DIM0436

16. Cálculo WP

Richard Bonichon

20140925

Sumário

Resumo de semântica

Semântica concreta

- Formalização de **todos os comportamentos** possíveis do programa
- **Função** que associa um elemento do **domínio concreto** considerado a um programa

Um analisador ideal

O analisador é:

Estático	não é preciso executar o programa
Automático	um clique só!
Exato	ele calcula sem aproximação a informação desejada

Uma análise estática automática ?

Uma análise estática automática exata é impossível.

Teorema (Rice, 1953)

Toda propriedade

- ***extensional***
  - *que depende unicamente da semântica do programa e não da sintaxe dele*
- ***não trivial***
  - *nem sempre verdadeira, nem sempre falsa*

é

- ***indecidível***
  - *todo algoritmo que decide si essa propriedade é falsa ou verdadeira*
    - *ou não termina (laço infinito)*
    - *ou erra no mínimo numa infinidade de programas*

Exemplo (Problema da parada)

Técnicas de análise

Neutralizar o teorema de Rice

- **Testes**
  - Muito usados na industria até hoje. Não garante a correção do código mas acha contraexemplos
  - *Program testing can be used to show the presence of bugs, but never to show their absence!*
    – Dijkstra, 1969
- **Model-checking**
  - Analisar não o programa mesmo mas modelos dele (por exemplo, a política de segurança)
  - Perda de precisão possível
- **Métodos dedutivos** (esta aula)
  - Raciocinar precisamente, por dedução, sobre o programa, e extrair os problemas "difíceis" (condições de verificação)
  - Geralmente não automático
- **Interpretação abstrata**
  - Realizar aproximações da semântica concreta
  - Perda de precisão possível

Sintaxe de `While`

Categorias sintáticas

- n, n_i, n' = elementos numéricos (Num)
- x = variáveis (Var)
- a = expressões aritméticas ($\mathcal{A}_e xp$)
- b = expressões booleanas ($\mathcal{B}_e xp$)
- S = instruções

BNF

$$
\begin{array}{rcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2 \\
b & ::= & \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
S & ::= & x := a \mid \text{skip} \mid S_1 ; S_2 \\
 & \mid & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & \mid & \text{while } b \text{ do } S
\end{array}
$$

Fundamentos

Triplas de Hoare

$$\{P\}s\{Q\}$$

- Se a propriedade P é verdadeira num ponto do programa, a propriedade Q será verdadeira após a execução da instrução s

- Análise proposta por C.A.R. Hoare (1969) e W. Floyd (1967)
- Um conjunto de regras indica quais são as condições para que um tripleto $\{P\}s\{Q\}$ seja válido

Cálculo de WP

A análise de Floyd-Hoare fornece um **cálculo de pré-condição mais fraca**

- Wp(s, Q) é a propriedade mais fraca tal que $\{Wp(s, Q)\}s\{Q\}$ seja verdadeira.
- se $\{P\}s\{Q\}$ é verdadeiro, $P \Rightarrow Wp(s, Q)$

Lógica de Hoare e interpretação abstrata

O **cálculo de pré-condição mais fraca** e a **análise por interpretação abstrata** são dois meios complementários para estabelecer propriedades de um programa.

Vista sintética

Lógica de Hoare / WP	Interpretação abstrata
✔Verificação de propriedades arbitrárias ✔Modular ✘ Escrever especificações formais ✘ Precisa interagir com o usuário	✘ Restrito às propriedades expressáveis no reticulado usado ✘ Análise de uma aplicação completa ✔Precisa só do código fonte ✔Automática
Propriedades funcionais	**Ausência de erro na execução**

Regras da lógica de Hoare: a base

$$\frac{}{\{P\}\text{skip}\{P\}}\ \text{Ax}$$

$$\frac{\{P\}S_1\{R\} \qquad \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}\ \text{Seq}$$

$$\frac{}{\{P[x \leftarrow \mathcal{A}[\![a]\!]]\}x := a\{P\}}\ \text{Ass}$$

Regras da lógica de Hoare: instruções

$$\frac{\{P \land \mathcal{B}[\![b]\!]\}S_1\{Q\} \qquad \{P \land \neg\mathcal{B}[\![b]\!]\}S_2\{Q\}}{\{P\} \text{ if } b \text{ then } S_1 \text{ else } S_2\{Q\}} \text{ If}$$

$$\frac{P \Rightarrow I \qquad \{I \land \mathcal{B}[\![b]\!]\}S\{I\} \qquad I \land \neg\mathcal{B}[\![b]\!] \Rightarrow Q}{\{P\} \text{ while } b \text{ do } S\{Q\}} \text{ While}$$

$$\frac{P \Rightarrow P' \qquad Q' \Rightarrow Q \qquad \{P'\}s\{Q'\}}{\{P\}s\{Q\}} \text{ S/W}$$

Exercícios

- Mostrar
  $\{\emptyset\}$
  `if (n <= 0) then result := (-n) else result := n;`
  $\{result \geq 0\}$
- Mostrar
  $\{x > 0\}$
  `y:=1; while` $\neg$ `(x = 1) do (y := y * x; x := x - 1)`
  $\{y >= 1\}$
- Mostrar
  $\{x = n\}$
  `y:=1; while` $\neg$ `(x = 1) do (y := y * x; x := x - 1)`
  $\{y = n! \wedge n > 0\}$

Lógica de Hoare clássica e memória

Lógica de Hoare clássica

A regra

$$\{P[x \leftarrow \mathcal{A}[\![a]\!]]\} x := a \{P\}$$

tem uma hipótese implícita:

A *location* x no programa é representada por uma variável x só

Problema

Algumas linguagens não respeitam essa hipótese!

Em C

Manipulação explícita da memória

As linguagens — como C — que manipulam ponteiros de objetos armazenados na memória não validam essa hipótese:

- {?} `t[3] = 0;` {"o elemento na índice 3 de t contem 0" }
- {?} `s -> campo = 3` {"o conteúdo do campo de s é 3"}
- {?} `*p = 42` {*p == 42}

- É preciso ter um **modelo da memória**, que traduz na lógica os acessos à memória, em leitura e em escrita.
- É geralmente usado uma representação em forma de vetores

Axiomatização

Funções de acesso

- Tipos dos vetores (polimórficos): $\forall\alpha.\alpha$ array
- Uma função de leitura : **select**: $\forall\alpha.\alpha$ array $\times$ int $\rightarrow \alpha$
- Uma função de escrita: **store** : $\forall\alpha.\alpha$ array $\times$ int $\times\alpha \rightarrow \alpha$ array

Axiomas

- $\forall\alpha\forall$ a, i, v. a : α array, i : int, v : α, select(store(a, i, v), i) = v
- $\forall\alpha\forall$ a, i, j, v. a : α array, i, j : int, v : α, i $\neq$ j $\Rightarrow$ select(store(a, i, v), j) = select(a, j)

Exemplo

```c
int x[2];

//@ ensures x[0] == 0 && x[1] == 1;
int main () {
  int i = 0;
  x[i] = i;
  i = i + 1;
  x[i] = i;
}
```

Exemplo

```
int x[2];

//@ ensures x[0] == 0 && x[1] == 1;
int main () {
  // select(store(store(x,0,0),1,1),0) = 0 &&
  // select(store(store(x,0,0),1,1),1) = 1
  int i = 0 & 1;
  /* select(store(store(x,i,i),i+1,i+1),0) = 0 &&
     select(store(store(x,i,i),i+1,i+1),1) = 1 */
  x[i] = i;
  // select(store(x, i+1, i+1), 0) &&
  // select(store(x, i+1, i+1), 1) == 1
  i = i + 1;
  // select(store(x, i, i), 0) == 0 &&
  // select(store(x, i, i), 1) == 1
  x[i] = i;
 // select(x, 0) == 0 && select(x, 1) == 1
}
```

Validade das leitureas

Problema

- `select(a, i)` gera um valor para todo i $\in \mathbb{Z}$
- `store(a, i, v)` funciona para todos os índices

Solução

- Usar uma função `length:` $\forall\ \alpha.\ \alpha$ array $\rightarrow$ int
- Com 2 axiomas
  - `length_pos:` $\forall\ \alpha.\ \forall$ a: α array length(a) ≥ 0
  - `store_length:` $\forall\ \alpha.\ \forall$ a: α array, i:int, v: α. length(store(a, i, v)) $=$ length(a)

Limite dos vetores

Especificação defensiva

- `select_store_eq`: $\forall\alpha\forall$ a, i, v. a : α array, i : int, v : α, $0 \leq$ i $<$ length(a) $\Rightarrow$ select(store(a, i, v), i) $=$ v
- `select_store_neq`: $\forall\alpha\forall$ a, i, j, v. a : α array, i, j : int, v : α, i $\neq$ j $\Rightarrow 0 \leq$ i $<$ length(a) $\Rightarrow$ select(store(a, i, v), j) $=$ select(a, j)

Exemplo

```
int x[2]; // axiom: length(x) == 2;

//@ ensures x[0] == 0 && x[1] == 1;
int main () {
  int i = 0;
  x[i] = i;
  i = i + 1;
 // select(x, 0) == 0 && select(x, 1) == 1 && 0 <= i < length(x)
  x[i] = i;
 // select(x, 0) == 0 && select(x, 1) == 1
}
```

Ponteiros

Representação dos ponteiros

Primeira ideia: um ponteiro = acesso a um vetor

- ponteiro $\approx$ endereço de base + deslocamento (índice)
- variável cujo endereço é usado $\approx$ vetor
- permite a representação das chamadas por referência

Exemplo

```
int x; // axiom: length(X) = 1

/*@ requires \valid(p);
    0 <= p < length(P0);
    ensures *p == \old(*p) + 1;
*/
void incr (int* p) {
 // select(store(P, p,select(P0,p)+1),p)=select(P_0,p)+1
 (*p)++;
 // select(P, p) == select(P0, p) + 1
}


/*@ ensures x == 1; */
int main () {
  // select(X, 0) == select(X0, 0) + 1 ==> select(X, 0) == 1
  incr(&x);

  return x;
  // select(X, 0) == 1
}
```

Aliasing

Problema

- O modelo precedente supõe que, em todo momento, o acesso à uma zona da memória (*location*) é feita através um **único** caminho.
- Não podemos ter ***aliasing***

Exemplo

```
int x;
/*@ requires \valid(p);
    ensures *p == x + 1;
*/
void incr (int* p) {
  *p = x;
  (*p)++;
}

int main(){
  incr(&x);
}
```

Exemplo

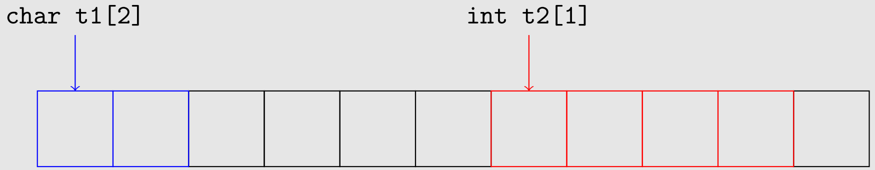
```
int x;
/*@ requires \valid(p);
   0 <= p < length(P0);
   ensures *p == x + 1;
*/
void incr (int* p) {
  // select(store(store(P0,p,x),p,select(...)+1)= select(X, 0) +1
  *p = x;
  // select(store(P,p,select(P,p)+1) = select(X, 0) +1
  (*p)++;
  // select(P,p) = select(X, 0) + 1
}

int main(){
  incr(&x);
}
```

Modelo baixo nível

Memória = vetor

- ✔ Aliasing possível
- ✔ Manipulação dos ponteiros permitida
- ✘ Todo é na mesma memória geral : um `store` pode potencialmente mudar o conteudo de qualquer outro ponteiro
- ✘ A estrutura dos dados é perdida
- ✘ Inusável em prática (obrigações de prova gigante)

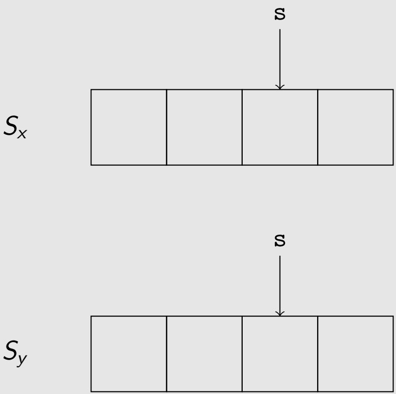


Burstall-Bornat

- Usar tipos para separar os ponteiros
- ✔ Um `store` não muda tudo o conteúdo da memória
- ✔ A estrutura dos dados é conservada
- ✘ Não pode misturar valores de tipos estáticos diferentes **Nem cast, nem** `union`

```
struct S
  { int x; int y; };

int main () {
...
s->x = 3;
s->y = 2;
...
```



Lógica de separação

Ideia fundamental (O'Hearn, Reynolds, Yang)

Falar explicitamente da memória nas fórmulas

- Introdução de novos conectivos ...
- e de novas regras de dedução para as triplas de Hoare

Conectivos de base

- *Emp*, memória vazia
- $l \mapsto v$: Memória com uma única *location* l armazenando o valor v
- $e_1 * e_2$: Memória composta de duas partes **separadas**, uma verificando a condição e_1, a outra e_2.
- $e_1 \mathrel{-\!\!*} e_2$: Se existir uma parte da memória verificando e_1, uma parte **separada** verifica e_2

Uso

Regras de dedução

As regras da lógica de Hoare + regras para definir operadores manipulando explicitamente a memória:

- $\{Q\}$ `malloc(n)` $\{l \mapsto \texttt{block} * Q\}$
- ${Q * l} `free(l)` $\{Q\}$

Regra de separação das zonas

A *frame rule*, parecida ao enfraquecimento, se as *locations* de S, P e Q não aparacem em R:

$$\frac{\{P\}S\{Q\}}{\{P * R\}S\{Q * R\}}$$

Estado do arte

Que sabemos fazer hoje ?

- Raciocínios matemáticos (i.e. formais) sobre programas com ponteiros
- Obrigação de descrever precisamente o estado da memória
- **Nenhum provador automático dedicado**
- Conceitos nas linguagens de especificação
  - `\separated`
  - `\fresh`
  - `\freed`

Análise de região

- Noção de região $\approx$ separação
  - Ponteiras que pertencem a duas regiões diferentes apontam sobre *locations* disjuntas
- Ao contrário de Burstall-Bornat, que usa a informação sintática, o cálculo das regiões usa o fluxo de controle.
- Vários algoritmos de cálculo de região (Steensgard, Talpin)
- O outro plugin de WP de Frama-C (Jessie) usa o cálculo de região

Resumo

Referências

- Richard Bornat, *Proving pointer programs in hoare logic*, Mathematics of Program Construction, 5th International Conference, MPC 2000, Ponte de Lima, Portugal, July 3-5, 2000, Proceedings, 2000, pp. 102–126.
- Rod M. Burstall, *Some techniques for proving correctness of programs which alter data structures*, Machine Intelligence (1972).
- Hanne Riis Nielson and Flemming Nielson, *Semantics with applications: An appetizer*, Undergraduate Topics in Computer Science, Springer, 2007.
- Flemming Nielson, Hanne Riis Nielson, and Chris Hankin, *Principles of program analysis (2. corr. print)*, Springer, 2005.
- Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang, *Local reasoning about programs that alter data structures*, Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings, 2001, pp. 1–19.

Perguntas ?

http://dimap.ufrn.br/~richard/dim0436