Compiling Techniques

Lecture 11: Type Analysis (Part 2)

First ChocoPy Typing Rule that use the Environment

$$\frac{O(id) = T;\ \text{where } T \text{ is not a function type.}}{O, R \vdash id : T}\ [\text{VAR-READ}]$$

"If the variable **`id`** is in the type environment `O` with type **`T`**, and
T is not a function type
then we can conclude that
in the same type environment `O` and `R`
*the expression **`id`** is well typed and has type **`T`**"*

Example of Type Checking with Environment

```
________________________________________ [?]
   {x: int}, R ⊢ -x : int
```

```
O(id) = T; where T is not
           a function type.
____________________________ [VAR-READ]
O, R ⊢ id : T
```

```
O, R ⊢  e : int
_____________________ [NEGATE]
O, R ⊢ -e : int
```

Example of Type Checking with Environment

```
______________________________[?]
   {x: int}, R ⊢ x : int
__________________________________________________[NEGATE]
   {x: int}, R ⊢ -x : int
```

```
O(id) = T; where T is not
           a function type.
________________________ [VAR-READ]
O, R ⊢ id : T
```

```
O, R ⊢  e : int
___________________ [NEGATE]
O, R ⊢ -e : int
```

Example of Type Checking with Environment

```
{x: int}(x) = int; where int is not
                     a function type
——————————————————————————————————————[VAR-READ]
 {x: int}, R ⊢ x : int
——————————————————————————————————————————————————[NEGATE]
 {x: int}, R ⊢ -x : int
```

```
O(id) = T; where T is not
           a function type.
——————————————————————————— [VAR-READ]
O, R ⊢ id : T
```

```
O, R ⊢  e : int
——————————————————— [NEGATE]
O, R ⊢ -e : int
```

First ChocoPy Typing Rule that use the Environment

$$\frac{\begin{array}{l} O(id) = T \\ O, R \vdash e_1 : T_1 \\ T_1 \leq_a T \end{array}}{O, R \vdash id = e_1} \text{ [VAR-ASSIGN-STMT]}$$

First ChocoPy Typing Rule that use the Environment

$$\frac{\begin{array}{l} O(id) = T \\ O, R \vdash e_1 : T_1 \\ T_1 \leq_a T \end{array}}{O, R \vdash id = e_1} \text{[VAR-ASSIGN-STMT]}$$

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

- $T_1 \leq_a T_2$, 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 `<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`

First ChocoPy Typing Rule that use the Environment

$$\frac{\begin{array}{l} O(id) = T \\ O, R \vdash e_1 : T_1 \\ T_1 \leq_a T \end{array}}{O, R \vdash id = e_1} \text{[VAR-ASSIGN-STMT]}$$

"If the variable **id** is in the type environment O with type **T**, and expression e_1 has type T_1 in the same type environment O and R, and T_1 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

$$\frac{\begin{array}{l} O, R \vdash e_0 : \mathrm{bool} \\ O, R \vdash e_1 : T_1 \\ O, R \vdash e_2 : T_2 \end{array}}{O, R \vdash e_1 \ \textbf{if}\ e_0 \ \textbf{else}\ e_2 : T_1 \sqcup T_2}\ [\mathrm{COND}]$$

ChocoPy Typing Rule for Conditional Expressions

$$\frac{\begin{array}{l} O, R \vdash e_0 : \texttt{bool} \\ O, R \vdash e_1 : T_1 \\ O, R \vdash e_2 : T_2 \end{array}}{O, R \vdash e_1 \textbf{ if } e_0 \textbf{ else } e_2 : T_1 \sqcup T_2} \quad [\text{COND}]$$

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 $T_1 \sqcup T_2$) 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

$$\frac{\begin{array}{l} O, R \vdash e_0 : \texttt{bool} \\ O, R \vdash e_1 : T_1 \\ O, R \vdash e_2 : T_2 \end{array}}{O, R \vdash e_1 \ \textbf{if}\ e_0 \ \textbf{else}\ e_2 : T_1 \sqcup T_2} \ [\text{COND}]$$

"*If* the expression e_0 has type **bool** in the type environment O and R, and
the expression e_1 has type T_1 in the same type environment O and R, and
the expression e_2 has type T_2 in the same type environment O and R,
then we can conclude that
in the same type environment O and R
the expression e_1 **if** e_0 **else** e_2 *is well typed and has type* $T_1 \sqcup T_2$."

Example of Type Checking for Conditional Expressions

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

`[bool]`

`object`

$T_1 \leq_a T_2$

- $T_1 \leq T_2$ (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

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

`[bool]`   `object`

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

`[bool]`   `object`

$T_1 \leq_a T_2$

- $T_1 \leq T_2$ (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

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

`[bool]` `object`

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

`[bool]` `object`

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

`[bool]` `object`

$T_1 \leq_a T_2$

- $T_1 \leq T_2$ (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

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

`[bool]` `object`

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

`[bool]` `object`

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

`[bool]` `object`

$T_1 \leq a\ T_2$

- $T_1 \leq T_2$ (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`

ChocoPy Function Definition Typing Rule

$$\frac{\begin{array}{l} T = T_0 \text{ if return type is present, <None> otherwise} \\ O(f) = \{T_1 \times \ldots \times T_n \rightarrow T;\ x_1, \ldots, x_n;\ v_1 : T'_1, \ldots, v_m : T'_m\} \\ O[T_1/x_1]\ldots[T_n/x_n][T'_1/v_1]\ldots[T'_m/v_m], T \vdash b \end{array}}{O, R \vdash \texttt{def } f(x_1 : T_1, \ldots, x_n : T_n)\ [\![\rightarrow T_0]\!]? : b} \text{ [FUNC-DEF]}$$

ChocoPy Function Definition Typing Rule

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

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

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

$$\frac{O[T_1/x_1]\ldots[T_n/x_n][T'_1/v_1]\ldots[T'_m/v_m],\ T \vdash b}{O,\ R \vdash \texttt{def}\ f(x_1 \colon T_1,\ \ldots,\ x_n \colon T_n)\ [\![\rightarrow T_0]\!]? : b}\ [\text{FUNC-DEF}]$$

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

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

$$\frac{O[T_1/x_1]\ldots[T_n/x_n][T'_1/v_1]\ldots[T'_m/v_m],\ T \vdash b}{O,\ R \vdash \text{def } f(x_1 : T_1, \ldots, x_n : T_n)\ [\![\rightarrow T_0]\!]? : b}\ \text{[FUNC-DEF]}$$

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

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

$$\frac{O[T_1/x_1]\ldots[T_n/x_n][T'_1/v_1]\ldots[T'_m/v_m],\ T \vdash b}{O,\ R \vdash \mathtt{def}\ f(x_1: T_1, \ldots, x_n: T_n)\ [\![\rightarrow T_0]\!]? : b}\ [\text{FUNC-DEF}]$$

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.:

```
# [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
```

$$\frac{O,\ R \vdash e : \texttt{int}}{O,\ R \vdash -e : \texttt{int}}\ [\texttt{NEGATE}]$$

- 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!

$$\frac{O, R \vdash e_1 : \mathtt{int} \quad O, R \vdash e_2 : \mathtt{int} \quad op \in \{+, -, *, //, \%\}}{O, R \vdash e_1 \ op \ e_2 : \mathtt{int}} \text{ [ARITH]}$$

$$\frac{O, R \vdash e_2 : \mathtt{str}}{O, R \vdash e_1 + e_2 : \mathtt{str}} \text{ [STR-CONCAT]}$$

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