Value Definitions & Functions

Contents

Value Definitions

Value Declarations — Syntax

value
  $\text{value\_definition}_1$
  ⋮
  $\text{value\_definition}_n$

Value Definitions

- different forms:
  - axiomatic: using signatures (i.e. typings) and axioms
  - explicit definition of constants
  - implicit definition of constants
  - explicit definition of functions
  - implicit definition of functions
- under specification
- four last forms can be transformed to first form
- axiomatic versus operational interpretation 12

Definition of constants — Examples

- explicit

  value $x : \textbf{Int} = 1$

- implicit

  value $x : \textbf{Int} \cdot x > 0$

- axiomatic

  value $x : \textbf{Int}$
  axiom $x > 0$

Definition of functions – Examples

- explicit

  value
  $f : \textbf{Int} \to \textbf{Int}$
  $f(x) \equiv x + 1$

- implicit

  value
  $f : \textbf{Int} \to \textbf{Int}$
  $f(x)$ **as** r **post** $r > x$

- axiomatic

  value $f : \textbf{Int} \to \textbf{Int}$
  axiom $\forall\, x : \textbf{Int} \cdot f(x) > x$

Underspecification

$\mathbf{value}\ \ x : \mathbf{Int}$
$\mathbf{axiom}\ x > 0$

x is underspecified, may be refined to:

$\mathbf{value}\ \ x : \mathbf{Int} = 1$

or

$\mathbf{value}\ \ x : \mathbf{Int} = 2$

or ...

Expansion of Explicit Value Definition

$\mathbf{value}\ \ x : \mathbf{Int} = 1$

equivalent to

$\mathbf{value}\ \ x : \mathbf{Int}$
$\mathbf{axiom}\ x \equiv 1$

Generally:

$\mathbf{value}\ \mathrm{binding} : \mathrm{type\_expr} = \mathrm{value\_expr}$

equivalent to

$\mathbf{value}\ \mathrm{binding} : \mathrm{type\_expr}$
$\mathbf{axiom}\ \mathrm{binding} \equiv \mathrm{value\_expr}$

Expansion of Implicit Value Definitions

$\mathbf{value}\ \ x : \mathbf{Int} \cdot x > 0$

equivalent to

$\mathbf{value}\ \ x : \mathbf{Int}$
$\mathbf{axiom}\ \ x > 0$

Generally:

$\mathbf{value}\ \mathrm{binding} : \mathrm{type\_expr} \cdot \mathrm{value\_expr}$

equivalent to

$\mathbf{value}\ \mathrm{binding} : \mathrm{type\_expr}$
$\mathbf{axiom}\ \mathrm{value\_expr}$

Expansion of Explicit Function Definitions

$\mathbf{value}$
$\quad f : \mathbf{Int} \to \mathbf{Int}$
$\quad f(x) \equiv x + 1$

equivalent to

$\mathbf{value}$
$\quad f : \mathbf{Int} \to \mathbf{Int}$
$\mathbf{axiom}\ \forall\ x : \mathbf{Int} \cdot f(x) \equiv x + 1$

Generally:

$\mathbf{value}$
$\quad \mathrm{id} : \mathrm{type\_expr}_1 \to \mathrm{type\_expr}_2$
$\quad \mathrm{id}(x) \equiv \mathrm{value\_expr}$

equivalent to

$\mathbf{value}$
$\quad \mathrm{id} : \mathrm{type\_expr}_1 \to \mathrm{type\_expr}_2$
$\mathbf{axiom}$
$\quad \forall\ x : \mathrm{type\_expr}_1 \cdot \mathrm{id}(x) \equiv \mathrm{value\_expr}$

Expansion of Implicit Function Definitions

value
  f : **Int** → **Int**, f(x) **as** r **post** r > x

equivalent to

value f : **Int** → **Int**
axiom ∀ x : **Int** • f(x) **as** r **post** r > x

Generally:

value
  id : type_expr1 → type_expr2
  id(x) **as** y **post** value_expr

equivalent to

value id : type_expr1 → type_expr2
axiom ∀ x : type_expr1 • id(x) **as** y **post** value_expr

Example of Expansion

scheme SET_DATABASE =
  class
    type
      Database = Person-**set**,
      Person = **Text**

    value
      empty : Database = {},

      register : Person × Database → Database
      register(p,db) ≡ db ∪ {p},

      check : Person × Database → **Bool**
      check(p,db) ≡ p ∈ db
  end

expands to

scheme SET_DATABASE =
  class
    type
      Database = Person-**set**,
      Person = **Text**
    value
      empty : Database,
      register : Person × Database → Database,
      check : Person × Database → **Bool**,
    axiom
      empty ≡ {},
      ∀ p : Person, db : Database •
        register(p, db) ≡ db ∪ {p},
      ∀ p : Person, db : Database •
        check(p, db) ≡ p ∈ db
  end

Axiomatic versus Operational Interpretation

Two different ways to of interpreting function definitions in languages:

- operational (as in programming languages)
- axiomatic (as in *some* specification languages, e.g. RSL)

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Operational versus Axiomatic Interpretation (I)

value
f: **Int** → **Int**
f(i) ≡ i+1

Operational interpretation:

f(5) is computed by

1. substituting 5 for i in the body: i+1
2. evaluating the resulting expression: 5+1 to get 6

Axiomatic interpretation:

The function is described by two properties:

1. its type: it must map integer arguments to integer results
2. an equivalence: f(i) must be semantically equivalent to i+1 for all integers i

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

RSL has axiomatic interpretation

value
f: **Int** → **Int**
f(i) ≡ i+1

is just a short form of:

value
f: **Int** → **Int**
axiom
∀ i:**Int** • f(i) ≡ i+1

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Operational versus Axiomatic Interpretation (II)

value
f: **Int** $\xrightarrow{\sim}$ **Int**
f(i) ≡ f(i)

Axiomatic interpretation:

all partial functions from **Int** to **Int**.

Operational interpretation:

only the infinite loop (corresponding to **chaos**)

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Operational versus Axiomatic Interpretation (III)

value
f: **Int** → **Int**
f(i) ≡ f(i)

The axiomatic interpretation requires f to be an arbitrary *total* function from **Int** to **Int** and the operational interpretation does *not* fulfill this.

Functions

Contents

Partial and Total Functions

Functions are values

Functions are first class values!

Hence, RSL allows:

- *higher-order* functions that take functions as argument and/or return functions.
- quantification over function types

Function Type Expressions

- total functions

  $\mathrm{type\_expr}_1 \rightarrow \mathrm{type\_expr}_2$

- partial functions

  $\mathrm{type\_expr}_1 \stackrel{\sim}{\rightarrow} \mathrm{type\_expr}_2$

$\forall\, \mathrm{f_{tot}} : \mathrm{T}_1 \rightarrow \mathrm{T}_2,\ \mathrm{f_{par}} : \mathrm{T}_1 \stackrel{\sim}{\rightarrow} \mathrm{T}_2,\ \mathrm{x} : \mathrm{T}_1 \bullet$

	defined (not **chaos**)	deterministic
$\mathrm{f_{tot}}(\mathrm{x})$	yes	yes
$\mathrm{f_{par}}(\mathrm{x})$	might be	might be

$\exists!\ \mathrm{y} : \mathrm{T}_2 \bullet \mathrm{f_{tot}}(\mathrm{x}) \equiv \mathrm{y}$

Function Type Expressions

Examples

$\mathbf{Bool} \rightarrow \mathbf{Bool}$

denote the type consisting of the following totale functions:

$\lambda\, \mathrm{b} : \mathbf{Bool} \bullet \mathbf{true}$
$\lambda\, \mathrm{b} : \mathbf{Bool} \bullet \mathbf{false}$
$\lambda\, \mathrm{b} : \mathbf{Bool} \bullet \mathrm{b}$
$\lambda\, \mathrm{b} : \mathbf{Bool} \bullet \sim\mathrm{b}$

$\mathbf{Bool} \stackrel{\sim}{\rightarrow} \mathbf{Bool}$

denote the type consisting of the following partial functions:

$\lambda\, \mathrm{b} : \mathbf{Bool} \bullet \mathbf{true}$
$\lambda\, \mathrm{b} : \mathbf{Bool} \bullet \mathbf{false}$
$\lambda\, \mathrm{b} : \mathbf{Bool} \bullet \mathrm{b}$
$\lambda\, \mathrm{b} : \mathbf{Bool} \bullet \sim\mathrm{b}$
$\lambda\, \mathrm{b} : \mathbf{Bool} \bullet \mathbf{chaos}$
$\lambda\, \mathrm{b} : \mathbf{Bool} \bullet \mathbf{true} \lceil\rceil \mathbf{false}$
...

Function Value Expressions

- names of user-defined function, e.g.:
  distance
- lambda abstractions (anonymous functions), e.g.:
  λ b:**Bool** • ∼b

Lambda Abstraction

Basic form:

λ binding : type_expr • value_expr

Examples:

λ b : **Bool** • ∼b
λ (x,y) : **Int** × **Int** • x + y
λ (b,(x,y)) : **Bool** × (**Nat** × **Nat**) •
  if b **then** x **else** y **end**

Semantics:

represents function of type: type_expr $\stackrel{\sim}{\to}$ T,
where T = type_of(value_expr)

Derived form:

λ (typing$_1$,…,typing$_n$) • value_expr, n $\geq$ 0

Examples:

λ (x : **Int**, y : **Int**) • x + y,
λ (b : **Bool**, x,y : **Nat**) • **if** b **then** x **else** y **end**

Function Application Expressions

Examples:

distance((1.0,2.3), (5.13, 5.13))
(λ(x,y) : **Nat** × **Nat** • x = y)(2,7)

Typical form:

function-expr(expr$_1$, … , expr$_n$), n $\geq$ 0

Context conditions:

(expr$_1$, … , expr$_n$)

must be of the argument type of expr

Associated Built-in Operators

$=$, $\neq$, $\circ$

$\circ$: $(T_2 \stackrel{\sim}{\to} T_3) \times (T_1 \stackrel{\sim}{\to} T_2) \to (T_1 \stackrel{\sim}{\to} T_3)$
f $\circ$ g $\equiv$ λ x : T_1 • f(g(x))

Higher Order Functions

Example

3 ways of defining twice:

value
twice : $(\mathbf{Int} \xrightarrow{\sim} \mathbf{Int}) \to \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$
$\mathrm{twice}(f) \equiv f \circ f$

value
twice : $(\mathbf{Int} \xrightarrow{\sim} \mathbf{Int}) \to \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$
$\mathrm{twice}(f) \equiv \lambda\ i : \mathbf{Int} \cdot f(f(i))$

value
twice : $(\mathbf{Int} \xrightarrow{\sim} \mathbf{Int}) \to \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$
$\mathrm{twice}(f)(i) \equiv f(f(i))$

Applications of twice:

$\mathrm{twice}(\lambda\ i : \mathbf{Int} \cdot i + 1) \equiv \lambda\ i : \mathbf{Int} \cdot i + 2$
$\mathrm{twice}(\lambda\ i : \mathbf{Int} \cdot i + 1)(1) \equiv 3$