Compiler Construction

Type Inference in Practice

Type Checking and Type Inference

- Type Checking is the process of verifying fully typed programs
- Type Inference is the process of filling in missing type information
- The two are different, but are often used interchangeably!

Inference Rule

- **Types do not need to be explicit to have static typing.**
  With inference rules, we can infer types!

- We use an appropriate formalism to express inference rules!
  - Given a proper notation we can check the accuracy of the rules
  - Given a proper notation, we can easily translate it into programs.

From English to an Inference Rule

If e_1 has type *Int* and e_2 has type *Int* then $e_1 + e_2$ has type *Int*.

(e_1 has type *Int* $\wedge$ e_2 has type *Int*) $\implies$ $e_1 + e_2$ has type *Int*.

$$(e_1 : Int \wedge e_2 : Int) \implies e_1 + e_2 : Int.$$

Generalization

The statement:

- $(e_1 \colon \mathit{Int} \land e_2 \colon \mathit{Int}) \implies e_1 + e_2 \colon \mathit{Int}.$

...is a special case of:

- (Hypothesis$_1$: $\mathit{Int} \land \ldots \land$ Hypothesis$_n$: Int) $\implies$ Conclusion

This is an inference rule!

Notation

By tradition inferences rules are written:

$$\frac{\vdash \text{Hyp}_1 \qquad \ldots \qquad \vdash \text{Hyp}_n}{\text{Conclusion}}$$

$\vdash$ means *is provable that ...*

Example

Detect the type of a variable:

$$\frac{\vdash \text{i is an integer}}{\vdash \text{i: int}}$$

Example

Detect the type of a variable:

$$\frac{\vdash \text{i is an integer}}{\vdash \text{i: int}}$$

Detect the type of an expression:

$$\frac{\vdash e_1 : \text{int} \qquad \vdash e_2 : \text{int}}{\vdash e_1 + e_2 \text{: int}}$$

Applied to Tiger: if-then-else (1/2)

Type checking for if-then-else

Rule 1.

$$\frac{\Gamma \vdash \text{c} : \text{int} \qquad \Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash \text{if c then } e_1 \text{ else } e_2 \text{: int}}$$

Rule 2.

$$\frac{\Gamma \vdash \text{c} : \text{int} \qquad \Gamma \vdash e_1 : \text{string} \qquad \Gamma \vdash e_2 : \text{string}}{\Gamma \vdash \text{if c then } e_1 \text{ else } e_2 \text{: string}}$$

Applied to Tiger: if-then-else (2/2)

How to handle user-defined types?
Need for a generalization

Generalization.

$$\frac{\Gamma \vdash \text{c} : \text{int} \qquad \Gamma \vdash e_1 : \text{T} \qquad \Gamma \vdash e_2 : \text{T}}{\Gamma \vdash \text{if c then } e_1 \text{ else } e_2 \text{: T}}$$

All other situation must return an error

Applied to Tiger: if-then

How to type **if-then**?

Rule.

$$\frac{\Gamma \vdash \text{c} : \text{int} \qquad \Gamma \vdash e_1 : \text{Void}}{\Gamma \vdash \text{if c then } e_1 : \text{Void}}$$

All other situations must return an error

Applied to Tiger: :=

How to type :=?

Rule.

$$\frac{\Gamma \vdash \mathrm{c} : \mathrm{T} \qquad \Gamma \vdash e_1 : \mathrm{T}}{\Gamma \vdash \mathrm{c} := e_1 : \mathrm{Void}}$$

All other situations must return an error

Applied to Tiger: let-in-end

How to type **let-in-end**?

```
let in s1 ; s2 ; e1 end
```

Rule.

$$\frac{\Gamma \vdash s_1 : ? \qquad \Gamma \vdash s_2 : ? \qquad \Gamma \vdash e_1 : \mathsf{T}}{\Gamma \vdash \text{let in } s_1;s_2;e_1 \text{ end: } \mathsf{T}}$$

Applied to Tiger: let-in-end

How to type **let-in-end**?

```
let in s1 ; s2 ; e1 end
```

Rule.

$$\frac{\Gamma \vdash s_1 : ? \qquad \Gamma \vdash s_2 : ? \qquad \Gamma \vdash e_1 : \mathsf{T}}{\Gamma \vdash \text{let in } s_1;s_2;e_1 \text{ end: } \mathsf{T}}$$

Can we avoid type checking s_1 and s_2?

Applied to Tiger: let-in-end

How to type **let-in-end**?

```
let in s1 ; s2 ; e1 end
```

Rule.

$$\frac{\Gamma \vdash s_1 : ? \qquad \Gamma \vdash s_2 : ? \qquad \Gamma \vdash e_1 : \mathsf{T}}{\Gamma \vdash \text{let in } s_1;s_2;e_1 \text{ end: } \mathsf{T}}$$

Can we avoid type checking s_1 and s_2?
$\Rightarrow$ NO!!!

Remark on let-in-end chunks

How to handle the let-in declaration part?

1. Visit the headers of all types in the block.
2. Visit the bodies of all types in the block

```
let type one = { hd : int }
    type two = array of one
in
    ...
end
```

Reporting errors

Reporting errors

When there are several type errors, it is admitted that some remain hidden by others.

For loops are Void type but the type checker shall ensure loop index variables are read-only.

```
for i := 10 to 1 do
  i := i - 1
```

Type Checking in Practice

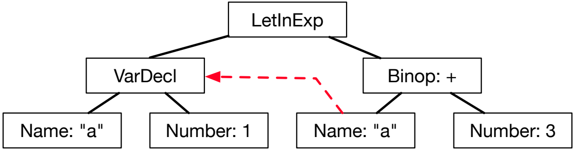
Type checking is done compositionally:

1. break down expressions into their subexpressions
2. type-check the subexpressions
3. ensure that the top-level compound expression can then be given a type itself

Throughout the process, a type environment is maintained which records the types of all variables in the expression.

Type Checking in Practice

```
let a := 1 in a + 3 end
```



Step 1. Fill Gamma. $\Gamma : \{a : int\}$

Step 2. Apply type inference

$$\dfrac{\dfrac{\Gamma \vdash a : \text{int} \qquad \vdash 3 : \text{int}}{\Gamma \vdash \text{a+3: int}}}{\Gamma \vdash \text{let a := 1 in a + 3 end: int}}$$

Summary

Type Inference

Type Checking

Chunks

Error reporting