Compiling Techniques

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

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

\[ \text{[VAR-READ]} \]

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

“If the variable \texttt{id} is in the type environment \( O \) with type \( T \), and

\( T \) is not a function type

then we can conclude that

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

\textit{the expression } \texttt{id} \textit{ is well typed and has type } \( T \)”
Example of Type Checking with Environment

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

\[ \begin{align*}
O, R \vdash id : T \\
O, R \vdash e : \text{int}
\end{align*} \] [VAR-READ]

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

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

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

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

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

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

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

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

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

\[
\text{O, R ⊢ e : int}
\]

\[
\text{O, R ⊢ -e : int}
\]

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

\[
\text{O, R ⊢ x : int}
\]

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

\[ O(id) = T \]
\[ O, R \vdash e_1 : T_1 \]
\[ T_1 \leq_a T \]
\[ \text{----------------- [VAR-ASSIGN-STMT]} \]
\[ O, R \vdash id = e_1 \]
First ChocoPy Typing Rule that use the Environment

What is this?
Assignment compatibility

- Besides the subtyping relationship, ChocoPy introduces another relation between two types: *assignment compatibility* (≤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 $T_2$

- $T_1 ≤a T_2$, iff at least one of the following is true:
  - $T_1 ≤ T_2$ (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> ≤a T`
First ChocoPy Typing Rule that use the Environment

\[ O(id) = T \]
\[ 0, R \vdash e_1 : T_1 \]
\[ T_1 \leq a T \]
\[ \text{----------------- [VAR-ASSIGN-STMT]} \]
\[ 0, R \vdash id = e_1 \]

"If the variable \textit{id} is in the type environment 0 with type T, and expression \( e_1 \) has type \( T_1 \) in the same type environment 0 and R, and \( T_1 \) is assignment compatible with T, then we can conclude that in the same type environment 0 and R \( \text{the expression } id = e_1 \text{ is well typed} \)"

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

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

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

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 $T_1 \leq a T_2$
  - $T_1$ if $T_2 \leq a T_1$
  - `object` otherwise, as it is the *least common ancestor* of $T_1$ and $T_2$
ChocoPy Typing Rule for Conditional Expressions

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

\[0, R \vdash \begin{cases} 
e_1 & \text{if } e_0 \\ e_2 & \text{else} \end{cases} : T_1 \sqcup T_2\]

“\(\text{If the expression } e_0 \text{ has type } \text{bool} \text{ in the type environment } O \text{ and } R, \text{ and the expression } e_1 \text{ has type } T_1 \)\
\(\text{in the same type environment } O \text{ and } R, \text{ and the expression } e_2 \text{ has type } T_2 \text{ in the same type environment } O \text{ and } R, \text{ then we can conclude that} \)
\(\text{the expression } e_1 \text{ if } e_0 \text{ else } e_2 \text{ is well typed and has type } T_1 \sqcup T_2.\)”
ChocoPy Function Definition Typing Rule

$T = T_0$ if return type is present, <None> otherwise

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

$O[T_1/x_1] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_n/v_n], T \vdash b$  \hspace{1cm} [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 $<\text{None}>$

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

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

$$O[T_1/x_1] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_n/v_n], \ T \vdash b$$

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

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

2. Get information about $f$ from the environment

$T = T_0$ if return type is present, <None> otherwise

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

$O[T_1/x_1] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_n/v_n], T \vdash b$

------------------------------------------------------------------ [FUNC-DEF]

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

1. Set \( T \) to be return the return type, or <None>

2. Get information about \( f \) from the environment

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

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

\[
O[T_i/x_i][T'_i/v_i], T \vdash b
\]

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!

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

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

To decide which rule to invoke, I need to know the type of \(e_1\) or \(e_2\)!