Compiler Construction

Sibylle Schupp[1]

[1]Institute for Software Systems/Institut für Softwaresysteme
Hamburg University of Technology (TUHH)

SoSe16

Lecture 5

Outline

Why type checking

- Error detection early, at compile time
- Type-based specialization
  - Resolve operators: what does + refer to?
- Type-based optimization

Motivation

```
int a, b;
float f;
myclass c;
int foo(int);
...

foo(a);
foo(b+f);
if (a = c);
```

- What is the type of each of those expressions?

Type checkers

Unlike lexers and parsers, type checkers are not automatically generated.

- A type checker is implemented by hand.
  - Research prototypes exist for type-checking generators (Saga, Dedukti)
- It is based on the *type system* of the language. A type system consists of
  - A set of types
  - Typing rules
  - Rules for applying the rules

The set of type *identifiers* is already specified in the grammar.

Type inference rules

Typing rules are formally specified through *type inference rules*.

$$\frac{J_1 \cdots J_n}{J}$$

- An inference rule consists of a set of premises and a conclusion.
- The premises and the conclusion of a rule must be *judgments*.
- The most common judgments in type systems are of the form

$$e : T,$$

which is read as: the expression `e` is of type `T`. We also say "e has been annotated with type `T`."

Example (type inference rule)

$$\frac{a : bool \qquad b : bool}{a \&\& b : bool}$$

- The rule has two premises and one conclusion.
- Read the rule as follows:
  - "if `a` has type `bool` and `b` has type `bool`, then `a && b` has type `bool`."
  - (Forward): from the two premises, we can conclude that `a && b` has type `bool`.
  - (Backward): to check `a && b`, check `a` and `b`.

Type checking vs. type inference

Type inference rules can be used in different ways:

- In *type checking*, an expression e and a type T are given and one checks whether $e : T$ ("e has type T").
- In *type inference*, an expression e is given and one seeks a type T such that $e : T$.
- Both operations use the type inference rules, though in different ways.
  - For type checking, the rules are read backwards.
  - For type inference, the rules are read forwards.

From typing rules to type-checking code

An inference rule can be transformed to pseudo code in a straightforward way:

$$\frac{J_1 \cdots J_n}{J}$$

- Pattern-match on the type of the conclusion
- Repeat recursively with each premise

Example (type-checking code)

$$\frac{a : bool \qquad b : bool}{a \&\& b : bool}$$

Type checker

```
check(a && b: bool) :
      check(a, bool)
      check(b, bool)
```

- The type in the conclusion defines the pattern to match against.
- `_&&_` marks one case of pattern matching.
- If the type checker can establish that `a && b` is of that type, it has succeeded. If pattern matching fails, type checking fails.

Note that within a compiler, the type checker operates at the level of abstract syntax: `EAnd a b`

Example (type-inference code)

$$\frac{a : bool \qquad b : bool}{a \&\& b : bool}$$

The pseudo code for type inference for && is similar:

```
infer(a&&&b):
      check(a, bool)
      check(b, bool)
      return bool
```

- The premises are recursive calls (as in type checking).
- Different from type checking, type inference returns a value (if the checks of the premises have succeeded).
- The type of the conclusion becomes the return type of the function.

Outline

Motivation

- Lexical analysis and syntactic analysis are *context-free*.
- Type checking (and semantic analysis in general) requires a context.
  - Ex.: **if** the context contains one `+`-function, `int +(int,int)`, then `a+b` requires `a`, `b` of type `int`.
  - If the context is empty `a+b` might not have a type.

Contexts

- *Contexts* are a simple form of symbol table (environment) where (name, type)-pairs are kept. Contexts are denoted by $\Gamma, \Gamma_1, \Gamma'$.
- In their general form, typing judgments then look as follows

$$\Gamma \vdash e : T$$

Read: expression e has type T in context Γ.

Context handling

Type checking includes also handling of the context:

- The context needs to be initialized.
  - Variable types from declarations (in typed programming languages)
  - Function types from preludes, standard libraries
- When typing a subexpression, the type checker looks up the context for type information.
- When a subexpression `e` obtained a type `t`, the context may be extended by the pair `(e,t)`.

The context can be stored in the nodes of the AST ("annotated AST") or separately.

Example (context)

What is the type of the following expression? Does it have one at all?

```
x + y > y
```

- Depends on the types of `x`, `y`, `+`, and `>`.
- Assume the following context Γ:

$$
\begin{aligned}
x &: \text{int} \\
y &: \text{int} \\
+ &: (\text{int},\text{int}) \to \text{int} \\
> &: (\text{int},\text{int}) \to \text{bool}
\end{aligned}
$$

- The following judgment holds:

$$\Gamma \vdash x + y > y : \text{bool}$$

Read "in context Γ, `x + y > y` is an expression of type `bool`." In another context, the expression might have a different or no type.

Accessing the context Γ

As a data type, the context supports two operations:

- Insertion: written as comma-separated list
  - $\Gamma, x : T$ extends the context Γ by the pair x, T where x is a variable, T its type. Often, the extended context does not get a separate name.
  - The empty context is denoted as [].
- Lookup: often written as function, $\Gamma(x)$, and returns the type with which x is stored.
  - In the previous example: $\Gamma(x)$ returns *int* since x : int is in the context, $\Gamma(z)$ returns nothing.

Context in typing rules

Since the type of an expression depends on the context, the context must be part of the judgments, thus of the typing rules.

We revise the typing rule for &&

$$\frac{\Gamma \vdash a : bool \qquad \Gamma \vdash b : bool}{\Gamma \vdash a \&\& b : bool}$$

- In many rules, the context is not just passed around but changed.

The typing rule for variable expressions

Consider the typing rule for variable expressions:

$$\frac{}{\Gamma \vdash x : T} \text{ IF } x{:}T \text{ IN } \Gamma$$

- The rule is an axiom: it does not depend on other checks.
- The if-condition to the right is not a judgement, thus not part of the premise. It is a *side condition*. Consider the corresponding pseudo code:

```
infer(Gamma, x):
    t := lookup(x,Gamma)
    return t
```

- The type of `x` is determined by looking up the context.

The typing rule for function applications

For the same reason, the check of a function application (a.k.a. function invocation) requires looking up the context.

Here is the typing rule:

$$\frac{\Gamma \vdash a_1 : T_1 \quad \cdots \quad \Gamma \vdash a_n : T_n}{\Gamma \vdash f(a_1, \ldots, a_n) : T} \text{ IF } f\colon (T_1, \ldots, T_n) \to T \text{ IN } \Gamma$$

where

$$T_1 \ldots T_n \to T$$

denotes the type of the function f (a.k.a. signature, header).
Typing of a function *invocation* requires

- Looking up the function in Γ (as a side condition)
- Recursive typing of the arguments

Proof trees

The sequence of steps during typing checking (type inference) forms a tree, the *proof tree*. A proof tree is a tree where nodes are judgments.

- The root of the tree is the expression to be type-checked (or typed). This expression must match the conclusion of a typing rule.
- Each premise of that rule constitutes a child of the root.
- Since each premise serves as conclusion of another rule, each child has children on its own.
- Axioms form the leaves of the proof tree.

Proof trees are displayed upside-down, since the proof is constructed backwards.

Example (proof tree)

$$\Gamma \vdash x + y > y : \text{bool}$$

Proof tree (assume a typing rule for $>, +$-expressions,
set $\Gamma := x : \text{int}, y : \text{int}, + : (\text{int},\text{int}) \to \text{int}, > : (\text{int},\text{int}) \to \text{bool}$)

$$\dfrac{\dfrac{\Gamma \vdash x : \text{int} \qquad \Gamma \vdash y : \text{int}}{\Gamma \vdash x + y : \text{int}} \qquad \Gamma \vdash y : \text{int}}{\Gamma \vdash x + y > y : \text{bool}}$$

Outline

Validity checks for statements

An expression has a type, a statement has no type. The type checker therefore can only check whether a statement is *valid*.

We extend the inference rules by a new judgment form:

$$\Gamma \vdash s \text{ valid}$$

Read: the statement s is valid in the context Γ.

$$\frac{\Gamma \vdash e : bool \qquad \Gamma \vdash s \;\; valid}{\Gamma \vdash while(e)\; s \;\; valid}$$

- A while-statement is valid if its body is valid and the termination condition is of type `bool`.
- Statement validation and type checking are often intertwined.

Valid function definitions

A function definition is valid if the statements in its body are valid:

$$\frac{x_1 : T_1, \ldots, x_m : T_m \vdash s_1 \cdots s_n \ \mathit{valid}}{T\ f(T_1\, x_1, \ldots, T_m\, x_m)\{s_1, \ldots, s_n\} \mathit{valid}}$$

The function body is checked for validity in the context of the function parameters.

- The body statements cannot be checked separately: the context may change during the check.
- Strictly speaking, validity should also ensure that the parameters are pairwise distinct. This check is omitted here.
- Further, a validity check might require a function to include a return statement as the last statement, with an expression of the right type.

Valid expressions

The statement grammar often includes expressions. In this case, an expression is subject to a validity check as well.

Validity checks for expressions are simple:

- It only matters that the expression is well-typed.
- Afterwards, the type can be omitted.

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e \ \mathit{valid}}$$

Outline

Scope

The *scope* of a declared identifier is the extent of the program where the identifier refers to that declaration.

- Variable names, function names, module names, . . . all have a scope.
- Common scopes: function scope, class scope, file scope
- The most basic scope is block scope.
  - Every *block-structured* language introduces special delimiters for a block. Ex. "{" "}" in C++ and Java.
  - A block is a sequence of statements enclosed in block delimiters.

Example (scope)

```
{
    int x = 3;
    {
        print x;
        int z = 4;
        x = z;
        print x;
     }
     print x;
     print z;
}
```

- Assume blocks are marked by "{" and "}". Then, the fragment contains two (block) scopes, an inner and an outer scope.
- Assume variables from the outer scope are visible in the inner scope. What gets printed?
  - What happens if the outer scope variables are not visible in the inner scope?
  - And the other way around: what if an inner variable is visible in the outer scope?

Nested scoping

- The scope of an identifier includes the local scope and all inner scopes. It excludes all outer scopes.
  - Typically, it includes the local scope from the point of the declaration onwards.
- In each scope, not more than one definition may exist.
- An identifier may be redeclared in an inner scope. In that inner scope, the identifier refers to the inner declaration.
  - The outer declaration is *shadowed*. Some languages provide a way to access the outer declaration.

In-class exercise

Type checking and scoping

The type of a variable and its well-typedness depend on the scope.
Therefore, the type checker needs to take scope into account.

- If the program has only one scope, the context Γ can be just one big lookup table.
- In case of several scopes, one needs to have one context per scope.
- Since scopes are nested, the contexts can be organized in a stack.
  - Notation: $\Gamma_1.\Gamma_2$ denotes a stack of two contexts where Γ_1 is the outer context and Γ_2 the inner context. The stack grows to the right.
- Lookup of an identifier must be modified so that it follows the stack structure.

Variable declarations

The declaration rule says: a declaration followed by statements $s_2, \ldots s_n$ is valid if $s_2, \ldots s_n$ is valid in a context extended by that variable.

$$\frac{\text{X NOT IN TOP-MOST CONTEXT} \quad \Gamma, x : T \vdash s_2 \cdots s_n \ \mathit{valid}}{\Gamma \vdash T\ x; s_2, \ldots, s_n \ \mathit{valid}}$$

A declaration extends the context in which the statements are checked:

- Need to tie the declaration to its scope.
- Need to look at all statements, not just individual ones.

Outline

From specification to pseudo code

The specification of a rule lends itself directly to an implementation.

- We start with pseudo code and first just check for type correctness.
- Important helper structure: the data type `Env` (environment). It keeps a
  - lookup table for functions
  - stack of contexts (also lookup tables)

Top-level check

Recall: "A program is a sequence of definitions."

PDefs . Program ::= [Def] ;

The top-level check is performed in two passes.

- First, the environment is extended by the function definitions one by one.
- In the resulting environment, each function is checked.

```
check (d1, ..., dn):
  Gamma_0 := emptyEnv();
  for i=1,..,n:
      Gammma_i := extend(Gamma_i-1, di)
  for i=1,..,n:
      check(Gamma_n,di)
```

- The organization in two passes allows for mutually recursive functions.

Checking function definitions

The check of a single function definition requires context extension and a recursive check:

- First, the context is extended by pairs (variable, type).
- Then, in the resulting (stack of) context(s), the sequence of statements of the function body is checked.

```
check(Gamma,t f (t1 x1,...,tm xm){s1 ...sn}) :
    Gamma_0 := Gamma
    for i = 1,...,m :
        Gamma_i := extend(Gamma_i-1,xi,ti)
    check(Gamma_m, s1 . . . sn)
```

Checking statement lists

Checking a statement list depends on the particular statement.

- Here is the pseudo code for the check of declarations:

```
check(Gamma,t x;s2 ...sn) :
  if x is not yet in Gamma
      Gamma_0 := extend (Gamma, x, t)
  else error ``x redeclared''
  check(Gamma_0, s2 . . . sn)
```

- If legal, the context is extended and the statements are checked in the new context.

Checking block statements

Block statements:

- The sequence $r_1 \cdots r_m$ is checked in the new environment.
- The sequence $s_1 \cdots s_n$ is checked in the original environment

```
check(Gamma,{r1 ...rm}s2 ...sn) :
  Gamma_0 := newBlock(Gamma)
  check(Gamma_0, r1 . . . rm)
  check(Gamma, s2 . . . sn)
```

The rest

Remaining functions

- Proceed in a similar way for statements, expression statements, and expressions.
- Note: all those rules only look up the environment, but do not change it.

Synopsis (type-checking functions)

Main functions

Type	infer	(Env Γ, Exp e)
Void	check	(Env Γ, Exp e, Type t)
Void	check	(Env Γ, Stms s)
Void	check	(Env Γ, Def d)
Void	check	(Program p)

Auxiliary functions and data structures

- Data types `Env`, `FunType`
- Error reporting (see later lectures for more)

Type	lookup	(Ident x, Env Γ)
FunType	lookup	(Ident f, Env Γ)
Env	extend	(Env Γ, Ident x, Type t)
Env	extend	(Env Γ, Def d)
Env	newBlock	(Env Γ)
Env	emptyEnv	()

Summary

- Typing judgments; context and context handling, annotated ASTs
- Type inference, type checking; proof trees
- Valid statements
- Scoping

References

- IPL, Ch.4.1-4.5 and Ch.4.7-4.10