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(t₁, ..., tₙ) | ⟨t₁, t₂⟩ | fst(t) | snd(t) | lam x.t | app(t₁, t₂) |
            | if(t₁, t₂, t₃) | fix f(x).t
values     v ::= b | i | x | ⟨v₁, v₂⟩ | lam x.t
types      T ::= bool | int | T₁ ∗ T₂ | T₁ → T₂
contexts   Γ ::= ∅ | Γ, 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(t₁, ..., tₙ) for the application of a constant function c to n arguments t₁, ..., tₙ. We use ⟨t₁, t₂⟩ for forming a pair and fst(t) and snd(t) for the first and second projections. In addition, we use lam x.t for lambda-abstraction and app(t₁, t₂) 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-abstract is also a value.

The types bool and int are for booleans and integers, respectively. Given types T₁ and T₂, we can form a type T₁ ∗ T₂ for pairs whose first and second components are of types T₁ and T₂, respectively; also we can form a type T₁ → T₂ for functions that returns a value of type T₂ when applied to an argument of type T₁.

We use Γ ⊢ t : T for a typing judgment meaning that the term t can be assigned the type T under the context Γ. The rule for deriving typing judgments are given in Figure 1.

Lemma 1 (Canonical Forms) Assume that ∅ ⊢ v : T is derivable.

1. If T = bool, then v is a boolean value b.
2. If T = int, then v is an integer value b.
3. If T = T₁ ∗ T₂, then v is of the form ⟨v₁, v₂⟩.
4. If T = T₁ → T₂, then v is of the form lam x.t.

Lemma 2 (Substitution) Assume that Γ ⊢ t₁ : T₁ and Γ, x : T₁ ⊢ t₂ : T₂ are derivable. Then Γ ⊢ t₂[x ← t₁] : T₂ is also derivable.

We write t₁ → t₂ to mean that t₁ reduces to t₂ in one step, and the reduction rules are given in Figure 2.
\[
\frac{\Gamma \vdash b : \text{bool}}{
\quad \text{(bool)}
}
\]
\[
\frac{\Gamma \vdash i : \text{int}}{
\quad \text{(int)}
}
\]
\[
\frac{x : T \in \Gamma \quad \Gamma \vdash x : T}{
\quad \text{(var)}
}\]
\[
\frac{\Gamma \vdash c(T_1, \ldots, T_n) : T \quad \Gamma \vdash t_1 : T_1 \quad \ldots \quad \Gamma \vdash t_n : T_n}{
\quad \text{(const)}
}\]
\[
\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{
\quad \text{(tup)}
}\]
\[
\frac{\Gamma \vdash t : T_1 \ast T_2}{
\quad \text{(fst)}
}\]
\[
\frac{\Gamma \vdash t : T_1 \ast T_2}{
\quad \text{(snd)}
}\]
\[
\frac{\Gamma, x : T_1 \vdash t : T_2}{
\quad \text{(lam)}
}\]
\[
\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{
\quad \text{(app)}
}\]
\[
\frac{\Gamma \vdash t_1 : \text{bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{
\quad \text{(if)}
}\]
\[
\frac{\Gamma, f : T_1 \rightarrow T_2, x : T_1 \vdash t : T_2}{
\quad \text{(fix)}
}\]

Figure 1: The typing rules
\[
\begin{align*}
t_i & \rightarrow t'_i \\
c(v_1, \ldots, v_{i-1}, t_i, t_{i+1}, \ldots, t_n) & \rightarrow c(v_1, \ldots, v_{i-1}, t'_i, t_{i+1}, \ldots, t_n) \\
c(v_1, \ldots, v_n) & = v \\
c(v_1, \ldots, v_n) & \rightarrow v \\
t_1 & \rightarrow t'_1 \\
\langle t_1, t_2 \rangle & \rightarrow \langle t'_1, t_2 \rangle \\
t_2 & \rightarrow t'_2 \\
\langle v_1, t_2 \rangle & \rightarrow \langle v_1, t'_2 \rangle \\
\text{fst}(\langle v_1, v_2 \rangle) & \rightarrow v_1 \\
\text{snd}(\langle v_1, v_2 \rangle) & \rightarrow v_2 \\
t_1 & \rightarrow t'_1 \\
\text{app}(t_1, t_2) & \rightarrow \text{app}(t'_1, t_2) \\
t_2 & \rightarrow t'_2 \\
\text{app}(v_1, t_2) & \rightarrow \text{app}(v_1, t'_2) \\
\text{app}(\text{lam } x. t, v) & \rightarrow t[x \mapsto v] \\
t_1 & \rightarrow t'_1 \\
\text{if}(t_1, t_2, t_3) & \rightarrow \text{if}(t'_1, t_2, t_3) \\
\text{if}(\text{true}, t_1, t_2) & \rightarrow t_1 \\
\text{if}(\text{false}, t_1, t_2) & \rightarrow t_2 \\
\text{fix } f(x) . t & \rightarrow \text{lam } x. t[f \mapsto \text{fix } f(x). t]
\end{align*}
\]

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

Theorem 2 (Progress) Assume that $\emptyset \vdash t : T$ 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:

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

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. $\text{fst}((v_1, v_2))$ is a redex, and its reduction is $v_1$.
4. $\text{snd}((v_1, v_2))$ 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.

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

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

The new forms of evaluation rules are given as follows:

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