

COMP 321: Design of Programming Language, Fall 2026

Homework 5: Type Checking

1 Progress and Preservation

For this problem, refer to the sections on Immutable Variables, Progress, and Preservation in the lecture notes from Wednesday's class for the typing and operational semantics rules that we discussed in those lectures.

Task 1 (20%). Do the four cases of the progress theorem for the following four typing rules:

$$\overline{\Gamma \vdash \text{True} : \text{bool}}$$

$$\overline{\Gamma \vdash \text{False} : \text{bool}}$$

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{If}(e, e_1, e_2) : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{Let}(e_1, x.e_2) : \tau_2}$$

Task 2 (20%). Do the three cases of the Preservation theorem for the following three stepping rules:

$$\frac{e \mapsto e'}{\text{If}(e, e_1, e_2) \mapsto \text{If}(e', e_1, e_2)}$$

$$\overline{\text{If}(\text{True}, e_1, e_2) \mapsto e_1}$$

$$\overline{\text{If}(\text{False}, e_1, e_2) \mapsto e_2}$$

Hand in a file `hw05.pdf` in a `hw05` folder of your handin folder.

2 Implementation

In this problem, you will implement a type checker for the IMP language from Homework 3. The type system we consider in this problem is C-like, in that it should rule out uses of a variable that is not in scope, uses of a function that is not in scope, and applying a function to an incorrect number of arguments, but will not ensure memory-safety — well-typed programs can still access memory that they are not supposed to by getting an address that is not in the domain of the memory. This means that the type system will *not* satisfy progress and preservation; we will look at fixing this later in the course.

2.1 Concrete Syntax

For this homework, the concrete syntax of Imp has changed as follows:

- Local variables are now written `{mut t x; s}`, e.g.

```
{mut int x;  
  x := 20;  
  x := (get(x) + 1);  
}
```

Read this as “new mutable local variable of type `t` named `x`”.

- Function arguments are prefixed with their types:

```
f(int x, int y, int* ret) {  
  get(ret) := (get(x) + get(y));  
}  
  
main() {  
  { mut int ret;  
    f(12, 14, ret);  
  }  
}
```

You can find other examples of the concrete syntax in the `test-X.imp`.

2.2 Type System

Most of the typing rules were discussed in class, and are given in Section 2 of the lecture notes. For conditionals and loops, you will need to minorly adapt the rules for `If` above for a conditional whose branches are expressions.

These typing rules distinguish between integers (`int`) and pointers (`int*`). For example,

```
{mut int x;  
  x := (get(x) + 1);  
}
```

should type check, but

```
{mut int x;  
  get(x) := (get(x) + 1);  
}
```

should not, because the left-hand side of an assignment is supposed to have type `int*` not `int`. This might catch some bugs where, for example, the programmer applies an incorrect number of gets to a variable. E.g. when `x:int*`, `x := (x + 1);` will fail to type check, under the presumption that the programmer is more likely to have forgotten to write `get(x)+1` than to have meant “reassign `x` to be the address 1 to the right of its own address”, even though both are potentially-sensible IMP programs.

This rules out programs that use an explicit memory address (`get (0)`) or that use a computed value as a memory address (e.g. `get (get (A) + get (i))` when $A : \text{int}^*$ and $i : \text{int}^*$, which is supposed to compute an array element access $A[i]$).¹ To allow these programs, we will add a “backdoor” to the type system called a *cast* and written `cast [t] e` in concrete syntax. This has the following rules:

$$\frac{}{\Gamma \vdash \text{Cast}(\tau, e) : \tau} \quad \frac{}{\text{Cast}(\tau, e) \mapsto e}$$

The cast allows the programmer to assert that any expression has any type, and at run-time the cast disappears. For example, while `get (0)` is not well-typed, `get (cast [int*] 0)` is well-typed. Read the cast as “I, the programmer, promise that address 0 is a valid memory address”. In a later task, you will think about which casts are needed to make address arithmetic like `get (get (A) + get (i))` be well-typed.

In general, the operational semantics might do some checking of casts, rather than simply ignoring them. For example, the cast might check that memory addresses are allocated, or that the actual type of e is somehow compatible with τ (e.g. they are represented by the same size in memory). For this week, since the only types in the language are integers and pointers, and the operational semantics does not distinguish the two, we allow all casts to succeed. This has the effect of making all types contain all values, trivializing the type system to essentially have only one type `int`. But the type annotations can still be useful as a lightweight expression of programmer intent, even though they can be circumvented by casts whenever the programmer wants to do so.

2.3 Tasks

Task 1 (40%). In `check.sml`, implement the type checker for expressions (`check_exp`), statements (`check_statement`), and function declarations (`check_decl`). Hand in `check.sml` to a hw05 folder of your handin folder. Specifically:

- `check_exp` takes a typing context Γ , and an expression e , and should return the type τ such that $\Gamma \vdash e : \tau$, or raise the exception `TypeError` if e is not well-typed in Γ .
- `check_statement` takes a list of declarations which is the complete program p , a context Γ , and a statement s , and should return `()` (the sole value of the `unit` type) if $\Gamma \vdash s$, ok, or raise the `TypeError` exception if the statement is not well-formed. Note that there is no interesting return value of a function whose output type is `unit`, so the only reason to run `check_statement (p, G, e)` is to make sure that there are no `TypeError` exceptions raised. You can use `let` to sequence multiple expressions of type `unit`, e.g.

```
let val () = check_statement (p, G, s1)
    val () = check_statement (p, G, s2)
in
  ()
end
```

- `check_decl` takes a program p and a declaration d and returns `()` if d ok in p or raises the `TypeError` exception if not.

¹Remember that a function argument `f(int* A, int i)` produces mutable local variables with one additional level of pointer type.

If you want to include the program or type in an error message, you can use `Print.typToString`, `Print.expToString`, and `Print.stmtToString` to convert them to a string.

Task 2 (10%). You can load a file with

```
Top.file_print "test-X.imp";
```

If the program type checks, you should see “Program is well-typed.” and the start of an evaluation trace. If the program does not type check, you should see the error message you specified for the type error. Make sure that all of the `test-X.imp` files type check with your type checker. Make a copy each file named `test-X-fail.imp`, change the program so that it has a type error, and make sure that the type checker catches the error. Between all of the files, try to cover as many cases where your code raises `TypeError` as you can. Hand these new test files in to the `hw05` folder of your handin folder.

Task 2 (10%). Adapt your `reverse.imp` from Homework 3 so that it type checks by inserting type annotations for the variables and the necessary casts. Hand in `reverse.imp` to the `hw05` holder.

If you want to step the programs, you can copy your implementation of the operational semantics from Homework 3 into `step.sml`, lightly modify the cases for local variables and function calls to propagate the type information, and add a case for `Cast(t, e)` stepping to `e`. Then run `Top.file_eval` as usual. But you don’t need to hand this in.