Compiler Construction

Sibylle Schupp[1]

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

SoSe16

Lecture 6

Outline

STS
Software
Technology
Systems

Structural recursion

Structural recursion is a form of recursion that goes along substructures of its input.

- Ex.: tree traversal, list processing
- Recursive functions over *algebraic data types* are often structurally recursive.
- If the recursive step does not refer to a subset of the initial argument, the recursion is called *generative*.
  - Ex. quicksort, which works on lists different from the input list

What is syntax-directed translation?

A translation is *syntax-directed* if it is a structural recursion over an (abstract) syntax tree.

- The major compilation phases can be implemented by syntax-directed translation:
  - NFA generation
  - Syntax analysis
  - Type checking
  - Code generation
- Other applications
  - Interpreter
  - Program transformation
  - Circuit design
  - ...

Syntax-directed translation is a very powerful technique!

Major implementation techniques

At implementation level, syntax-directed translation requires a tree traversal and a tree transformation. Different paradigms provide different support:

- In OOP, one may use the visitor pattern. The tree can simply be updated in-place.
- In FP, one uses pattern matching. For in-place updates, one employs a monadic style.

Outline

The Visitor pattern

The visitor pattern separates traversal of a composed structure from operations on its components. It includes two class hierarchies: for the visiting classes and for the visited classes.

- The visited classes contain a method `accept` that takes a visitor object.
  - `accept` is a callback function that passes itself back to its visitor. It is mere boilerplate code.
  - The client calls this method.
- The visitor class contains one `visit` function per substructure.
- A visitor framework provides two abstract base classes, for the visitor and the visited classes each.

STS Software Technology Systems TUHH

The expression grammar, again

Here is the expression grammar again:

```
EAdd. Exp  ::= Exp  "+" Exp1 ;
ESub. Exp  ::= Exp  "-" Exp1 ;
EMul. Exp1 ::= Exp1 "*" Exp2 ;
EDiv. Exp1 ::= Exp1 "/" Exp2 ;
EInt. Exp2 ::= Integer ;
```

- The expressions are the objects that are visited.
- The visitor contains one visit-method per BNFC label ("constructor").

STS Software Technology Systems

BNFC-generated framework

The abstract base classes depend only on the grammar.

BNFC generates the complete code for the abstract classes of the visitor automatically.

- For C++ , see `Absyn.[CH]`
- For Java, see `Absyn/Exp.java`

Generated base class for the visitor: C++

```
class Visitor {
public:
  virtual ~Visitor() {}
  virtual void visitExp(Exp *p) = 0;
  virtual void visitEAdd(EAdd *p) = 0;
  virtual void visitESub(ESub *p) = 0;
  virtual void visitEMul(EMul *p) = 0;
  virtual void visitEDiv(EDiv *p) = 0;
  virtual void visitEInt(EInt *p) = 0;

  virtual void visitInteger(Integer x) = 0;
  virtual void visitChar(Char x) = 0;
  virtual void visitDouble(Double x) = 0;
  virtual void visitString(String x) = 0;
  virtual void visitIdent(Ident x) = 0;
};
```

- There is one visit-method per label (plus BNFC built-in tokens).
- Note the C++ syntax for abstract methods.

Generated base class `Visitable` in C++

```
class Visitable {
public:
  virtual ~Visitable() {}
  virtual void accept(Visitor *v) = 0;
};
```

- The base class for the visited hierarchy does not even depend on the grammar but is always the same.
- Note the argument type of accept, `Visitor`.

Generated concrete visited classes

bnfc further generates all concrete classes of the visited hierarchy. Those classes depend on the particular grammar.

```
class Exp: public Visitable {...}

class EAdd: public Exp {...}
class ESub: public Exp {...}
class EDiv: public Exp {...}
class EMul: public Exp {...}
class EInt: public Exp {...}
```

- The start symbol, Exp, becomes the root class of the hierarchy: it inherits from Visitable.
- There is one class per BNFC label. Note the correspondence to the visiting methods.
- Its implementation is boilerplate code.

STS Software Technology Systems TUHH

Generated concrete visited classes (2)

```
class Exp: public Visitable {...}

class EAdd: public Exp {
    ...
    accept(Visitor *v) {
        v->visitEAdd(this);
    }
};
class ESub: public Exp {
    ...
    accept(Visitor *v) {
        v->visitESub(this);
    }
};
```

- Each visited class implements an `accept` method by calling the appropriate visitor method.
- The name of the visitor method is known (since it is also derived from the grammar).

Visitor pattern, so far

Thus far, we have seen three components of a visitor framework:

- The abstract classes `Visitor` and `Visitable`, and the concrete visited classes for the particular grammar (`Exp, EAdd`,..).

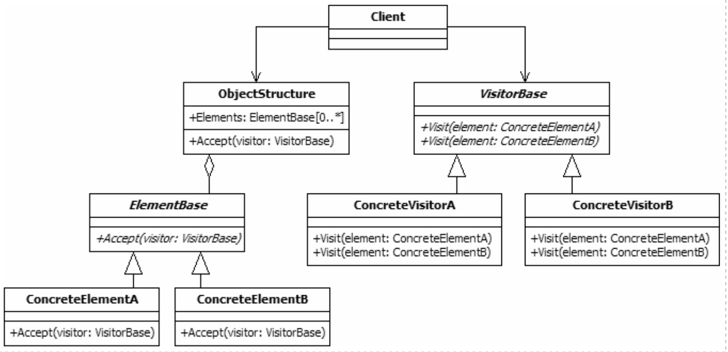
There are two more components: concrete *visitor* classes and a client.

- What is the purpose of traversing the AST?
  - Pretty-printing
  - Type checking
  - Evaluating
  - Translating
  - ...

  Each constitutes a *concrete* visitor of its own.
- The client is a small method or class that ties the two hierarchies.

The Visitor pattern: UML specification

http://www.blackwasp.co.uk/Visitor.aspx



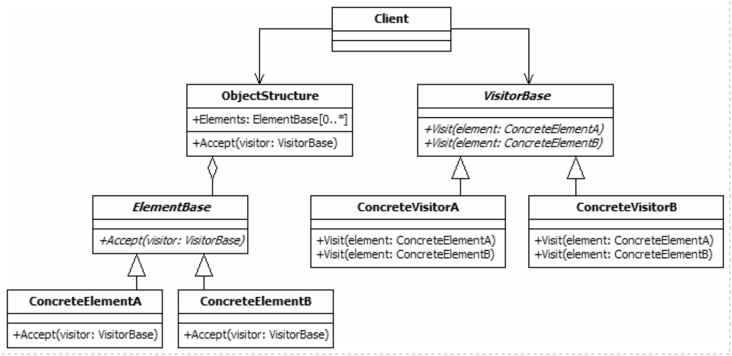
In the previous example:

- In `bnfc`, `ElementBase` represents `Exp`, `ConcreteElementA` represents `EAdd`, `VisitorBase` represents `Visitor`.
- `Visitable` has no counterpart in the diagram.

STS Software Technology Systems

The Visitor pattern (cont'd)

http://www.blackwasp.co.uk/Visitor.aspx



Assume the client is an interpreter, with main function `eval`. Call sequence

- `eval` → `Accept` → `Visit` → `eval` → `Accept` → ... where the second eval is called on a substructure.

Call sequence (example, pseudo bnfc)

We did not look yet into the implementation of the interpreter (eval visitor), but a possible call sequence might look as follows:

$$
\begin{array}{ll}
 & \text{eval(EAdd(EInt(2),EInt(3)))} \\
\rightarrow & \text{EAdd(EInt(2),EInt(3)).Accept} \\
\rightarrow & \text{Visit(EAdd(EInt(2),EInt(3))} \\
\rightarrow & \text{eval(EInt(2))} + \text{eval(EInt(3))} \\
\rightarrow & \text{EInt(2).Accept} + \text{EInt(3).Accept} \\
\rightarrow & \text{Visit (EInt(2))} + \text{Visit(EInt(3))} \\
\rightarrow & 2 + 3
\end{array}
$$

Concrete visitor: skeleton interface

A concrete visitor still contains generated code: The names and signatures of the methods are known.

bnfc generates this code automatically.

```
#include "Absyn.H"

class Skeleton : public Visitor
{
public:
  void visitExp(Exp* p);
  void visitEAdd(EAdd* p);
  void visitESub(ESub* p);
  void visitEMul(EMul* p);
  void visitEDiv(EDiv* p);
  void visitEInt(EInt* p);

  ...
};
```

Concrete visitor: skeleton implementation

Even the implementation is partially known: The callback into the visited class can be generated automatically.

```
#include "Skeleton.H"

void Skeleton::visitEAdd(EAdd *eadd)
{
  /* Code For EAdd Goes Here */

  eadd->exp_1->accept(this);
  eadd->exp_2->accept(this);

}
```

- The visitor-specific code goes into the comments. Note: it might also be necessary to modify the generated invocation of `accept`.

STS Software Technology Systems TUHH

The visitor pattern in BNFC

- BNFC generates the complete code for the abstract classes of the visitor automatically. It further generates the abstract class `Visitable` and its concrete implementations.
  - For C++ , see `Absyn.H`
  - For Java, see `Absyn/Exp.java`
- For the concrete visitor classes it generates the boilerplate code
  - For C++ , see `Skeleton.[CH]`
  - For Java, see `VisitSkel.java`
- Manually, one only has to write the client and to complete the concrete visitor classes.

For Haskell, BNFC also generates a skeleton, but the boilerplate is very different (see below).

Writing your own visitor: work flow in C++

- Copy `Skeleton.[CH]` and rename.
- Add a client method that takes as parameter a `Visitable*`, add instance variable if needed.
- Edit the visit-methods.
- Copy `Test.C` to `Test2.C` and extend or modify it accordingly.
- Copy `Makefile` to `Makefile.new` and replace `Test.C` by `Test2.C` (and rename the main executable).
- Call `make -f Makefile.new`.

STS Software Technology Systems

Example (visitor)

An interpreter is easy to build as visitor:

1. Rename Skeleton.[CH] to Interpreter.[CH].
2. Add a client method that takes a Visitable* and calls accept on it. Since evaluation returns a value, we introduce an instance variable, val.

```
Integer Interpreter::eval(Visitable* v){
  v->accept(this);
  return val;
}
```

3. Edit the visit-methods. All accept and visit-methods return void. We therefore update val as a side effect:

```
void Interpreter::visitEAdd(EAdd *eadd)
{
  /* Code For EAdd Goes Here */
  // eadd->exp_1->accept(this);
  // eadd->exp_2->accept(this);
 val = eval(eadd->exp_1) + eval(eadd->exp_2);
}
```

STS Software Technology Systems

Example (visitor)

```
#include "Interpreter.H"

Interpreter::Interpreter() {}
Interpreter::~Interpreter() {}
Integer Interpreter::eval(Visitable* v){
  v->accept(this);
  return val;
}

void Interpreter::visitExp(Exp* t) {} //abstract class

void Interpreter::visitEAdd(EAdd *eadd)
{
  /* Code For EAdd Goes Here */
  // eadd->exp_1->accept(this);
  // eadd->exp_2->accept(this);
 val = eval(eadd->exp_1) + eval(eadd->exp_2);
}
void Interpreter::visitInteger(Integer x)
{
  /* Code for Integer Goes Here */
  val = x;
}
```

Building

```
> make -f Makefile
g++ -g -c Absyn.C
g++ -g -c Lexer.C
Calc.l:42:68: warning: control reaches end of non-void
function [-Wreturn-type]
int initialize_lexer(FILE *inp) { yyrestart(inp);
 BEGIN YYINITIAL; }
                                                                   ^←


1 warning generated.
g++ -g -c Parser.C
g++ -g -c Printer.C
g++ -g -c Test2.C
g++ -g -c Interpreter.C
Linking ...
g++ -g Absyn.o Lexer.o Parser.o Printer.o Interpreter.o Test2.o
    -o TestInterpreter
```

STS Software Technology Systems

Example (visitor)

```
> ./TestInterpreter input

Parse Successful!

[Abstract Syntax]
(EAdd (EInt 1) (EMul (EInt 3) (EInt 4)))

[Linearized Tree]
1+ 3* 4

[Value after Evaluation]
13
```

A visitor for type checking

Type checking can be realized by an appropriate visitor.

- The visited hierarchy is as before.
- The visitor, `TypeChecker`, follows the same logic as before.
- The biggest difference to the example before is not in the logic but in the data structures and types
  - The type checker relies on an environment: `Env`
  - The environment requires that its entries are encapsulated in types: `FunType`, ...

The class TypeChecker

```
class TypeChecker : public Visitor
{
public:
  ~TypeChecker();
  Type* typecheck(Visitable* v);
private:
  Type* ty_;

  class Env {
     ...
  };

  Env env_;
};

TypeChecker::~TypeChecker() {}
Type* TypeChecker::typecheck(Visitable* v) {
    v->accept(this);
    return ty_;
}
```

Example (type checking)

```
void TypeChecker::visitSInit(SInit *sinit)
{
  /* Code For SInit Goes Here */
  env_.updateVar(sinit->id_, sinit->type_); // id_, type_ ↩
      appropriate


}
void TypeChecker::visitId(Id x)
{
  /* Code for Id Goes Here */
  ty_ = env_.lookup(x);   // error handling omitted
}
```

- The environment is updated in initalization statements.
- It is looked up whenever an identifier is used.

STS Software Technology Systems TUHH

Outline

Syntax-directed type checking, part II

The alternative solution is based on pattern matching and a monadic style.

- Pattern matching is available in many functional languages.
  - Crucial ingredient: algebraic data types
  - Pattern matching on the constructors of an algebraic data type
- For *stateful* programming, functional languages employ special machinery: monads
  - IO monad, Err monad, State monad

What is an algebraic data type?

An *algebraic data types (ADT)* is a type that is defined as a *sum* type.

- Sum types are similar to union types, but each value can be only of one type (disjoint union of types).
- In OO-speak, “instanceOf” a value returns always the same type.
- Mathematical foundation in category theory

STS Software Technology Systems TUHH

Example (ADT)

In Haskell, the keyword data is used for the definition of an ADT, the |
operator is the sum operator.

```
data Value = VInt Integer | VDouble Double | VUndef
```

- The type `Value` has 3 constructors: `VInt`, `VDouble`, `VUndef`.
- Constructors can take arguments; the type of their arguments must be known.
- ADTs can be recursive types.

Example (ADTs and instances)

```
data Value = VInt Integer | VDouble Double | VUndef
> :type VInt 1
VInt 1 :: Value
> :type VUndef
VUndef :: Value
> :type VDouble 2
VDouble 2 :: Value
> VInt 1
VInt 1
> VDouble 2
VDouble 2.0
> VUndef
VUndef
```

- Values (instances) of type `Value` must be constructed using one of the three constructors.
- A constructor expression has a value. It cannot be further evaluated.

Another example (ADT)

- An ADT can be parameterized. In Haskell, type parameters are signified by lower-case characters.

```
data Err a = Ok a | Bad String      -- in the Haskell Prelude
> :type Ok (VInt 3)
Ok (VInt 3) :: Err Value
> :type Ok (VDouble 3)
Ok (VDouble 3) :: Err Value
> :type Ok (VUndef)
Ok (VUndef) :: Err Value
> :type Bad "ill-defined"
Bad "ill-defined" :: Err a
> Bad "ill-defined"
Bad "ill-defined"
```

- The type `Err` is parameterized by the expected type. It has two constructors: `Ok` takes a value of the expected type, `Bad` requires a string.
- `Err ()` is used when the Ok-value does not matter.

Example: the language Mini

```
-- Mini.cf

Prog. Program ::= [Stm] ;

terminator Stm "" ;

SDecl.  Stm ::= Type Ident ";"  ;
SAss.   Stm ::= Ident "=" Exp ";" ;
SBlock. Stm ::= "{" [Stm] "}" ;
SPrint. Stm ::= "print" Exp  ";" ;

EVar.    Exp1 ::= Ident ;
EInt.    Exp1 ::= Integer ;
EDouble. Exp1 ::= Double ;
EAdd.    Exp  ::= Exp "+" Exp1 ;

coercions Exp 1 ;

TInt.    Type ::= "int" ;
TDouble. Type ::= "double" ;
```

Implementing a grammar

We have already seen the ADTs for productions in BNFC:

```
data Program =
   Prog [Stm]
  deriving (Eq,Ord,Show,Read)
data Stm =
   SDecl Type Ident
  | SAss Ident Exp
  | SBlock [Stm]
  | SPrint Exp
  deriving (Eq,Ord,Show,Read)
data Exp =
   EVar Ident
  | EInt Integer
  | EDouble Double
  | EAdd Exp Exp
  deriving (Eq,Ord,Show,Read)
data Type =
   TInt
  | TDouble
  deriving (Eq,Ord,Show,Read)
```

ADTs and pattern matching

- ADTs suggest writing functions by pattern matching on the constructor.
- Ex.: statements are checked by pattern matching on the statement labels (constructors). (Recall that [Stm] refers to a list of statement.)

```
data Stm =
    SDecl Type Ident
  | SAss Ident Exp
  | SBlock [Stm]
  | SPrint Exp
   deriving (Eq,Ord,Show,Read)

checkStm env s =
      case s of
        SDecl t x        -> ...
        SAss x e         -> ...
        SBlock stms      -> ...
        SPrint e         -> ...
```

ADTs and pattern matching (2)

Similarly, expressions are checked/inferred/... by pattern matching on the expression labels.

```
data Exp =
   EVar Ident
 | EInt Integer
 | EDouble Double
 | EAdd Exp Exp
  deriving (Eq,Ord,Show,Read)

inferExp env e =
    case e of
      EVar x          -> ...
      EInt _          -> ...
      EDouble _       -> ....
      EAdd e1 e2      -> ...
```

Type checker: function types

http://www.cse.chalmers.se/edu/year/2011/course/TIN321/laborations/mini/haskell

```
module TypeChecker where

import AbsMini
import PrintMini
import ErrM

typecheck :: Program -> Err ()
checkStms :: Env -> [Stm] -> Err ()
checkStm  :: Env -> Stm -> Err Env
checkExp  :: Env -> Exp -> Type -> Err ()
inferExp  :: Env -> Exp -> Err Type
```

- The type checker module contains one function per syntactic category.
- The functions depend on the environment and return either an `Ok` value or a `Bad` message.
- The arrow notation is called *currying*. It is widely used in functional programming (see below for more).

STS Software Technology Systems

Pattern matching: checking statements

Statements are checked by pattern matching on the statement labels.

```
checkStm :: Env -> Stm -> Err Env
checkStm env s =
    case s of
      SDecl t x        -> addVar env x t
      SAss x e         -> do t <- lookupVar env x
                             checkExp env e t
                             return env
      SBlock stms      -> do checkStms (addScope env) stms
                             return env
      SPrint e         -> do inferExp env e
                             return env
```

- checkStm takes two arguments and pattern-matches on the second.
- Each of the four BNFC labels of Stm defines a case. The cases take different arguments.
- The logic is close to the typing rules.
- For the do/ return-notation, see below.

Pattern matching: infering expressions

Expression inference pattern-matches on the expression labels.

```
inferExp :: Env -> Exp -> Err Type
inferExp env e =
    case e of
      EVar x         -> lookupVar env x
      EInt _         -> return TInt
      EDouble _      -> return TDouble
      EAdd e1 e2
         -> do t1 <- inferExp env e1
               t2 <- inferExp env e2
               if t1 == t2
                 then return t1
                 else fail (printTree e1 ++ " has type " ++
                    printTree t1  ++ " but " ++ printTree e2
                    ++ " has type " ++ printTree t2)
```

- Each of the four syntactic categories constitute a case.
- Inference proceeds recursively. The base cases represent constants (EInt, EDouble, and variables). In Mini, operands of a binary expression must have the same type.

Record type

- A record is an algebraic data type with one constructor ("product type").
- In Haskell, records have a special syntax: `{..}` and symbolic names.
- Example:

```
data Env = Env {
     context ::  [[(Ident, Type)]]
}
```

  - The definition is equivalent to
    `data Env = Env [[(Ident, Type)]]`

- In the example, the record contains one field only. Typically, records are used to group several fields.

STS Software Technology Systems

Access functions for records

In Haskell, for every field name of a record, an access function is automatically generated.

```
data Env = Env {
    context :: [[(Ident, Type)]]
}

> :type context
context :: Env -> [[(Ident, Type)]]
```

The environment: initialization and extension

- The initial environment initializes all fields:

```
emptyEnv :: Env
emptyEnv = Env {
     context = [[]]
}
```

- The environment gets populated when declarations are processed and type checking commences.
- The extension can be organized in-place, using the `State` monad.

STS Software Technology Systems TUHH

Outline

`State` in Haskell

- For in-place updates, use the type `State`:

```
State s
```

  where `s` denotes the type of the state, in our case: `Env`.
- For our purposes, you get by with two functions:
  - `get`: returns the state
  - `modify (s -> s)`: updates the state (by applying its argument to the old state)
- For the driver, you need an additional function
  - `execState` takes the initial environment `emptyEnv`, runs the check, and returns the final environment.
- The `State` type is a monad. Monads are a special feature in Haskell to allow for computations with side effects ("actions").

Example modify

- Adding a pair (x,t) to the environment in-place:

```
addVar :: Ident -> Type -> State Env ()
addVar x t = do
  (scope:rest) <- gets context
  case lookup x scope of
    Nothing ->
       modify (\env -> env{ context=((x,t):scope):rest })
    Just _  ->
       fail ("Variable " ++ printTree x ++
             " already declared.")
```

- The instruction gets retrieves the current context.
- The pair (x,t) is cons'ed with the current scope of the scope stack the context access function returns.
- modify changes the context field of that particular instance of Env.

Another example: gets

- Since the environment is stateful, one must use gets to get it.

```
lookupVar :: Ident -> State Env Type
lookupVar x = do
    (scope:rest) <- gets context
    return $ look x (scope:rest)
 where
    look x []               = error $ "Unknown variable " ++
                                      printTree x ++ "."
    look x (scope:rest)  = case lookup x scope of
                             Nothing -> look x rest
                             Just t  -> t
```

- The do-notation is used in monads to sequence computations. The layout rule applies.
- The method return is defined in the type class Monad. It is used to "inject" a value in a monad. It has nothing to do with the return statement in C-like languages.

```
:type return
return :: Monad m => a -> m a
```

Copy vs. monad

```
-- stateful environment
inferExp :: Exp -> State Env Type
inferExp e =
    case e of
        EVar x        -> lookupVar x
        EInt _        -> return TInt
        EDouble _     -> return TDouble
        EAdd e1 e2    -> do t1 <- inferExp e1
                            t2 <- inferExp  e2
                            if t1 == t2
                              then return t1
                              else fail (printTree e1 ++ "...")
-- functional environment
inferExp :: Env -> Exp -> Err Type
inferExp env e =
   case e of
      EVar x          -> lookupVar env x
      EInt _          -> return TInt
      EDouble _       -> return TDouble
      EAdd e1 e2       -> do t1 <- inferExp env e1
                             t2 <- inferExp env e2
                             if t1 == t2
```

STS Software Technology Systems

Function types

- The monadic style has in impact on the function types of the type checker.
- If the environment is updated in-place, it needs no longer be passed around.

```
-- stateful environment
checkStms :: [Stm] -> State Env ()
checkStm  :: Stm -> State Env ()
checkExp  :: Exp -> Type -> State Env ()
inferExp  :: Exp -> State Env Type
-- functional environment
checkStms :: Env -> [Stm] -> Err ()
checkStm  :: Env -> Stm -> Err Env
checkExp  :: Env -> Exp -> Type -> Err ()
inferExp  :: Env -> Exp -> Err Type
```

Summary

- Implementation via visitor resp. pattern matching
- Visitor pattern: two hierarchies, accept/visit callback
- Pattern matching on constructors of ADTs
- Monadic style, the State monad

References

- IPL, Ch. 4.11, 4.12