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 a *static environment*

a static environment defines the information associated with the variables available at a specific point in the program:

- the declared variables, in particular their names
- the types associated with them
- the scope of their declarations

Scope of a variable declaration in the static semantics

the scope of a variable declaration is the code fragment where the user can refer to it

Variable declarations

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/dynamic semantics

Possible approaches

How can the static/dynamic 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, concise, use of proof assistants
  - cons: based on complex concepts, not directly executable, not suitable for some details
- declarative programming language (functional or logic)
  - pros: abstract, non ambiguous, concise, executable
  - cons: requires knowledge of the used programming language

Definition of the static semantics

Our choice

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

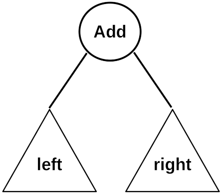
Typing rules

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

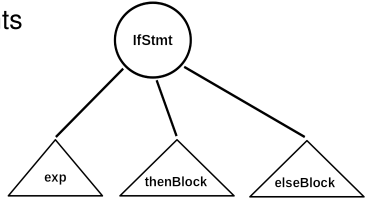
Typing rules

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

How can we model static environments?

abstractly, env_s is a partial function: $env_s : \mathit{Variable} \rightarrow \mathit{Type}$

- if env_s`(x)=IntType`, then `x` can be referred and has type `IntType`
- if env_s`(x)` is undefined, then `x` cannot be correctly referred

Static environment

A more operational view

- static environment = scope chain = list of scope levels (scopes for short)
- scope level = a dictionary (a.k.a. map) where keys are variables and values are types
- scope levels are sorted: inner scope levels come first
- reason: nested declarations hide variables declared in outer scopes
- Remark: the first scope in the list is the current scope level which is always the most nested one

Static environments: examples in F#

Technical details

- definition of static types

```
type staticType = | IntType | BoolType | PairType of staticType * staticType
```

- definition of static environments in F#:

```
type staticEnv = Map<variable, staticType> list
```

- example of scope with `x` of type **`int`** and `y` of type **`bool`**:

```
Map.add (Name "x") IntType Map.empty |> Map.add (Name "y") BoolType
```

where

```
Map.add: ('a -> 'b -> Map<'a,'b> -> Map<'a,'b>)
```

Static environments: examples in F#

Example of a static environment with two nested scopes

```
let topLevelScope = Map.add (Name "x") BoolType Map.empty

let currentAndNestedScope =
    Map.add (Name "x") IntType Map.empty |> Map.add (Name "y") BoolType

let env: staticEnv = [ currentAndNestedScope; topLevelScope ]
```

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

```
let topLevelScope =
    Map.add (Name "x") BoolType Map.empty |> Map.add (Name "z") IntType
let currentAndNestedScope =
    Map.add (Name "x") IntType Map.empty |> Map.add (Name "y") BoolType
let env: staticEnv = [ currentAndNestedScope; topLevelScope ]

assert (lookup (Name "x") env=IntType)
assert (lookup (Name "y") env=BoolType)
assert (lookup (Name "z") env=IntType)
```

`lookup (Name "w") env` **raises exception** `UndeclaredVariable (Name "w")`

Definition of the static semantics in F#

Generic functions for handling the environment

```
enterScope: 'a environment -> 'a environment

exitScope: 'a environment -> 'a environment

lookup: variable -> 'a environment -> 'a

dec: variable -> 'a -> 'a environment -> 'a environment

update: variable -> 'a -> 'a environment -> 'a environment
```

Definition of the static semantics in F#

Remarks

- functions are polymorphic to manage both the static and dynamic environment
- the type parameter `'a` corresponds to the information type associated with variables:
  - `'a = staticType` for static environments
  - `'a = value` for dynamic environments
- `dec` *x* *ty* *env* returns a new environment where the declaration of *x* with type *ty* is added in *env* at the current scope level
  an exception is raised if the variable has been already declared in the current scope level
- `update` and `exitScope` are used in the dynamic semantics (see the next slides)

Definition of the static semantics in F#

Main static semantic functions

```
typecheckProg: prog -> unit

typecheckStmtSeq: staticEnv -> stmtSeq -> unit

typecheckBlock: staticEnv -> block -> unit

typecheckStmt: staticEnv -> stmt -> staticEnv

typecheckExp: staticEnv -> exp -> staticType
```

Definition of the static semantics in F#

Remarks

- the static semantics is correct if and only if typechecking succeeds
- all functions raise an exception if typechecking fails
- `typecheckProg`, `typecheckStmtSeq`, and `typecheckBlock`
  do not return any value
  they simply check that the static semantics is correct
- `typecheckStmt` *env* *stmt* returns a new static environment env':
  - if *stmt* is **`var`** x=*exp* then env'=`dec` x *ty* *env*, where *ty* is the type of *exp* in *env*
  - $env' = env$ for all other types of statement
- `typecheckExp` *env* *exp* returns the static type of *exp* in *env*

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
  - ...

Remark: dynamic errors of several kinds can occur during the execution of the program

Definition of the dynamic semantics

Provided by executable F# 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: where the variable can be referred (similar definition as in the static semantics)
  - time: when the variable is allocated and de-allocated

Variable declarations

Example for the temporal dimension

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

- when the execution of the then-block starts, a new nested scope level is added to allocate variables `y` and `z` declared within the block (function `enterScope`)
- when the execution of the then-block finishes, the scope level of the block is removed to de-allocate variables `y` and `z` (function `exitScope`)
- Remark: the then-block can be executed more times; for instance, in case the **`if`** statement is contained in a loop

Dynamic environment

How can we model dynamic environments?

analogously as static environments
abstractly, env_d is a partial function: $env_d : Variable \to Value$

- if env_d`(x)=42`, then `x` can be referred and has value 42
- if env_d`(x)` is undefined, then `x` cannot be correctly referred

Dynamic environment

A more operational view

Analogous to that for static environments

- dynamic environment = scope chain = list of scope levels (scopes for short)
- scope level = a dictionary (a.k.a. map) where keys are variables and keys are associated with the values contained in the variables
- scope levels are sorted: inner scope levels come first
- reason: nested declarations hide variables declared in outer scopes
- Remark: the first scope in the list is the current scope level which is always the most nested one

Dynamic environment

Technical details

- values

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

- definition of dynamic environments in F#:

```
type dynamicEnv = Map<variable, value> list
```

Variable update

Example

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

the dynamic environment `env1` before the execution of `x=x+y` is

```
let topLevelScope1 = Map.add (Name "x") (IntValue 1) Map.empty
let thenBlockScope = Map.add (Name "y") (IntValue 2) Map.empty
let env1: dynamicEnv = [thenBlockScope ; topLevelScope1 ]
```

the dynamic environment `env2` after the execution of `x=x+y` is

```
let topLevelScope2 = Map.add (Name "x") (IntValue 3) Map.empty
let env2: dynamicEnv = [thenBlockScope ; topLevelScope2 ]
```

Variable update

Remarks

- `x=x+y` is executed when the current scope level is `thenBlockScope` but `topLevelScope1` is affected
- this cannot happen for the static semantics because
  - declarations at the current level cannot affect outer scope levels
  - the type associated with a variable cannot be changed

Variable update

Variable assignment

- assignment allows programmers to change the values stored in variables
- operations `update` and `exitScope` are used for dynamic environments

  the generic functions

  ```
  update: variable -> 'a -> 'a environment -> 'a environment
  exitScope: 'a environment -> 'a environment
  ```

  for dynamic environments become

  ```
  update : variable -> value -> dynamicEnv -> dynamicEnv
  exitScope: dynamicEnv -> dynamicEnv
  ```

- in the static semantics `update` and `exitScope` are not needed

Definition of the dynamic semantics is F#

Main dynamic semantic functions

```
executeProg: prog -> unit

executeStmtSeq: dynamicEnv -> stmtSeq -> dynamicEnv

executeBlock: dynamicEnv -> block -> dynamicEnv

executeStmt: dynamicEnv -> stmt -> dynamicEnv

evalExp : dynamicEnv -> exp -> value
```

Definition of the dynamic semantics is F#

Remark

Note the difference between

```
executeStmtSeq: dynamicEnv -> stmtSeq -> dynamicEnv
executeBlock: dynamicEnv -> block -> dynamicEnv
```

and

```
typecheckStmtSeq: staticEnv -> stmtSeq -> unit
typecheckBlock: staticEnv -> block -> unit
```

- `executeStmtSeq` and `executeBlock` must return a new dynamic environment because execution in nested levels can affect outer scope levels
- `typecheckStmtSeq` and `typecheckBlock` do not need to return a new static environment because typechecking in nested levels cannot affect outer scope levels