Sviluppo di Software Sicuro - S^3
Condizioni di verifica - IV

Corso di Laurea Magistrale in
Sicurezza Informatica: Infrastrutture e Applicazioni
Università di Pisa – Polo di La Spezia
C. Montangero
Anno accademico 2009/10

Sommario

- Data refinement – introduzione
- Caratterizzazione del tipo astratto
- Definizione della rappresentazione
- Relazione tra tipo astratto e concreto
- Codifica del tipo concreto
- Verifica del codice

S^3 2009/10 – Condizioni di verifica – Parte III

INTRODUZIONE

Il problema

- Mettere a disposizione un TDA
  - visione astratta di un tipo di dato, semplice da utilizzare
  - esempi:
    - Integer invece di sequenze di bit
    - stack invece di array e puntatore
- e realizzarlo in modo efficiente (accesso, memoria)
- ma... **corretto**!

Procedimento

- Caratterizzazione del tipo astratto
  - sintassi: interface in Java, package spec in Ada
    - costanti, costruttori, selettori e predicati
  - semantica: modello, assiomi
    - significato alle operazioni, in senso astratto
- Definizione della rappresentazione
  - struttura dati concreta
    - variabili private di una classe implements, private in Ada
- Relazione tra tipo astratto e concreto
  - corrispondenza matematica tra i due insiemi di valori
- Codifica del tipo concreto
  - classe in Java, package body in Ada
- Verifica del codice

S^3 2009/10 – Condizioni di verifica – Parte III

CARATTERIZZARE UN TIPO ASTRATTO

Esempio: IntStack

```
package Stack
  --# own State ;
  --# initializes State;
is
  function Empty return Boolean;
  --# global State;
  --# return ? ;

  procedure Clear;
  --# global out State;
  --# derives State from ;
  --# post ? ;

  procedure Pop (X : out Integer);
  --# global in out State;
  --# derives State from *
  --# &       X     from State;
  --# pre  ? ;
  --# post ? ;
```

- Come diciamo:
  - il risultato di Empty è true sse lo stack è vuoto?
  - lo stato dopo Clear è ure vuoto?
  - ^Pop richiede che lo stack non sia vuoto?
  - ^Pop assicura che X vale ciò che stava in cima allo stack, e che X è stato eliminato?

Modello (canonico)

- Il tipo StackT (il tipo dei valori che può assumere State) è così definito:
  - "nil" è uno stack
  - se s è uno stack, e X un intero, "push(?X,?s)" è uno stack
  - non ci sono altri stack

```
nil
... push(-1,nil) push(0,nil) push(1,nil) ...
... push(- 1, push(-1,nil)) push(0,push(-1,nil)) ...
...
```

- Semantica delle costanti e dei costruttori.

Modello (canonico) (2)

- Selettori e predicati vengono definiti sul modello
  - empty(s) = true se s="nil", false altrimenti
  - pop(s) = s' se s="push(?X,?s')" per qualche X
  - top(s) = X se s="push(?X,?s')" per qualche s'
- Potremmo allora definere le procedure
  (NB Ada nomi maiuscoli, modello minuscoli)

Esempio: IntStack

```
package Stack
   --# own State ;
   --# initializes State;
is
   function Empty return Boolean;
   --# global State;
   --# return State = "nil" ;

   procedure Clear;
   --# global out State;
   --# derives State from ;
   --# post State = "nil" ;

   procedure Pop (X : out Integer);
   --# global in out State;
   --# derives State from *
   --# &       X     from State;
   --# pre  State /= "nil" ;
   --# post State = "push(?v,?s)"
   --#      -> X = v and State = s ;
```

- Come diciamo:
  - il risultato di Empty è true sse lo stack è vuoto?
  - lo stato dopo Clear è ure vuoto?
  - ^Pop richiede che lo stack non sia vuoto?
  - ^Pop assicura che X vale ciò che stava in cima allo stack, e che X è stato eliminato?

Modello assiomatico

- Il tipo StackT è definito implicitamente
- Dalle proprietà delle operazioni
  - nil: -> StackT empty: StackT -> Boolean
  - push: Int x StackT -> StackT
  - pop : StackT -> StackT top : StackT -> Int
- come
  - empty(nil) = true empty(push(X,s)) = false
  - pop(push(X,s)) = s
  - top(push(X,s)) = X

Esempio: IntStack

```
package Stack
   --# own State ;
   --# initializes State;
is
   function Empty return Boolean;
   --# global State;
   --# return empty(State) ;

   procedure Clear;
   --# global out State;
   --# derives State from ;
   --# post empty(State) ;

   procedure Pop (X : out Integer);
   --# global in out State;
   --# derives State from *
   --# &       X     from State;
   --# pre  not empty(State);
   --# post State~ = push(Y,State)
   --#       -> X = Y;
```

- Come diciamo:
  - il risultato di Empty è true sse lo stack è vuoto?
  - lo stato dopo Clear è ure vuoto?
  - ^Pop richiede che lo stack non sia vuoto?
  - ^Pop assicura che X vale ciò che stava in cima allo stack, e che X è stato eliminato?

Pregi e difetti

- Modello canonico è meno astratto
- Gli assiomi corrono il rischio di essere incompleti o contradditori
- Sotto certe condizioni dagli assiomi si ricava un modello canonico (molto simile a quello visto)
- Sul modello canonico si possono verificare le proprietà assiomatiche (modello di una teoria logica)
- e SPARK ?

Tipi astratti in SPARK

- Annotazioni per dichiarare
  - tipi astratti
  - funzioni (di prova, esplicite)
- Esempi

```
--# type StackAT is abstract;

--# function push_a(X : Integer; S : StackAT)
--#                return StackAT;
```

- Ma nessun modo di dare semantica…

Esempio IntStack

```
package Stack
   --# own State : StackAT;   -- promessa!
   --# initializes State;
is
   --# type StackAT is abstract;
   --# function empty_a(S: StackAT) return Boolean;
   --# function push_a(X : Integer; S : StackAT)
   --#                          return StackAT;
   --# function pop_a(S : StackAT) return StackAT;
   --# function top_a(S : StackAT) return Integer;
   --# function size_a(S : StackAT) return Natural;

   MaxSize : constant Natural := 100;
```


con queste definizioni

```
is
 function Empty return Boolean;
   --# global State;
   --# return empty_a(State);

   procedure Clear;
   --# global out State;
   --# derives State from ;
   --# post empty_a(State) ;

   procedure Pop (X : out Integer);
   --# global in out State;
   --# derives State from *
   --# &       X     from State;
   --# pre  not empty_a(State);
   --# post State~ = push_a(X, State);
```

- Come diciamo:
  - il risultato di Empty è true sse lo stack è vuoto?
  - lo stato dopo Clear è ure vuoto?
  - ^Pop richiede che lo stack non sia vuoto?
  - ^Pop assicura che X vale ciò che stava in cima allo stack, e che X è stato eliminato?

S³ 2009/10 – Condizioni di verifica – Parte III

DEFINIZIONE DELLA RAPPRESENTAZIONE

Finora...

```
package body Stack
--# own State is Pointer, Vector;
is
   subtype PointerT is Natural
             range Natural'First .. MaxSize;
   Pointer : PointerT;

   type IntVectorT is array (Integer range <>) of Integer;
   subtype StackRange is Positive range 1 .. MaxSize;
   subtype IntStackT is IntVectorT (StackRange);
   Vector : IntStackT;
   EmptyVector : constant IntStackT :=
                    IntStackT'(StackRange => 0);
```

- Un valore concreto è una coppia di valori
- Efficiente, ma difficile da trattare

Il tipo concreto

```
type IntStackCT is record
   point : PointerT;         -- come sopra
   vect  : IntStackT;        -- come sopra
end record;

ConcreteStack : IntStackCT;
EmptyConcreteStack : constant IntStackCT :=
     IntStackCT'(point => 0, vect => EmptyVector);
```

- e quindi

```
package body Stack
--# own State is ConcreteStack; -- rappresentazione
```

- un valore concreto per ogni astratto

S³ 2009/10 – Condizioni di verifica – Parte III

RELAZIONE ASTRATTO-CONCRETO

Idealmente

- Siano
  - A il tipo astratto, con valori a, a', ...
  - C il tipo concreto, con valori c, c', ...
  - α: C -> A, la funzione di astrazione
  - γ: A -> C, quella di concretizzazione
- con le condizioni (ovvie)
  - $\alpha(\gamma(a)) = a \quad \gamma(\alpha(c)) = c$
- ogni P^A : A -> Bool definisce P^C : C -> Bool
- $P^C = \alpha \circ P^A$ (viceversa $P^A = \gamma \circ P^C$)
- Sia $\mathcal{T}$ questa trasformazione applicata a pre/post condizioni

Integrità del raffinamento

```
procedure P(X : C)
 --# pre  ^PC;
 --# post PC^;
```

- *raffina* onestamente (integrity)

```
procedure P(X : A)
 --# pre  ^PA;
 --# post PA^;
```

- purchè

  $\mathscr{T}$ ^P^A -> ^P^C

  $\mathscr{T}$ P^C^ -> P^A^

Correttezza della realizzazione

```
procedure P(X : C)
 --# pre  ^PC;
 --# post PC^;
```

- *implementa* correttamente

```
procedure P(X : A)
 --# pre  ^PA;
 --# post PA^;
```

- purchè *il raffinamento sia integro*, e P^C *sia corretta*, ossia valga, al solito:

  ^P^C -> wp[body]P^C^

Praticamente in Spark...

- La correttezza del raffinamento si esprime
  - con *rule* in file .rlu
  - che Simplifier usa per collegare
    - le pre/post-condizioni della specifica e del body
- Si tratta di una *specifica contratta* di $\mathscr{T}$, α e γ
- A *completo carico* del progettista sw!
- Questo risolve anche la questione delle due variabili
  - si assume a livello astratto, in FDL

Raffinamento per Examiner

- La dichiarazione (nel body)

```
--# own State is Pointer, Vector;
```

- viene interpretata da Examiner come un raffinamento: genera vc per collegare le pre/post condizioni di specifica e body, nella forma

$$\hat{P}^A \rightarrow \hat{P}^C \quad \text{e} \quad P^{C\wedge} \rightarrow P^{A\wedge}$$

- Le regole servono per semplificarle

Integrità del raffinamento

```
procedure  Pop(X: out Integer);   Pop (X : out Integer)
--# global in out State;         in out Pointer; in Vector;
…                                 ...
--# pre    not empty_a(State);    Pointer > 0 ;
```

For checks of refinement integrity:
procedure_pop_4.
H1: not empty_a(state) .
->
C1: fld_pointer(state) > 0 .

Integrità del raffinamento

```
procedure  Pop(X: out Integer);   Pop (X : out Integer)
--# global in out State;         in out Pointer; in Vector;
…                                 ...
--# pre    not empty_a(State);    Pointer > 0 ;
```

For checks of refinement integrity:
procedure_pop_4.
H1: not empty_a(state) .
->
C1: fld_pointer(state) > 0 .

fld sta per field :
fld_pointer è quindi il campo pointer del record che rappresenta lo stack;
NB: state è implicitamente astratto in H1, concreto in C1.

Integrità del raffinamento

```
procedure  Pop(X: out Integer);
--# global in out State;
…
--# pre    not empty_a(State);
```

```
Pop (X : out Integer)
in out Pointer; in Vector;
...
Pointer > 0 ;
```

For checks of refinement integrity:
procedure_pop_4.
H1: not empty_a(state) .
->
C1: fld_pointer(state) > 0 .

𝒯 not empty_a(state) -> fld_pointer(state) > 0 ?

Integrità del raffinamento

For checks of refinement integrity:
procedure_pop_4.
H1: not empty_a(state) .
->
C1: fld_pointer(state) > 0 .

```
Pop (X : out Integer)
in out Pointer; in Vector;
...
Pointer > 0 ;
```

𝒯 not empty_a(state)
≡
not 𝒯 empty_a(state)
≡ converto e applico la funzione concreta
not (𝒯 empty_a)(γ state)
≡ λx.(𝒯 empty_a) = λx.(fld_pointer(x) = 0), per definizione di raffinamento
not (fld_pointer(γ state) = 0)

In SPARK, si abbrevia il tutto con le regole utente, in un file .rlu:

empty_a(X) may_be_replaced_by fld_pointer(γ state) = 0

Integrità del raffinamento

```
procedure  Pop(X: out Integer);
--# global in out State;
…
--# post State=pop_a(State~) and
--#      X = top_a(State~);
```

```
Pop (X : out Integer)
in out Pointer; in Vector;
...
--# post Pointer=Pointer~ -1
--#  and X = Vector(Pointer~);
```

procedure_pop_5.
H1: not empty_a(state~) .
H3: pointer~ = fld_pointer(state~) .
H4: fld_vector(state~) = fld_vector(state) .
->
C1: state = pop_a(state~) .
C2: element(fld_vector(state~), [pointer~]) = top_a(state~) .

Integrità del raffinamento

```
  𝒯 (= (X, Vector(Pointer~)))
≡    def: top_c = λx.element(fld_vector(x),[fld_pointer(x)])
     e H4
  𝒯 (= (X, top_c(mk(Pointer~, Vector~))))
≡
 (𝒯 =)(X,(𝒯 top_c) ∝ mk(Pointer~, Vector~))
≡  λx.(𝒯 fld_pointer) = λx.top_a    (def raff)
 = (X, top_a(State~))
```

element(fld_vector(X, [fld_pointer(X)]) may_be_replaced_by top_a(X)

```
procedure_pop_5.
H1:   not empty_a(state~) .
H3:   pointer~ = fld_pointer(state~) .
H4:   fld_vector(state~) = fld_vector(state) .
      ->
C2:   element(fld_vector(state~), [pointer~]) = top_a(state~) .
```

S3: VC - C.Montangero - Copyright 2010 31

S³ 2009/10 – Condizioni di verifica – Parte III

SIMPLIFIER: REGOLE UTENTE

S3: VC- C.Montangero - Copyright 2010 32

file .rlu

- uno per il progetto (e.g. stack.rlu)
  - con le regole di raffinamento
- uno per ogni sottoprogramma (e.g. pop.rlu)
  - con eventuali regole specifiche
- utilizzati dopo le semplificazioni automatiche
  - regole mirate alle condizioni restanti in .siv
  - alcune tattiche possono essere inibite: utile?

S3: VC - C.Montangero - Copyright 2010 33

Regole d'inferenza

- rulename(#): Goal **may_be_deduced_from** Conditions .
- f_property(2): f(X,Y) >= Z may_be_deduced_from
- [X >= W, Y >= W, W>=Z]
- Goal e ogni Condition sono termini Prolog (espressi in FDL)
- Le variabili (maiuscole) sono dette wildcards (jolly)
  - assegnamento per patter matching
- f_property(2) è utile per C1: f(a, b * b) >= c + 1 .
- ma non per C1: f(a, b * b) > c + 1 .
- the Simplifier will attempt to find a means of instantiating all of the wildcards in Goal such that it becomes an exact match for an *existing undischarged conclusion*: the main operator of the goal formula in the rule must be the same as that of the conclusion if we are to be able to pattern-match the two together.
- poi si procede con le condizioni: per C1
  a >= W, b*b >= W, W>=c+1

 34

Regole di riscrittura

- rulename(#): Old may_be_replaced_by New if Conditions.
  - parte if opzionale
  - numeri ovviamente diversi all'interno della famiglia
- If Old can be pattern-matched to the toplevel conclusion formula, this is a match.
- Otherwise, we look at its immediate subexpressions, and try pattern-matching against these in turn (in left-to-right order), and to the subexpressions of these expressions, and so on.
- Finding a match will instantiate the wildcards in Old, and will generally instantiate some (or all) of those in New and in the Conditions list.
- La sostituzione è

 35