IfTest Here \( e = if e_1 \text{then} e_2 \text{else} e_3 \) \( \text{fi} \) and \( e' = if e'_1 \text{then} e'_2 \text{else} e'_3 \) \( \text{fi} \). By inversion we have that \( e_1 : \text{bool} \), \( e_2 : \tau \) and \( e_3 : \tau \). By inductive hypothesis \( e'_1 : \text{bool} \), and hence \( e' : \tau \).

AppFun Here \( e = \text{apply} (e_1, e_2) \) and \( e' = \text{apply} (e'_1, e_2) \). By inversion \( e_1 : \tau_2 \rightarrow \tau \) and \( e_2 : \tau_2 \), for some type \( \tau_2 \). By induction \( e'_1 : \tau_2 \rightarrow \tau \), and hence \( e' : \tau \).

AppArg Here \( e = \text{apply} (v_1, e_2) \) and \( e' = \text{apply} (v_1, e'_2) \). By inversion, \( v_1 : \tau_2 \rightarrow \tau \) and \( e_2 : \tau_2 \), for some type \( \tau_2 \). By induction \( e'_2 : \tau_2 \), and hence \( e' : \tau \).

Lemma: Canonical Forms. The type of a closed value "predicts" its form.
Suppose that \( v : \tau \) is a closed, well-formed value.

1. If \( \tau = \text{bool} \), then either \( v = \text{true} \) or \( v = \text{false} \).
2. If \( \tau = \text{int} \), then \( v = n \) for some \( n \).
3. If \( \tau = \tau_1 \rightarrow \tau_2 \), then \( v = \text{fun} f (x : \tau_1) : \tau_2 \text{is} e \text{end} \) for some \( f, x \), and \( e \).

Proof. By induction on the typing rules, using the fact that \( v \) is a value.

Theorem: Progress. If \( e : \tau \), then either \( e \) is a value, or there exists \( e' \) such that \( e \rightarrow e' \).

Proof. The proof is by induction on the typing rules.

VarTyp Cannot occur, since \( e \) is closed.

NumTyp, TrueTyp, FalseTyp, FunTyp In each case \( e \) is a value, which completes the proof.

OpTyp Here \( e = o (e_1, \ldots, e_n) \) and \( e_i : \tau_i \) for each \( 1 \leq i \leq n \), where the primitive operation \( o \) takes arguments of types \( \tau_1, \ldots, \tau_n \), and yields values of type \( \tau \). By inductive hypothesis we have for each \( 1 \leq i \leq n \), either \( e_i \) is a value, or there exists \( e'_i \) such that \( e_i \rightarrow e'_i \). If all arguments are values, then by the assumption that the primitive operations are total, there exists \( v \) such that \( e \rightarrow v \). Otherwise, supposed that \( e_1, \ldots, e_{i-1} \) are all values, and that \( e_i \) is not a value. Then by the \( i \)th inductive hypothesis there exists \( e'_i \) such that \( e_i \rightarrow e'_i \). But then \( e \rightarrow e' \), where \( e' = o (e_1, \ldots, e_{i-1}, e'_i, \ldots, e_n) \), by rule OpArg.

IfTyp Here \( e = if e_1 \text{then} e_2 \text{else} e_3 \text{fi} \), with \( e_1 : \text{bool} \), \( e_2 : \tau \), and \( e_3 : \tau \). By the first inductive hypothesis, either \( e_1 \) is a value, or there exists \( e'_1 \) such that \( e_1 \rightarrow e'_1 \). If \( e_1 \) is a value, then we have by the Canonical Forms Lemma, either \( e_1 = \text{true} \) or \( e_1 = \text{false} \). In the former case \( e \rightarrow e_2 \), and in the latter \( e \rightarrow e_3 \), as required. If \( e_1 \) is not a value, then \( e \rightarrow e' \), where \( e' = if e'_1 \text{then} e_2 \text{else} e_3 \text{fi} \), by rule IfTest.

AppTyp Here \( e = \text{apply} (e_1, e_2) \), with \( e_1 : \tau_2 \rightarrow \tau \) and \( e_2 : \tau_2 \). By the first inductive hypothesis, either \( e_1 \) is a value, or there exists \( e'_1 \) such that \( e_1 \rightarrow e'_1 \). If \( e_1 \) is not a value, then \( e \rightarrow e'_1 \), by rule AppFun, as required. By the second inductive hypothesis, either \( e_2 \) is a value, or there exists \( e'_2 \) such that \( e_2 \rightarrow e'_2 \). If \( e_2 \) is not a value, then \( e \rightarrow e' \), where \( e' = \text{apply} (e_1, e'_2) \), as required. Finally, if both \( e_1 \) and \( e_2 \) are values, then by the Canonical Forms Lemma, \( e_1 = \text{fun} f (x : \tau_2) : \tau \text{is} e'' \text{end} \), and \( e \rightarrow e' \), where \( e' = [e_1, e_2 / f, x] e'' \), by rule CallFun.

Theorem: Type safety. If \( e : \tau \) and \( e \rightarrow e' \) then either \( e' \) is a value or there exists \( e'' \) such that \( e' \rightarrow e'' \).

Proof. By progress and preservation.