Lecture notes for chapter 8

Peter Thiemann

2024-06-18

Type checking Lfun

- top-level functions with type annotations

```
def inc(x : int) -> int:
  return x + 1
```

- so far: type checking expressions was enough
- now we need to check statements in a significant way:
  - want to ensure that return types are obeyed
  - want to ensure that `return` statement appear in all execution paths of a function (c.f., Python type checkers)

What are the issues?

```
def f() -> int:
    x = 42
```

Should not type check (missing `return`)

```
def f(b: bool) -> int:
    if b:
        return 1
    else:
        x = 4
```

Should not type check because the function may not return an integer.

```
def f(b: bool) -> int:
    while b:
        return 1
```

Should not type check because `f(False)` does not return an integer.

```
def f(b: bool) -> int:
    if b:
        return 1
```

```
    else:
        return 4
```

Ok!

```
def f(b: bool) -> int:
    if b:
        return 1
    else:
        return True
```

Not ok: return types have to be consistent (with each other and the type annotation).

Typing judgment

Judgment

```
ctx, rty |- s : r
```

- `ctx` typing context
- `rty` return type (from function declaration)
- `s` statement
- `r` is `Y` (this statement definitely returns) or `N` (statement may or may not return)

Let's say that `Y` < `N`.

Typing rules for statements

```
ctx |- e : t
----------------------------------------
ctx, rty |- e : N


ctx |- e : int
----------------------------------------
ctx, rty |- print(e) : N


ctx |- e : t
ctx(x) = t
----------------------------------------
ctx, rty |- x = e : N

ctx |- e : rty
```

```
---------------------------------------
ctx, rty |- return e : Y

ctx |- e : Bool
ctx, rty |- ss1 : r1
ctx, rty |- ss2 : r2
---------------------------------------
ctx, rty |- if e: ss1 else: ss2 : max(r1, r2)

ctx |- e : Bool
ctx, rty |- ss : r
---------------------------------------
ctx, rty |- while e: ss : N
```

Typing for lists of statements

Judgment

`ctx, rty |- ss : r`

- `ss` list of statements

  ```
  ctx, rty |- [] : N

  ctx, rty |- s : r1
  ctx, rty |- ss : r2
  if r1=Y then r=Y else r=r2
  ---------------------------------------
  ctx, rty |- s::ss : r
  ```

Implementation We store `rty` in `env` under a special name which is not a valid identifier (e.g., `@ret`).

Alternative designs for the type checker

Stop checking after return

The above rule requires that we check all statement even if they follow a definite `return` statement.

Alternatively, we could stop checking statements in a statement list as soon as as we find a `return`. The corresponding stop-check rules would look like this:

```
ctx, rty |- s : Y
---------------------------------------
ctx, rty |- s::ss : Y
```

```
ctx, rty |- s : N
ctx, rty |- ss : r
---------------------------------------
ctx, rty |- s::ss : r
```

Handling of the void type

We omitted the `TVoid` type from the book, which means that all functions have to return a value.

The easiest way to add this type would be to add an expression like `None` that "creates" a value of type `TVoid`. With the stop-check rule for statement lists in force, a transformation could add `return None` to the end of any function body to obtain a Python-like behavior. As yet another alternative, we could make the expression of the `return` statement optional and treat `return` like `return None` (with the advantage that we don't have to create a new expression).

Tail calls

Example translation (hand optimized) of the `tail_sum` example in section 8.2.2.

```
tail_sum:
    addi sp, sp, -16
    sd ra, +8(sp)
    sd fp, 0(sp)
    addi fp, sp, 16
tail_sum_tail:
    # create space for callee-saved registers+locals
    beq a0, L.1
    add a1, a1, a0
    addi a0, a0, -1
    # delete space for callee-saved registers+locals
    j tail_sum_tail
L.1:
    mv a0, a1
tail_sum_epilog:
    # delete space for callee-saved registers+locals
    ld fp, 0(sp)
    ld ra, 8(sp)
    addi sp, sp, 16
    ret
```