Programming Languages and Compilers

Specification of Dynamic Semantics

Dr. Peter Pfahler
Based on the lecture by Prof. Dr. Uwe Kastens

Universität Paderborn
Fakultät EIM
Institut für Informatik

Winter 2016/2017

Dynamic Semantics

The *Dynamic semantics* of a language describes the effect of executing a program. For that purpose the behaviour ot the various language constructs is specified.
Informal natural language specifications are commonly used:

> *Each variable has a storage cell, suitable to store values of the type of the variable. An assignment $v := e$ is executed by the following steps: determine the storage cell of the variable v, evaluate the expression e yielding a value x, and store x in the storage cell of v.*

Often the language specification does not explicitly state, what happens if an erroneous program construct is executed:

> *The execution of an input statement is undefined if the next value of the input is not a value of the type of the variable in the statement.*

Denotational Semantics

Denotational Semantics is a formal calculus for specification of dynamic semantics. It maps language constructs of the abstract syntax to functions, thus defining their effect. For a given structure tree the functions associated to the tree nodes are composed yielding a semantic function of the whole program - statically.

Denotational Semantics allow to

- prove dynamic properties of a program formally,
- reason about the function of the program - rather than about is operational execution,
- reason about dynamic properties of language constructs formally.

A denotational semantics specification of a programming language consists of

- the specification of semantic domains to model the program state
- a function E that maps all expression constructs on semantic functions
- a function C that maps all statement constructs on semantic functions

Semantic Domains

Semantic domains describe the domains and ranges of the semantic functions of a particular language. For an imperative language the central semantic domain describes the program state.

Semantic domains of a very simple imperative language

```
State = Memory X Input X Output   // Program State
Memory = Ident -> Value           // Memory
Input = Value*                    // Input Stream
Output = Value*                   // Output Stream
Value = Numeral | Bool            // Legal Values
```

Consequences for the language specified using these semantic domains:

- The language can allow only global variables, because a 1:1-mapping is assumed between identifiers and storage cells. In general the storage has to be modelled:

```
Env = Ident -> Location
Memory = Location -> Value
```

- Undefined values and an error state are not modelled; hence, behaviour in erroneous cases and exception handling can not be specified with these domains.

Mapping of Expressions

Let `Expr` be the set of all constructs of the abstract syntax that represent expressions, then the function `E` maps `Expr` to functions which describe expression evaluation:

```
E: Expr -> (State -> Value)
```

If additionally *side-effects* of expression evaluation must be modeled, the evaluation function has to return a potentially changed state:

```
E: Expr -> (State -> (State X Value))
```

The mapping `E` is defined by enumerating the cases of the abstract syntax in the form

```
E [abstract syntax construct] state = evaluation result
```

Example of a simple expression mapping

```
E [e1 + e2] s = (E [e1] s) + (E [e2] s)
E [e1 * e2] s = (E [e1] s) * (E [e2] s)
E [Number] s = Number
E [Ident] (m, i, o) = m Ident
```


Mapping of Statements

Let `Command` be the set of all constructs of the abstract syntax that represent statements, then the function `C` maps `Command` to functions which describe statement execution:

```
C: Command -> (State -> State)
```

The `C` function computes a state transition. To additionally model jumps and labels in statement execution an additional argument would be needed, which models the continuation after execution of the specified construct (*continuation semantics*).

The mapping `C` is defined by enumerating the cases of the abstract syntax in the form

```
C [abstract syntax construct] state = resulting state
```

Example of a simple statement mapping

```
C [stmt1; stmt2] s = C [stmt2] (C [stmt1] s)
C [v = e] (m, i, o) = (m [(E [e] (m, i, o)) / v], i, o) // change memory
C [if x then stmt1 else stmt2] s = E[x]s -> C [stmt1] s, C [stmt2] s
C [while x do stmt] s = E[x]s -> C [stmt; while x do stmt] s, s
C [print x] (m, i, o) = (m, i, E [x](m,i,o).o) // prepend value to output
```