Métodos Formales y Análisis de Herramientas para la Producción de Software

Aristides Dasso, Ana Funes

{arisdas, afunes}@unsl.edu.ar
Universidad Nacional de San Luis

Universidad Nacional de San Luis
2006

RSL

Raise
Specification
Language

RSL es Formal

- Sintaxis precisa
- Reglas de buena formación (typing, ámbitos)
- Semántica matemática, formal
- Lógica subyacente (permite razonar sobre propiedades)

RSL es de Amplio espectro

- Puede ser aplicado a distintos niveles de abstracción.
- Incluye distintos estilos:
  - orientado
    - al modelo
    - a la propiedad
  - secuencial / concurrente
  - aplicativo / imperativo

Requerimientos, ejemplo

Base de datos que soporta la administración de un registro electoral.

Soporta las siguientes funciones:

- Registrar una persona en la BD
- Chequear si una persona está en la BD

Abstracción

- "Qué" más bien que "Cómo"

RSL permite:

- Abstracción de datos
- Abstracción de operaciones

Ejemplos de Abstracción de Datos

type Database // abstracto, sort

type Database = Person-**set** // concreto

type Database = Person* // concreto, list

type Person // abstracto, sort

type Person = **Text** // concreto

type Person = **Nat** // concreto

Ejemplo de abstracción de operación

value
-- abstracta
sqrt: **Real** $\xrightarrow{\sim}$ **Real**
sqrt(x) **as** r
post r * r = x $\wedge$ r $\geq$ 0.0
pre x $\geq$ 0.0

value
-- concreta
sqrt: **Real** $\xrightarrow{\sim}$ **Real**
sqrt(x) $\equiv$ x $\uparrow$ 0.5
pre x $\geq$ 0.0

Especificación orientada a la propiedad

scheme ABS_DATABASE =
class
type
Database,
Person
value
empty : Database,
register : Person × Database → Database,
check : Person × Database → **Bool**
axiom
∀ p : Person ·
check(p, empty) ≡ **false**,
∀ p,p′ : Person, db : Database ·
check(p′, register(p, db)) ≡ p′ = p ∨ check(p′, db)
end

Ejemplo de Especificación RSL (orientada al modelo)

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

value
empty : Database = {},

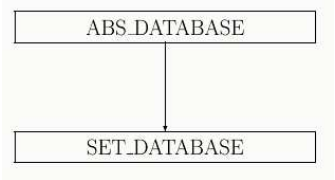
register : Person × Database → Database
register(p,db) ≡ db ∪ {p},

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

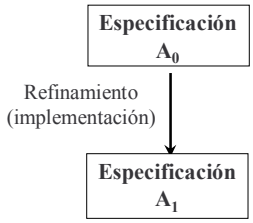
Preguntas

- ¿Necesitamos otras funciones?
- ¿Son las que necesitamos?
- ¿Son las definiciones correctas?
- ¿Sirven para registrar los alumnos de la clase?
- ¿Sirven para registrar los habitantes del país?
- ¿**Text** es un buen modelo para Person?

Un paso de desarrollo



Refinement or Implementation Relation



Relación de Refinamiento, condiciones

- El nuevo modelo (A_1) debe incluir TODAS las entidades del viejo (A_0)
  - types, values, objects, variables y channels con los mismos nombres y tipos maximales.
- A_1 puede tener más entidades que A_0.
  (Implementación estática, garantiza monotonicidad).
- Puede ser chequeada estáticamente.

Relación de Refinamiento, condiciones

- Todas las propiedades de A_0 se mantienen en A_1.
  - axiomas, defs. de funciones, constantes, restricciones en subtipos.
- No puede ser chequeada estáticamente, necesita demostración (Recordar que RAISE es riguroso.)

Especificación orientada a la propiedad

```
scheme ABS_DATABASE =
  class
    type
      Database,
      Person
    value
      empty : Database,
      register : Person × Database → Database,
      check : Person × Database → Bool
    axiom
      ∀ p : Person ·
        check(p, empty) ≡ false,
      ∀ p,p′ : Person, db : Database ·
        check(p′, register(p, db)) ≡ p′ = p ∨ check(p′, db)
  end
```


Ejemplo de Especificación RSL (orientada al modelo)

```
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
```


Condiciones de Refinamiento

$$\lfloor \forall\, p : \text{Person} \cdot \text{check}(p, \text{empty}) \equiv \text{false} \rfloor$$

$$\lfloor \forall\, p,p' : \text{Person}, db : \text{Database} \cdot \text{check}(p', \text{register}(p, db)) \equiv p' = p \vee \text{check}(p', db) \rfloor$$

Verificación

⌊check(p′, register(p, db)) ≡ p′ = p ∨ check(p′, db)⌋
unfold register:
⌊check(p′, db ∪ {p}) ≡ p′ = p ∨ check(p′, db)⌋
unfold check:
⌊p′ ∈ (db ∪ {p}) ≡ p′ = p ∨ check(p′, db)⌋
unfold check:
⌊p′ ∈ (db ∪ {p}) ≡ p′ = p ∨ p′ ∈ db⌋
isin_union:
⌊p′ ∈ db ∨ p′ ∈ {p} ≡ p′ = p ∨ p′ ∈ db⌋
isin_singleton:
⌊p′ ∈ db ∨ p′ = p ≡ p′ = p ∨ p′ ∈ db⌋
or_commutativity:
⌊p′ = p ∨ p′ ∈ db ≡ p′ = p ∨ p′ ∈ db⌋
is_annihilation:
⌊**true**⌋
qed

Desarrollo Alternativo

- La segunda implementación podría haber usado otros tipos:
- Lists
- Maps
- Etc.

Resumen (1)

RAISE
- Método
  - Refinamiento por pasos sucesivos
  - "Invent and Verify"
  - Riguroso
- Lenguaje de especificación (RSL)
  - Formal
  - Amplio espectro
  - Con facilidades de estructuración
- Herramientas

Resumen (2)

RSL soporta diversos estilos:
- Aplicativo
- Imperativo
- Concurrente

Resumen (3)

- La mayor parte del curso usará el estilo aplicativo, el cual es cercano a la programación funcional y a la matemática.
- El estilo imperativo es similar a la programación imperativa tradicional, con variables, asignaciones, secuencias, loops, etc.
- El estilo concurrente soporta la especificación de aspectos concurrentes.

Aspectos Sintácticos de RSL

Especificación RSL

Una especificación RSL consiste de:

- definiciones de módulos relacionados

Un módulo puede contener definiciones de:

- Tipos (type)
- Valores (value)
- Axiomas (axiom) No hay orden
- Objetos (object)
- Variables (variable)
- Canales (channel)

Módulos

- Tipos de módulos:
  - Esquemas (**scheme**): clase (de modelo)
  - Objetos (**object**): instancia de una clase (de modelo).
- Expresión de clase:
  - Básica
  - Extendida
  - Renombrada
  - Con **hide**
  - Con **with**
  - Instanciada

Especificación de una Base de Datos para una Elección

Requerimientos:

Adminstrar el padrón electoral. Para ello se deberá contar con las operaciones:

- *register:* para registrar una persona en la base de datos.
- *check:* para verificar si una persona se encuentra ya registrada o no.
- *number*: contar el número de personas que se encuentran registradas.

Especificación Formal

```
scheme DATABASE =
  class
    type Person, Database = Person-set

    value
      empty : Database,
      register : Person × Database → Database,
      check : Person × Database → Bool,
      number : Database → Nat

    axiom
      empty ≡ {},
      ∀ p : Person, db : Database • register(p, db) ≡ {p} ∪ db,
      ∀ p : Person, db : Database • check(p, db) ≡ p ∈ db,
      ∀ db : Database • number(db) ≡ card db
  end
```


Expresión Básica de Clase

class
  declaración$_1$
  ⋮
  declaración$_n$
end

- $n \geq 0$

Cada *declaración* puede ser:

- declaración de **type**
- declaración de **value**
- declaración de **axiom**
- declaración de **object**
- declaración de **variable**
- declaración de **channel**

Declaraciones de Tipos

- Un tipo es una colección de valores lógicamente relacionados.
- Pueden ser:
  - Built-in (o predefinidos, p.e. **Nat**, **Real**, ...)
  - Definidos por el usuario (sorts, usando constrs.)

Declaraciones de Tipos

- Una declaración de tipos tiene la forma:

type

definición_de_tipo$_1$,

⋮

definición_de_tipo$_n$

$n \geq 1$

Declaraciones de Tipos

Una *definición_de_tipo* puede ser:

(1) *id = expresión_de_tipo*

(2) *id*

(3) record, variant, union

(1) id es una abreviatura para el tipo definido después de '='. p.e. Database = Person-**set**.

(2) Para tipos de datos abstractos: tipo que no tiene operadores predefinidos, excepto igualdad (=) y desigualdad (≠). p.e. Person

(3) Definición de tipos record, variant o union.

Declaraciones de Values

- Una declaración de values tiene la forma:

value

definición_de_value$_1$,

⋮

definición_de_value$_n$

$n \geq 1$

Declaraciones de Values

Una *definición_de_value* tiene la forma:

id: expresión_de_tipo

Ejemplos:

value

empty: Database, -- constante

register: Person × Database → Database -- función

Declaraciones de axiomas

- Los axiomas expresan propiedades sobre los values.

axiom

value_exp$_1$,

...

value_exp$_n$

- Las *values_exp* son expresiones booleanas que por definición evaluan siempre a true

Declaraciones de axiomas

Ejemplo

/* axioma que expresa que la constante empty es equivalente al conjunto vacío.*/

axiom

empty ≡ {}

Definiciones de Values

- Diferentes formas para definir constantes y funciones
  - Axiomática (Signatura + axioma)
  - Explicita
  - Implícita
- Las dos últimas pueden ser transformadas a la primera.

Declaraciones de valores constantes

- Definición axiomática: Signatura + axioma(s)

value x: **Int**	**value** x: **Int**
axiom x ≡ 1	**axiom** x > 0

- Definición explícita
  value x: **Int** = 1
- Definición implícita

value x: **Int** • x = 1	**value** x: **Int** • x > 0

Definición de valores de función

- Definición axiomática: Signatura + axioma(s)

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

- Definición de función explícita
  value
  f: **Int** → **Int**
  f(x) ≡ x + 1
- Definición de función implícita
  value
  f: **Int** → **Int**
  f(x) **as** r **post** r > x

Tipos Predefinidos

Type	Example values	Operators
Bool	**true**, **false**	∧, ∨, ⇒ ∼
Int	..., -1, 0, 1, ...	+, −, ∗, /, \, ↑, <, ≤, >, ≥, **abs**, **real**
Nat	0, 1, ...	Same as for **Int**
Real	..., -4.3, . . . , 0.0, . . .	+, −, ∗, /, ↑, <, ≤, >, ≥, **abs**, **int**
Char	'a', . . .	
Text	"", "Alice", ...	As for lists of **Char**
Unit	()	

Tipos Compuestos

- Productos (×)
- Funciones ($\rightarrow$, $\xrightarrow{\sim}$)
- Conjuntos (**-set**, **-infset**)
- Listas (*, $^\omega$)
- Mapas ($\underset{m}{\rightarrow}$, $\underset{m}{\xrightarrow{\sim}}$)

El tipo **Bool**

- El literal **Bool** denota el tipo que contiene los valores **true** y **false**.
- Expresiones **if**:
  if *expr* **then** *expr1* **else** *expr2* **end**
  Ejemplo:
  value
  f: **Int** × **Int** → **Int**
  f(x, y) ≡ **if** x > y **then** x – y **else** y – x **end**

chaos

¿Qué pasa con las expresiones que no terminan? p.e.:

value
  factorial_pobre: **Int** →**Int**
  factorial_pobre(x) ≡
    if x = 0 **then** 1 **else** x * factorial_pobre(x – 1) **end**

¿A qué evalúa factorial_pobre(-2)?

En RSL el comportamiento no terminante, o caótico es representado por la expresión **chaos**.

chaos

Expresión	Valor
true	**true**
1+2	3
factorial_pobre(-1)	**?**
15/0	**?**

- **chaos** no es miembro de ningún tipo.
- Puede aparecer donde expresiones de distintos tipos son esperadas.

Propiedades de la expresión **if**

if true then *expr1* **else** *expr2* **end** ≡ *expr1*
if false then *expr1* **else** *expr2* **end** ≡ *expr2*

- Evaluación no estricta: Una importante propiedad de las expresiones **if** es que cuando aplicadas a **chaos** no necesariamente equivalen a **chaos**:

if true then *expr* **else chaos end** ≡ *expr*
if false then chaos else *expr* **end** ≡ *expr*

Propiedades de la expresión **if**

- Sin embargo, si la condición del **if** es equivalente a **chaos**, entonces toda la expresión equivale a **chaos**:

if chaos then *expr1* **else** *expr2* **end** ≡ **chaos**

Lógica Condicional

~ *expr* ≡ **if** *expr* **then false else true end**
$expr_1 \wedge expr_2$ ≡ **if** $expr_1$ **then** $expr_2$ **else false end**
$expr_1 \vee expr_2$ ≡ **if** $expr_1$ **then true else** $expr_2$ **end**
$expr_1 \Rightarrow expr_2$ ≡ **if** $expr_1$ **then** $expr_2$ **else true end**

- Se debe tener en cuenta el orden de evaluación cuando se usan las conectivas infijo.
- Es condicional porque la 2da. expr. es evaluada sólo si el valor de la 1ra. no es suficiente para determinar el valor de la expresión completa.

Lógica Condicional

p: **Real** →**Bool**
p(x) ≡ (x ≠ 0) ∧ (1.0 / x < epsilon)

¿a qué evalúa p(0)?

Lógica Condicional

p(0) ≡
(0 ≠ 0) ∧ (1.0 / 0 < epsilon) ≡
if (0 ≠ 0) **then** (1.0 / 0.0 < epsilon) **else false end** ≡
if false then (1.0 / 0.0 < epsilon) **else false end** ≡
false

Lógica Condicional

El significado de las conectivas lógicas es resumido en las siguientes tablas:

∧	true	false	chaos
true	**true**	**false**	**chaos**
false	**false**	**false**	**false**
chaos	**chaos**	**chaos**	**chaos**

Lógica Condicional

∨	true	false	chaos
true	**true**	**true**	**true**
false	**true**	**false**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

Lógica Condicional

⇒	true	false	chaos
true	**true**	**false**	**chaos**
false	**true**	**true**	**true**
chaos	**chaos**	**chaos**	**chaos**

Notar ∧, ∨ y ⇒ que no son conmutativas en el caso de expresiones no terminantes (**chaos**).

≡ y =

≡	true	false	chaos
true	**true**	**false**	**false**
false	**false**	**true**	**false**
chaos	**false**	**false**	**true**

≡ y =

=	true	false	chaos
true	**true**	**false**	**chaos**
false	**false**	**true**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

≡ y =

- Notar que:
  - las ≡ y = coinciden sólo para el caso de las expresiones terminantes.
  - No hay necesidad de ⇔ ya que la tabla de la igualdad aplica para la doble implicación.

Expresiones cuantificadas Ejemplos

$\forall x : \mathbf{Nat} \cdot (x = 0) \ \vee (x > 0)$

$\exists x : \mathbf{Int} \cdot x = 7$

$\exists! \ x : \mathbf{Int} \cdot (x \geq 0) \ \wedge (x \leq 0)$

$\forall x : \mathbf{Nat} \cdot x = -7$

$\forall x, y : \mathbf{Nat} \cdot (\exists! \ z : \mathbf{Nat} \cdot x+y = z)$

Expresiones cuantificadas Forma general

cuantificador $typing_1, \ldots, typing_n$ • *expresion_logica*

El tipo **Int**

El literal **Int** denota el tipo conteniendo los números enteros: ..., -2, -1, 0, 1, 2, ...

Operadores Prefijos:

abs: **Int** → **Nat**

ejemplos:

abs –5 = 5

abs 5 = 5

El tipo **Int**

Operadores Infijo

>: **Int** × **Int** → **Bool**
<: **Int** × **Int** → **Bool**
≥: **Int** × **Int** → **Bool**
≤: **Int** × **Int** → **Bool**
+: **Int** × **Int** → **Int**
- : **Int** × **Int** → **Int**
* :**Int** × **Int** → **Int**
/ : **Int** × **Int** $\rightarrow\!\!\!\!\!\sim$ **Int** // división entera
\ : **Int** × **Int** $\rightarrow\!\!\!\!\!\sim$ **Int** // resto entero
↑: **Int** × **Int** $\rightarrow\!\!\!\!\!\sim$ **Int** // exponenciación

El tipo **Int**

7/3 = 2
-7/3 = -2
7/-3 = -2
-7/-3 = 2
7\3 = 1
-7\3 = -1
7\-3 = 1
-7\-3 = -1
↑ No está definido cuando ambos argumentos son 0 o si el segundo es negativo.

El tipo **Nat**

El literal **Nat** denota el tipo conteniendo los números enteros positivos: 0, 1, 2, ...

Nat es un subtipo de **Int**, en consecuencia todos los operadores definidos para **Int** están definidos para **Nat**.

El tipo **Real**

El literal **Real** denota el tipo conteniendo los números reales: ..., -4.33, ..., 1.0,..., 12.277,...

Int no es un subtipo de **Real** => Operadores de conversión:

int: **Real** → **Int** // retorna valor más cercano a cero

real: **Int** → **Real**

El tipo **Real**

int 4.6 = 4

int -4.6 = -4

real 5 = 5.0

real ((**int** 5.2) /2) = 2.0

El tipo **Real**

abs: **Real** → **Real**
>: **Real** × **Real** → **Bool**
<: **Real** × **Real** → **Bool**
≥: **Real** × **Real** → **Bool**
≤: **Real** × **Real** → **Bool**
+: **Real** × **Real** → **Real**
- : **Real** × **Real** → **Real**
* :**Real** × **Real** → **Real**
/ : **Real** × **Real** ⇸ **Real**
↑: **Real** × **Real** ⇸ **Real**

Los tipos **Char** y **Text**

El literal **Char** denota el tipo conteniendo los caracteres: 'A', 'b', etc.

El literal **Text** denota el tipo conteniendo strings de caracteres.

p.e.: "esto es un texto", ""

Text es una abreviatura para **Char*** (lista de **Char**), en consecuencia se le pueden aplicar todos los operadores definidos para listas en RSL.

El tipo **Unit**

El literal **Unit** denota el tipo que contiene el único valor ().

Es útil cuando se trabaja con especificaciones imperativas y concurrentes (similar a void en C).

Se usa para poner un tipo en los parámetros de las funciones que no tienen parámetros y para proveer un tipo de retorno en las funciones que no retornan valores.

1. Below are some equivalences which always hold (are true) in classical logic. Which of them hold in RAISE's conditional logic?

   (a) $\sim(\sim a) \equiv a$
   (b) **true** $\vee$ a $\equiv$ **true**
   (c) a $\vee$ **true** $\equiv$ **true**
   (d) a $\Rightarrow$ b $\equiv \sim$a $\vee$ b
   (e) a $\vee \sim$a $\equiv$ **true**
   (f) (a $\wedge$ b) $\wedge$ c $\equiv$ a $\wedge$ (b $\wedge$ c)
   (g) (a $\vee$ b) $\vee$ c $\equiv$ a $\vee$ (b $\vee$ c)
   (h) (a = a) $\equiv$ **true**
   (i) (a $\equiv$ a) $\equiv$ **true**

2. Reduce the following expressions to simpler expressions:

   (a) **if true then false else chaos end** $\equiv$?
   (b) **if** a **then** $\sim$(a $\equiv$ **chaos**) **else false end** $\equiv$?

Hint:
Use the following equivalences:

if true then a **else** b **end** $\equiv$ a

if false then a **else** b **end** $\equiv$ b

if a **then** e1 **else** e2 **end** $\equiv$ **if** a **then** e1[**true**/a] **else** e2[**false**/a] **end**

if a **then true else false end** $\equiv$ a

3. Which of the following expressions are true:

   (a) $\forall$ i : **Int** • $\exists$ j : **Int** • i + j = 0
   (b) $\forall$ i : **Int** • $\exists$ j : **Nat** • i + j = 0
   (c) $\exists$ i : **Int** • $\forall$ j : **Int** • i + j = 0

4. Write an RSL value expression which expresses the fact that there is not a largest integer.

5. Complete the following definition of a function which tests whether a natural number is even.

```
is_even : Nat → Bool
is_even(n) ≡ ...
```


Productos

Producto

La expresión de tipo Producto Cartesiano

$tipo_1 \times ... \times tipo_n \qquad n \geq 2$

representa el tipo conteniendo los productos

$(v_1, ..., v_n)$

donde v_i es un valor de tipo $tipo_i$

Producto, ejemplos

Bool $\times$ **Bool** representa los productos

(**true**, **true**), (**true**, **false**), (**false, true**), (**false, false**)

(Nat $\times$ **Nat)** $\times$ **Bool** representa los productos

((0, 0), **true**), ((0, 0), **false**), ((0, 1), **true**), ((0, 1), **false**), ...

Nat $\times$ **Nat** $\times$ **Bool** representa los productos

(0, 0, **true**), (0, 0, **false**), (0, 1, **true**), (0, 1, **false**), ...

Ejemplo: Un sistema de Coordenadas (1)

- Un sistema de coordenadas provee un conjunto de posiciones (x, y) donde *x* e *y* son números reales.
- El centro del sistema de coordenadas es (0.0, 0.0) y es referenciado como el *origen*.
- La *distancia* entre dos posiciones es obtenida por el teorema de Pitágoras.

Ejemplo: Un sistema de Coordenadas (2)

```
SYSTEM_OF_COORDINATES =
  class
    type
      Position = Real × Real
    value
      origin : Position = (0.0,0.0),
      distance : Position × Position → Real
      distance((x1,y1),(x2,y2)) ≡
        ((x2−x1)↑2.0 + (y2−y1)↑2.0)↑0.5
  end
```


Ejemplo: Un sistema de Coordenadas (3)

```
SYSTEM_OF_COORDINATES =
  class
    type
      Position = Real × Real
    value
      origin : Position = (0.0,0.0),
      distance : Position × Position → Real
      distance(p1,p2) ≡
        let
          (x1,y1) = p1,
          (x2,y2) = p2
        in
          ((x2−x1)↑2.0 + (y2−y1)↑2.0)↑0.5
        end
  end
```


Expresiones Let (1)

Usadas especialmente en dos formas:

1. Para destruir un producto, p.e.:

   let (x, y) = (1, 2) **in** x + y **end**

   que evaluará a 3.

- Primero se evalúa la expresión que sigue al =.
- Luego se asocia *x* a la primera parte e *y* a la segunda.
- Finalmente se evalúa la expresión que sigue a **in**.

Producto, ejercicios

(a) Define a type, 'Complex', with an appropriate representation for complex numbers.
(b) Define a value, 'zero', which represents the complex number 0 + 0i.
(c) Define a value, 'c', which represents a complex number of the form x + xi, i.e. one where the real and imaginary parts are equal in magnitude.
(d) Define functions, 'add' and 'mult', for addition and multiplication of complex numbers.
(e) Define a function, 'f', which takes a complex number as argument and returns some complex number which is different.

Producto, ejercicios (hints)

Part (b) can be done with an explicit value definition, part (c) with an implicit value definition, part (d) with explicit function definitions, and part (e) with an implicit function definition.

(x1,y1) + (x2,y2) = (x1 + x2, y1 + y2)

(x1,y1) (x2,y2) =

(x1 · x2 - y1 · y2 , x1 · y2 + y1 · x2)