Chapter 4: Type Checking

Aarne Ranta

Slides for the book "Implementing Programming Languages. An Introduction to Compilers and Interpreters", College Publications, 2012.

Checking that a program makes sense

Traditional notions of type checking as in C and Java

Typing rules

Syntax-directed translation

Getting started with implementation in Haskell and Java

Everything that is needed for Assignment 2

The purposes of type checking

Finding errors at compile time

- the development of programming languages shows a movement to more and more type checking, e.g. from C to C++

Resolving ambiguities to get better machine code

- e.g. to find out if `+` is to be `iadd` or `dadd`

Compiling `x + y` needs the **context** to look up the types of `x` and `y`

Specifying a type checker

An implementation-language-independent way: **type system** with **inference rules**.

Example:

$$\frac{a : \mathit{bool} \;\; b : \mathit{bool}}{a \,\&\&\, b : \mathit{bool}}$$

Read: *if a has type bool and b has type bool. then a && b has type bool.*

Inference rules

An inference rule has a set of **premisses** $J_1, \ldots, J_n$ and one **conclusion** J, separated by a horizontal line:

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

It can be read in many ways:

From the premisses $J_1, \ldots, J_n$, we can conclude J.

If $J_1, \ldots, J_n$, then J.

To check J, check $J_1, \ldots, J_n$.

Judgements

The premisses and conclusions are called **judgements**.

The most common judgements in type systems have the form

$$e : T$$

which is read, *expression e has type T*.

From typing rules to pseudocode

Convert the rule

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

to a pseudocode program

J :
  J_1
  $\ldots$
  J_n

Thus the conclusion becomes a case for pattern matching, and the premisses become recursive calls.

Type checking and type inference

Two kinds of programs:

- **Type checking**: given an expression e and a type T, decide if $e : T$.
- **Type inference**: given an expression e, find a type T such that $e : T$.

Both programs may be needed. They are both derivable from typing rules.

Example: type checking for `&&`

$$
\begin{array}{l}
\mathit{check}(a \texttt{\&\&} b, \mathit{bool}) : \\
\quad \mathit{check}(a, \mathit{bool}) \\
\quad \mathit{check}(b, \mathit{bool})
\end{array}
$$

No patterns matching other types than *bool*, so type checking fails for them.

Type inference for `&&`

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

Notice that the function must also check that the operands are of type bool.

From pseudocode to code

We have use concrete syntax notation for expression patterns - that is, $a\&\&\&b$ rather than ($\mathit{EAnd}\, a\, b$).

In real type checking code, abstract syntax must of course be used. E.g. Haskell

```
inferExp :: Exp -> Type
inferExp (EAnd a b) = ...
```

Java:

```
public static class InferExp implements Exp.Visitor<Type> {
  public Type visit(EAnd p) ...
  }
```

Contexts

A variable can have *any* of the types available in the language.

In C and Java, the type is determined by the **declaration** of the variable.

In inference rules, the variables are collected to a **context**.

In the compiler, the context is a **symbol table** of (variable,type) pairs.

In inference rules, the context is denoted by the Greek letter Γ, Gamma.

The judgement form for typing is generalized to

$$\Gamma \vdash e : T$$

Read: *expression e has type T in context* Γ.

Example:

$$x : \texttt{int}, y : \texttt{int} \vdash x\texttt{+}y\texttt{>}y : \texttt{bool}$$

This means:

> `x + y` $>$ `y` *is a boolean expression in the context where* `x` *and* `y` *are integer variables.*

Notice the notation for contexts:

$$x_1 : T_1, \ldots, x_n : T_n$$

When we add a new variable to the context Γ, we write

$$\Gamma, x : T$$

Most judgements have the same Γ to all judgements, because the context doesn't change.

$$\frac{\Gamma \vdash a : \mathit{bool} \;\; \Gamma \vdash b : \mathit{bool}}{\Gamma \vdash a \,\&\&\, b : \mathit{bool}}$$

But the context does change in the typing rules for declarations.

Typing rule for variable expressions

This is where contexts are needed.

$$\frac{}{\Gamma \vdash x : T} \text{ if } x : T \text{ in } \Gamma$$

Notice: the condition "if $x : T$ in Γ" is not a judgement but a sentence in the **metalanguage** (English).

In the pseudocode, it is not a recursive call, but uses a *lookup* function:

infer(Γ, x) :
 $t :=$ *lookup*(x, Γ)
 return t

In Haskell code,

```
lookupVar :: Ident -> Context -> Err Type

inferExp  :: Context -> Exp -> Err Type

inferExp gamma (EVar x) = do
  typ <- lookupVar x gamma
  return typ
```

Type checking function applications

Need to look up the type of the function

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

Notation:

$$(T_1, \ldots, T_n) \to T$$

even though there is no such type in the language described.

Proofs in a type system

Proof tree: a trace of the steps that the type checker performs, built up rule by rule.

$$\dfrac{\dfrac{}{x : \texttt{int}, y : \texttt{int} \vdash x : \texttt{int}}\,x \quad \dfrac{}{x : \texttt{int}, y : \texttt{int} \vdash y : \texttt{int}}\,y}{\dfrac{x : \texttt{int}, y : \texttt{int} \vdash x\texttt{+}y : \texttt{int}}{x : \texttt{int}, y : \texttt{int} \vdash x\texttt{+}y\texttt{>}y : \texttt{bool}}\,\texttt{>} \quad \dfrac{}{x : \texttt{int}, y : \texttt{int} \vdash y : \texttt{int}}\,y}\,\texttt{+}$$

Each judgement is a conclusion from the ones above with some of the rules, indicated beside the line. This tree uses the variable rule and the rules for + and >:

$$\frac{}{\Gamma \vdash x : T}\,x \qquad \frac{\Gamma \vdash a : \mathit{int} \;\; \Gamma \vdash b : \mathit{int}}{\Gamma \vdash a + b : \mathit{int}}\,+ \qquad \frac{\Gamma \vdash a : \mathit{int} \;\; \Gamma \vdash b : \mathit{int}}{\Gamma \vdash a > b : \mathit{bool}}\,>$$

Overloading

The binary arithmetic operations (`+ - * /`) and comparisons (`== != < > <= >=`) are in many languages **overloaded**, which means: usable for different types.

If the possible types are `int`, `double`, and `string`, the typing rules become:

$$\frac{\Gamma \vdash a : t \quad \Gamma \vdash b : t}{\Gamma \vdash a \texttt{+} b : t} \text{ if } t \text{ is } \texttt{int} \text{ or } \texttt{double} \text{ or } \texttt{string}$$

$$\frac{\Gamma \vdash a : t \quad \Gamma \vdash b : t}{\Gamma \vdash a \texttt{==} b : \mathit{bool}} \text{ if } t \text{ is } \texttt{int} \text{ or } \texttt{double} \text{ or } \texttt{string}$$

Type inference for overloading

First infer the type of the first operand, then check the second operand with respect to this type:

infer$(a + b)$:
 $t :=$ *infer*(a)
 // *check that* $t \in \{\texttt{int}, \texttt{double}, \texttt{string}\}$
 check(b, t)
 return t

For other arithmetic operations, only `int` and `double` are possible.

Type conversions

Example: an integer can be **converted** into a double, i.e. used as a double.

May sound trivial in mathematics, as integers are a subset of reals.

But for most machines, integers and doubles have totally different binary representations and different sets of instructions.

Therefore, the compiler usually has to generate a special instruction for type conversions.

Converting from smaller to larger type

No loss of information.

$$\frac{\Gamma \vdash a : t \quad \Gamma \vdash b : u}{\Gamma \vdash a + b : \mathit{max}(t, u)} \text{ if } t, u \in \{\texttt{int}, \texttt{double}, \texttt{string}\}$$

Assume the following ordering:

$$\texttt{int} < \texttt{double} < \texttt{string}$$

For example:

$$\max(\texttt{int}, \texttt{string}) = \texttt{string}$$

Thus `2 + "hello"` gives the result `"2hello"`, because string addition is the maximum.

Quiz: what is the result of

```
1 + 2 + "hello" + 1 + 2
```

Recall that `+` is left associative!

The validity of statements

When type-checking a statement, we are not interested in a type, but just in whether the statement is **valid**.

A new judgement form:

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

Read, *statement s is valid in environment* Γ.

Example: `while` statements

$$\frac{\Gamma \vdash e : \mathit{bool} \quad \Gamma \vdash s \ \mathit{valid}}{\Gamma \vdash \texttt{while}\ (e)\ \ s \ \mathit{valid}}$$

Checking validity may thus involve type checking some expressions.

Expression statements

We don't need to care about what the type of the expression is, just that it has one.

That is, that we can infer a type.

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

This typically covers assignments and function calls.

The validity of function definitions

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

The variables declared as parameters of the function define the context.

The body statements $s_1 \ldots s_n$ are checked in this context.

Notice that the context may change within the body, because of declarations.

The type checker also has to make sure that all variables in the parameter list are distinct.

Return statements

When checking a function definition, we could check that the function body contains a `return` statement of expected type.

A more sophisticated version of this could also allow returns in `if branches`, as in

```
if (fail()) return 1 ; else return 0 ;
```

Declarations and block structures

Each declaration has a **scope**, which is within a certain **block**.

Blocks in C and Java correspond (roughly) to parts of code between curly brackets, { and }.

Two principles regulate the use of variables:

1. A variable declared in a block has its scope till the end of that block.
2. A variable can be declared again in an inner block, but not otherwise.

Example:

```
{
  int x ;
  {
    x = 3 ;      // x : int
    double x ;   // x : double
    x = 3.14 ;
    int z ;
  }
  x = x + 1 ;    // x : int, receives the value 3 + 1
  z = 8 ;        // ILLEGAL! z is no more in scope
  double x ;     // ILLEGAL! x may not be declared again
  int z ;        // legal, since z is no more in scope
}
```

Stack of contexts

We need to refine the notion of context to deal with blocks:

Instead of a simple lookup table, Γ must be a **=stack of lookup tables**.

We separate the tables with dots, for example,

$$\Gamma_1.\Gamma_2$$

where Γ_1 is an old (i.e. outer) context and Γ_2 an inner context.

The innermost context is the top of the stack.

Refining the lookup to work in a block structure

With just one context, lookup goes everywhere.

With a stack of contexts, it starts by looking in the top-most context and goes deeper in the stack only if it doesn't find the variable.

Refining the declaration rule

A declaration introduces a new variable in the current scope.

This variable is checked to be fresh with respect to the context.

But how do we express that the new variable is added to the context in which the later statements are checked?

We need to have rules for **sequences of statements**, not just individual statements:

$$\Gamma \vdash s_1 \ldots s_n \textit{ valid}$$

A declaration extends the context used for checking the statements that follow:

$$\frac{\Gamma, x : T \vdash s_2 \ldots s_n \textit{ valid}}{\Gamma \vdash T\, x; s_2 \ldots s_n \textit{ valid}} \; x \text{ not in the top-most context in } \Gamma$$

In other words: a declaration followed by some other statements $s_2 \ldots s_n$ is valid, if these other statements are valid in a context where the declared variable is added.

This addition causes the type checker to recognize the effect of the declaration.

Checking block statements

Push a new context on the stack.

$$\frac{\Gamma. \vdash r_1 \ldots r_m \textit{ valid} \quad \Gamma \vdash s_2 \ldots s_n \textit{ valid}}{\Gamma \vdash \{r_1 \ldots r_m\} s_2 \ldots s_n \textit{ valid}}$$

Only the statements inside the block are affected by the declaration, not the statements after.

Implementing a type checker

Our first large-scale lesson in **syntax-directed translation**.

Inference and checking functions, plus some auxiliary functions for dealing with contexts.

A summary of the functions needed

Type	*infer*	(*Env* Γ, *Exp* e)	*infer type of Exp*
Void	*check*	(*Env* Γ, *Exp* e, *Type* t)	*check type of Exp*
Void	*check*	(*Env* Γ, *Stms* s)	*check sequence of stms*
Void	*check*	(*Env* Γ, *Def* d)	*check function definition*
Void	*check*	(*Program* p)	*check a whole program*
Type	*lookup*	(*Ident* x, *Env* Γ)	*look up variable*
FunType	*lookup*	(*Ident* f, *Env* Γ)	*look up function*
Env	*extend*	(*Env* Γ, *Ident* x, $Type$ t)	*extend Env with variable*
Env	*extend*	(*Env* Γ, *Def* d)	*extend Env with function*
Env	*newBlock*	(*Env* Γ)	*enter new block*
Env	*emptyEnv*	()	*empty environment*

The *check* functions return a *Void*: they go through the code and silently return if the code is correct.

If any of the functions encounters an error, it emits an error message.

Most of the types above come from the abstract syntax of the implemented language, hence ultimately from its BNF grammar. The exceptions are *FunType*, and *Env*.

We don't need the definitions of these types in the pseudocode. But we will show possible Haskell and Java definitions below.

Checking a whole program

A program is a sequence of function definitions. It is checked in two passes:

1. collect the type signatures of each function by running `extend`
2. check each function definition in the created environment

$$
\begin{array}{l}
\mathit{check}(d_1, \ldots, d_n) : \\
\quad \Gamma_0 := \mathit{emptyEnv}() \\
\quad \mathit{for}\ i = 1, \ldots, n : \Gamma_i := \mathit{extend}(\Gamma_{i-1}, d_i) \\
\quad \mathit{for\ each}\ i = 1, \ldots, n : \mathit{check}(\Gamma_n, d_i)
\end{array}
$$

Having the two passes permits **mutually recursive functions**.

The variables in a definition are not visible to other definitions.

Checking a function definition

$$
\begin{aligned}
&\mathit{check}(\Gamma, t\ f\ (t_1\, x_1, \ldots, t_m\, x_m)\{s_1 \ldots s_n\}: \\
&\quad \Gamma_0 := \Gamma \\
&\quad \mathit{for}\ i = 1, \ldots, m : \Gamma_i := \mathit{extend}(\Gamma_{i-1}, x_i, t_i) \\
&\quad \mathit{check}(\Gamma_m, s_1 \ldots s_n)
\end{aligned}
$$

Checking statement lists and block statements

check$(\Gamma, t\ x; s_2 \ldots s_n)$:
 // here, check that x *is not yet in* Γ
 $\Gamma' :=$ *extend*(Γ, x, t)
 check$(\Gamma', s_2 \ldots s_n)$

check$(\Gamma, \{r_1 \ldots r_m\} s_2 \ldots s_n)$:
 $\Gamma' :=$ *newBlock*(Γ)
 check$(\Gamma', r_1 \ldots r_m)$
 check$(\Gamma, s_2 \ldots s_n)$

Other statements work in a constant environment.

Annotating type checkers

Usually the type checker is expected to return a more informative syntax tree to the later phases, a tree with **type annotations**.

Here are the type signatures of an annotating type checker:

Exp	*infer*	(*Env* Γ, *Exp* e)	*infer type of Exp*
Exp	*check*	(*Env* Γ, *Exp* e, *Type* t)	*check type of Exp*
Stms	*check*	(*Env* Γ, *Stms* s)	*check sequence of stms*
Def	*check*	(*Env* Γ, *Def* d)	*check function definition*
Program	*check*	(*Program* p)	*check a whole program*

The abstract syntax needs to be extended with a constructor for type-annotated expressions. We will denote them with $[e : t]$ in the pseudocode.

Type inference for addition expression (without type conversions but just overloadings)

$$
\begin{array}{l}
\mathit{infer}(\Gamma, a + b) : \\
\quad [a' : t] := \mathit{infer}(\Gamma, a) \\
\quad \textit{// here, check that } t \in \{\mathit{int}, \mathit{double}, \mathit{string}\} \\
\quad [b' : t] := \mathit{check}(\Gamma, b, t) \\
\quad \textbf{return } [a' + b' : t]
\end{array}
$$

After running the type checker, syntax trees will have type annotations all over the place, because every recursive call returns an annotated subtree.

Annotated trees

An easy way to add type-annotated expressions in the abstract syntax is to use an **internal rule** in BNFC:

```
internal ETyped. Exp ::= "[" Exp ":" Typ "]" ;
```

An internal rule is not added to the parser, but only to the abstract syntax, the pretty-printer, and the syntax-directed translation skeleton.

If type conversions are wanted, they can be added by the C++ style rule

```
EConv. Exp ::= Typ "(" Exp ")" ;
```

A conversion is added to the operand that does not have the maximal type:

$\mathit{infer}(\Gamma, a + b)$:

$[a' : u] := \mathit{infer}(\Gamma, a)$

$[b' : v] := \mathit{infer}(\Gamma, b)$

// here, check that $u, v \in \{\mathit{int}, \mathit{double}, \mathit{string}\}$

if $(u < v)$

  return $[v(a') + b' : v]$

else if $(v < u)$

  return $[a' + u(b') : u]$

else

  return $[a' + b' : u]$

Type checker in Haskell

(Java programmers can safely skip some slides now)

To implement the type checker in Haskell, we need three things:

- Define the appropriate auxiliary types and functions.
- Implement type checker and inference functions.
- Put the type checker into the compiler pipeline.

The compiler pipeline

Calls the lexer within the parser, reports a syntax error if the parser fails.

Proceed to type checking, showing an error message at failure and saying "OK" if the check succeeds.

Later compiler phases take over from the OK branch of type checking.

The compiler is implemented in the **IO monad**.

Internally, it also uses an **error monad**, which can be implemented by the **error type** defined in the BNFC generated code (the file `ErrM`).

```
compile :: String -> IO ()
compile s = case pProgram (myLexer s) of
  Bad err  -> do
    putStrLn "SYNTAX ERROR"
    putStrLn err
    exitFailure
  Ok tree -> case typecheck tree of
    Bad err -> do
      putStrLn "TYPE ERROR"
      putStrLn err
      exitFailure
    Ok _ -> putStrLn "OK"  -- or go to next compiler phase
```

Monads

The error type

```
data Err a = Ok a | Bad String
```

The value is either `Ok` of the expected type or `Bad` with an error message.

Whatever monad is used, its actions can be **sequence**d. For instance, if

```
checkExp :: Env -> Exp -> Type -> Err ()
```

then several checks one after the other are combined with `do`

```
do checkExp env exp1 typ
   checkExp env exp2 typ
```

You can **bind** variables returned from actions, and **return** values.

```
do typ1 <- inferExp env exp1
   checkExp env exp2 typ1
   return typ1
```

If you are only interested in side effects, you can use the dummy value type `()` (corresponds to `void` in C and `void` or `Object` in Java).

Symbol tables

We use an **environment** with separate parts for the function type table and the stack of variable contexts.

We use the `Map` type for symbol tables, and a list type for the stack.

```
type Env = (Sig,[Context])       -- functions and context stack
type Sig = Map Id ([Type],Type) -- function type signature
type Context = Map Id Type       -- variables with their types
```

Auxiliary operations on the environment have the following types:

```
lookupVar :: Env -> Id -> Err Type
lookupFun :: Env -> Id -> Err ([Type],Type)
updateVar :: Env -> Id -> Type -> Err Env
updateFun :: Env -> Id -> ([Type],Type) -> Err Env
newBlock  :: Env -> Env
emptyEnv  :: Env
```

Data abstraction

You should keep the environment datatype abstract, that is, use it only via these operations.

Then you can switch to another implementation if needed, for instance to make it more efficient or add more things in the environment.

You can also more easily modify your type checker code to work as an interpreter or a code generator, where the environment is different but the same operations are needed.

Pattern matching for type checking and inference

Here is type inference for some expression forms, following the BNFC case skeleton:

```
inferExp :: Env -> Exp -> Err Type
inferExp env x = case x of
  ETrue   -> return Type_bool
  EInt n  -> return Type_int
  EId id  -> lookupVar env id
  EAdd exp1 exp2 ->
    inferBin [Type_int, Type_double, Type_string] env exp1 exp2
```

Overloaded operators

A generic auxiliary for overloaded binary operations:

```
inferBin :: [Type] -> Env -> Exp -> Exp -> Err Type
inferBin types env exp1 exp2 = do
  typ <- inferExp env exp1
  if elem typ types
    then
      checkExp env exp2 typ
    else
      fail $ "wrong type of expression " ++ printTree exp1
```

The BNFC-generated function

```
printTree :: a -> String
```

converts a syntax tree of any type `a` to a string using the pretty-printer.

Checking in terms of inference

Checking expressions is defined in terms of type inference:

```
checkExp :: Env -> Type -> Exp -> Err ()
checkExp env typ exp = do
  typ2 <- inferExp env exp
  if (typ2 = typ) then
      return ()
    else
      fail $ "type of " ++ printTree exp ++
             "expected " ++ printTree typ ++
             "but found " ++ printTree typ2
```

Checking statements

```
checkStm :: Env -> Type -> Stm -> Err Env
checkStm env val x = case x of
  SExp exp  -> do
    inferExp env exp
    return env
  SDecl typ x  ->
    updateVar env id typ
  SWhile exp stm  -> do
    checkExp env Type_bool exp
    checkStm env val stm
```

Checking statement lists

As the statement checker may change the environment, statement lists are simple:

```
checkStms :: Env -> [Stm] -> Err Env
checkStms env stms = case stms of
  [] -> return env
  x : rest -> do
    env' <- checkStm env x
    checkStms env' rest
```

A seasoned Haskell programmer would of course simply write

```
checkStms = foldM checkStm
```

Type checker in Java

(Haskell programmers can safely skip the end of this chapter.)

Main problem: how to implement pattern matching.

We want to be modular - to write separate modules for a type checker, a code generator, an interpeter perhaps some optimizations, perhaps an interpreter.

The solution is the **visitor pattern**. It is supported by BNFC, which generates the **visitor interface** and skeleton code to implement a visitor.

Each compiler component can be put to a separate class, which implements the visitor interface.

The visitor pattern

Three ingredients:

- `Visitor<R,A>`, the interface to be implemented by each application
- `R visit(Tree p, A arg)`, the interface methods in `Visitor` for each constructor
- `R accept(Visitor v, A arg)`, the abstract class method calling the visitor

These are generated by BNFC.

The class parameters `A` and `R` make the visitor applicable to different tasks:

- in type inference `A` is a context and `R` is a type. Let us look at the code:

```
public abstract class Exp {
  public abstract <R,A> R accept(Exp.Visitor<R,A> v, A arg);
  public interface Visitor <R,A> {
    public R visit(Arithm.Absyn.EAdd p, A arg);
    public R visit(Arithm.Absyn.EMul p, A arg);
    public R visit(Arithm.Absyn.EInt p, A arg);
  }
}
public class EAdd extends Exp {
  public final Exp exp_1, exp_2;
  public <R,A> R accept(Arithm.Absyn.Exp.Visitor<R,A> v, A arg) {
    return v.visit(this, arg);
  }
}
public class EInt extends Exp {
  public final Integer integer_;
  public <R,A> R accept(Arithm.Absyn.Exp.Visitor<R,A> v, A arg) {
    return v.visit(this, arg);
  }
}
```

Example: the calculator

```
public class Interpreter {
  public Integer eval(Exp e) {
    return e.accept(new Value(), null ) ;
  }
  private class Value implements Exp. Visitor<Integer, Object> {
    public Integer visit (EAdd p, Object arg) {
      return eval(p.exp_1) + eval(p.exp_2) ;
    }
    public Integer visit (EMul p, Object arg) {
      return eval(p.exp_1) * eval(p.exp_2) ;
    }
    public Integer visit (EInt p, Object arg) {
      return p.integer_ ;
    }
  }
}
```

In the calculator,

- the return type `R` is `Integer`.
- the additional argument `A` is just `Object`; we don't need it for anything.
- the main class is `Interpreter` and contains
  - the public main method, `Integer eval(Exp e)`, calling the visitor with `accept`
  - the private class `Value`, which implements `Visitor` by making the `visit` method evaluate the expression

Running the interpreter

To understand the code, look at what happens when the calculator is run on the expression `2 + 3`:

`eval(EAdd(EInt(2),(EInt(3))))`	$\longrightarrow$	*eval calls accept*
`EAdd(EInt(2),(EInt(3))).accept(v,null)`	$\longrightarrow$	*accept calls visit*
`visit(EAdd(EInt(2),(EInt(3))),null)`	$\longrightarrow$	*visit calls eval*
`eval(EInt(2)) + eval(EInt(3))`	$\longrightarrow$	*eval calls accept, etc*

The logic is less direct than in Haskell's pattern matching:

`eval (EAdd (EInt 2) (EInt 3))`	$\longrightarrow$	*eval calls eval*
`eval (EInt 2) + eval (EInt 3)`	$\longrightarrow$	*eval calls eval, etc*

Type checker components

To implement the type checker in Java, we need three things:

- define the appropriate `R` and `A` classes;
- implement type checker and inference visitors with `R` and `A`;
- put the type checker into the compiler pipeline.

For the return type `R`, we use the class `Type` from the abstract syntax.

We also need a representation of function types:

```
public static class FunType {
  public LinkedList<Type> args ;
  public Type val ;
}
```

The environment

A symbol table (`HashMap`) of function type signatures

A stack (`LinkedList`) of variable contexts

Lookup and update methods:

```
public static class Env {
  public HashMap<String,FunType> signature ;
  public LinkedList<HashMap<String,Type>> contexts ;

  public static Type lookupVar(String id) { ...} ;
  public static FunType lookupFun(String id) { ...} ;
  public static void updateVar (String id, Type ty) {...} ;
  // ...
}
```

Comparing types for equality

(Something that Haskell gives for free)

A special enumeration type of **type codes**:

```
public static enum TypeCode { CInt, CDouble, CString, CBool, CVoid } ;
```

The headers of the main classes and methods

```
public void typecheck(Program p) {
  }
public static class CheckStm implements Stm.Visitor<Env,Env> {
  public Env visit(SDecl p, Env env) {
  }
  public Env visit(SExp p, Env env) {
  }
  // ... checking different statements
public static class InferExp implements Exp.Visitor<Type,Env> {
  public Type visit(EInt p, Env env) {
  }
  public Type visit(EAdd p, Env env) {
  }
  // ... inferring types of different expressions
}
```

The top level

On the top level, the compiler ties together the lexer, the parser, and the type checker. Exceptions are caught at each level:

```
try {
  l = new Yylex(new FileReader(args[0]));
  parser p = new parser(l);
  CPP.Absyn.Program parse_tree = p.pProgram();
  new TypeChecker().typecheck(parse_tree);
} catch (TypeException e) {
    System.out.println("TYPE ERROR");
    System.err.println(e.toString());
    System.exit(1);
} catch (IOException e) {
    System.err.println(e.toString());
    System.exit(1);
} catch (Throwable e) {
    System.out.println("SYNTAX ERROR");
    System.out.println ("At line " + String.valueOf(l.line_num())
 + ", near \"" + l.buff() + "\" :");
    System.out.println("     " + e.getMessage());
    System.exit(1);
}
```

Visitors for type checking

Written by modifying a copy of the BNFC-generated file `VisitSkel.java`

Statements, with declarations and expression statements as examples:

```
public static class CheckStm implements Stm.Visitor<Env,Env> {
  public Env visit(SDecl p, Env env) {
    env.updateVar(p.id_,p.type_) ;
    return env ;
  }
  public Env visit(SExp s, Env env) {
    inferExp(s.exp_, env) ;
    return env ;
  }
 //...
}
```

Type inference, for overloaded multiplication expressions:

```
public static class InferExpType implements Exp.Visitor<Type,Env> {
  public Type visit(demo.Absyn.EMul p, Env env) {
    Type t1 = p.exp_1.accept(this, env);
    Type t2 = p.exp_2.accept(this, env);
    if (typeCode(t1) == TypeCode.CInt &&
        typeCode(t2) == TypeCode.CInt)
       return TInt;
    else
    if (typeCode(t1) == TypeCode.CDouble &&
        typeCode(t2) == TypeCode.CDouble)
       return TDouble;
    else
      throw new TypeException("Operands to * must be int or double.");
    }
  //...
}
```

The function `typeCode` converts source language types to their type codes:

```
public static TypeCode typeCode (Type ty) ...
```

It can be implemented by writing yet another visitor :-)