Not Quite Standard ML

Chris Stone

September 20, 2000

This document gives a full definition for the small functional programming language presented in class.

1 Abstract Syntax

1.1 Naming conventions

It is conventional that if, for example, the abstract syntax defines e to represent an arbitrary expression, then metavariables like e' and e'' and e_1 and e_4'' also refer to arbitrary expressions.

1.2 Grammar

$$
\begin{array}{rcll}
x, y, f & ::= & \texttt{x} \mid \texttt{y} \mid \texttt{foo} \mid \cdots & \text{variables} \\
n & ::= & \overline{\texttt{0}} \mid \overline{\texttt{1}} \mid \cdots & \text{integer constants} \\
v & ::= & n & \\
 & \mid & \overline{\texttt{tt}} & \\
 & \mid & \overline{\texttt{ff}} & \\
 & \mid & \texttt{fun } f(x{:}t_1){:}t_2 \texttt{ is } e_2 & \\
e & ::= & v & \\
 & \mid & x & \\
 & \mid & e_1 \texttt{ + } e_2 & \\
 & \mid & e_1 \texttt{ < } e_2 & \\
 & \mid & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \\
 & \mid & \texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2 & \\
 & \mid & e_1\, e_2 & \\
t, u & ::= & \texttt{Int} & \\
 & \mid & \texttt{Bool} & \\
 & \mid & t_1\texttt{->}t_2 &
\end{array}
$$

1.3 Bound variables and scope

In the expression

$$\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2$$

the variable x is bound, and its scope is e_2. Similarly, in the expression

$$\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2$$

the variables f and x are bound, and their scope is e_2. Expressions in this language which differ only in the names of bound variables are considered identical.

Thus we can define the function FV to calculate the free variables of an expression. This is defined by induction on the structure of the expression:

$$\begin{array}{ll}
\mathrm{FV}(n) & = \emptyset \\
\mathrm{FV}(\overline{\texttt{tt}}) & = \emptyset \\
\mathrm{FV}(\overline{\texttt{ff}}) & = \emptyset \\
\mathrm{FV}(e_1 \texttt{+} e_2) & = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 \texttt{<} e_2) & = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) & = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \cup \mathrm{FV}(e_3) \\
\mathrm{FV}(e_1\, e_2) & = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(x) & = \{x\} \\
\mathrm{FV}(\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2) & = \mathrm{FV}(e_1) \cup (\mathrm{FV}(e_2) \setminus \{x\}) \\
\mathrm{FV}(\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2) & = \mathrm{FV}(e_2) \setminus \{f, x\}
\end{array}$$

(Here the notation $A \setminus B$ denotes the set of elements of A which do not appear in B.)

We can now define substitution, where $e[x{\mapsto}v]$ stands for the substitution of v for free occurrences of x in e. This is again defined by induction on e.

$$\begin{array}{ll}
n[x{\mapsto}v] & = n \\
\overline{\texttt{tt}}[x{\mapsto}v] & = \overline{\texttt{tt}} \\
\overline{\texttt{ff}}[x{\mapsto}v] & = \overline{\texttt{ff}} \\
(e_1 \texttt{+} e_2)[x{\mapsto}v] & = (e_1[x{\mapsto}v]) \texttt{+} (e_2[x{\mapsto}v]) \\
(e_1 \texttt{<} e_2)[x{\mapsto}v] & = (e_1[x{\mapsto}v]) \texttt{<} (e_2[x{\mapsto}v]) \\
(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)[x{\mapsto}v] & = \texttt{if}\ (e_1[x{\mapsto}v])\ \texttt{then}\ (e_2[x{\mapsto}v])\ \texttt{else}\ (e_3[x{\mapsto}v]) \\
x[x{\mapsto}v] & = v \\
y[x{\mapsto}v] & = y \qquad \text{if } x \neq y \\
(\texttt{let}\ y\ \texttt{be}\ e_1\ \texttt{in}\ e_2)[x{\mapsto}v] & = \texttt{let}\ y\ \texttt{be}\ e_1[x{\mapsto}v]\ \texttt{in}\ e_2[x{\mapsto}v] \\
 & \qquad \text{if } x \neq y \text{ and } y \notin \mathrm{FV}(v) \\
(\texttt{fun}\ f(y{:}t_1){:}t_2\ \texttt{is}\ e_2)[x{\mapsto}v] & = \texttt{fun}\ f(y{:}t_1){:}t_2\ \texttt{is}\ (e_2[x{\mapsto}v]) \\
 & \qquad \text{if } x \notin \{f, y\} \text{ and } \{f, y\} \cap \mathrm{FV}(v) = \emptyset
\end{array}$$

Note that despite the side-conditions, for any x, v, and e there is a well-defined meaning for $e[x{\mapsto}v]$.

2 Static Semantics

The following rules define a three-place relation $\Gamma \vdash e : t$ between typing environments, expressions, and types. It is important to remember that a judgment $\Gamma \vdash e : t$ holds *if and only if there is a proof of this using the following rules.*

The metavariable Γ is used to denote an arbitrary typing environment. The notation $\Gamma(x)$ means the type which Γ assigns to x, while the notation $\Gamma, x{:}t$ means the type environment which agrees with Γ except that it also assigns type t to the variable x. Finally, $\mathrm{dom}(\Gamma)$ is defined to be the set of variables to which the environment Γ assigns types.

Type environments may be written as finite lists; for example,

```
x:Int, y:Bool, z:Int->(Int->Bool)
```

represents an environment that associates the given types with the variables `x`, `y`, and `z`. Then

$$\mathrm{dom}(\texttt{x:Int, y:Bool, z:Int->(Int->Bool)}) = \{\texttt{x}, \texttt{y}, \texttt{z}\}$$

and

$$(\texttt{x:Int, y:Bool, z:Int->(Int->Bool)})(\texttt{y}) = \texttt{Bool}.$$

$$\frac{}{\Gamma \vdash n : \texttt{Int}} \tag{1}$$

$$\frac{}{\Gamma \vdash \overline{\texttt{tt}} : \texttt{Bool}} \tag{2}$$

$$\frac{}{\Gamma \vdash \overline{\texttt{ff}} : \texttt{Bool}} \tag{3}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \quad (x \in \mathrm{dom}(\Gamma)) \tag{4}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 \texttt{+} e_2 : \texttt{Int}} \tag{5}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 \texttt{<} e_2 : \texttt{Bool}} \tag{6}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) : t} \tag{7}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad (\Gamma, x{:}t_1) \vdash e_2 : t}{\Gamma \vdash (\texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2) : t} \quad (x \notin \mathrm{dom}(\Gamma)) \tag{8}$$

$$\frac{\Gamma \vdash e_1 : t_2\texttt{->}t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\, e_2 : t} \tag{9}$$

$$\frac{(\Gamma, f{:}t_1\texttt{->}t_2, x{:}t_1) \vdash e_2 : t_2}{\Gamma \vdash \texttt{fun } f(x{:}t_1){:}t_2 \texttt{ is } e_2 : t_1\texttt{->}t_2} \quad (x \neq f \text{ and } x, f \notin \mathrm{dom}(\Gamma)) \tag{10}$$

Proposition 1 (Determinacy of Typing)
Given Γ and e there is at most one type t such that $\Gamma \vdash e : t$.

Lemma 2 (Inversion)

1. *If $\Gamma \vdash x : t$ then $t = \Gamma(x)$.*
2. *If $\Gamma \vdash e_1 \texttt{+} e_2 : t$ then $t = \texttt{Int}$ and $\Gamma \vdash e_1 : \texttt{Int}$ and $\Gamma \vdash e_2 : \texttt{Int}$.*
3. *If $\Gamma \vdash e_1 \texttt{<} e_2 : t$ then $t = \texttt{Bool}$ and $\Gamma \vdash e_1 : \texttt{Int}$ and $\Gamma \vdash e_2 : \texttt{Int}$.*
4. *If $\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t$ then $\Gamma \vdash e_1 : \texttt{Bool}$ and $\Gamma \vdash e_2 : t$ and $\Gamma \vdash e_3 : t$.*
5. *If $\Gamma \vdash \texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2 : t$ then there exists a type t_1 such that $\Gamma \vdash e_1 : t_1$ and $(\Gamma, x{:}t_1) \vdash e_2 : t_2$.*
6. *If $\Gamma \vdash e_1\, e_2 : t$ then there exists a type t_2 such that $\Gamma \vdash e_1 : t_2\texttt{->}t$ and $\Gamma \vdash e_2 : t_2$.*
7. *If $\Gamma \vdash \texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2 : t$ then $t = t_1\texttt{->}t_2$ and $(\Gamma, f{:}t_1\texttt{->}t_2, x{:}t_1) \vdash e_2 : t_2$.*

We say that $\Gamma' \supseteq \Gamma$ if for every variable that Γ gives a type to, Γ' assigns the variable the same type. That is, if

$$\forall x \in \mathrm{dom}(\Gamma). \quad \Gamma(x) = \Gamma'(x).$$

The type environment Γ' may additionally contain types for other variables. So, for example, we have

$$\begin{array}{ll} \texttt{x:Int}, \texttt{y:Bool} & \supseteq \texttt{x:Int}, \texttt{y:Bool} \\ \texttt{x:Int}, \texttt{y:Bool}, \texttt{z:Bool} & \supseteq \texttt{x:Int}, \texttt{y:Bool} \\ \texttt{x:Int}, \texttt{y:Bool}, \texttt{z:Bool} & \supseteq \texttt{y:Bool} \end{array}$$

Proposition 3 (Weakening)
If an expression is well-typed under a certain set of typing assumptions, and we throw in even more typing assumptions, then the expression remains well-typed with the same type. Formally, if $\Gamma \vdash e : t$ and $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash e : t$.

Proof: By induction on the proof that $\Gamma \vdash e : t$, and cases on the final rule used in this proof. I show just a few examples.

- Case: the proof ends with a use of Rule 1. Then e must be an integer constant n and t must be the type `Int`. But by this same rule, $\Gamma' \vdash n : \texttt{Int}$.
- Case: the proof ends with a use of Rule 4. Then e must be a variable x and t must be the type $\Gamma(x)$. But by assumption $\Gamma'(x) = \Gamma(x)$, so by this same rule $\Gamma' \vdash x : \Gamma'(x)$, or equivalently $\Gamma' \vdash x : \Gamma(x)$.
- Case: the proof ends with a use of Rule 5. Then e must be of the form $e_1 \texttt{+} e_2$, t must be the type `Int`, and there are sub-proofs $\Gamma \vdash e_1 : \texttt{Int}$ and $\Gamma \vdash e_2 : \texttt{Int}$. By the inductive hypothesis, $\Gamma' \vdash e_1 : \texttt{Int}$ and $\Gamma' \vdash e_2 : \texttt{Int}$. Hence by Rule 5 we have $\Gamma' \vdash e_1 \texttt{+} e_2 : \texttt{Int}$.

- Case: the proof ends with a use of Rule 8. Then e must be of the form `let` x `be` e_1 `in` e_2, and there are sub-proofs of $\Gamma \vdash e_1 : t_1$ and $(\Gamma, x{:}t_1) \vdash e_2 : t$. Then $(\Gamma', x{:}t_1) \supseteq (\Gamma, x{:}t_1)$, so inductively we have $\Gamma' \vdash e_1 : t_1$ and $(\Gamma', x{:}t_1) \vdash e_2 : t$. Since x here is a local bound variable which can always be renamed, without loss of generality we can assume that $x \notin \mathrm{dom}(\Gamma')$. Therefore by Rule 8 we have $\Gamma' \vdash (\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2) : t$.

- The other cases are similar and left to the reader.

∎

Proposition 4 (Substitution)
If $\Gamma_1, x{:}t, \Gamma_2 \vdash e : t'$ *and* $\Gamma_1 \vdash v : t$ *then* $\Gamma_1, \Gamma_2 \vdash e[x{\mapsto}v] : t'$.

Proof: By induction on the proof that $\Gamma_1, x{:}t, \Gamma_2 \vdash e : t'$, and cases on the last rule used.

- Case: the proof ends with a use of Rule 1. Then e must be a constant integer n and $t' = \texttt{Int}$. Then $e[x{\mapsto}v] = n[x{\mapsto}v] = n$, so $\Gamma_1, \Gamma_2 \vdash e[x{\mapsto}v] : t'$ as required.

- Case: the proof ends with a use of Rule 4. Then e is a variable; there are two subcases, depending on whether e is the variable x or some other variable.

  - Subcase: $e = x$ and $t = t'$. Then $e[x{\mapsto}v] = v$ and by assumption we have $\Gamma_1 \vdash v : t$. By Weakening we have $\Gamma_1, \Gamma_2 \vdash v : t$. Thus $\Gamma_1, \Gamma_2 \vdash e[x{\mapsto}v] : t'$ as required.
  - Subcase: $e = y \neq x$ and $t' = (\Gamma_1, x{:}t, \Gamma_2)(y)$. Then $e[x{\mapsto}v] = y[x{\mapsto}v] = y$, and $(\Gamma_1, x{:}t, \Gamma_2)(y) = (\Gamma_1, \Gamma_2)(y)$, so by Rule 4 we have $\Gamma_1, \Gamma_2 \vdash y : (\Gamma_1, \Gamma_2)(y)$. That is, $\Gamma_1, \Gamma_2 \vdash e[x{\mapsto}v] : t'$.

- Case: the proof ends with a use of Rule 6. Then e is $e_1 \texttt{ < } e_2$ and $t = \texttt{Bool}$, and there are sub-proofs $\Gamma_1, x{:}t, \Gamma_2 \vdash e_1 : \texttt{Int}$ and $\Gamma_2, x{:}t, \Gamma_2 \vdash e_2 : \texttt{Int}$. By the inductive hypothesis $\Gamma_1, \Gamma_2 \vdash e_1[x{\mapsto}v] : \texttt{Int}$ and $\Gamma_1, \Gamma_2 \vdash e_2[x{\mapsto}v] : \texttt{Int}$. Thus $\Gamma_1, \Gamma_2 \vdash (e_1[x{\mapsto}v]) \texttt{ < } (e_2[x{\mapsto}v]) : \texttt{Bool}$. That is, $\Gamma_1, \Gamma_2 \vdash ((e_1 \texttt{ < } e_2)[x{\mapsto}v]) : \texttt{Bool}$.

- The remaining cases are similar and left to the reader.

∎

3 Dynamic Semantics

3.1 Definition

The following rules define a *small-step* operational semantics, i.e., a binary relation $\rightarrow$ between programs (which in this case are simply expressions) such that $e \rightarrow e'$ if and only if starting with e and doing one step of execution yields e'.

The transitive, reflexive closure of $\rightarrow$ is denoted $\rightarrow^*$. Formally, this is the smallest binary relation (the relation that relates the fewest expressions) that is reflexive, transitive, and

satisfies "if $e \to e'$ then $e \to^* e'$". More intuitively, $e \to^* e'$ means that there is a sequence of zero or more expressions satisfying

$$e \to^* e_1 \to e_2 \to \cdots \to e_n \to e'$$

or equivalently that e evaluates in zero or more steps to e'.

The style of definition is the Structural Operational Semantics (SOS) framework due to Gordon Plotkin. In general, evaluation is similar to SML in that it is left-to-right and eager (call-by-value).

$$\frac{e_1 \;\to\; e_1'}{e_1 + e_2 \;\to\; e_1' + e_2} \tag{11}$$

$$\frac{e_2 \;\to\; e_2'}{v_1 + e_2 \;\to\; v_1 + e_2'} \tag{12}$$

$$\frac{}{\overline{\mathtt{m}_1} + \overline{\mathtt{m}_2} \;\to\; \overline{\mathtt{m}_1 + \mathtt{m}_2}} \tag{13}$$

$$\frac{e_1 \;\to\; e_1'}{e_1 < e_2 \;\to\; e_1' < e_2} \tag{14}$$

$$\frac{e_2 \;\to\; e_2'}{v_1 < e_2 \;\to\; v_1 < e_2'} \tag{15}$$

$$\frac{}{\overline{\mathtt{m}_1} < \overline{\mathtt{m}_2} \;\to\; \overline{\mathtt{m}_1 < \mathtt{m}_2}} \tag{16}$$

$$\frac{e_1 \;\to\; e_1'}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \;\to\; \mathtt{if}\ e_1'\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3} \tag{17}$$

$$\frac{}{\mathtt{if}\ \overline{\mathtt{tt}}\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \;\to\; e_2} \tag{18}$$

$$\frac{}{\mathtt{if}\ \overline{\mathtt{ff}}\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \;\to\; e_3} \tag{19}$$

$$\frac{e_1 \;\to\; e_1'}{\mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2 \;\to\; \mathtt{let}\ x\ \mathtt{be}\ e_1'\ \mathtt{in}\ e_2} \tag{20}$$

$$\frac{}{\mathtt{let}\ x\ \mathtt{be}\ v_1\ \mathtt{in}\ e_2 \;\to\; e_2[x \mapsto v_1]} \tag{21}$$

$$\frac{e_1 \;\to\; e_1'}{e_1\, e_2 \;\to\; e_1'\, e_2} \tag{22}$$

$$\frac{e_2 \;\to\; e_2'}{v_1\, e_2 \;\to\; v_1\, e_2'} \tag{23}$$

$$\frac{}{(\mathtt{fun}\ f(x{:}t_1){:}t_2\ \mathtt{is}\ e_2)\, v \;\to\; (e_2[x \mapsto v])[f \mapsto (\mathtt{fun}\ f(x{:}t_1){:}t_2\ \mathtt{is}\ e_2)]} \tag{24}$$

3.2 Properties

Proposition 5
Values cannot be further evaluated. That is, for any value v there is no e satisfying $v \to e$.

Proof: By inspection of the dynamic semantics; there are no rules that would allow you to conclude that a value evaluates to anything. ∎

Proposition 6 (Determinacy of Evaluation)
Given an expression e there is at most one expression e' such that $e \to e'$. Equivalently, if $e \to e'$ and $e \to e''$ then $e' = e''$.

Proof: By induction on the structure of e (or by induction on the length of the expression e) and by cases on the form of e. ∎

Corollary 7
If $e \to^* v_1$ and $e \to^* v_2$ then $v_1 = v_2$.

4 Type Soundness

Proposition 8 (Type Preservation)
If $\vdash e : t$ and $e \to e'$ then $\vdash e' : t$.

Proof: By induction on the proof that $e \to e'$.

- Case: Rule 11. That is, $e = e_1 \texttt{+} e_2$ and $e' = e_1' \texttt{+} e_2'$ where $e_1 \to e_1'$. By Inversion applied to the assumption $\vdash e : t$, we have that $t = \texttt{Int}$ and that $\vdash e_1 : \texttt{Int}$ and that $\vdash e_2 : \texttt{Int}$. Thus by the inductive hypothesis applied to e_1 we have $\vdash e_1' : \texttt{Int}$. By Rule 5, then, $\vdash e_1' \texttt{+} e_2 : \texttt{Int}$.

- Case: Rule 12. Analogous to the previous case.

- Case: Rule 13. Then $e = \overline{\texttt{m}_1} \texttt{+} \overline{\texttt{m}_2}$ and $e' = \overline{\texttt{m}_1 \texttt{+} \texttt{m}_2}$. By Inversion, $t = \texttt{Int}$. By Rule 1, we have $\vdash \overline{\texttt{m}_1 \texttt{+} \texttt{m}_2} : \texttt{Int}$.

- Case: Rule 18. Then $e = \texttt{if } \overline{\texttt{tt}} \texttt{ then } e_2 \texttt{ else } e_3$ and $e' = e_2$. By Inversion, $\vdash e_2 : t$, which is what we needed to show.

- Case: Rule 20. Then $e = (\texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2)$ and $e' = (\texttt{let } x \texttt{ be } e_1' \texttt{ in } e_2)$ where $e_1 \to e_1'$. By Inversion, there exists a type t_1 such that $\vdash e_1 : t_1$ and $x{:}t_1 \vdash e_2 : t$. By the inductive hypothesis applied to e_1, we have $\vdash e_1' : t_1$. Thus by Rule 8, we have $\vdash \texttt{let } x \texttt{ be } e_1' \texttt{ in } e_2 : t$.

- Case: Rule 21. Then $e = (\texttt{let } x \texttt{ be } v_1 \texttt{ in } e_2)$ and $e' = e_2[x \mapsto v_1]$. By Inversion, there exists a type t_1 such that $\vdash v_1 : t_1$ and $x{:}t_1 \vdash e_2 : t_2$. By Substitution, $\vdash e_2[x \mapsto v_1] : t$.

- Case: Rule 24. Then $e = (\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2)\, v_1$ and $e' = (e_2[x{\mapsto}v_1])[f{\mapsto}(\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2)]$. By Inversion, there exists a type u such that $\vdash (\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2) : u\texttt{->}t$ and $\vdash v_1 : u$. By Inversion again, $u = t_1$ and $t = t_2$ and $f{:}t_1\texttt{->}t_2, x{:}t_1 \vdash e_2 : t_2$. By Weakening $f{:}t_1\texttt{->}t_2 \vdash v_1 : t_1$, so by Substitution we have $f{:}t_1\texttt{->}t_2 \vdash (e_2[x{\mapsto}v_1]) : t_2$. By Substitution again, $\vdash (e_2[x{\mapsto}v_1])[f{\mapsto}(\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2)] : t_2$.
- The remaining cases are analogous and left to the reader.

∎

Lemma 9 (Canonical Forms)

1. *If* $\vdash v : \texttt{Int}$ *then* v *is an integer constant* n.
2. *If* $\vdash v : \texttt{Bool}$ *then* v *is either* $\overline{\texttt{tt}}$ *or* $\overline{\texttt{ff}}$.
3. *If* $\vdash v : t_1\texttt{->}t_2$ *then* v *is a function of the form* $\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2$.

Proof: By inspection of the typing rules for values. ∎

Proposition 10 (Progress)
If $\vdash e : t$ *then either* e *is a value or there exists* e' *such that* $e \to e'$.

Proof: By induction on the proof of $\vdash e : t$, and cases on the last rule used.

- Case: Rule 1. Then $e = n$ is a value.
- Case: Rule 2 or 3. e is again a value.
- Case: Rule 4. This case cannot occur, since we are assuming that e is well-typed in an *empty* typing context.
- Case: Rule 5. Then $e = e_1 \texttt{+} e_2$ and $t = \texttt{Int}$ and there are sub-proofs $\vdash e_1 : \texttt{Int}$ and $\vdash e_2 : \texttt{Int}$.
  - Subcase: e_1 is not a value. By the inductive hypothesis, there exists e_1' such that $e_1 \to e_1'$. Thus by Rule 11 we have $e_1 \texttt{+} e_2 \to e_1' \texttt{+} e_2$.
  - Subcase: e_1 is a value, but e_2 is not. By the inductive hypothesis there exists e_2' such that $e_2 \to e_2'$. By Rule 12 we have $e_1 \texttt{+} e_2 \to e_1 \texttt{+} e_2'$.
  - Subcase: e_1 and e_2 are both values. By the Canonical Forms Lemma, e_1 and e_2 must be integer constants $\overline{\texttt{m}_1}$ and $\overline{\texttt{m}_2}$. Thus $e_1 \texttt{+} e_2 \to \overline{\texttt{m}_1 + \texttt{m}_2}$ by Rule 13.
- Case: Rule 6. Analogous to the previous case.
- Case: Rule 7. Then $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ and there are sub-proofs of $\vdash e_1 : \texttt{Bool}$ and $\vdash e_2 : t$ and $\vdash e_3 : t$. There are two subcases to consider.
  - Subcase: e_1 is not a value. By the inductive hypothesis there exists e_1' such that $e_1 \to e_1'$, and so by Rule 17 we have $e \to \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3$.

- Subcase: e_1 is a value. By the Canonical Forms Lemma, either $e_1 = \overline{\texttt{tt}}$ or $e_1 = \overline{\texttt{ff}}$. In the former case $e \rightarrow e_2$ and otherwise $e \rightarrow e_3$.

- Case: Rule 8. Then $e = \texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2$ and there are sub-proofs of $\vdash e_1 : t_1$ and $x{:}t_1 \vdash e_2 : t_2$. There are two subcases to consider.
  - Subcase: e_1 is not a value. Then by the inductive hypothesis there exists e_1' such that $e_1 \rightarrow e_1'$ and so $e \rightarrow \texttt{let}\ x\ \texttt{be}\ e_1'\ \texttt{in}\ e_2$.
  - Subcase: e_1 is a value. then $e \rightarrow e_2[x{\mapsto}e_1]$ by Rule 21.

- Case: Rule 9. Then $e = e_1\, e_2$ and there are subproofs of $\vdash e_1 : t_2\texttt{->}t$ and $\vdash e_2 : t_2$. There are three subcases to consider.
  - Subcase: e_1 is not a value. Then by the inductive hypothesis $e_1 \rightarrow e_1'$ for some e_1', and so by Rule 22 we have $e \rightarrow e_1'\, e_2$.
  - Subcase: e_1 is a value but e_2 is not. Then by the inductive hypothesis $e_2 \rightarrow e_2'$ for some e_2', and so by Rule 23 we have $e \rightarrow e_1\, e_2'$.
  - Subcase: e_1 and e_2 are both values. By the Canonical Forms Lemma, e_1 must be a function $\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2'$. Therefore by Rule 24 we have $e \rightarrow (e_2'[x{\mapsto}e_2])$.

- Case: Rule 10. Then e is a function value.

∎

Theorem 11 (Type Safety)
If $\vdash e : t$ then either evaluation of e terminates with a value or there is an infinite sequence of evaluation steps. In particular, evaluation of e can never reach a point where it gets stuck.

Proof: A consequence of Type Preservation and Progress. ∎