Compiling Techniques

Lecture 11: Type Analysis (Part 2)
First ChocoPy Typing Rule that use the Environment

\[ O(id) = T; \text{ where } T \text{ is not a function type.} \]

\[ O, R \vdash id : T \]

“If the variable \textbf{id} is in the type environment \( O \) with type \( T \), and \( T \) is not a function type then we can conclude that \( \text{in the same type environment } O \text{ and } R \) the expression \textbf{id} \text{ is well typed and has type } T”
Example of Type Checking with Environment

O(id) = T; where T is not a function type.

O, R ⊢ id : T

O, R ⊢ e : int

O, R ⊢ -e : int

{x: int}, R ⊢ -x : int
Example of Type Checking with Environment

\[O(id) = T; \text{ where } T \text{ is not a function type.}\]

\[O, R \vdash id : T\]

\[O, R \vdash e : \text{int}\]

\[\{x: \text{int}\}, R \vdash -x : \text{int}\]

\[O, R \vdash -e : \text{int}\]
Example of Type Checking with Environment

O(id) = T; where T is not a function type.

\[ O, R \vdash id : T \]

\[ O, R \vdash e : \text{int} \]

\[ O, R \vdash -e : \text{int} \]

\{x: \text{int}\}(x) = \text{int}; where \text{int} is not a function type

\[ \{x: \text{int}\}, R \vdash x : \text{int} \]

\[ \{x: \text{int}\}, R \vdash -x : \text{int} \]
First ChocoPy Typing Rule that use the Environment

\[ 0(id) = T \]
\[ 0, R \vdash e_1 : T_1 \]
\[ T_1 \leq a T \]
\[ \begin{array}{c}
0, R \vdash \text{VAR-ASSIGN-STMT} \\
0, R \vdash id = e_1
\end{array} \]
First ChocoPy Typing Rule that use the Environment

\[ O(id) = T \]
\[ O, R \vdash e_1 : T_1 \]
\[ T_1 \leq a T \]

\[ \text{----------------- [VAR-ASSIGN-STMT]} \]
\[ O, R \vdash id = e_1 \]

What is this?
Assignment compatibility

- Besides the subtyping relationship, ChocoPy introduces another relation between two types: *assignment compatibility* \( (\leq a) \)

- The idea is that we may assign a value of type \( T_1 \) to something of type \( T_2 \) iff \( T_1 \) is assignment compatible with \( T_2 \)

\[ T_1 \leq a T_2 \text{, iff at least one of the following is true:} \]
- \( T_1 \leq T_2 \) (i.e., \( T_1 \) is a subtype of \( T_2 \))
- \( T_1 \) is \(<\text{None}>\) and \( T_2 \) is not \( \text{int, bool, or str} \)
- \( T_2 \) is a list type \([T]\) and \( T_1 \) is \(<\text{Empty}>\)
- \( T_2 \) is a list type \([T]\) and \( T_1 \) is \([<\text{None}>]\), where \(<\text{None}> \leq a T \)
First ChocoPy Typing Rule that use the Environment

If the variable `id` is in the type environment `O` with type `T`, and expression `e₁` has type `T₁` in the same type environment `O` and `R`, and `T₁` is assignment compatible with `T`, then we can conclude that

**in the same type environment `O` and `R`**

*the expression `id = e₁` is well typed* 

Note: we are checking a statement that has no type!
ChocoPy Typing Rule for Conditional Expressions

\[ \begin{align*}
0, R \vdash e_0 & : \text{bool} \\
0, R \vdash e_1 & : T_1 \\
0, R \vdash e_2 & : T_2
\end{align*} \]

\[ \begin{align*}
\hline
& \text{COND} \\
\hline
0, R \vdash e_1, \textbf{if} e_0 \textbf{ else} e_2 & : T_1 \sqcup T_2
\end{align*} \]
ChocoPy Typing Rule for Conditional Expressions

\[
\begin{align*}
0, R \vdash e_0 &: \text{bool} \\
0, R \vdash e_1 &: T_1 \\
0, R \vdash e_2 &: T_2 \\
\hline
\vdash e_1 \quad \text{if } e_0 \quad \text{else } e_2 &: T_1 \cup T_2
\end{align*}
\]

What is this?
Join of Types

- Sometimes (e.g., when type checking a conditional expression), we need to find a single type that can be used to represent the two original types. For this, we define the join operator

- The join of two types $T_1$ and $T_2$ (written as $T_1 \sqcap T_2$) is:
  - $T_2$ if $T_1 \leq T_2$
  - $T_1$ if $T_2 \leq T_1$
  - object otherwise, as it is the least common ancestor of $T_1$ and $T_2$
ChocoPy Typing Rule for Conditional Expressions

\[\begin{align*}
0, R &\vdash e_0 : \text{bool} \\
0, R &\vdash e_1 : T_1 \\
0, R &\vdash e_2 : T_2 \\
\hline
[\text{COND}] \\
0, R &\vdash e_1 \text{ if } e_0 \text{ else } e_2 : T_1 \sqcup T_2
\end{align*}\]

“If the expression \(e_0\) has type \(\text{bool}\) in the type environment \(0\) and \(R\), and
the expression \(e_1\) has type \(T_1\) in the same type environment \(0\) and \(R\), and
the expression \(e_2\) has type \(T_2\) in the same type environment \(0\) and \(R\),
then we can conclude that

\textit{in the same type environment \(0\) and \(R\)}

\textit{the expression} \(e_1 \text{ if } e_0 \text{ else } e_2\) \textit{is well typed and has type} \(T_1 \sqcup T_2\).”
Example of Type Checking for Conditional Expressions

Regular type checking for conditional expressions:

```
O, R ⊢ [True] if True else [] : [bool] ⊡ <Empty>
```

- `T₁` is `<Empty>` and `T₂` is a list type `[T]` and `T₁` is `<None>`, where `<None>` ≤ `T`
- `T₁ ≤ T₂` (i.e., `T₁` is a subtype of `T₂`)
- `T₁` is `<None>` and
  - `T₂` is not `int`, `bool`, or `str`
- `T₂` is a list type `[T]` and
  - `T₁` is `<Empty>`
- `T₂` is a list type `[T]` and
  - `T₁` is `[<None>]`, where `<None>` ≤ `T`
Example of Type Checking for Conditional Expressions

O, R ⊢ [True] if True else [] : [bool] ⊨ <Empty>

<table>
<thead>
<tr>
<th>[bool]</th>
<th>object</th>
</tr>
</thead>
</table>

O, R ⊢ [True] if True else None : [bool] ⊨ <None>

<table>
<thead>
<tr>
<th>[bool]</th>
<th>object</th>
</tr>
</thead>
</table>

T₁ ≤a T₂

- T₁ ≤ T₂ (i.e., T₁ is a subtype of T₂)
- T₁ is <None> and T₂ is not int, bool, or str
- T₂ is a list type [T] and T₁ is <Empty>
- T₂ is a list type [T] and T₁ is [<None>], where <None> ≤a T
Example of Type Checking for Conditional Expressions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>O, R ⊢ [True] if True else [] : [bool] ⊑ &lt;Empty&gt;</code></td>
<td>[bool] object</td>
</tr>
<tr>
<td><code>O, R ⊢ [True] if True else None : [bool] ⊑ &lt;None&gt;</code></td>
<td>[bool] object</td>
</tr>
<tr>
<td><code>O, R ⊢ [True] if True else [None] : [bool] ⊑ [&lt;None&gt;]</code></td>
<td>[bool] object</td>
</tr>
</tbody>
</table>

- $T_1 \leq_a T_2$
- $T_1 \leq T_2$ (i.e., $T_1$ is a subtype of $T_2$)
- $T_1$ is `<None>` and $T_2$ is not `int`, `bool`, or `str`
- $T_2$ is a list type `[T]` and $T_1$ is `<Empty>`
- $T_2$ is a list type `[T]` and $T_1$ is `[<None>]`, where `<None>` $\leq a T$
Example of Type Checking for Conditional Expressions

\[
\begin{align*}
O, R \vdash [True] \text{if True else } [] : [\text{bool}] \sqsubseteq \text{<Empty>}
\end{align*}
\]

\[
\begin{align*}
\text{[bool]} & \quad \text{object}
\end{align*}
\]

\[
\begin{align*}
O, R \vdash [True] \text{if True else None} : [\text{bool}] \sqsubseteq \text{<None>}
\end{align*}
\]

\[
\begin{align*}
\text{[bool]} & \quad \text{object}
\end{align*}
\]

\[
\begin{align*}
O, R \vdash [True] \text{if True else [None]} : [\text{bool}] \sqsubseteq [\text{<None>}] \\
\end{align*}
\]

\[
\begin{align*}
\text{[bool]} & \quad \text{object}
\end{align*}
\]

\[
\begin{align*}
T_1 \leq a T_2 \\
\bullet & \quad T_1 \leq T_2 \text{ (i.e., } T_1 \text{ is a subtype of } T_2) \\
\bullet & \quad T_1 \text{ is } \text{<None>} \text{ and } T_2 \text{ is not } \text{int, bool, or str} \\
\bullet & \quad T_2 \text{ is a list type } [T] \text{ and } T_1 \text{ is } \text{<Empty>} \\
\bullet & \quad T_2 \text{ is a list type } [T] \text{ and } T_1 \text{ is } [\text{<None>}], \text{ where } \text{<None>} \leq a T
\end{align*}
\]
ChocoPy Function Definition Typing Rule

\[ T = T_0 \text{ if return type is present, } \langle \text{None} \rangle \text{ otherwise} \]

\[ O(f) = \{ T_1 \times \cdots \times T_n \to T; x_1, \ldots, x_n; v_1: T'_1, \ldots, v_m: T'_m \} \]

\[ O[T_1/x_1] \cdots [T_n/x_n][T'_1/v_1] \cdots [T'_m/v_m], T \vdash b \]

\[ 0, R \vdash \text{def } f(x_1: T_1, \ldots, x_n: T_n) \Rightarrow T_0 : b \]

[ FUNC-DEF ]
ChocoPy Function Definition Typing Rule

1. Set $T$ to be return the return type, or $<\text{None}>$

$$T = T_0 \text{ if return type is present, } <\text{None}> \text{ otherwise}$$

$$O(f) = \{T_1 \times \cdots \times T_n \rightarrow T; x_1, \ldots, x_n; v_1: T'_1, \ldots, v_m: T'_m\}$$

$$0[T_i/x_i][T_n/x_n][T'_i/v_i][T'_m/v_m], T \vdash b$$

$[\text{FUNC-DEF}]$

$$0, R \vdash \text{def } f(x_1: T_1, \ldots, x_n: T_n) [\Rightarrow T_0]? : b$$
ChocoPy Function Definition Typing Rule

1. Set $T$ to be return the return type, or $\langle \text{None} \rangle$

2. Get information about $f$ from the environment

\[ T = T_o \text{ if return type is present, } \langle \text{None} \rangle \text{ otherwise} \]

\[ O(f) = \{T_1 \times \ldots \times T_n \rightarrow T ; x_1, \ldots, x_n ; v_1 : T'_1, \ldots, v_m : T'_m \} \]

\[ O[T_1/x_1] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_m/v_m], T \vdash b \]

\[ O, R \vdash \text{def } f(x_1 : T_1, \ldots, x_n : T_n) \iff T_o \} : b \]
ChocoPy Function Definition Typing Rule

1. Set $T$ to be return the return type, or $<$None$>$

2. Get information about $f$ from the environment

   \[
   T = T_o \text{ if return type is present, } <\text{None}> \text{ otherwise}
   \]

   \[
   O(f) = \{T_1 \times \ldots \times T_n \rightarrow T; x_1, \ldots, x_n; v_1: T'_1, \ldots, v_m: T'_m\}
   \]

   \[
   O[T/x_1][T_n/x_n][T'_1/v_1][T'_m/v_m], T \vdash b
   \]

   [FUNC-DEF]

   \[
   0, R \vdash \text{def } f(x_1: T_1, \ldots, x_n: T_n) \# T_o? : b
   \]

3. Type check function body $b$ with an adjusted environment, where

   - $x_i$ has type $T_i$ and $v_i$ has type $T'_i$ (notation: $O[T/c](c) = T$; $O[T/c](d) = O(d)$ if $d \neq c$)
   - $T$ is used instead of $R$
Implementing ChocoPy Typing Rules

Basic idea

- Implement one Python function for each typing rule, e.g.:

```python
# [NEGATE] rule
# O, R, ⊢ - e : int
def negate_rule(o: LocalEnvironment, r: Type, e: Operation) -> Type:
    # O, R, ⊢ e : int
    check_type(check_expr(o, r, e), expected=int_type)
    return int_type

- Have a `dispatch` function that decides which typing rule to invoke.
```
Implementing dispatch function

Basic idea

- Implement one Python function for each typing rule.
- Have a *dispatch* function that decides which typing rule to invoke:

```python
def check_expr(o: LocalEnvironment, r: Type, op: Operation) -> Type:
    if isinstance(op, choco_ast.UnaryExpr):
        unary_expr = op
        op = unary_expr.op.data
        e = unary_expr.value.blocks[0].ops[0]
        if op == "+":
            return negate_rule(o, r, e)
        else:
            raise Exception("Not implemented yet")
    else:
        raise Exception("Not implemented yet")
```
Dispatch of Typing Rules

- There are three different dispatch functions:
  - `def check_stmt_or_def_list(o, r, ops: List[Operation])` for list of statements and definitions
  - `def check_stmt_or_def(o, r, op: Operation)` for statements and definitions
  - `def check_expr(o, r, op: Operation) → Type` for expressions

- **Challenge:**
The syntax alone is not always enough to decide which typing rule to invoke!

To decide which rule to invoke, I need to know the type of `e1` or `e2`!

\[
\begin{align*}
O, R ⊢ e_1 : \text{int} \\
O, R ⊢ e_2 : \text{int} \\
\text{op} ∈ \{+, -, *, //, %\} \\
\hline
\text{[ARITH]} \\
O, R ⊢ e_1 \text{ op } e_2 : \text{int}
\end{align*}
\]

\[
\begin{align*}
O, R ⊢ e_2 : \text{str} \\
\hline
\text{[STR-CONCAT]} \\
O, R ⊢ e_1 + e_2 : \text{str}
\end{align*}
\]