Verificação Formal

Maria João Frade Jorge Sousa Pinto

Março de 2017

O objectivo geral do trabalho é o desenvolvimento completo de um verificador de programas de uma linguagem de programação simples. Pretende-se explorar o tratamento de um **mecanismo de excepções** ("try-catch").

O verificador deve compreender um *parser* e um *gerador de condições de verificação* (VC-Gen), e deve ainda gerir a prova destas condições, interagindo com pelo menos uma ferramenta de prova automática (como por exemplo Z3, Alt-Ergo, CVC3, CVC4 ...). É importante sublinhar que ficam a cargo dos grupos toda uma série de escolhas, desde a sintaxe concreta da linguagem de programação alvo, até à tecnologia de implementação utilizada. Por esta razão, os grupos terão todo o interesse em iniciar desde já o processo da familiarização com as tecnologias a utilizar, para permitir o desenvolvimento atempado do projecto, preferencialmente incluindo algumas das extensões propostas.

O trabalho deverá ser desenvolvido por grupos de duas pessoas; o relatório e código deverão ser entregues até ao dia **?? de Junho**. Submissões posteriores serão penalizadas. A apresentação para avaliação terá lugar no dia **?? de Junho**.

Deliverables a entregar:

- código comentado;
- relatório justificando todas as opções e reportando todos os resultados;
- apresentação a realizar em momento de avaliação no final do semestre, incluindo demonstração de utilização.

1 Linguagem Alvo

A linguagem de programação a considerar deve conter pelo menos as seguintes construções:

- Variáveis de tipo inteiro e expressões de tipo inteiro e Booleano.
- Instrução de atribuição
- Estruturas de controlo: sequenciação; condicional (1 ou 2 ramos); pelo menos uma forma de ciclo (*while*)

A sintaxe concreta a utilizar não será fixada, podendo ser definida pelos grupos.

Na sua forma básica a linguagem não necessitará de ter qualquer noção de sub-rotina (procedimentos ou funções). No entanto, poder-se-á optar pela utilização de um *parser* de uma linguagem de programação imperativa standard (como C ou Pascal), e neste caso será útil considerar-se que o código imperativo a considerar (com as construções acima listadas) está contido num procedimento ou função principal.

Mecanismo de Excepções

É comum nas linguagens modernas a presença de um mecanismo de controlo de fluxo baseado em excepções. Na sua forma mais simples, com uma única excepção, o mecanismo oferece uma instrução elementar **throw**, que levanta a excepção, e uma construção **try** C **catch** C'. A ideia é tentar executar o bloco C. Caso, durante a execução de C, seja executado o comando **throw**, então a execução de C é interrompida, e é executado em seguida o bloco C'.

Existem pois duas formas de terminação; *normal* e *excepcional*. Para lidar com excepções axiomaticamente, é necessário modificar a noção de triplo de Hoare, passando a haver também duas pós-condições, uma para terminação normal, e outra para terminação excepcional desencadeada pela execução do comando **throw**.

Como exemplo de utilização deste mecanismo, o seguinte programa DIV calcula a divisão de x por y, atribuindo um valor constante ao resultado caso o valor de y seja 0.

```
try {
   if (y = 0) throw;
   r := x;
   q := 0;
   while (y <= r) {
      r := r-y;
      q := q+1;
   }
}
catch {
   q := INT_MAX; r:= 0;
}
```

Um triplo de Hoare válido para este programa será

$$\{(x \geq 0 \wedge y \geq 0\}\ DIV\ \{\!|(0 \leq r < y \wedge q * y + r = x) \vee (y = 0 \wedge q = INT\_MAX \wedge r = 0), \mathsf{false}|\!\}$$

A terminação excepcional não é possível, porque a excepção é capturada pelo bloco *catch*. O triplo:

$$\{(x \geq 0 \wedge y \geq 0\}\ C\ \{\!|0 \leq r < y \wedge q * y + r = x, y = 0|\!\}$$

com C o corpo do comando *try*, é também válido, e admite terminação excepcional.

O seguinte sistema de inferência capta a semântica axiomática da linguagem:

$$\frac{}{\{\phi\}\ \mathbf{skip}\ \{\!|\phi, \mathsf{false}|\!\}} \quad \text{(skip)}$$

$$\frac{}{\{\psi[e/x]\}\ x := e\ \{\!|\psi, \mathsf{false}|\!\}} \quad \text{(assign)}$$

$$\frac{\{\phi\}\ C_1\ \{\!|\theta, \epsilon|\!\} \qquad \{\theta\}\ C_2\ \{\!|\psi, \epsilon|\!\}}{\{\phi\}\ C_1\ ;\ C_2\ \{\!|\psi, \epsilon|\!\}} \quad \text{(seq)}$$

$$\frac{\{\phi \wedge b\}\ C_t\ \{\!|\psi, \epsilon|\!\} \qquad \{\phi \wedge \neg b\}\ C_f\ \{\!|\psi, \epsilon|\!\}}{\{\phi\}\ \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \{\!|\psi, \epsilon|\!\}} \quad \text{(if)}$$

$$\frac{}{\{\phi\}\ \mathbf{throw}\ \{\!|\mathsf{false}, \phi|\!\}} \quad \text{(throw)}$$

$$\frac{\{\phi\}\ C\ \{\!|\psi, \theta|\!\} \qquad \{\theta\}\ C_c\ \{\!|\psi, \epsilon|\!\}}{\{\phi\}\ \mathbf{try}\ C\ \mathbf{catch}\ C_c\ \{\!|\psi, \epsilon|\!\}} \quad \text{(try-catch)}$$

$$\frac{\{\phi\}\ C\ \{\!|\psi, \epsilon|\!\}}{\{\phi'\}\ C\ \{\!|\psi', \epsilon'|\!\}}\ \text{if}\quad \begin{array}{l}\phi' \to \phi \ \text{ and} \\ \psi \to \psi' \ \text{ and } \epsilon \to \epsilon'\end{array} \quad \text{(conseq)}$$

Atente-se na regra do comando **throw**: a pré-condição normal é false, uma vez que não é possível este comando terminar normalmente. A regra de **skip** é dual desta. Também os comandos **if** e **try** se comportam de formal dual, diferindo na pós-condição (normal ou excepcional) que é passada ao comando seguinte.

2 Infraestrutura de Desenvolvimento

O desenvolvimento da aplicação necessitará, além de uma linguagem de programação à escolha, de uma API da ferramenta de prova *para essa linguagem* (em alternativa a comunicação com esta ferramenta poderá ser feita através do formato SMT-LIB), e ainda de uma qualquer tecnologia para a criação de um *parser* para a linguagem de programação alvo.

Algumas sugestões:

- Desenvolvimento em Haskell; desenvolvimento do *parser* com *parsec*[1].
- Desenvolvimento em Python; desenvolvimento do *parser* com recurso a combinadores de parsing[2].

[1] `http://www.haskell.org/haskellwiki/Parsec`

[2] Ver por exemplo `http://www.jayconrod.com/posts/37/a-simple-interpreter-from-scratch-in-python-part-1`, `http://www.jayconrod.com/posts/38/a-simple-interpreter-from-scratch-in-python-part-2`, `http://www.jayconrod.com/posts/39/a-simple-interpreter-from-scratch-in-python-part-3`

- Desenvolvimento em Java; desenvolvimento do *parser* com recurso a um gerador standard[3].

Quanto à comunicação com a ferramenta de prova, tratando-se de um SMT-*solver* é sempre possível a utilização do formato SMT-LIB, mas algumas ferramentas disponibilizam interfaces de programação com *bindings* para diversas linguagens. Por exemplo no caso particular da ferramenta Z3, existem APIs/*bindings* para C, C++, Java, OCaml e Python[4], e ainda para Haskell[5].

3 Linguagem de Especificação

A sintaxe concreta da linguagem deverá prever um mecanismo para a anotação de *pré-condições*, *pós-condições*, e *invariantes de ciclo*. Para ilustrar uma possível sintaxe concreta veja-se o seguinte exemplo de um programa anotado muito simples :

```
pre x > 100

while (x < 1000) do
      { 100 < x and x <= 1000 }
      x := x+1
end;

post x = 1000
```

Este programa daria origem às condições de verificação seguintes:

1. `x > 100 ==> 100 < x and x <= 1000`
2. `100 < x and x <= 1000 and x < 1000 ==> 100 < x+1 and x+1 <= 1000`
3. `100 < x and x <= 1000 and not(x < 1000) ==> x = 1000`

4 Etapas

1. (12 valores) A aplicação deverá implementar o seguinte *workflow* a partir de uma invocação na linha de comando:
   (a) leitura de um ficheiro contendo um programa imperativo anotado
   (b) construção da respectiva árvore de sintaxe (AST)
   (c) geração das condições de verificação (VCs) do programa por travessia da AST
   (d) apresentação das VCs
   (e) tentativa de prova de cada VC, utilizando a ferramenta externa. As expressões de tipo inteiro dos programas deverão ser modeladas como inteiros matemáticos (*unbounded*)

[3] Por exemplo `http://www.antlr.org`
[4] `https://github.com/Z3Prover/z3`
[5] `http://hackage.haskell.org/package/z3`

(f) apresentação dos resultados da verificação

(g) Validação da aplicação através de um número razoável de testes.

2. (6 valores) Tratamento de excepções

   (a) extensão do algoritmo de geração de condições de verificação para triplos com duas pós-condições, e com os comandos **throw** e **try**.

   (b) Validação desta extensão através de testes adequados.

3. (2 valores) Bonificação por melhoramentos adicionais: GUI; utilização de interface Why3[6] para gestão da interface com diversas ferramentas de prova.

[6] `http://why3.lri.fr`