Semantic analysis for Dolphin/ Phase 1

Compilers 2024

Aslan Askarov

Dolphin Phase 1 AST

```
type ident = Ident of {name : string;}
type typ = Int | Bool
type binop = Plus | Minus | Mul | Div | Rem | Lt | Le | Gt | Ge | Lor | Land | Eq | NEq
type unop = Neg | Lnot

type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr}
| UnOp of {op : unop; operand : expr}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr;}
| Call of {fname : ident; args : expr list}
and lval =
| Var of ident
```

```
type statement =
| VarDeclStm of {name : ident;
      tp : typ option; body : expr}
| ExprStm of {expr : expr option}
| IfThenElseStm of {cond : expr;
      thbr : statement;
      elbro : statement option}
| CompoundStm of {
      stms : statement list}
| ReturnStm of {ret : expr}

type program = statement list
```

Dolphin Phase 1 AST

```
type ident = Ident of {name : string;}
```

Identifier are just strings, but are wrapped in a record for stronger “typing hygiene” to avoid confusing other uses of strings

Dolphin Phase 1 AST

```
type ident = Ident of {name : string;}
type typ = Int | Bool
type binop = Plus | Minus …
type unop = Neg | Lnot
```

Two types: integer and booleans
- standard binary operations
- a couple of standard unary operations per type

Dolphin Phase 1 AST

A program is a list of statements

```
type statement =
| VarDeclStm of {name : ident;
      tp : typ option; body : expr}
| ExprStm of {expr : expr option}
| IfThenElseStm of {cond : expr;
      thbr : statement;
      elbro : statement option}
| CompoundStm of {
      stms : statement list}
| ReturnStm of {ret : expr}

type program = statement list
```

Dolphin Phase 1 AST

Statements are standard for a C/Java style imperative language

- var declaration, with optional type annotation
- expression statement that can be empty
- conditional, with optional else branch
- compound is a list of statements
- return of an expression

```
type statement =
| VarDeclStm of {name : ident;
      tp : typ option; body : expr}
| ExprStm of {expr : expr option}
| IfThenElseStm of {cond : expr;
      thbr : statement;
      elbro : statement option}
| CompoundStm of {
      stms : statement list}
| ReturnStm of {ret : expr}

type program = statement list
```

Dolphin Phase 1 AST

Expressions are also standard

```
type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr}
| UnOp of {op : unop; operand : expr}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr;}
| Call of {fname : ident; args : expr list}
and lval =
| Var of ident
```

L-values (definition, Appel pg. 515)

An *l-value* is a location whose value may be read or assigned. Variables, procedure parameters, fields of records and elements of arrays are all *l-values*

Examples of l-values in (full) Dolphin

x

b[i]

a.foo

a.bar[j].p

Dolphin Phase 1 AST

Expressions are also standard

```
type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr}
| UnOp of {op : unop; operand : expr}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr;}
| Call of {fname : ident; args : expr list}
and lval =
| Var of ident
```

L-values (definition, Appel pg. 515)

An *l-value* is a location whose value may be **read** or assigned. Variables, procedure parameters, fields of records and elements of arrays are all *l-values*

Examples of l-values in (full) Dolphin

y = x

y = b[i]

y = a.foo

y = a.bar[j].p

Dolphin Phase 1 AST

Expressions are also standard

```
type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr}
| UnOp of {op : unop; operand : expr}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr;}
| Call of {fname : ident; args : expr list}
and lval =
| Var of ident
```

L-values (definition, Appel pg. 515)

An *l-value* is a location whose value may be read or **assigned**. Variables, procedure parameters, fields of records and elements of arrays are all *l-values*

Examples of l-values in (full) Dolphin

x = e

b[i] = e

a.foo = e

a.bar[j].p = e

Dolphin Phase 1 AST

Expressions are also standard

L-values (definition, Appel pg. 515)

An *l-value* is a location whose value may be read or assigned. Variables, procedure parameters, fields of records and elements of arrays are all *l-values*

```
type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr}
| UnOp of {op : unop; operand : expr}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr;}
| Call of {fname : ident; args : expr list}
and lval =
| Var of ident
```

The only l-values we can have now are identifiers, but it helps to have a placeholder for the future extension in the AST

Semantic analysis: from the AST to the typed AST

```ocaml
(* -- Use this in your solution without modifications *)
type ident = Ident of {name : string;}

type typ = | Int | Bool

type binop =
| Plus | Minus | Mul | Div | Rem | Lt | Le | Gt | Ge | Lor | Land |
Eq | NEq

type unop = | Neg | Lnot

type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr}
| UnOp of {op : unop; operand : expr}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr;}
| Call of {fname : ident; args : expr list}
and lval =
| Var of ident

type statement =
| VarDeclStm of {name : ident; tp : typ option; body : expr}
| ExprStm of {expr : expr option}
| IfThenElseStm of {cond : expr; thbr : statement; elbro : statement
option}
| CompoundStm of {stms : statement list}
| ReturnStm of {ret : expr}

type program = statement list
```

```ocaml
(* -- Use this in your solution without modifications *)
module Sym = Symbol

type ident = Ident of {sym : Sym.symbol}

type typ = Void | Int | Bool | ErrorType

type binop = Plus | Minus | Mul | Div | Rem | Lt | Le | Gt | Ge | Lor | Land | Eq
| NEq

type unop = Neg | Lnot

type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr; tp : typ}
| UnOp of {op : unop; operand : expr; tp : typ}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr; tp : typ}
| Call of {fname : ident; args : expr list; tp : typ}
and lval =
| Var of {ident : ident; tp : typ}

type statement =
| VarDeclStm of {name : ident; tp : typ; body : expr}
| ExprStm of {expr : expr option}
| IfThenElseStm of {cond : expr; thbr : statement; elbro : statement option}
| CompoundStm of {stms : statement list}
| ReturnStm of {ret : expr}

type param = Param of {paramname : ident; typ : typ}

type funtype = FunTyp of {ret : typ; params : param list}

type program = statement list
```

Semantic analysis

- Detailed conditions to check for the program being well-typed: see the handbook
- Implementation idea
  - traversal of the AST and reconstruction of the program in the Typed AST
  - accumulation of error messages to report
- Things to be aware of
  - environment for variable declarations/declaration shadowing, error management
  - auxiliary information/functions
  - it’s not always just type checking (next slide)…

Semantic analysis of expressions: type checking vs type inference

In variable declarations, types may be omitted

Examples

Examples	`var x: int = 5;`	ok, explicit type
	`var x = true;`	type omitted, also ok
	`var x: int = true;`	error

In general

`var x: τ = e;`

Type checking: because type τ is explicitly given, check that e has no other type errors and is indeed of type τ. Extend variable environment so that x has type τ

`var x = e;`

Type inference: because type is omitted only check that e has no type errors and infer its type. Extend variable environment so that x has the inferred type

Type checking can be implemented as type inference plus extra checks on the inferred type

```
exception Unimplemented
(* your code should eventually compile without this exception *)

let typecheck_typ = function
| Ast.Int -> TAst.Int
| Ast.Bool -> TAst.Bool

(* should return a pair of a typed expression and its
   inferred type. you can/should use typecheck_expr inside
  infertype_expr. *)
let rec infertype_expr env expr =
  match expr with
  | _ -> raise Unimplemented
and infertype_lval env lvl =
  match lvl with
  | _ -> raise Unimplemented
(* checks that an expression has the required type tp by
   inferring the type and comparing it to tp. *)
and typecheck_expr env expr tp =
  let texpr, texprtp = infertype_expr env expr in
  if texprtp <> tp then raise Unimplemented;
  texpr
```

Type checking is implemented as type inference plus extra checks on the inferred type.

```
exception Unimplemented
(* your code should eventually compile without this exception *)

let typecheck_typ = function
| Ast.Int -> TAst.Int
| Ast.Bool -> TAst.Bool

(* should return a pair of a typed expression and its
   inferred type. you can/should use typecheck_expr inside
  infertype_expr. *)
let rec infertype_expr env expr =
  match expr with
  | _ -> raise Unimplemented
and infertype_lval env lvl =
  match lvl with
  | _ -> raise Unimplemented
(* checks that an expression has the required type tp by
   inferring the type and comparing it to tp. *)
and typecheck_expr env expr tp =
  let texpr, texprtp = infertype_expr env expr in
  if texprtp <> tp then raise Unimplemented;
  texpr
```

Main work-horse function for the semantic analysis of expressions. This is where we pattern match on the expression.

```
exception Unimplemented
(* your code should eventually compile without this exception *)

let typecheck_typ = function
| Ast.Int -> TAst.Int
| Ast.Bool -> TAst.Bool

(* should return a pair of a typed expression and its
   inferred type. you can/should use typecheck_expr inside
  infertype_expr. *)
let rec infertype_expr env expr =
  match expr with
  | _ -> raise Unimplemented
and infertype_lval env lvl =
  match lvl with
  | _ -> raise Unimplemented
(* checks that an expression has the required type tp by
   inferring the type and comparing it to tp. *)
and typecheck_expr env expr tp =
  let texpr, texprtp = infertype_expr env expr in
  if texprtp <> tp then raise Unimplemented;
  texpr
```

To infer the type of an lvalue means just looking them up in the environment

Type checking of statements / program

```
(* should check the validity of a statement and produce the
   corresponding typed statement. Should use typecheck_expr
   and/or infertype_expr as necessary. *)
let rec typecheck_statement env stm =
  match stm with
  | _ → raise Unimplemented
(* should use typecheck_statement to check the block of statements. *)
and typecheck_statement_seq env stms = raise Unimplemented

(* the initial environment should include all the library
   functions, no local variables, and no errors. *)
let initial_environment = raise Unimplemented

(* should check that the program (sequence of statements) ends
    in a return statement and make sure that all statements are
    valid as described in the assignment.
    Should use typecheck_statement_seq. *)
let typecheck_prog prg = raise Unimplemented
```

Shadowing, scoping, and compound statements

```
var x: int = 1;
var x: int = 2;
return x; (* 2 *)
```

```
var x: int = 1;
var y: int = 0;
{
  y = x;
  var x: int = 2;
  y = y + x;
}
return x + y;  (* 4 *)
```

The scoping rules impact how we program our environments

Environments and scoping

Environment maintenance during the AST traversal

There are two classical approaches to environments : imperative vs functional (see Appel's book, pg 104-110)
Alternative taxonomy: destructive vs non-destructive

	Pros	Cons
Destructive (Imperative)	no need to "thread" the environment throughout the traversal	difficult to "undo"
Non-destructive (Functional)	easy to undo – just use the earlier version	the extra burden of having to thread it

env.ml

Mixed-style implementation

```ocaml
(* Env module *)
module TAst = TypedAst
module Sym = Symbol

(* your code should eventually compile without this exception *)
exception Unimplemented

type varOrFun =
| Var of TAst.typ
| Fun of TAst.funtype

type environment = { vars_and_funs : varOrFun Sym.Table.t
                   ; errors : Errors.error list ref}
```

```ocaml
type varOrFun =
| Var of TAst.typ
| Fun of TAst.funtype

type environment = { vars_and_funs : varOrFun Sym.Table.t
                   ; errors : Errors.error list ref}

(* create an initial environment with the given functions defined *)
let make_env function_types =
  let emp = Sym.Table.empty in
  let env =
    List.fold_left (fun env (fsym, ftp)
      -> Sym.Table.add fsym (Fun ftp) env) emp function_types
  in
  {vars_and_funs = env; errors = ref []}

(* insert a local declaration into the environment *)
let insert_local_decl env sym typ =
  let {vars_and_funs; _} = env in
  {env with vars_and_funs = Sym.Table.add sym (Var typ) vars_and_funs}

let insert_error env err =
  let {errors; _} = env in
  errors := err :: !errors

(* lookup variables and functions. Note: it must first
   look for a local variable and if not found then look
   for a function. *)
let lookup_var_fun env sym =
  let {vars_and_funs; _} = env in
  Sym.Table.find_opt sym vars_and_funs
```

Testing and dev strategies

Advise: make sure you have a working pipeline and that your compiler works for the features you think you have implemented

- Step 0: have a compiler that accepts no programs (!) but compiles itself.
- Step 1: Think of the simplest possible program in your language that you want to accept; extend your compiler to support that

Advise: identify complexity, break it down, and tackle it one piece at a time

Example 4-program progression for you to start your development

```
return 5;
```

programs consisting of a single return statements

```
return (1 + 4);
```

+ basic binops

```
var x: int = 5;
return x;
```

+ explicit var declaration
+ var-expressions

```
var x: int = 5;
var y = 1;
return (x + y);
```

+ omitted type in declaration