Static semantics

In a nutshell

- defined for statically typed languages
- implemented by a typechecker

Motivations

- parsers check only syntactic errors
- typecheckers can detect bugs that manifest at runtime

Reminder

- static means “before execution”
- dynamic means “during the execution” (that is, “at runtime”)

Static semantics

Syntax versus static semantics

- theoretical limitation: static semantics cannot be specified with regular expressions and CF grammars
- practical issue: implementation simpler and code better organized if checks are separated
  - phase 1: syntax checks and AST generation (parser)
  - phase 2: type checks (typechecker)

Static semantics in a nutshell

Main ingredients

- static types (or types, if there are no ambiguities) specify sets of values
- typing rules:
  - define the expressions and statements which are type correct and those which are not
  - define a type for each type correct expression
  - remark: statements do not have types

Simple examples

Expressions and statements which are type correct

`1*-2+1`	has type **`int`**
`fst (true,2)`	has type **`bool`**
`if(2==1+1){print 1}`	is type correct

Expressions and statements which are not type correct

`1*true`	error: found **`bool`**, expected **`int`**
`fst 2`	error: found **`int`**, expected **`pair`**
`if(2){print 1}`	error: found **`int`**, expected **`bool`**

More complex examples with variables

Expressions and statements which are type correct

`1*x`	has type **int** if `x` is declared and has type **int**
`fst y`	has type t_1 if `y` is declared and has type $t_1 * t_2$
`var z=true;` `if(z){print 1}`	is type correct

Programs which are not type correct

`if(z){print 1}`	error: undeclared variable `z`
`var z=1;` `if(z){print 1}`	error: found **int**, expected **bool**
`var z=false;` `if(z){var z=1; print true && z}`	error: found **int**, expected **bool**

Variable declarations

Typing rules depend on variable declarations

Required information associated with a variable declaration:

- the declared variable, in particular its name
- the type associated with the variable
- the scope of the declaration

Scope of a variable declaration

in the static semantics the scope of a variable declaration is
the code fragment where the declared variable is accessible

Standard typing rules on variables are used for our toy language

- variables must be declared before their use
- scopes can be nested by using blocks
- variables with the same name cannot be re-declared in the same scope
- variables declared in a nested scope hide the variables declared with the same name in outer scopes

Nested scopes

Variables with the same name can be declared in nested scopes

Blocks introduce nested scopes

A type correct program:

```
1 var x=1;
2 if(x==1){
3     var x=true;     // nested scope, outer x will be hidden
4     print x && true // type correct
5 };
6 print x + 1         // type correct
```

Remarks:

- the scope of **var** x=1 includes lines 2, 3 and 6
- the scope of **var** x=**true** includes line 4
- **var** x=1 is hidden by **var** x=**true** in line 4

Definition of the static semantics

Possible approaches

How can the static semantics of a language be defined?

- natural language
  - pros: requires minimal technical skills
  - cons: ambiguous, verbose, not executable, not suitable for technical details
- mathematical language
  - pros: very abstract, non ambiguous, very concise
  - cons: based on complex concepts, not directly executable
- declarative programming language (functional or logic)
  - pros: abstract, non ambiguous, concise, executable
  - cons: requires knowledge of the used programming language

Our choice

- a functional language, as OCaml, is a reasonable compromise
- Remark: although executable, the OCaml program is used only as a definition, a typechecker can be implemented more efficiently in Java

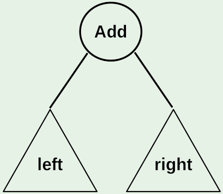
Typing rules

Rules defined on the abstract syntax (= AST)

- typing rules defined for each type of node of the AST
- typechecking of a program = a visit of its AST

Example of simple typing rules

- expressions with addition:



if `left` and `right` have type **`int`**,
then `Add(left,right)` has type **`int`**,
otherwise `Add(left,right)` is not type correct

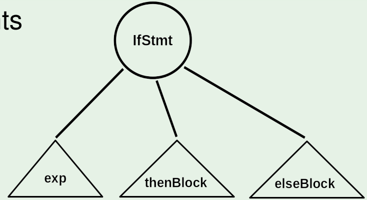
Typing rules

Rules defined on the abstract syntax (= AST)

- typing rules defined for each type of node of the AST
- typechecking of a program = a visit of its AST

Example of simple typing rules

- if-then-else statements



if `exp` has type **`bool`**, `thenBlock` and `elseBlock` are type correct,
then `IfStmt(exp,thenBlock,elseBlock)` is type correct
otherwise `IfStmt(exp,thenBlock,elseBlock)` is not type correct

Static environment

Static types

```
type static_type = IntType | BoolType | PairType of static_type * static_type;;
```

Definition of static environment

- a static environment *env* defines the variables accessible at a specific point in the program
- abstractly, *env* is a function: $env : Variable \rightarrow Type$
  - if *env* `(x)=IntType`, then `x` is accessible and has type `IntType` at the current point
  - if *env* `(x)` is undefined, then `x` is not accessible at the current point

Implementation of static environments

- static environment = a sorted list of scopes (also called scope chain)
- scope = a dictionary where keys are variables and values are types
- scopes are sorted: inner scopes come first
- reason: nested declarations hide variables declared at outer scopes

Static environments: examples in OCaml

Implementation details

- simple way to implement a scope in OCaml: a list of pairs `(variable,type)`
- this implementation is called association list, with predefined functions `List.mem_assoc` and `List.assoc`
- example of scope with `x` of type **`int`** and `y` of type **`bool`**:

```
[(Name "x",IntType);(Name "y",BoolType)]
```

A static environment with two nested scopes

List of association lists in OCaml

```
[
  [(Name "x",IntType);(Name "y",BoolType)];
  [(Name "x",BoolType)]
]
```

Static environments: variable lookup

Definition

- function `lookup` checks whether a variable is declared in the environment
- if the variable is found, then its associated type is returned
- otherwise an exception is raised

Examples of variable lookup

```
env=[
  [(Name "x",IntType);(Name "y",BoolType)];
  [(Name "x",BoolType);(Name "z",IntType)]
]

lookup (Name "x") env=IntType
lookup (Name "y") env=BoolType
lookup (Name "z") env=IntType
lookup (Name "w") env raises exception UndeclaredVariable (Name "w")
```

Definition of the static semantics in OCaml

Functions handling the environment

```
lookup : variable -> static_env -> static_type

dec : variable -> static_type -> static_env -> static_env

enter_scope : static_env -> static_env
```

Definition of the static semantics in OCaml

Semantic functions

```
typecheckProg : prog -> unit

typecheckStmt : static_env -> stmt -> static_env

typecheckBlock : static_env -> block -> unit

typecheckStmtSeq : static_env -> stmt_seq -> unit

typecheckExp : static_env -> exp -> static_type
```

Remark: all functions raise an exception if typechecking fails

Dynamic semantics

In a nutshell

- defines the behavior of a program at runtime (= when it is executed)
- implemented by an interpreter or a compiler

Reminder

- an interpreter directly executes the program; in other words, the program is executed on a virtual machine
- a compiler "translates" the program into "executable" lower-level code

Dynamic semantics

What does a program do when it is executed?

- evaluates expressions (= computes their values)
- performs I/O; e.g., prints strings on the standard output
- modifies the memory
    - adds new variables in the current scope
    - adds a new scope, to allocate new variables
    - removes a scope, to de-allocate their variables
    - modifies the content of variables
    - ...

Definition of the dynamic semantics

Provided by executable OCaml code, as done for the static semantics

Variable declarations

Scope of a variable declaration

- the notion of scope of a variable declaration in the dynamic semantics is more complex than in the static semantics
- two different dimensions are required
  - space: the code portion where the variable is accessible (similar definition as in the static semantics)
  - time: when the variable is allocated and de-allocated

Example for the temporal dimension

```
if(x>0) { var y=1; var z=false; ... }
```

- each time the execution of the if-block starts, a new nested scope is added to allocate variables `y` and `z`
- each time the execution of the if-block finishes, the corresponding scope is removed to de-allocate variables `y` and `z`
- remark: the same if-block can be executed more times; for instance, in case the **`if`** statement is contained in a loop

Dynamic environment

Definition, analogous to static environment

- a dynamic environment env_d defines the variables accessible at a specific point and time in the program
- abstractly, env_d is a function: $env_d : Variable \to Value$
- examples:
  - if env_d`(x)=42`, then `x` is accessible and has value 42 at the current point and time
  - if env_d`(x)` is undefined, then `x` is not accessible at the current point and time

Implementation of dynamic environments

Analogous to the implementation of static environments

- dynamic environment = a sorted list of scopes (also called scope chain)
- scope = a dictionary where keys are variables associated with values
- scopes are sorted: inner scopes come first
- reason: nested declarations hide variables declared at outer scopes

Variable update

Variable assignment

- assignment allows programmers to change the values stored in variables
- an update operation is required on dynamic environments

```
update : variable -> value -> dynamic_env -> dynamic_env
```

- in the static semantics environment, function `update` is not needed: the typing rule for assignment prohibits to change the type of a variable

Example

```
var x=1;
if(!(x==0)){var y=2;x=x+y};
print x // prints 3
```

Definition of the dynamic semantics is OCaml

Values and functions handling the environment

```
type value = IntValue of int | BoolValue of bool | PairValue of value*value;;

lookup : variable -> dynamic_env -> value

dec : variable -> value -> dynamic_env -> dynamic_env

enter_scope : dynamic_env -> dynamic_env

update : variable -> value -> dynamic_env -> dynamic_env

exit_scope : dynamic_env -> dynamic_env
```

Semantic functions

```
evalExp : dynamic_env -> exp -> value

executeStmt : dynamic_env -> stmt -> dynamic_env

executeBlock : dynamic_env -> block -> dynamic_env

executeStmtSeq : dynamic_env -> stmt_seq -> dynamic_env

executeProg : prog -> unit
```