Reinterpret Types

1 INTRO

Here is a formalization of our type system.

2 CORE LANGUAGE

First, we'll define a small core language with basic integers, booleans, and functions.

$$
\begin{array}{rcll}
e & ::= & \mathbb{Z} \mid \mathbb{B} \mid x \mid \texttt{fun } x \texttt{ -> } e \mid e\ e & \textit{expressions} \\
x & ::= & \textit{(identifiers)} & \textit{variables} \\
v & ::= & \mathbb{Z} \mid \mathbb{B} \mid \texttt{fun } x \texttt{ -> } e \mid x & \textit{values} \\
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \tau \texttt{ -> } \tau & \textit{types}
\end{array}
$$

Fig. 1. Core language grammar

The typing rules of the system is defined as following:

Definition 2.1 (Typing rules).

(1) $\models e : \texttt{int}$ iff $e \Longrightarrow v, v \in \mathbb{Z}$.
(2) $\models e : \texttt{bool}$ iff $e \Longrightarrow v, v \in \mathbb{B}$.
(3) $\models e : \tau_1 \texttt{ -> } \tau_2$ iff $\forall v$ such that $\models v : \tau_1, \models e\ v : \tau_2$.

3 LANGUAGE EXTENSIONS

Next, we will define a couple of languages extensions and their corresponding typing rules.

$$
\begin{array}{rcll}
e & ::= & \cdots \mid a & \textit{expressions} \\
v & ::= & \cdots \mid a & \textit{values} \\
\tau & ::= & \cdots \mid \alpha \mid \tau \cup \tau \mid \tau \cap \tau \mid \{\tau \mid e\} \mid (x : \tau) \texttt{ -> } \tau \mid \mu\alpha.\tau & \textit{types}
\end{array}
$$

Fig. 2. Extended language grammar

Definition 3.1 (More typing rules).

(1) $\models e : \alpha$ iff $e \Longrightarrow a$, where $\textsc{typeof}(a) = \alpha$.
(2) $\models e : \tau_1 \cup \tau_2$ iff $\models e : \tau_1$ or $\models e : \tau_2$.
(3) $\models e : \tau_1 \cap \tau_2$ iff $\models e : \tau_1$ and $\models e : \tau_2$.
(4) $\models e : \{\tau \mid p\}$ iff $\models e : \tau$ and $p\ e \Longrightarrow \texttt{true}$.
(5) $\models e : (x : \tau_1) \texttt{ -> } \tau_2$ iff $\forall v$ such that $\models v : \tau_1, \models e\ v : \tau_2[v/x]$.
(6) $\models e : \mu\alpha.\tau$ iff ?.

We will now extend the language with records.

$$
\begin{array}{rcll}
e & ::= & \cdots \mid \{\overline{\ell = e}\}^{\{\bar{\ell}\}} \mid e.\ell & \textit{expressions} \\
v & ::= & \cdots \mid \{\overline{\ell = v}\}^{\{\bar{\ell}\}} & \textit{values} \\
\tau & ::= & \cdots \mid \{\overline{\ell : \tau}\} & \textit{types}
\end{array}
$$

Fig. 3. Extended language grammar (with records)

Definition 3.2 (Record typing rules).

(1) $\models e : \{\ell_1 : \tau_1, \cdots, \ell_m : \tau_m\}$ iff $e \Longrightarrow \{\ell_1 = v_1, \cdots, \ell_m = v_m, \cdots, \ell_n = v_n\}^{\{\ell_1, \cdots, \ell_p\}}$ where $\models v_i : \tau_i$ for $i \in \{1, \cdots, m\}$, $n \geq p \geq m$.

e	::=	$\mathbb{Z} \mid \mathbb{B} \mid x \mid$ `fun` x `->` $e \mid e\ e \mid e \odot e \mid a \mid \{\overline{\ell = e}\}^{\{\bar{\ell}\}} \mid e.\ell$ $\mid$ `if` e `then` e `else` $e \mid$ $\texttt{pick}_i \mid \texttt{pick}_b \mid$ `mzero` $\mid$ `ERROR` $\mid$ `let` x `=` e `in` $e \mid$ `let f x = e in e` $\mid e \sim p$	*expressions*
x	::=	*(identifiers)*	*variables*
p	::=	`int` $\mid$ `bool` $\mid$ `fun` $\mid$ `any` $\mid a \mid \{\bar{\ell}\}$	*patterns*
v	::=	$\mathbb{Z} \mid \mathbb{B} \mid$ `fun` x `->` $e \mid x \mid a \mid \{\overline{\ell = v}\}^{\{\bar{\ell}\}}$	*values*
τ	::=	`int` $\mid$ `bool` $\mid \tau$ `->` $\tau \mid \alpha \mid \tau \cup \tau \mid \tau \cap \tau \mid \{\tau \mid e\} \mid (x : \tau)$ `->` $\tau \mid \mu\alpha.\tau \mid \{\overline{\ell : \tau}\}$	*types*

Fig. 4. Complete language grammar

4 TYPE AS VALUES

In this section, we will demonstrate how to represent each type using a tuple of functions `generator` and `checker`.

Definition 4.1 (Semantic interpretation of types). We define the semantic interpretation of types as $[\![\tau]\!] = \{$`gen = generator(`τ`), check = fun` e `->` `checker(`τ, e`)`$\}$.

Definition 4.2 (Defining Generator in the core language).

(1) `generator(int)` : $\texttt{pick}_i$.
(2) `generator(bool)` : $\texttt{pick}_b$.
(3) `generator(`τ_1 `->` τ_2`)` : `fun` x `->` `generator(`τ_2`)`.

Definition 4.3 (Defining Checker in the core language).

(1) `checker(int,`e`)` : $e \sim$ `int`.
(2) `checker(bool,`e`)` : $e \sim$ `bool`.
(3) `checker(`τ_1 `->` τ_2, e`)` : `let arg = generator(`τ_1`) in checker(`τ_2`, (e arg))`.

Definition 4.4 (Defining Generator in the extended language).

(1) `generator(`α_i`)` : a_i.
(2) `generator(`$\tau_1 \cup \tau_2$`)` : $\texttt{pick}_b$. `if` b `then generator(`τ_1`) else generator(`τ_2`)`.
(3) `generator(`$\tau_1 \cap \tau_2$`)` where τ_1, τ_2 are not arrow types or record types : `pick` $b \in \mathbb{B}$.
```
if b then
  let gend = generator(τ1) in
  if checker(τ2, gend) then gend else mzero
else
  let gend = generator(τ2) in
  if checker(τ1, gend) then gend else mzero
```
(4) `generator(`$\tau_1 \cap \tau_2$`)` where $\tau_1 = \tau_{dom1}$ `->` $\tau_{cod1}, \tau_2 = \tau_{dom2}$ `->` τ_{cod2} :
```
fun x ->
  if checker(τdom1, x) then generator(τcod1) else generator(τcod2).
```
(5) `generator(`$\tau_1 \cap \tau_2$`)` where
$\tau_1 = \{\ell_1 : \tau'_1, \cdots, \ell_n : \tau'_n, \cdots, \ell_{11} : \tau'_{11}, \cdots, \ell_{1m} : \tau'_{1m}\}$,
$\tau_2 = \{\ell_1 : \tau''_1, \cdots, \ell_n : \tau''_n, \cdots, \ell_{21} : \tau''_{21}, \cdots, \ell_{2n} : \tau''_{2n}\}$:
$\{\ell_1 =$ `generator(`$\tau'_1 \cap \tau''_1$`)`$, \cdots, \ell_n =$ `generator(`$\tau'_n \cap \tau''_n$`)`$, \cdots, \ell_{11} = \tau_{11}, \cdots, \ell_{2n} = \tau'_{2n}\}$.
(6) `generator(`$\{\tau \mid p\}$`)` :
`let gend = generator(`τ`) in if (p gend) then gend else mzero`.

(7) $\texttt{generator}((x : \tau_1) \texttt{ -> } \tau_2)$:
$\texttt{fun } x' \texttt{ -> if checker}(\tau_1, x') \texttt{ then generator}(\tau_2[x'/x]) \texttt{ else ERROR}$.
(8) $\texttt{generator}(\mu\alpha.\tau) : \texttt{generator}(\tau[\alpha/\mu\alpha.\tau])$.
(9) $\texttt{generator}(\{\ell_1 : \tau_1, \cdots, \ell_n : \tau_n\})$:
$\texttt{let } v_1 = \texttt{generator}(\tau_1) \texttt{ in } \cdots \texttt{ let } v_n = \texttt{generator}(\tau_n) \texttt{ in } \{\ell_1 = v_1, \cdots, \ell_n = v_n\}$.

Definition 4.5 (Defining Checker in the extended language).

(1) $\texttt{checker}(\alpha_i, e) : e \sim a_i$.
(2) $\texttt{checker}(\tau_1 \cup \tau_2, e) : \texttt{checker}(\tau_1, e) \texttt{ or checker}(\tau_2, e)$.
(3) $\texttt{checker}(\tau_1 \cap \tau_2, e) : \texttt{checker}(\tau_1, e) \texttt{ and checker}(\tau_2, e)$.
(4) $\texttt{checker}(\{\tau \mid p\}, e) : \texttt{checker}(\tau, e) \texttt{ and eval}(e) = \texttt{true}$.
(5) $\texttt{checker}((x : \tau_1) \texttt{ -> } \tau_2, e) : \texttt{let arg = generator}(\tau_1) \texttt{ in checker}(\tau_2[\texttt{arg}/x], \texttt{(e arg)})$.
(6) $\texttt{checker}(\mu\alpha.\tau, e) : \texttt{checker}(\tau[\mu\alpha.\tau/\alpha], e)$.
(7) $\texttt{checker}(\{\ell_1 : \tau_1, \cdots, \ell_m : \tau_m\}, e) : \texttt{eval}(e) = \{\ell_1 = v_1, \cdots, \ell_m = v_m, \cdots, \ell_n = v_n\}^{\{\ell_1, \cdots, \ell_m\}}$
$\texttt{and checker}(\tau_1, v_1) \cdots \texttt{ and checker}(\tau_m, v_m)$.

5 SELECTIVE TYPECHECKING

We allow users to declare types in their program selectively. If an expression doesn't have a type declaration, we assume that the user does not wish for us to check its type. In other words, we will only be checking explicitly declared types in the user program.

$$e \quad ::= \quad \cdots \mid \texttt{let f (x : } \tau \texttt{) : } \tau \texttt{ = e in e} \mid \texttt{let (} x \texttt{ : } \tau \texttt{) = } e \texttt{ in } e \qquad \textit{expressions}$$

Fig. 5. Updated language grammar