A Simple Type System

Types play a pivotal rôle in the design of modern programming languages. We here present a simple type system $ST$ and establish its type soundness. The syntax of $ST$ is given as follows:

- **booleans** $b$ ::= $true$ | $false$
- **integers** $i$ ::= $0$ | $1$ | $-1$ | $2$ | $-2$ | ...$
- **terms** $t$ ::= $b$ | $i$ | $x$ | $c(t_1, \ldots, t_n)$ | $\langle t_1, t_2 \rangle$ | $\text{fst}(t)$ | $\text{snd}(t)$ | $\text{lam} \, x \, t$ | $\text{app} \, (t_1, t_2)$ | $\text{if}(t_1; t_2; t_3)$ | $\text{fix} \, f(x).t$
- **values** $v$ ::= $b$ | $i$ | $x$ | $\langle v_1, v_2 \rangle$ | $\text{lam} \, x \, t$
- **types** $T$ ::= $\text{bool}$ | $\text{int}$ | $T_1 \, \ast \, T_2$ | $T_1 \rightarrow T_2$
- **contexts** $\Gamma$ ::= $\emptyset$ | $\Gamma, x : T$

We use $b$ and $i$ for booleans and integers, respectively, and $x$ for variables. There are terms and types in $ST$, and we are to present rules for assigning types to terms. We say that a term $t$ is of type $T$ if $T$ can be assigned to $t$ according to such type assignment rules. We use $c$ for a builtin constant function such as addition $+$ and subtraction $-$, and $c(t_1, \ldots, t_n)$ for the application of a constant function $c$ to $n$ arguments $t_1, \ldots, t_n$. We use $\langle t_1, t_2 \rangle$ for forming a pair and $\text{fst}(t)$ and $\text{snd}(t)$ for the first and second projections. In addition, we use $\text{lam} \, x \, t$ for lambda-abstraction and $\text{app} \, (t_1, t_2)$ for function application.

We use $v$ for values, which are a special form of terms; both booleans and integers are values; a pair of values is a value; a lambda-abstraction is also a value.

The types $\text{bool}$ and $\text{int}$ are for booleans and integers, respectively. Given types $T_1$ and $T_2$, we can form a type $T_1 \, \ast \, T_2$ for pairs whose first and second components are of types $T_1$ and $T_2$, respectively; also we can form a type $T_1 \rightarrow T_2$ for functions that returns a value of type $T_2$ when applied to an argument of type $T_1$.

We use $\Gamma \vdash t : T$ for a typing judgment meaning that the term $t$ can be assigned the type $T$ under the context $\Gamma$. The rule for deriving typing judgments are give in Figure 1.

**Lemma 1 (Canonical Forms)** Assume that $\emptyset \vdash v : T$ is derivable.

1. If $T = \text{bool}$, then $v$ is a boolean value $b$.
2. If $T = \text{int}$, then $v$ is an integer value $b$.
3. If $T = T_1 \, \ast \, T_2$, then $v$ is of the form $\langle v_1, v_2 \rangle$.
4. If $T = T_1 \rightarrow T_2$, then $v$ is of the form $\text{lam} \, x \, t$.

**Lemma 2 (Substitution)** Assume that $\Gamma \vdash t_1 : T_1$ and $\Gamma, x : T_1 \vdash t_2 : T_2$ are derivable. Then $\Gamma \vdash t_2[x \rightarrow t_1] : T_2$ is also derivable.

We write $t_1 \rightarrow t_2$ to mean that $t_1$ reduces to $t_2$ in one step, and the reduction rules are given in Figure 2.
\[ \begin{align*}
\Gamma & \vdash b : \text{bool} \quad \text{(bool)} \\
\Gamma & \vdash i : \text{int} \quad \text{(int)} \\
x & : T \in \Gamma \\
\Gamma & \vdash x : T \quad \text{(var)} \\
\vdash c(T_1, \ldots, T_n) : T & \quad \Gamma \vdash t_1 : T_1 \quad \ldots \quad \Gamma \vdash t_n : T_n \quad \text{(const)} \\
\Gamma & \vdash c(t_1, \ldots, t_n) : T \\
\Gamma & \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad \text{(tup)} \\
\Gamma & \vdash \langle t_1, t_2 \rangle : T_1 \times T_2 \\
\Gamma & \vdash t : T_1 \times T_2 \\
\Gamma & \vdash \text{fst}(t) : T_1 \quad \text{(fst)} \\
\Gamma & \vdash t : T_1 \times T_2 \\
\Gamma & \vdash \text{snd}(t) : T_2 \quad \text{(snd)} \\
\Gamma, x : T_1 & \vdash t : T_2 \\
\Gamma & \vdash \text{lam} x : t : T_1 \rightarrow T_2 \quad \text{(lam)} \\
\Gamma & \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1 \\
\Gamma & \vdash \text{app}(t_1, t_2) : T_2 \quad \text{(app)} \\
\Gamma & \vdash \text{bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T \\
\Gamma & \vdash \text{if}(t_1, t_2, t_3) : T \quad \text{(if)} \\
\Gamma, f : T_1 & \rightarrow T_2, x : T_1 \vdash t : T_2 \\
\Gamma & \vdash \text{fix} f(x). t : T_1 \rightarrow T_2 \quad \text{(fix)}
\end{align*} \]

Figure 1: The typing rules
Figure 2: The reduction rules
Theorem 1 (Subject Reduction) Assume that $\varnothing \vdash t : T$ is derivable and $t \rightarrow t'$ holds. Then $\varnothing \vdash t' : T$ is also derivable.

Theorem 2 (Progress) Assume that $\varnothing \vdash t : T$ is derivable. Then $t$ is either a value, or $t \rightarrow t'$ for some term $t'$.

We now present a slightly different means to assign dynamic semantics to terms in ST. We first introduce the notion of evaluation contexts:

$$
E ::= [] | c(v_1, \ldots, v_n, E, t_{i+1}, \ldots, t_n) | \langle E, t \rangle | \langle v, E \rangle | \text{fst}(E) | \text{snd}(E) | \text{app}(E, t) | \text{app}(v, E) | \text{if}(E, t_2, t_3)
$$

We then introduce the notion of redexes as follows:

Definition 3 We define redexes and their reductions as follows.

1. $c(v_1, \ldots, v_n)$ is a redex if $c$ is a built-in function and $c(v_1, \ldots, v_n)$ is defined to equal $v$, and the reduction of $c(v_1, \ldots, v_n)$ is $v$.
2. $\text{app}(\text{lam} x : t, v)$ is a redex, and its reduction is $t[x \mapsto v]$.
3. $\text{fst}(\langle v_1, v_2 \rangle)$ is a redex, and its reduction is $v_1$.
4. $\text{snd}(\langle v_1, v_2 \rangle)$ is a redex, and its reduction is $v_2$.
5. $\text{if}(\text{true}, t_1, t_2)$ is a redex, and its reduction is $t_1$.
6. $\text{if}(\text{false}, t_1, t_2)$ is a redex, and its reduction is $t_2$.
7. $\text{fix} f(x).t$ is a redex and its reduction is $\text{lam} x . t[f \mapsto \text{fix} f(x).t]$.

Given $t_1 = E[t]$ and $t_2 = E[t']$ for some redex $t$ and its reduction $t'$, we write $t_1 \rightarrow t_2$ and say that $t_1$ reduces to $t_2$ in one step. Let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$.

We now make some additional adjustments in order to support the language constructs callcc and throw.

$$
t ::= \ldots | \text{callcc} (\text{lam} x . t) | \text{throw}(t_1, t_2) | *E
$$

$$
E ::= \ldots | \text{throw}(E, t) | \text{throw}(v, E)
$$

The new forms of evaluation rules are given as follows:

$$
E[\text{callcc}(\text{lam} x . t)] \rightarrow E[t[x \mapsto *E]] \quad E[\text{throw}(*E', v)] \rightarrow E'[v]
$$