Refutation-based Typechecking via Symbolic Execution

1 INTRO

In this section, we will first provide the model theory definition of types in our language. However, since in real programs, we cannot effectively perform certain mathematical enumerations (such as enumeration of functions), we will need an operational-semantics-based definition of how typechecking works in actual programs. As such, we will provide a proof theory definition of types, and prove equivalence of the two systems.

2 CORE LANGUAGE

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

$$
\begin{array}{rcll}
e & ::= & \mathbb{Z} \mid \mathbb{B} \mid x \mid \texttt{fun}\ x\ \texttt{->}\ e \mid e\ e & \textit{expressions} \\
 & & \mid e \odot e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e & \\
 & & \mid \texttt{pick}_i \mid \texttt{pick}_b \mid \texttt{ERROR} & \\
x & ::= & \textit{(identifiers)} & \textit{variables} \\
v & ::= & \mathbb{Z} \mid \mathbb{B} \mid \texttt{fun}\ x\ \texttt{->}\ e & \textit{values} \\
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \tau\ \texttt{->}\ \tau & \textit{types}
\end{array}
$$

Fig. 1. Core language grammar

2.1 Modeling Types

The typing rules of the system is defined as following:

Definition 2.1 (Typing rules in model theory).

(1) $\models e : \texttt{int}$ iff $\forall v. e \longrightarrow^* v, v \in \mathbb{Z}$.
(2) $\models e : \texttt{bool}$ iff $\forall v. e \longrightarrow^* v, v \in \mathbb{B}$.
(3) $\models e : \tau_1\ \texttt{->}\ \tau_2$ iff $\forall v_f. e \longrightarrow^* v_f$, and that $\forall v.$ if $\models v : \tau_1$, then $\models v_f\ v : \tau_2$.

Note that the rules do not actually check the types of the subexpressions. In fact, we can have an expression such as `Y (fun this -> fun n -> if n = 0 then 0 else this (n-1))` and assign the type `int -> int` to it, in spite of the fact that we cannot assign types to any of its subexpressions.

2.2 Proving Types

In this section, we will provide the proof theory definition of typechecking.

Definition 2.2 (Type Generator).

(1) $\texttt{generator(int)} = \texttt{pick}_i$
(2) $\texttt{generator(bool)} = \texttt{pick}_b$
(3) $\texttt{generator}(\tau_1\ \texttt{->}\ \tau_2) = \texttt{fun}\ x\ \texttt{->}\ \texttt{let}\ \_\ \texttt{=}\ [\![x]\!]_{\tau_1}\ \texttt{in}\ \texttt{generator}(\tau_2)$

Definition 2.3 (Type Checker).

(1) $\texttt{checker}(\texttt{int}, e) = \texttt{if}\ e \sim \texttt{int}\ \texttt{then}\ e\ \texttt{else}\ \texttt{ERROR}$
(2) $\texttt{checker}(\texttt{int}, e) = \texttt{if}\ e \sim \texttt{bool}\ \texttt{then}\ e\ \texttt{else}\ \texttt{ERROR}$

(3) $\texttt{checker}(\tau_1 \texttt{ -> } \tau_2, e) =$
$\quad \texttt{let arg = generator}(\tau_1) \texttt{ in checker}(\tau_2, \texttt{(e arg)})$

Definition 2.4 (Typing rules in proof theory).
$\vdash e : \tau$ iff $\forall v$. if $e \longrightarrow^* v$, then $\texttt{checker}(\tau, v) \not\longrightarrow^* \texttt{ERROR}$.

2.3 Completeness and Soundness

In this section, we will show that the two definitions are equivalent, by showing completeness and soundness.

Completeness. $\forall e$ if $\nvdash e : \tau$, then $\not\models e : \tau$.
This is equivalent to showing: if $\exists v.e \longrightarrow^* v$ and $\texttt{checker}(\tau, v) \longrightarrow^* \texttt{ERROR}$, then $\not\models e : \tau$.
We will prove it by induction on the structure of τ.
Base case: $\tau = \texttt{int}$
Given $\exists v.e \longrightarrow^* v$ and $\texttt{checker}(\texttt{int}, e) \longrightarrow^* \texttt{ERROR}$, we know that there are two potential sources for ERROR:

(1) $e \longrightarrow^* \texttt{ERROR}$. In this case, $\not\models e : \texttt{int}$ is trivial.
(2) $e \longrightarrow^* v$. In this case, we know that $v \sim \texttt{int} \longrightarrow^* \texttt{false}$, which means $v \notin \mathbb{Z}$. Thus $\not\models e : \texttt{int}$.

Inductive step: $\tau = \tau_1 \texttt{ -> } \tau_2$
Given $\texttt{checker}(\tau_1 \texttt{ -> } \tau_2, e) \longrightarrow^* \texttt{ERROR}$, since $\texttt{arg = generator}(\tau)$ is guaranteed to evaluate to a value, we know that ERROR must come from $\texttt{checker}(\tau_2, \texttt{(e arg)})$. This suggests that $\exists v.\texttt{(e arg)} \longrightarrow^* v$ and $\nvdash v : \tau_2$. By induction hypothesis, we know that $\not\models v : \tau_2$. By lemma 1, we know that $\models \texttt{generator}(\tau_1) : \tau_1$. Thus we have found a witness $\models v : \tau_1$ such that $\not\models (e\ v) : \tau_2$, proving that $\not\models e : \tau_1 \texttt{ -> } \tau_2$.

□

3 LANGUAGE EXTENSIONS

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

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

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.

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

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 $i \in \{1, \cdots, m\}$, $n \geq p \geq m$.

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

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 $[\![\tau]\!] = \{\texttt{gen = generator}(\tau)\texttt{, check = fun}\ e\ \texttt{-> checker}(\tau, e)\}$.

Definition 4.2 (Defining Generator in the extended language).

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

(1) $\texttt{checker}(\alpha_i, e) : e \sim a_i$.

(2) $\texttt{checker}(\tau_1 \cup \tau_2, e) : \texttt{checker}(\tau_1, e)$ `or` $\texttt{checker}(\tau_2, e)$.
(3) $\texttt{checker}(\tau_1 \cap \tau_2, e) : \texttt{checker}(\tau_1, e)$ `and` $\texttt{checker}(\tau_2, e)$.
(4) $\texttt{checker}(\{\tau \mid p\}, e) : \texttt{checker}(\tau, e)$ `and` $\texttt{eval}(e) = \texttt{true}$.
(5) $\texttt{checker}((x : \tau_1)$ `->` $\tau_2, e) :$ `let arg = generator`(τ_1) `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) $\texttt{checker}(\{\ell_1 : \tau_1, \cdots, \ell_m : \tau_m\}, e) : \texttt{eval}(e) = \{\ell_1 = v_1, \cdots, \ell_m = v_m, \cdots, \ell_n = v_n\}^{\{\ell_1, \cdots, \ell_m\}}$ `and` $\texttt{checker}(\tau_1, v_1) \cdots$ `and` $\texttt{checker}(\tau_m, v_m)$.

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