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.

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

Fig. 1. Core language grammar

2.1 Modeling Types Mathematically

The typing rules of the system is defined as following:

Definition 2.1 (Typing rules).

(1) $\models e : \texttt{int}$ iff $e \not\longrightarrow^* \texttt{ERROR}$ and $\forall v.e \longrightarrow^* v, v \in \mathbb{Z}$.
(2) $\models e : \texttt{bool}$ iff $e \not\longrightarrow^* \texttt{ERROR}$ and $\forall v.e \longrightarrow^* v, v \in \mathbb{B}$.
(3) $\models e : \tau_1 \texttt{ -> } \tau_2$ iff $e \not\longrightarrow^* \texttt{ERROR}$ and $\forall v_f$. if $e \longrightarrow^* v_f$, then $\forall v$. 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 Modeling Types Practically

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) $\texttt{generator}(\tau_1 \texttt{ -> } \tau_2) = \texttt{fun } x \texttt{ -> let \_ = checker}(\tau_1, x) \texttt{ in generator}(\tau_2)$

Definition 2.3 (Type Checker).

(1) $\texttt{checker(int},e) = \texttt{if}\ e \sim \texttt{int then}\ e\ \texttt{else ERROR}$
(2) $\texttt{checker(bool},e) = \texttt{if}\ e \sim \texttt{bool then}\ e\ \texttt{else ERROR}$
(3) $\texttt{checker}(\tau_1\ \texttt{->}\ \tau_2, e) =$
$\quad \texttt{let arg = generator}(\tau_1)\ \texttt{in let \_ = checker}(\tau_2, \texttt{(e arg)) in e}$

Definition 2.4 (Updated typing rule).
$\models_p e : \tau$ iff $\texttt{checker}(\tau, e) \not\longrightarrow^* \texttt{ERROR}$.

2.3 Completeness and Soundness

In this section, we will show that the two definitions are equivalent.

THEOREM 2.5. $\forall e.\ \not\models_p e : \tau$ *iff* $\not\models e : \tau$.

IF DIRECTION. $\forall e.$ if $\not\models_p e : \tau$, then $\not\models e : \tau$.
This is equivalent to showing: if $\texttt{checker}(\tau, e) \longrightarrow^* \texttt{ERROR}$, then $\not\models e : \tau$.
To prove this statement, we'll need the following lemma:

LEMMA 2.6. $\models \texttt{generator}(\tau) : \tau$.

We will prove the conjunction of the completeness statement and the lemma by induction on the structure of τ.

Base case: $\tau = \texttt{int}$

First, we will show that $\models \texttt{generator(int)} : \texttt{int}$.

Since $\texttt{generator(int)} = pick_i$, by definition of $\texttt{pick}_i$, we know that $\forall v.$ if $\texttt{pick}_i \longrightarrow^* v$, then $v \in \mathbb{Z}$. Thus, we have shown that $\models \texttt{generator(int)} : \texttt{int}$.

Next, we'll prove that for an arbitrary e, if $e : \texttt{int}$, then $\not\models e : \texttt{int}$.

By definition of $\vdash e : \texttt{int}$, we know that $\not\models_p e : \texttt{int}$ suggests $\texttt{checker(int}, e) \longrightarrow^* \texttt{ERROR}$. Examining the definition of $\texttt{checker(int}, e)$, we can see 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 $\not\models e : \texttt{int}$.

Proof for the case where $\tau = \texttt{bool}$ is very similar, so we'll omit it here for brevity.

Inductive step: $\tau = \tau_1\ \texttt{->}\ \tau_2$

First, we will show that $\models \texttt{generator}(\tau_1\ \texttt{->}\ \tau_2) : \tau_1\ \texttt{->}\ \tau_2$.

To prove this, we need to show that $\forall v.$ if $v : \tau_1$, then $\models (\texttt{generator}(\tau_1\ \texttt{->}\ \tau_2)\ v) : \tau_2$.

By definition, $(\texttt{generator}(\tau_1\ \texttt{->}\ \tau_2))\ v = \texttt{let \_ = checker}(\tau_1, v)\ \texttt{in generator}(\tau_2)$.

By inductive hypothesis, $\forall v.$ if $\models v : \tau_1$, then $\vdash v : \tau_1$, which means $\texttt{checker}(\tau_1, v) \not\longrightarrow^* \texttt{ERROR}$. In the case that $\texttt{checker}(\tau_1, v)$ diverges, $(\texttt{generator}(\tau_1\ \texttt{->}\ \tau_2))\ v$ will diverge, too, making the statement $\models (\texttt{generator}(\tau_1\ \texttt{->}\ \tau_2)\ v) : \tau_2$ trivially true. If $\texttt{checker}(\tau_1, v)$ doesn't diverge, we only have to consider $\texttt{generator}(\tau_2)$. By inductive hypothesis, we know that $\models \texttt{generator}(\tau_2) : \tau_2$. Therefore, we have shown that $\forall v.$ if $v : \tau_1$, then $\models (\texttt{generator}(\tau_1\ \texttt{->}\ \tau_2)\ v) : \tau_2$, which means $\models \texttt{generator}(\tau_1\ \texttt{->}\ \tau_2) : \tau_1\ \texttt{->}\ \tau_2$.

Next, we'll prove that for an arbitrary e, if $\not\models_p e : \tau_1\ \texttt{->}\ \tau_2$, then $\not\models e : \tau_1\ \texttt{->}\ \tau_2$.

By definition, $\texttt{checker}(\tau_1\ \texttt{->}\ \tau_2, e) = \texttt{let arg = generator}(\tau_1)\ \texttt{in checker}(\tau_2, \texttt{(e arg))}$. Since $\texttt{generator}(\tau_1)$ is guaranteed to evaluate to a value, we know that ERROR must come from $\texttt{checker}(\tau_2, \texttt{(e arg))}$. This suggests that $\not\models_p (e\ \texttt{arg}) : \tau_2$. By induction hypothesis, we know that $\not\models (e\ \texttt{arg}) : \tau_2$, and that $\models \texttt{generator}(\tau_1) : \tau_1$. Thus we have found a witness $\models \texttt{arg} : \tau_1$ such that $\not\models (e\ \texttt{arg}) : \tau_2$, proving that $\not\models e : \tau_1\ \texttt{->}\ \tau_2$.

□

Soundness. $\forall.e$ if $\not\models e : \tau$, then $\not\models_p e : \tau$.
This is equivalent to showing: if $\not\models e : \tau$, then $\exists v.e \longrightarrow^* v$ and $\texttt{checker}(\tau, v) \longrightarrow^* \texttt{ERROR}$.
Consider the case $\exists v_f.e \longrightarrow^* v_f$ and $\exists v. \models v : \tau_1$

Definition 2.7. $e_1 \subseteq_\tau e_2$ is defined as the following by case analysis:
$e_1 \subseteq_{\texttt{int}} e_2$ iff $\forall v_1$. if $e_1 \longrightarrow^* v_1$, then $e_2 \longrightarrow^* v_1, v_1 \in \mathbb{Z}$.
$e_1 \subseteq_{\texttt{bool}} e_2$ iff $\forall v_1$. if $e_1 \longrightarrow^* v_1$, then $e_2 \longrightarrow^* v_1, v_1 \in \mathbb{B}$.
$e_1 \subseteq_{\tau_1 \texttt{ -> } \tau_2} e_2$ iff $\forall v_1$. if $e_1 \longrightarrow^* v_1$, then $\exists v_2.e_2 \longrightarrow^* v_2, \models v_1, v_2 : \tau_1 \texttt{ -> } \tau_2$, and $\forall v, v_{r1}$. if $\models v : \tau_1$ and $(v_1\ v) \longrightarrow^* v_{r1}$, then $\exists v_{r2}.(v_2\ v) \longrightarrow^* v_{r2}$ and $v_{r1} \subseteq_{\tau_2} v_{r2}$.

To prove this statement, we'll need the following lemma:

Lemma 2.8. *If* $\models v : \tau$, *then* $v \subseteq_\tau \texttt{generator}(\tau)$.

Definition 2.9. For all context C, $\models C[\bullet_\tau] : -$ iff $\forall v$. if $\models v : \tau$ then $C[v] \longrightarrow^* v'$.

Lemma 2.10. $\forall.v$ *if* $\models v : \tau$, *then* $\forall.C$ *if* $\models C[\bullet_\tau] : -$ *and* $C[v] \longrightarrow^* v_1$, *then* $C[\texttt{generator}(\tau)] \longrightarrow^* v_2$ *and* $v_1 \subseteq_\tau v_2$.

□

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 $e \not\longrightarrow^* \texttt{ERROR}$, and $\forall v$. if $e \longrightarrow^* v$, then $\models v : \tau_1$ or $\models v : \tau_2$.
(3) $\models e : \tau_1 \cap \tau_2$ iff $e \not\longrightarrow^* \texttt{ERROR}$, and $\forall v$. if $e \longrightarrow^* v$, then $\models v : \tau_1$ and $\models v : \tau_2$.
(4) $\models e : \{\tau \mid p\}$ iff $e \not\longrightarrow^* \texttt{ERROR}$, and $\forall v$. if $e \longrightarrow^* v$, then $\models v : \tau$ and $p\ v \longrightarrow^* \texttt{true}$.
(5) $\models e : (x : \tau_1) \texttt{ -> } \tau_2$ iff $e \not\longrightarrow^* \texttt{ERROR}$, and $\forall v_f$, if $e \longrightarrow^* v_f$, then $\forall v$, if $\models v : \tau_1$, then $\models v_f\ v : \tau_2[v/x]$.
(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 $i \in \{1, \cdots, m\}$, $n \geq p \geq m$.

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

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

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.

$$
\begin{array}{llll}
e & ::= & \cdots \mid \texttt{let f (x :}\ \tau\texttt{) :}\ \tau\ \texttt{= e in e} & \textit{expressions} \\
 & & \mid \texttt{let (}x\ \texttt{:}\ \tau\texttt{) =}\ e\ \texttt{in}\ e & \\
\tau & ::= & \cdots \mid \{\tau \mid e\} \mid (x : \tau)\ \texttt{->}\ \tau & \textit{types}
\end{array}
$$

Fig. 5. Updated language grammar

Definition 5.1 (Defining Generator in the extended language).

(1) $\texttt{generator}(\{\tau \mid p\})$:
   $\texttt{let gend = generator}(\tau)\ \texttt{in if (p gend) then gend else mzero}$.
(2) $\texttt{generator}((x : \tau_1)\ \texttt{->}\ \tau_2)$:
   $\texttt{fun}\ x'\ \texttt{-> if checker}(\tau_1, x')\ \texttt{then generator}(\tau_2[x'/x])\ \texttt{else ERROR}$.

Definition 5.2 (Defining Checker in the extended language).

(1) $\texttt{checker}(\{\tau \mid p\}, e)$:
   $\texttt{checker}(\tau, e)\ \texttt{and eval}(e) = \texttt{true}$.
(2) $\texttt{checker}((x : \tau_1)\ \texttt{->}\ \tau_2, e)$:
   $\texttt{let arg = generator}(\tau_1)\ \texttt{in checker}(\tau_2[\texttt{arg}/x], \texttt{(e arg)})$.

Definition 5.3 (Refinement type). $\models e : \{\tau \mid p\}$ iff $e \not\longrightarrow^* \texttt{ERROR}$, and $\forall v.$ if $e \longrightarrow^* v$, then $\models v : \tau$ and $p\ v \longrightarrow^* \texttt{true}$.

- $\texttt{checker}(\{\tau \mid p\}, e)$:
  $\texttt{let \_ = checker}(\tau, e)\ \texttt{in if}\ (p\ e)\ \texttt{then}\ e\ \texttt{else ERROR}$
- $\texttt{generator}(\{\tau \mid p\})$:
  $\texttt{let gend = generator}(\tau)\ \texttt{in if (p gend) then gend else mzero}$

Definition 5.4 (Dependent type). $\models e : (x : \tau_1)$ `->` τ_2 iff $e \not\longrightarrow^*$ `ERROR`, and $\forall v_f$, if $e \longrightarrow^* v_f$, then $\forall v$, if $\models v : \tau_1$, then $\models v_f\ v : \tau_2[v/x]$.

- `checker`$((x : \tau_1)$ `->` $\tau_2, e)$:
  `let arg = generator`(τ_1) `in checker`$(\tau_2[$`arg`$/x]$, `(e arg)`$)$
- `generator`$((x : \tau_1)$ `->` $\tau_2)$:
  `fun` x' `-> let _ = checker`(τ_1, x') `in generator`$(\tau_2[x'/x])$