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 an alternative definition of types that will serve as the basis of our implementation, and prove equivalence of the two systems with respect to typechecking.

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

Typechecking is defined by the following model relation:

Definition 2.1 (Model relation).

(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 relation does 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 an alternative definition of typechecking. But first, we will provide some auxilliary definitions that will be invoked in the typechecking definition.

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

Definition 2.4 (Updated model relation).
$\models_p e : \tau$ 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. *$\forall e. \models_p e : \tau$ iff $\models e : \tau$.*

COMPLETENESS. $\forall e.$ if $\models e : \tau$, then $\models_p e : \tau$.
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 $\not\models_p e : \texttt{int}$, then $\not\models e : \texttt{int}$.

By definition of $\models_p e : \texttt{int}$, we know that $\not\models_p e : \texttt{int}$ implies $\texttt{checker(int}, e) \longrightarrow^* \texttt{ERROR}$. 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 $\models_p v : \tau_1$, 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, by the above, we know that ERROR must come from $\texttt{checker}(\tau_2, \texttt{(e arg)})$. This implies 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 $\models_p e : \tau$, then $\models e : \tau$.

This is equivalent to showing: if $\not\models e : \tau$, then $\texttt{checker}(\tau, e) \longrightarrow^* \texttt{ERROR}$. Since we know that $\not\models e : \tau$, we can safely assume that $e \not\Uparrow$.

Consider the case where $e \longrightarrow^* \texttt{ERROR}$. By the operational semantics, we know that if e evaluates to ERROR, then $\texttt{checker}(\tau, e) \longrightarrow^* \texttt{ERROR}$.

Therefore, it suffices to show that if $e \longrightarrow^* v$ and $\not\models v : \tau$, then $\not\models_p e : \tau$. We will prove this by induction on the size of τ.

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

Given $\not\models e : \texttt{int}$ and $e \longrightarrow^* v$, we know that $v \notin \mathbb{Z}$. Unpacking the definition for $\texttt{checker}(\texttt{int}, e)$, we get $\texttt{if}\ e \sim \texttt{int then}\ e\ \texttt{else ERROR}$. Since $e \longrightarrow^* v$ and $v \notin \mathbb{Z}$, $e \sim \texttt{int}$ will evaluate to false, and the entire if expression will evaluate to ERROR, establishing this case.

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

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

To prove the case for function types, we need the following auxilliary definition.

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 $v_{r1} \subseteq_{\tau_2} v_{r2}$; if $\not\models v : \tau_1$, then $v_2\ v \longrightarrow^* \texttt{ERROR}$.

We will also need the following lemmas:

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

Proof. We will prove Lemma 2.8 by induction on the size of τ.

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

Since $\texttt{generator}(\texttt{int}) = \texttt{pick}_i$, by definition of $\texttt{pick}_i$, we know that $\texttt{pick}_i \longrightarrow^1 v$.

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

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

By Lemma 2.6, we know that $\models \texttt{generator}(\tau_1 \texttt{ -> } \tau_2) : \tau_1 \texttt{ -> } \tau_2$.

Then, we'll show that $\forall v_0$. if $\models v_0 : \tau_1$ and $v\ v_0 \longrightarrow^* v_{r1}$, then $(\texttt{generator}(\tau_1 \texttt{ -> } \tau_2))\ v_0 \longrightarrow^* v_{r2}$ and $v_{r1} \subseteq_{\tau_2} v_{r2}$.

By definition, $(\texttt{generator}(\tau_1 \texttt{ -> } \tau_2))\ v_0 = \texttt{let \_ = checker}(\tau_1, v_0)\ \texttt{in generator}(\tau_2)$. By completeness, we know that $\texttt{checker}(\tau_1, v_0) \not\longrightarrow^* \texttt{ERROR}$. Moreover, since we're given that $v\ v_0 \longrightarrow^* v_{r1}$, which indicates that $v_0 \not\Uparrow$, we can conclude that $(\texttt{generator}(\tau_1 \texttt{ -> } \tau_2))\ v_0 \longrightarrow^* \texttt{generator}(\tau_2)$. By Lemma 2.6, we know that $\models \texttt{generator}(\tau_2) : \tau_2$. By inductive hypothesis, we know that $v_{r1} \subseteq_{\tau_2} \texttt{generator}(\tau_2)$.

Finally, we need to show that $\forall v_0$. if $\not\models v_0 : \tau_1$, then $(\texttt{generator}(\tau_1 \texttt{ -> } \tau_2))\ v_0 \longrightarrow^* \texttt{ERROR}$.

By definition, $(\texttt{generator}(\tau_1 \texttt{ -> } \tau_2))\ v_0 = \texttt{let \_ = checker}(\tau_1, v_0)\ \texttt{in generator}(\tau_2)$. By induction hypothesis, we know that if $\not\models v_0 : \tau_1$, then $\texttt{checker}(\tau_1, v_0) \longrightarrow^* \texttt{ERROR}$. Therefore, we know that the application expression itself also evalutes to ERROR. □

Lemma 2.9. $\forall v$. *if* $\models v : \tau$ *and* $v \subseteq_\tau v'$, *then* $\forall C$. *if* $\not\models C[v] : \tau_0$, *then* $\not\models C[v'] : \tau_0$.

Proof. We will prove Lemma 2.9 by induction on the size of τ.

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

By definition of $\subseteq_{\texttt{int}}$, we know that $v = v'$, therefore $C[v] = C[v']$. It naturally follows that $\not\models C[v'] : \tau_0$.

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

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

Given that $\not\models C[v] : \tau_0$, there are two scenarios we need to consider:

(1) $C[v] \longrightarrow^* \texttt{ERROR}$,
(2) $C[v] \longrightarrow^* v_{r1}$ and $\not\models v_{r1} : \tau_0$.

The proof for these two cases are very similar, so we will only go over the proof for $C[v] \longrightarrow^* \texttt{ERROR}$. We will prove this by induction on the length of $C[v] \longrightarrow^* \texttt{ERROR}$.

Base case: $C[v] \longrightarrow^1 \texttt{ERROR}$

If C doesn't have holes, the statement will be trivially true for v and v'. We will only consider the case where C indeed contains holes to be filled by v and v' respectively. In this case, $C[v]$ can only be one of the following three:

(1) $v_1 + v_2$ where $v_1 = v$ or $v_2 = v$ (or both): In the first case, $C[v'] = v' + v_2$, and the other two cases follow similarly. According to the operational semantics, addition where one of the operands is a non-integer will evaluate to ERROR. Therefore, since v and v' are both function values, we know that $C[v'] \longrightarrow^1 \texttt{ERROR}$.
(2) $v_1\ v$ where v_1 isn't a function value: In this case, $C[v'] = v_1\ v'$. Again, by operational semantics, application of non-function will evaluate to ERROR, which means $C[v'] \longrightarrow^1 \texttt{ERROR}$.
(3) $\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$: In this case, $C[v'] = \texttt{if}\ v'\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. Both cases are conditoinal expressions with a non-boolean conditional, which by operational semantics will evaluate to ERROR.

All other cases will result in the computation taking more than one step to reach ERROR.

Inductive step: $C[v] \longrightarrow^*_n \texttt{ERROR}$

In this part of the proof, I'll be using some of the notations and lemmas introduced in *From Operational to Denotational Semantics* by Scott F. Smith.

Let's examine the first step in the given computation, which is effectively $C[v] \longrightarrow^1 e_1$ for some intermediate evaluation result, e_1.

By corollary 3.5 in the above mentioned paper, we know that there exists unique $R[\circ][\bullet]$ and $C'[\circ]$ such that $C[\circ] = R[\circ][C'[\circ]]$, and $C'[v]$ is a redex or $C'[\circ] = \circ$.

Since v is a value, we don't have to consider the case where $C'[\circ] = \circ$, since a value by itself cannot be a redex. This leaves us with the case where $C'[v]$ is a redex. There are two main scenarios to consider:

(1) If C' doesn't have holes: This impliess that neither v nor v' will appear in the redex, r. In this case, we know that $r \longrightarrow^* c$, and that $R[v][r] \longrightarrow^* R[v][c]$. Since $R[v][c] \longrightarrow^*_{n'} \texttt{ERROR}$ where $n' < n$, by induction hypothesis, we can conclude that $R[v'][c] \longrightarrow^* \texttt{ERROR}$. Since we know that $R[v'][r] \longrightarrow^* R[v'][c]$, we can conclude that $R[v'][r] \longrightarrow^* \texttt{ERROR}$.
(2) If C' does have holes: This impliess that v (and in turn, v') will appear in the redex. We'll proceed by case analysis on the redex. Since v, a function value, is in the redex, $C'[v]$, we know that $C'[v]$ cannot take the forms listed in the base case, since they will result in an error immediately. This leaves us with the following cases:
   (a) $C'[v] = v\ v_0$: In this case, $C[v'] = v'\ v_0$. Because $\models v : \tau_1\ \texttt{->}\ \tau_2$, we know that if $\models v_0 : \tau_1$, then $v\ v_0 \longrightarrow^* v_{r1}$ and $\models v_{r1} : \tau_2$.
   Since we know that $R[v][C'[v]] \longrightarrow^* R[v][v_{r1}]$, and that $R[v][C'[v]] \longrightarrow^* \texttt{ERROR}$, we know that $R[v][v_{r1}] \longrightarrow^* \texttt{ERROR}$. Since it's a smaller computation, by induction hypothesis, we can conclude that $R[v'][v_{r1}] \longrightarrow^* \texttt{ERROR}$.
   Because $v \subseteq_{\tau_1\ \texttt{->}\ \tau_2} v'$, we know that $\models v' : \tau_1\ \texttt{->}\ \tau_2$. This in turn gives us $v\ v_0 \longrightarrow^* v_{r2}$ and $\models v_{r2} : \tau_2$. Furthermore, we know that $v_{r1} \subseteq_{\tau_2} v_{r2}$ by definition of $v \subseteq_{\tau_1\ \texttt{->}\ \tau_2} v'$. Since τ_2 is a smaller type, by induction hypothesis, we can conclude that if $\not\models R[v'][v_{r1}] : \tau_0$,

then $\not\models R[v'][v_{r2}] : \tau_0$. Since we've proven that $R[v'][v_{r1}] \longrightarrow^* \texttt{ERROR}$, the premise is true, thus we can safely conclude $\not\models R[v'][v_{r2}] : \tau_0$.

(b) $C'[v] = v_f\ v$ where $v_f = \texttt{fun}\ x\ \texttt{->}\ e_f$: In this case, $C[v'] = v_f\ v'$. By operational semantics, we know that $R[v][C'[v]] \longrightarrow^1 R[v][e_f[v/x]]$. We can rewrite $e_f[v/x]$ as $C''[v]$, where the holes are where the x's were originally. Therefore, we have $R[v][C''[v]] \longrightarrow^*_{n-1} \texttt{ERROR}$. Since it's a smaller computation, we can use the induction hypothesis to conclude that $R[v'][C''[v']] \longrightarrow^* \texttt{ERROR}$. Since $C''[v'] = e_f[v'/x]$, we know that $R[v][C'[v']] \longrightarrow^1 R[v][C''[v]]$, thus proving $R[v][C'[v']] \longrightarrow^* \texttt{ERROR}$.

(c) $C'[v] = \texttt{if true then}\ e_1\ \texttt{else}\ e_2$: If $e_1 \neq v$, then both $C'[v]$ and $C'[v']$ will evaluate to e_2, and the rest of the computation will follow by inductive hypothesis. We only need to consider where $e_1 = v$. In this case, $C'[v] \longrightarrow^1 v$, thus $R[v][C'[v]] \longrightarrow^1 R[v][v]$ and $R[v][v] \longrightarrow^*_{n-1} \texttt{ERROR}$. By induction hypothesis, we have $R[v'][v'] \longrightarrow^* \texttt{ERROR}$. Since $C'[v'] = \texttt{if true then}\ v'\ \texttt{else}\ e_2$, we know that $R[v'][C'[v']] \longrightarrow^1 R[v'][v']$, showing that $R[v'][C'[v']] \longrightarrow^* \texttt{ERROR}$.

(d) $C'[v] = \texttt{if false then}\ e_1\ \texttt{else}\ e_2$: The proof for this is identical to the last case, so we'll omit it here for brevity.

□

Now we'll show that if $e \longrightarrow^* v_f$ and that $\not\models v_f : \tau_1\ \texttt{->}\ \tau_2$, then $\texttt{checker}(\tau_1\ \texttt{->}\ \tau_2, v_f) \longrightarrow^* \texttt{ERROR}$.

By definition of $\not\models v_f : \tau_1\ \texttt{->}\ \tau_2$, we know that there must exist some v_0 such that $\models v_0 : \tau_1$ and $\not\models v_f\ v_0 : \tau_2$.

Unpacking the checker definition, we have $\texttt{let arg = generator}(\tau_1)\ \texttt{in checker}(\tau_2, v_f\ \texttt{arg})$. By lemma 2.6, we have $\models \texttt{generator}(\tau_1) : \tau_1$. By lemma 2.8, we know that $v_0 \subseteq_{\tau_1} \texttt{generator}(\tau_1)$. Since $\not\models v_f\ v_0 : \tau_2$, by lemma 2.9, we can conclude that $\not\models v_f\ \texttt{generator}(\tau_1) : \tau_2$. Since τ_2 is a smaller type, by induction hypothesis, we can conclude that $\texttt{checker}(v_f\ \texttt{arg}, \tau_2) \longrightarrow^* \texttt{ERROR}$, thus the overall expression, $\texttt{checker}(\tau_1\ \texttt{->}\ \tau_2, v_f) \longrightarrow^* \texttt{ERROR}$.

□

3 LANGUAGE EXTENSIONS

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

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

Fig. 2. Extended language grammar

Definition 3.1 (More model relations).

(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.

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

Fig. 3. Extended language grammar (with records)

Definition 3.2 (Record model relation).

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

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

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 core language).

(1) $\texttt{generator(int)} : \texttt{pick}_i$.
(2) $\texttt{generator(bool)} : \texttt{pick}_b$.
(3) $\texttt{generator}(\tau_1\ \texttt{->}\ \tau_2) : \texttt{fun}\ x\ \texttt{->}\ \texttt{generator}(\tau_2)$.

Definition 4.3 (Defining Checker in the core language).

(1) $\texttt{checker(int}, e) : e \sim \texttt{int}$.
(2) $\texttt{checker(bool}, e) : e \sim \texttt{bool}$.
(3) $\texttt{checker}(\tau_1\ \texttt{->}\ \tau_2, e) : \texttt{let arg = generator}(\tau_1)\ \texttt{in checker}(\tau_2, \texttt{(e arg)})$.

Definition 4.4 (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 : 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(τ_{dom1}, x) then generator(τ_{cod1}) else generator(τ_{cod2}).
(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) generator($(x : \tau_1)$ -> τ_2) :
fun x' -> if checker(τ_1, x') then generator($\tau_2[x'/x]$) else ERROR.
(8) generator($\mu\alpha.\tau$) : generator($\tau[\alpha/\mu\alpha.\tau]$).
(9) generator($\{\ell_1 : \tau_1, \cdots, \ell_n : \tau_n\}$) :
let $v_1 =$ generator(τ_1) in $\cdots$ let $v_n =$ generator(τ_n) in $\{\ell_1 = v_1, \cdots, \ell_n = v_n\}$.

Definition 4.5 (Defining Checker in the extended language).

(1) checker(α_i, e) : $e \sim a_i$.
(2) checker($\tau_1 \cup \tau_2, e$) : checker(τ_1, e) or checker(τ_2, e).
(3) checker($\tau_1 \cap \tau_2, e$) : checker(τ_1, e) and checker(τ_2, e).
(4) checker($\{\tau \mid p\}, e$) : checker(τ, e) and eval(e) = true.
(5) checker($(x : \tau_1)$ -> τ_2, e) : let arg = generator(τ_1) in checker(τ_2[arg$/x$], (e arg)).
(6) checker($\mu\alpha.\tau, e$) : checker($\tau[\mu\alpha.\tau/\alpha], e$).
(7) checker($\{\ell_1 : \tau_1, \cdots, \ell_m : \tau_m\}, e$) : eval($e$) $= \{\ell_1 = v_1, \cdots, \ell_m = v_m, \cdots, \ell_n = v_n\}^{\{\ell_1, \cdots, \ell_m\}}$
and checker(τ_1, v_1) $\cdots$ and checker(τ_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 ::= $\cdots$ | let f (x : τ) : τ = e in e | let (x : τ) = e in e *expressions*

Fig. 5. Updated language grammar