Value Definitions & Functions

Contents

Value Definitions

Value Declarations — Syntax

value
value_definition$_1$
⋮
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 : **Int** = 1

- implicit

  value x : **Int** · x > 0

- axiomatic

  value x : **Int**
  axiom x > 0

Definition of functions – Examples

- explicit

  value
  f : **Int** → **Int**
  f(x) ≡ x + 1

- implicit

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

- axiomatic

  value f : **Int** → **Int**
  axiom ∀ x : **Int** · f(x) > x

Underspecification

value x : **Int**
axiom x > 0

x is underspecified, may be refined to:

value x : **Int** = 1

or

value x : **Int** = 2

or ...

Expansion of Explicit Value Definition

value x : **Int** = 1

equivalent to

value x : **Int**
axiom x ≡ 1

Generally:

value binding : type_expr = value_expr

equivalent to

value binding : type_expr
axiom binding ≡ value_expr

Expansion of Implicit Value Definitions

value x : **Int** • x > 0

equivalent to

value x : **Int**
axiom x > 0

Generally:

value binding : type_expr • value_expr

equivalent to

value binding : type_expr
axiom value_expr

Expansion of Explicit Function Definitions

value
f : **Int** → **Int**
f(x) ≡ x + 1

equivalent to

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

Generally:

value
id : $type\_expr_1$ → $type\_expr_2$
id(x) ≡ value_expr

equivalent to

value
id : $type\_expr_1$ → $type\_expr_2$
axiom
∀ x : $type\_expr_1$ • id(x) ≡ 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)

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

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

Operational versus Axiomatic Interpretation (II)

value
f: **Int** $\tilde{\to}$ **Int**
f(i) ≡ f(i)

Axiomatic interpretation:

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

Operational interpretation:

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

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

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

- partial functions

  $\text{type\_expr}_1 \xrightarrow{\sim} \text{type\_expr}_2$

$\forall\ f_{tot} : T_1 \rightarrow T_2,\ f_{par} :\ T_1 \xrightarrow{\sim} T_2,\ x : T_1 \cdot$

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

$\exists!\ y : T_2 \cdot f_{tot}(x) \equiv y$

Function Type Expressions
Examples

Bool $\rightarrow$ **Bool**

denote the type consisting of the following totale functions:

λ b : **Bool** $\cdot$ **true**
λ b : **Bool** $\cdot$ **false**
λ b : **Bool** $\cdot$ b
λ b : **Bool** $\cdot \sim$b

Bool $\xrightarrow{\sim}$ **Bool**

denote the type consisting of the following partial functions:

λ b : **Bool** $\cdot$ **true**
λ b : **Bool** $\cdot$ **false**
λ b : **Bool** $\cdot$ b
λ b : **Bool** $\cdot \sim$b
λ b : **Bool** $\cdot$ **chaos**
λ b : **Bool** $\cdot$ **true** $\sqcap$ **false**
...

Function Value Expressions

- names of user-defined function, e.g.:
  distance
- lambda abstractions (anonymous functions), e.g.:
  λ b:**Bool** $\cdot \sim$b

Lambda Abstraction

Basic form:

λ binding : type_expr $\cdot$ value_expr

Examples:

λ b : **Bool** $\cdot \sim$b
λ (x,y) : **Int** $\times$ **Int** $\cdot$ x + y
λ (b,(x,y)) : **Bool** $\times$ (**Nat** $\times$ **Nat**) $\cdot$
if b **then** x **else** y **end**

Semantics:

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

Derived form:

λ (typing$_1$,...,typing$_n$) $\cdot$ value_expr, n $\geq$ 0

Examples:

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

Function Application Expressions

Examples:

distance((1.0,2.3), (5.13, 5.13))
(λ(x,y) : **Nat** $\times$ **Nat** $\cdot$ 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$_2$ $\xrightarrow{\sim}$ T$_3$) $\times$ (T$_1$ $\xrightarrow{\sim}$ T$_2$) $\rightarrow$ (T$_1$ $\xrightarrow{\sim}$ T$_3$)
f $^\circ$ g $\equiv$ λ x : T$_1$ $\cdot$ f(g(x))

Higher Order Functions

Example

3 ways of defining twice:

value
  twice : (**Int** $\xrightarrow{\sim}$ **Int**) $\rightarrow$ **Int** $\xrightarrow{\sim}$ **Int**
  twice(f) $\equiv$ f $\circ$ f

value
  twice : (**Int** $\xrightarrow{\sim}$ **Int**) $\rightarrow$ **Int** $\xrightarrow{\sim}$ **Int**
  twice(f) $\equiv$ λ i : **Int** $\cdot$ f(f(i))

value
  twice : (**Int** $\xrightarrow{\sim}$ **Int**) $\rightarrow$ **Int** $\xrightarrow{\sim}$ **Int**
  twice(f)(i) $\equiv$ f(f(i))

Applications of twice:

twice(λ i : **Int** $\cdot$ i + 1) $\equiv$ λ i : **Int** $\cdot$ i + 2
twice(λ i : **Int** $\cdot$ i + 1)(1) $\equiv$ 3