Verificação Formal de Software

Maria João Frade Jorge Sousa Pinto

Março de 2014

O objectivo deste trabalho é o desenvolvimento completo de um verificador de programas de uma linguagem de programação imperativa simples. O verificador deve compreender um *parser* e um *gerador de condições de verificação* (VCGen), e deve ainda gerir a prova destas condições, interagindo com uma qualquer ferramenta de prova automática (como por exemplo Z3, Alt-Ergo, CVC3, CVC4 ...).

É proposto um enunciado-base e algumas sugestões de extensões, que permitirão diferenciar os trabalhos dos vários grupos.

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.

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.

O relatório e código deverão ser entregues até ao dia **19 de Junho**, data em que terá lugar a respectiva apresentação para avaliação. Submissões posteriores serão penalizadas.

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*)
- Tipos de dados compostos: *arrays* alocados com dimensão fornecida estaticamente.

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.

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].
- 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 Python, Java[4], Haskell[5], C, C++, e .Net.

[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`

[3] Por exemplo `http://www.antlr.org`

[4] `http://leodemoura.github.io/blog/2012/12/10/z3-for-java.html`

[5] `http://hackage.haskell.org/package/z3`

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 Enunciado-base

A aplicação deverá implementar o seguinte *workflow* a partir de uma invocação na linha de comando:

1. leitura de um ficheiro contendo um programa imperativo anotado
2. construção da respectiva árvore de sintaxe (AST)
3. geração das condições de verificação (VCs) do programa por travessia da AST
4. apresentação das VCs
5. tentativa de prova de cada VC, utilizando a ferramenta externa
6. apresentação dos resultados da verificação

A aplicação deverá ser validada através de um número razoável de testes.

5 Extensões

Apresenta-se uma lista de possíveis extensões deste trabalho, que devem ser exploradas de forma aberta e livre, por qualquer ordem.

1. Geração de *safety conditions* para operações aritméticas inválidas (divisão por 0) e acesso a posições fora da região alocada dos *arrays*.

2. Sub-rotinas: procedimentos *sem parâmetros*, equipados com *contratos*, devendo as chamadas de procedimento ser tratadas de acordo com os princípios do *design-by-contract*, como descrito em:

   J. B. Almeida, M. J. Frade, J. S. Pinto, and S. M. de Sousa. *Rigorous Software Development: An Introduction to Program Verification.* ISBN: 978-0-85729-017-5. Springer-Verlag London Ltd, 2011.

3. Implementação de um algoritmo de geração de condições de verificação alternativo, baseado na propagação directa de pós-condições.