Specification of Language-based Systems

Contents

Concepts:

- abstract syntax
- static semantics
- dynamic semantics
- denotational principle for semantics

Examples of specifications of language-based systems:

Language Description

A language is usually described by its

- syntax (*sentences* of the language)
- static semantics (which sentences are *wellformed*, e.g. wrt. scope rules)
- (dynamic) semantics (*meaning* of sentences)
- pragmatics (what is good *use* of the language)

Formal specification of syntax and semantics of programming/specification languages:

- programs/specifications are modelled as mathematical objects
- used in the design & development of
  - programming languages like Chill, Ada, ...
  - specification languages like VDM, RSL, Casl, Z, ...

In this course we will use RSL to specify language-based systems.

Concrete versus Abstract Syntax

concrete syntax

Concrete syntax $\sim$ strings of symbols.

We use grammars (e.g. BNF) to describe concrete syntax.

Bit ::= 0 | 1

abstract syntax

By an abstract syntax we understand rules for mathematically characterising the structure of sentences in terms of its decomposition. Hence, an abstract syntax abstracts from the symbols in the concrete syntax.

Abstract syntax $\sim$ parse trees.

We use RSL type definitions to define abstract syntax.

Bit == zero | one

Kinds of Semantics

- $\surd$ Denotational
- Mechanical/Computational/Operational
- *&c.*

Denotational Semantics — Principle

Every construct of the language must have its own denotation (meaning).

The semantics is described by functions $\mathcal{M}$ associating a denotation to each construct of the language. The denotation of a construct is defined in terms of the denotations of its subconstructs $c_1, c_2, \ldots, c_n$:

$$\mathcal{M}(S(c_1, c_2, \ldots, c_n)) = \mathcal{F}(\mathcal{M}(c_1), \mathcal{M}(c_2), \ldots, \mathcal{M}(c_n))$$

syntactic object $\xrightarrow[\mathcal{M}]{\text{denotes}}$ semantic object

1 + 2	3
value expr	value
+	addition
operator	function

RSL Specification of Language-based System

specification pattern

Abstract syntax:

type /* type of syntactic objects */
  Syn = ...

Static Semantics:

value /* is_wff(s) gives true when s is static correct */
  is_wff : Syn → **Bool**
  is_wff(s) ≡ ...

Dynamic Semantics:

type /* type of semantic objects */
  Sem = ...
value /* m(s) gives the denotation of s */
  m : Syn $\xrightarrow{\sim}$ Sem
  m(s) ≡ ...
  pre is_wff(s)

SAL1: A Simple Applicative Language

Concrete syntax examples:

5
x
e1 + e2, e1 − e2
− e1

Abstract Syntax

type
  Expr = Const | InfixExpr | PrefixExpr | Id,
  Const :: const : **Int**,
  InfixExpr :: l : Expr infixop : InfixOp r : Expr,
  PrefixExpr :: prefixop : PrefixOp expr : Expr,
  InfixOp == plus | minus,
  PrefixOp == minus,
  Id :: ide : Identifier,
  Identifier

Static semantics of SAL1

type /* dictionary of allowed identifiers */
  Dict = Identifier-**set**

value
  is_wff : Expr → Dict → **Bool**
  is_wff(e)(d) ≡
    case e **of**
      mk_Const(i) → **true**,
      mk_Id(id) → id ∈ d,
      mk_InfixExpr(e1, _, e2) →
        is_wff(e1)(d) ∧ is_wff(e2)(d),
      mk_PrefixExpr(_, e1) → is_wff(e1)(d)
    end

Dynamic Semantics of SAL1

type /* Environment */ Env = Identifier $\xrightarrow[m]{}$ **Int**

value

m : Expr $\rightarrow$ Env $\xrightarrow{\sim}$ **Int**

m(e)(ρ) $\equiv$

  case e **of**

    mk_Const(i) $\rightarrow$ i,

    mk_Id(id) $\rightarrow$ ρ(id),

    mk_InfixExpr(e1, op, e2) $\rightarrow$

      m(op)(m(e1)(ρ), m(e2)(ρ)),

    mk_PrefixExpr(op, e1) $\rightarrow$ m(op)(m(e1)(ρ))

  end

  pre is_wff(e)(**dom** ρ),

m : InfixOp $\rightarrow$ (**Int** $\times$ **Int** $\rightarrow$ **Int**)

m(op) $\equiv$

  case op **of**

    plus $\rightarrow$ λ (x, y) : **Int** $\times$ **Int** $\cdot$ x + y,

    minus $\rightarrow$ λ (x, y) : **Int** $\times$ **Int** $\cdot$ x $-$ y

  end,

m : PrefixOp $\rightarrow$ (**Int** $\rightarrow$ **Int**)

m(op) $\equiv$ (λ x : **Int** $\cdot$ 0 $-$ x)

Alternative formulation:

value

m : Expr $\rightarrow$ Env $\xrightarrow{\sim}$ **Int**

m(e)(ρ) $\equiv$

  case e **of**

    mk_Const(i) $\rightarrow$ i,

    mk_Id(id) $\rightarrow$ ρ(id),

    mk_InfixExpr(e1, op, e2) $\rightarrow$

      case op **of**

        plus $\rightarrow$ m(e1)(ρ) + m(e2)(ρ),

        minus $\rightarrow$ m(e1)(ρ) $-$ m(e2)(ρ)

      end,

    mk_PrefixExpr(= minus, e1) $\rightarrow$ 0 $-$ m(e1)(ρ)

  end

  pre is_wff(e)(**dom** ρ)

Example:

m(mk_InfixExpr(mk_Const(1), plus, mk_Const(2)))(ρ) $\equiv$

  m(plus)(m(mk_Const(1))(ρ),m(mk_Const(2))(ρ)) $\equiv$

  (λ (x,y) : **Int** $\times$ **Int** $\cdot$ x + y)(1,2) $\equiv$

  1 + 2 $\equiv$

  3

SAL2: A Simple Typed Applicative Language

Abstract Syntax (using unions and records)

type

Expr = Const | InfixExpr | PrefixExpr,

Const = IntConst | BoolConst,

IntConst :: **int** : **Int**,

BoolConst :: bool : **Bool**,

InfixExpr :: l : Expr infixop : InfixOp r : Expr,

PrefixExpr :: prefixop : PrefixOp ex : Expr,

InfixOp == plus | minus | and | or,

PrefixOp == minus | not

Alternative Abstract Syntax (using variants)

type

Expr ==

  mk_IntConst(i : **Int**) |

  mk_BoolConst(b : **Bool**) |

  mk_InfixExpr(l : Expr, infixop : InfixOp, r : Expr) |

  mk_PrefixExpr(prefixop : PrefixOp, ex : Expr) ,

InfixOp == plus | minus | and | or,

PrefixOp == minus | not

Static Semantics of SAL2

type Tp == integer | boolean

value
  type_of : Expr → Tp
  type_of(e) ≡
    case e **of**
      mk_IntConst(_) → integer,
      mk_BoolConst(_) → boolean,
      mk_InfixExpr(_, op, _) →
        case op **of**
          plus → integer,
          minus → integer,
          and → boolean,
          or → boolean
        end,
      mk_PrefixExpr(op, _) →
        case op **of**
          minus → integer,
          not → boolean
        end
    end,

  is_wff : Expr → **Bool**
  is_wff(e) ≡
    case e **of**
      mk_IntConst(_) → **true**,
      mk_BoolConst(_) → **true**,
      mk_InfixExpr(e1, op, e2) →
        is_wff(e1) ∧
        is_wff(e2) ∧
        let t1 = type_of(e1), t2 = type_of(e2) **in**
          case op **of**
            plus → (t1, t2) = (integer, integer),
            minus → (t1, t2) = (integer, integer),
            and → (t1, t2) = (boolean, boolean),
            or → (t1, t2) = (boolean, boolean)
          end
        end,
      mk_PrefixExpr(op, e1) →
        is_wff(e1) ∧
        let t = type_of(e1) **in**
          case op **of**
            minus → t = integer,
            not → t = boolean
          end
        end
    end

Dynamic Semantics of SAL2

Semantic domain and functions

type Val == mk_Int(intg : **Int**) | mk_Bool(bool : **Bool**)

value
  add : Val × Val $\xrightarrow{\sim}$ Val
  add(v1, v2) ≡ mk_Int(intg(v1) + intg(v2)),

  sub : Val × Val $\xrightarrow{\sim}$ Val
  sub(v1, v2) ≡ mk_Int(intg(v1) − intg(v2)),

  and : Val × Val $\xrightarrow{\sim}$ Val
  and(v1, v2) ≡ mk_Bool(bool(v1) ∧ bool(v2)),

  or : Val × Val $\xrightarrow{\sim}$ Val
  or(v1, v2) ≡ mk_Bool(bool(v1) ∨ bool(v2)),

  minus : Val $\xrightarrow{\sim}$ Val
  minus(v) ≡ mk_Int(0 − intg(v)),

  not : Val $\xrightarrow{\sim}$ Val
  not(v) ≡ mk_Bool(∼ bool(v))

Semantic meaning function

value
  m : Expr $\xrightarrow{\sim}$ Val
  m(e) ≡
    case e **of**
      mk_IntConst(i) → mk_Int(i),
      mk_BoolConst(b) → mk_Bool(b),
      mk_InfixExpr(e1, op, e2) →
        case op **of**
          plus → add(m(e1), m(e2)),
          minus → sub(m(e1), m(e2)),
          and → and(m(e1), m(e2)),
          or → or(m(e1), m(e2))
        end,
      mk_PrefixExpr(op, e1) →
        case op **of**
          minus → minus(m(e1)),
          not → not(m(e1))
        end
    end
    pre is_wff(e)

SIL1: A Simple Imperative Language

Concrete syntax examples

```
y := x + 1 ;
if b then y := 1 else y := 2 end;
x := 1 ; y := x
```

Abstract Syntax

```
type
  Prog == mk_Prog(stmt : Stmt),

  Stmt ==
    mk_Asgn(ide : Identifier,  expr : Expr) |
    mk_If(cond : Expr,  s1 : Stmt, s2 : Stmt) |
    mk_Seq(head : Stmt,  last : Stmt),

  Expr ==
    mk_Const(const : Int) |
    mk_Plus(fst : Expr,  snd : Expr) |
    mk_Id(ide : Identifier),

  Identifier
```


Dynamic Semantics of SIL1

Applicative Model

```
type /∗ storage for program variables ∗/
  Σ = Identifier -m-> Int

value
  m : Prog → Σ → Σ
  m(p)(σ)  ≡  m(stmt(p))(σ),

  m : Stmt → Σ → Σ
  m(s)(σ)  ≡
    case s of
      mk_Asgn(i, e) → σ † [i ↦ m(e)(σ)],
      mk_Seq(s1, s2) → m(s2)(m(s1)(σ)),
      mk_If(c, s1, s2) →
        if m(c)(σ) ≠ 0 then m(s1)(σ) else m(s2)(σ) end
    end,

  m : Expr → Σ → Int
  m(e)(σ)  ≡
    case e of
      mk_Const(n) → n,
      mk_Plus(e1, e2) → m(e1)(σ) + m(e2)(σ),
      mk_Id(id) → if id ∈ dom σ then σ(id) else 0 end
    end
```


Dynamic Semantics of SIL1

Imperative Model

```
type Σ = Identifier -m-> Int

variable σ : Σ

value
  m : Prog → write σ Unit
  m(p)  ≡  m(stmt(p)),

  m : Expr → read σ Int
  m(e)  ≡
    case e of
      mk_Const(n) → n,
      mk_Plus(e1, e2) → m(e1) + m(e2),
      mk_Id(id) → if id ∈ dom σ then σ(id) else 0 end
    end,

  m : Stmt → write σ Unit
  m(s)  ≡
    case s of
      mk_Asgn(i, e) → σ := σ † [i ↦ m(e)],
      mk_Seq(s1, s2) → m(s1) ; m(s2),
      mk_If(c, s1, s2) → if m(c) ≠ 0 then m(s1) else m(s2) end
    end,
```


SMALL: A Small Machine Language

SMALL programs specify computations on a simple machine consisting of a stack and a storage that stores the values of program variables.

Abstract syntax of SMALL

type
MProg = Inst*,
Inst ==
mk_Push(ide1 : Identifier) |
mk_Pop(**Unit**) |
mk_Add(**Unit**) |
mk_Cnst(val : **Int**) |
mk_Store(ide2 : Identifier) |
mk_Jumpfalse(off1 : **Int**) |
mk_Jump(off2 : **Int**)

An interpreter for SMALL instructions

type Σ = Identifier $\overrightarrow{m}$ **Int**, Stack = **Int***
value
I : MProg $\times$ **Int** $\times$ Stack $\to$ ($\Sigma \to \Sigma$)
I(mp, pc, s)(σ) $\equiv$
if pc $\leq$ 0 $\vee$ pc > **len** mp **then** σ **else**
case mp(pc) **of**
mk_Push(x) $\to$ **if** x $\in$ **dom** σ
then I(mp, pc + 1, $\langle\sigma$(x)$\rangle$ ^ s)(σ)
else I(mp, pc + 1, $\langle 0 \rangle$ ^ s)(σ) **end**,
mk_Pop(()) $\to$ **if len** s = 0 **then** σ
else I(mp, pc + 1, **tl** s)(σ) **end**,
mk_Cnst(n) $\to$ I(mp, pc + 1, $\langle$n$\rangle$ ^ s)(σ),
mk_Add(()) $\to$ **if len** s < 2 **then** σ
else I(mp, pc + 1,$\langle$s(1) + s(2)$\rangle$ ^ **tl tl** s)(σ) **end**,
mk_Store(x) $\to$ **if len** s = 0 **then** σ
else I(mp, pc + 1, s)(σ † [x $\mapsto$ s(1)]) **end**,
mk_Jumpfalse(n) $\to$ **if len** s = 0 **then** σ
elsif hd s $\neq$ 0 **then** I(mp, pc + 1, s)(σ)
else I(mp, pc + n, s)(σ) **end**,
mk_Jump(n) $\to$ I(mp, pc + n, s)(σ)
end
end

The interpretation function I operates on a program, mp, and carries with it an instruction counter pc, the stack s and storage σ.

A Compiler for SIL1

value
comp_Prog : Prog $\to$ MProg
comp_Prog(p) $\equiv$ comp_Stmt(stmt(p)),

comp_Stmt : Stmt $\to$ MProg
comp_Stmt(s) $\equiv$
case s **of**
mk_Asgn(id, e) $\to$ comp_Expr(e) ^ $\langle$ mk_Store(id), mk_Pop() $\rangle$,
mk_Seq(s1, s2) $\to$ comp_Stmt(s1) ^ comp_Stmt(s2),
mk_If(e, s1, s2) $\to$
let
ce = comp_Expr(e),
cs1 = comp_Stmt(s1), cs2 = comp_Stmt(s2)
in
ce ^
$\langle$ mk_Jumpfalse(**len** cs1 + 3) $\rangle$ ^
$\langle$ mk_Pop() $\rangle$ ^
cs1 ^
$\langle$ mk_Jump(**len** cs2 + 2) $\rangle$ ^
$\langle$ mk_Pop() $\rangle$ ^
cs2
end
end,

comp_Expr : Expr $\to$ MProg
comp_Expr(e) $\equiv$
case e **of**
mk_Const(n) $\to$ $\langle$ mk_Cnst(n) $\rangle$,
mk_Plus(e1, e2) $\to$
comp_Expr(e1) ^ comp_Expr(e2) ^ $\langle$ mk_Add() $\rangle$,
mk_Id(id) $\to$ $\langle$ mk_Push(id) $\rangle$
end

Compiler correctness

$\forall$ p : Prog, σ : Σ •
I(comp_Prog(p), 1, $\langle\rangle$)(σ) = m(p)(σ)

Exercise

Consider the following SIL1 program:

x := 2; **if** x **then** y := x + 3 **else** y := y+1 **end**

1. What is the corresponding abstract syntax value *p*?
2. What is the result *(comp_Prog(p))* of compiling the SIL1 program *p*?
3. Calculate *I(comp_Prog(p), 1, <>)*([$x \mapsto 2$, $y \mapsto 1$]).
4. Calculate *m(p)*([$x \mapsto 2$, $y \mapsto 1$]).