Universidade da Beira Interior

Departamento de Informática

Arquitectura de execução fiável e mobilidade do código: *Proof Carrying Code*

CLISS: Um compilador de LISS com anotações lógicas

João Gomes

Orientador do projecto:

Simão Melo de Sousa

Setembro 2006

Agradecimentos

Aos meus pais, pelas razões óbvias que justificam a minha existência, mas sobretudo por terem aceite sempre as minhas escolhas académicas menos ortodoxas.
Ao meu irmão, pela presença incondicional.
À grande Desertuna pela música, o prazer, a amizade, as serenatas, os risos, pelos amigos que me espalhou por cada canto, por me ter tirado tanto tempo para estudar, mas especialmente por ter completado o conhecimento científico que a universidade me proporcionou.
Aos companheiros da experiência de ERASMUS, pelas histórias, a amizade, as viagens e os momentos inesquecíveis deste último ano.
À Susana, por isto... mas também por aquilo...
Ao professor Simão pela grande disponibilidade que sempre teve durante este ano, pelo interesse com que recebeu as minhas ambições e as minhas dúvidas e pela orientação indispensável.
À Daniela da Cruz pela inesgotável paciência que demonstrou sempre para responder às minhas intermináveis questões sobre o `LISS`.
Aos colegas de curso, aos que me acompanharam desde o primeiro até ao último momento na universidade, mas também àqueles que os últimos anos me apresentaram. Pelo companheirismo, a partilha e as conversas. Pelas divagações científicas na sala de projecto, mas também pelas menos científicas.
Aos CD's do Peter Cincotti e ao trompete do Miles Davis por manterem a minha sanidade mental durante um Verão sem praia.
E finalmente, ao Bruce, pela presença indispensável nos trabalhos de grupo.

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos

Coco/R Compilers Compiler generating Recursive descent parsers

DBC Design By Contract

JML Java Modeling Language

LISS Language of Integer, Sequences and Sets

PCC Proof-Carrying Code

Glossário

Caduceus Ferramenta que extende o WHY de forma a processar programas C anotados.

Coco/R Gerador de compiladores criado na Universidade de Linz.

COQ Sistema de demonstrações assistidas por computador.

Java Linguagem de programação orientada a objectos, desenvolvida pela SUN.

JML Linguagem de anotações desenvolvida para anotar programas escritos em Java.

LISS Linguagem "brinquedo" que processa inteiros, conjuntos e sequências, criada com objectivos académicos.

LISSOM Arquitectura PCC ao nível do código fonte.

WHY Ferramenta que produz condições de verificação a partir de código anotado.

Capítulo 1

Introdução

Este documento foi elaborado no âmbito do projecto realizado na disciplina de *Projecto II* do curso de Engenharia Informática. Descreve as ideias e ambições previstas no início do projecto, a estrutura do trabalho, os objectivos que pretendia cumprir, as actividades desenvolvidas na sua elaboração, as justificações das decisões tomadas, os resultados obtidos e as conclusões retiradas do trabalho.

Neste capítulo, contextualiza-se e apresenta-se em detalhe o projecto escolhido.

1.1 Contexto

Este projecto surge integrado no projecto `LISSOM`, que tem como objectivo o desenvolvimento duma plataforma que implemente uma arquitectura `PCC` (*Proof Carrying Code*).

O `PCC` é um paradigma para a execução segura de código móvel. O `LISSOM` aborda o `PCC` com base na ideia da validação dos programas ao nível do código fonte. Através de anotações nos programas que indiquem o seu comportamento correcto, é possível utilizar ferramentas para provar a validade do código de um dado programa.

É neste ponto do `LISSOM` que se insere este projecto, tendo o objectivo de fornecer uma linguagem de anotação para a linguagem de programação `LISS` e uma ferramenta capaz de gerar condições de verificação a partir do código anotado.

1.1.1 Área de estudo

Este projecto pertence às áreas de Teoria da Computação e Compiladores. O trabalho desenvolvido está relacionado com programação, anotações de programas, semântica da linguagem, lógica computacional, métodos formais e demonstração assistida por computador.

1.1.2 Objectivos e estrutura do projecto

O projecto tem duas componentes, que contemplam dois objectivos ligados que me propus cumprir.

A primeira diz respeito à definição duma linguagem de anotações para o `LISS`.

A segunda compreende a construção duma ferramenta capaz de gerar condições de verificação a partir de programas anotados utilizando a linguagem anterior.

As duas componentes foram desenvolvidas em várias fases interligadas:

1. aprendizagem e compreensão da linguagem `LISS`;
2. estimativa dos desafios que seria necessário vencer e dos pontos críticos que era necessário estudar em pormenor;
3. definição da melhor abordagem e estratégia para desenvolver a ferramenta para gerar obrigações;
4. aprendizagem das ferramentas necessárias ao desenvolvimento do projecto, especialmente o `Coco/R`;
5. ajustar a linguagem de anotações às necessidades;
6. escrever a ferramenta para gerar obrigações.

Ferramentas utilizadas

Na elaboração deste projecto foram utilizadas as seguintes ferramentas:

- `Coco/R`[1], um gerador de compiladores;
- linguagem de programação `C#` com o IDE *Monodevelop*;

- WHY[4] e Caduceus[2], ferramentas para gerar condições de verificação;
- COQ[3], um sistema de demonstração assistida por computador.

1.1.3 Motivação da escolha

A principal razão para escolha deste projecto foi o facto da proposta ser bastante interessante e desta se envolver num projecto maior, resultante da interacção dos esforços de diferentes pessoas.

Também o facto de ter como objectivo produzir, de forma prática, uma ferramenta que me permitiu utilizar os conhecimentos teóricos, adquiridos durante o projecto do primeiro semestre, tornou especialmente interessante esta escolha.

Como base das justificações já descritas, está o aliciante de trabalhar nalgumas das áreas que me despertaram mais interesse durante a frequência do curso.

1.2 Organização do documento

Este documento está dividido em sete capítulos e um apêndice.

No primeiro capítulo, é apresentado e descrito de forma breve o projecto. Explica-se a sua estrutura e apresentam-se os objectivos a cumprir e o plano dos trabalhos desenvolvidos. É ainda descrito o documento.

No segundo capítulo, descreve-se a correcção de programas a nível do código fonte, explicam-se conceitos necessários para a sua compreensão e apresentam-se ferramentas estudadas.

No terceiro capítulo, descreve-se a linguagem LISS.

No quarto capítulo, descreve-se de forma breve a plataforma LISSOM e o papel nela desempenhado por este projecto.

No quinto capítulo, apresenta-se e descreve-se a linguagem de anotações para o LISS.

No sexto capítulo, descreve-se a ferramenta criada, explica-se o seu desenvolvimento e justificam-se as decisões tomadas.

No sétimo capítulo, apresentam-se em forma de conclusão, as dificuldades do projecto, o trabalho ainda em curso, os esforços futuros e outras considerações finais sobre o projecto.

No apêndice 1, são apresentados alguns esquemas da estrutura sintáctica da linguagem de anotação.

Capítulo 2

Correcção de programas a nível do código fonte

A necessidade de garantir a qualidade, a fiabilidade e a correcção de programas informáticos, vem tornando-se cada vez mais importante. A esta realidade, acrescenta-se um novo facto: já não é suficiente produzir sistemas informáticos que funcionem bem, é necessário garantir a validade dos resultados.

Para garantir a validade dos resultados dum programa, é necessário garantir a correcção do programa. Uma das formas de o fazer, é basear as garantias da correcção não no *output*, mas sim no código fonte.

As secções seguintes explicam, de forma sumária, como é feita a verificação do código fonte dum programa e descrevem algumas das ferramentas utilizadas neste processo.

Para um desenvolvimento mais detalhado deste capítulo, sugiro a leitura do projecto que apresentei no primeiro semestre, *Computer Programs Validation*[18].

2.1 Conceitos preliminares

Antes de explicar como é feita a correcção de programas informáticos[25], é necessário de descrever alguns conceitos essenciais para a compreensão das secções seguintes.

2.1.1 A *Lógica de Hoare*

O cientista britânico C. A. R. Hoare escreveu, em 1969, um documento intitulado "An axiomatic basis for computer programming"[19], onde descreveu a *Lógica de Hoare*.

A *Lógica de Hoare* é um sistema formal desenvolvido por Hoare, com o objectivo de disponibilizar um conjunto de regras lógicas, de forma a avaliar a correcção de programas informáticos através do rigor da lógica matemática.

A peça central da *Lógica de Hoare* é o *triplo de Hoare*. Este triplo descreve a forma como a execução duma parte de código influencia o estado da computação.

O *triplo de Hoare* tem a seguinte forma:

```
{ P }   C   { Q }
```

onde `P` e `Q` são asserções e `C` é um comando. `P` é chamado de pré-condição e `Q` de pós-condição. As asserções são fórmulas na forma de predicados lógicos. Podemos ler este triplo como:

> "Sempre que `P` for válido antes da execução de `C`, então `Q` será válido após a sua execução."

Se `C` não terminar, então `Q` pode ser qualquer asserção. `Q` pode ser definido como falso para descrever o caso em que `C` não termina. Este caso é chamado de "correcção parcial". Se `C` termina e `Q` é verdadeiro, é chamado de "correcção total".

A *Lógica de Hoare* tem axiomas e regras de inferência para todos os construtores duma linguagem imperativa simples. Desde a publicação original de Hoare, foram desenvolvidas, por Hoare e por outros investigadores, regras para outros construtores.

2.1.2 Programas anotados

A anotação de programas[15] assenta no conceito de permitir ao programador comentar o código do seu programa com asserções que, não só descrevem o comportamento desejado do módulo de código, como também facilitam a integração de diferentes módulos, garantindo que estes irão interagir de forma correcta entre si. Estas anotações podem ser vis-

tas como um resumo do módulo de código, onde é descrito aquilo que o programa necessita, o que vai produzir e as alterações que vai provocar.

A anotação dum programa é considerada correcta quando as asserções são demonstráveis segundo a *Lógica de Hoare*.

As anotações podem ser utilizadas para diferentes fins e utilizar diferentes metodologias. Algumas linguagens utilizam anotações para implementar `DBC`[21] (*Design By Contract*), como é o caso do `JML`[7].

Design by contract

Design by contract[21] é um método para desenvolver software, em que o conceito base assenta na ideia de que uma entidade de software (fornecedor) estabelece um "contracto" com outra entidade (cliente). O "contracto" é uma especificação funcional que cria obrigações que o cliente e o fornecedor têm de satisfazer. Para cada módulo de código é criado um "contracto". A execução do programa é vista como a interacção entre os intervenientes do "contracto".

Duma forma geral, os módulos têm pré-condições explícitas que o cliente tem de satisfazer e pós-condições que o fornecedor garante serem satisfeitas após a execução do programa.

Assim, o contracto pode ser descrito da seguinte forma,

> "Se forem cumpridas determinadas pré-condições, é devolvido um resultado que será consistente com determinadas regras. Mas se essas pré-condições não forem satisfeitas, não é garantido qualquer tipo de resultado."

Para informações mais detalhadas sobre `DBC` sugiro a consulta das publicações *Design by Contract with JML*[21] e *JML: A behavioral interface specification language for Java*[20].

2.2 Processo de validação

O processo de validação do código fonte compreende três passos:

- anotar o programar;
- utilizar uma ferramenta de verificação;

- demonstrar as condições de verificação geradas.

Escreve-se o programa contendo anotações lógicas que descrevem pré-condições e pós-condições, invariâncias e asserções ocasionais.

Utiliza-se uma ferramenta que, através do código anotado, verifica a correcção das anotações e gera condições de verificação.

Por último, é necessário demonstrar as condições geradas. Este passo pode ser feito com a ajuda dum sistema de demonstrações.

2.3 Aplicações

Existem algumas ferramentas para gerar condições de verificação através de código anotado. Descrevem-se, de seguida, o `WHY`[4], de Jean-Christophe Filliâtre, e a sua adaptação para linguagem `C`, o `Caduceus`[2] de Jean-Christophe Filliâtre, Claude Marché e Thierry Hubert.

2.3.1 Why

O `WHY`[4, 14] é uma ferramenta que produz condições de verificação a partir de programas anotados. Tem vários pontos em comum com outras ferramentas desenvolvidas para o mesmo propósito, no entanto, tem algumas características que o distinguem das outras.

O ponto mais visível em que se distingue é a sua independência em relação ao sistema de prova. Dito de outra forma, o `WHY` não tem o seu próprio sistema de prova. Em vez disso, permite gerar condições de verificação para sistemas de prova existentes, deixando assim ao utilizador, o privilégio da escolha pelo sistema que ele preferir. Isto é especialmente importante, quando a prova das condições de verificação exige a interactividade com o utilizador, o que, em programas com um mínimo de complexidade, acaba sempre por acontecer.

Actualmente, o `WHY` permite gerar condições de verificação para quatro sistemas de prova, `Coq`[3], `PVS`[11], `Mizar`[9] e `HOLLight`[6]. O mesmo acontece com procedimentos de decisão automática. Em vez de incorporá-los dentro da ferramenta de verificação, utiliza ferramentas existentes como

suporte. Actualmente, o WHY utiliza dois procedimentos de decisão, o Simplify[12] e o Harvey[5].

Tal como acontece com a saída, também no programa de entrada o WHY não está limitado a uma linguagem. Em vez disso, o WHY tem a sua própria linguagem interna. Qualquer outra linguagem existente pode ser compilada para a linguagem interna do WHY.

Esta linguagem é uma pequena linguagem semelhante ao ML[10], com características imperativas, excepções e anotações. Tal como o ML, junta expressões, instruções, variáveis locais e funções numa única classe, o que simplifica a manipulação de símbolos e limita o número de casos a considerar quando se calculam as pré-condições mais fracas ou as condições de verificação. Da mesma forma, as excepções são utilizadas para os casos de terminações inesperadas como return, break ou continue, em linguagens como C ou Java. Desta forma, afasta a necessidade de implementar regras especiais para estes construtores.

Para outros pormenores sobre o WHY, como a exclusão do *aliasing*, o cálculo da pré-condição mais fraca e o descartar automático de condições de verificação, sugiro a leitura do documento *Why: a multi-language multi-prover verification condition generator*[14].

2.3.2 Caduceus

O Caduceus[2, 13, 16] funciona como um compilador de programas C anotados para a linguagem interna do WHY.

O WHY não disponibiliza qualquer tipo de estrutura de dados, mas apenas tipos aplicativos (tipos primitivos como inteiros, booleanos, ... e tipos de dados abstractos) e variáveis mutáveis contendo valores aplicativos. Isto significa que o WHY não tem qualquer conhecimento de *arrays*, estruturas, ponteiros ou pilha de memória. Assim, o mecanismo de tradução necessita de introduzir definições de tipos de dados abstractos que modelem o estado da memória de um dado programa C.

Dessa forma, a semântica das instruções C tem que ser traduzida para instruções WHY que operem nesses tipos de dados abstractos.

O Caduceus suporta uma parte muito significativa da linguagem ANSI C. Todas as estruturas são suportadas, excepto o goto. São suportados programas com ponteiros, incluíndo a aritmética de ponteiros e a possibilidade

do *aliasing* de ponteiros. A principal característica do C não suportada é o *casting* de ponteiros.

O Caduceus é distribuído integrado no WHY. Assim, pode ser utilizado como uma ferramenta independente, recebendo programas C anotados e gerando condições de verificação para um dado sistema de prova, utilizando internamente para isso, o WHY.

Para informação mais detalhada sobre o Caduceus, sugiro a consulta dos documentos *The CADUCEUS verification tool for C programs. Tutorial and Reference Manual*.[13] e *Multi-Prover Verification of C Programs*[16].

2.4 Conclusão

Este capítulo permitiu-nos introduzir os conceitos e os processos gerais utilizados na correcção de programas a nível do código fonte. Nos capítulos seguintes, descrevem-se então os desenvolvimentos específicos envolvidos na correcção de programas escritos na linguagem LISS.

Capítulo 3

Linguagem de programação *LISS*

Neste capítulo descreve-se de forma sumária a linguagem `LISS`. São apresentados a estrutura, os tipos de dados, as diferentes operações e as instruções de controlo.

3.1 *LISS*

`LISS` são as iniciais para `Language of Integer, Sequences and Sets`.

Trata-se duma linguagem de estudo de alto nível, desenvolvida com objectivos pedagógicos, especialmente para o estudo de programação estruturada e imperativa. O seu principal objectivo é servir de caso de estudo para o ensino e a investigação na área de Compiladores.

A linguagem é simples, *case-insensitive* e segue o estilo *Pascal*, com palavras-chave completas e uma estrutura bem definida.

Apesar da sua simplicidade, o *LISS* disponibiliza tipos de dados e estruturas de controlo bastante complexos, habituais em linguagens mais evoluídas, como são exemplo os tipos de dados que implementam listas ligadas ou conjuntos e as estruturas semelhantes a ciclos *ForEach*.

3.2 Estrutura da linguagem

A estrutura dum programa `LISS` é bem definida, recorrendo a palavras-chave para limitar cada parte do programa.

A estrutura base obedece ao seguinte esquema:

```
program <identificador>
{
   Declarations
         ...

   Statements
         ...
}
```

Todas as variáveis utilizadas num programa LISS são obrigatoriamente declaradas imediatamente a seguir à palavra-chave `Declarations`. Opcionalmente, as variáveis podem ser inicializadas ao mesmo tempo que são declaradas. O exemplo seguinte demonstra a declaração de variáveis num programa LISS.

```
         ...
   Declarations
      a := 5, b, c := 0 -> Integer;
      f -> Integer;

   Statements
         ...
```

Todas as restantes operações são escritas após a palavra-chave `Statements`.

3.3 Tipos de variáveis

No LISS existem os seguintes tipos de variáveis:

- Inteiros (*Integer*)
- Booleanos (*Boolean*)
- Sequências Estáticas (*Array*)
- Sequências Dinâmicas (*Sequence*)
- Conjuntos (*Set*)

Ao serem declaradas, todas as variáveis têm atribuído um valor por defeito, `0`, `empty` ou `false`.

Nas subsecções seguintes descreve-se cada um dos tipos de variáveis.

3.3.1 Inteiros

Um inteiro é declarado da seguinte forma:

```
a := 5, b, c := 0 -> Integer;
```

Estão disponíveis as seguintes operações com inteiros:

- Operações algébricas: soma (+), subtracção (-), multiplicação (*) e divisão (/);
- Operações relacionais: ==, !=, <, >, <=, >=;
- Atribuição: =
- Sucessor (`succ`), incrementa o valor uma unidade e predecessor (`pred`), decrementa o valor uma unidade;
- Operações de entrada e saída:leitura (`read(a)`) e escrita (`write(a)` ou `writeln(a)`);
- Não está definida a divisão por zero.

3.3.2 Sequências estáticas

As sequências estáticas são semelhantes aos habituais *arrays* multidimensionais. São declaradas da seguinte forma:

```
vector := [1,2,3] -> Array size 5;
vector := [
          [[1],[4]],
          [[2,3],[5]]
          ] -> Array size 4,3,2;
```

Estão definidas as seguintes operações com sequências estáticas:

- Indexação:

```
array[i] = array[1] + array3[k];
array[i,j,k] = a*b;
```

- Atribuição:
```
array2 = array1;
```

- Comprimento:
```
intA = length(array1);
```

- Operações de saída:
```
write(array1);
writeln(array2);
```

Todos os índices têm de estar dentro do intervalo de valores definido na declaração do *array*.

O número de índices tem de corresponder às dimensões do *array*.

3.3.3 Sequências Dinâmicas

As sequências dinâmicas são listas ligadas de inteiros, sem tamanho fixo definido.

São declaradas da seguinte forma:
```
seq := <<1,2,3,50,6,23,3>> -> Sequence;
```
Estão definidas as seguintes operações com sequências:

- Inserção e remoção:
```
cons(2, seq);
del(3, seq);
```
Os valores inseridos são colocados no final da sequência;

- Cabeça e cauda:
```
i = head(seq);
seq2 = tail(seq);
```

- `IsMember` e `IsEmpty`:
```
if (IsMember(50, seq)) ...
While (IsEmpty(seq)) ...
```

- Indexação:
```
i = seq[2];
```

- Comprimento:
```
i = length(seq);
```

- Atribuição por referência, em que uma sequência passa a ser exactamente a outra:
```
seq2 = seq1;
```
- Atribuição por cópia, em que o conteúdo duma sequência é copiado para a outra:
```
copy(seqinit, seqdest);
```
- Operações de saída:
```
write(seq1);
writeln(seq2);
```

3.3.4 Conjuntos

Os conjuntos são definidos em compreensão:
```
conj = {x | x >= 100 && x < 500}
```
São declarados da seguinte forma:
```
conj1, conj2 := {x | x >= 100 && x < 500}, conj3 -> Set;
```
Estão definidas as seguintes operações para conjuntos:

- Atribuição:
```
conj2 = conj1;
```
- União e intersecção:
```
conj1 = conj2 ++ conj3;
conj4 = conj2 ** conj3;
```
- Membro (`in`):
```
While (n in conj) ...
```
- Operações de saída:
```
write(conj1);
writeln(conj2);
```

3.3.5 Booleanos

As variáveis Booleanas podem tomar os valores `false` e `true`. Podem ser declaradas da seguinte forma:
```
bool1, bool2 = true -> boolean;
```
Estão definidas as seguintes operações para booleanos:

- Atribuição:
```
bool1 = bool2;
```
- Conjunção:
```
bool1 && bool2
```
- Disjunção:
```
bool1 || bool2
```
- Negação:
```
not bool1
```
- Operações de saída:
```
write(bool1);
writeln(bool2);
```

3.3.6 Subprogramas

O LISS permite declarar subprogramas com zero ou mais argumentos de entrada. Esses subprogramas podem ou não devolver um valor. Podem ser declarados ao mesmo nível ou aninhados uns dentro dos outros.

Qualquer variável declarada dentro de um subprograma, torna-se "global" para todos os subprogramas também declarados dentro desse subprograma ou em níveis mais baixos ainda.

Os subprogramas podem chamar-se a si próprios tornando-se recursivos.

Exemplo da declaração de um subprograma:

```
1          ...
2  Declarations
3
4     d := [10,20,30,40], ev -> array size 4;
5     fac := 6 -> integer;
6
7     subProgram sub()
8     {
9        Declarations
10          c -> integer;
11
```

```
12          subprogram sub2(m -> array size 4) :: integer
13          {
14             Declarations
15                res -> integer;
16
17             Statements
18                res = m[2] + fac;
19                return res;
20          }
21
22       Statements
23          c = sub2(d);
24          writeln(c);
25
26          return d;
27    }
28
29 Statements
30          ...
```

3.4 Instruções de controlo

O LISS tem três tipos de instruções de controlo semelhantes às linguagens mais evoluídas: `If`, `While` e `For`.

3.4.1 Instrução *If*

A instrução `If` tem a seguinte estrutura:

```
   if (<expressão booleana>) Then
   {
      <instruções>
         ...
   }

 [ Else
   {
```

```
      <instruções>
   }
]
```

3.4.2 Instrução *While*

A instrução `While` tem a seguinte estrutura:

```
While (<expressão booleana>)
{
   <instruções>
      ...
}
```

3.4.3 Instrução *For*

A instrução `For` tem as seguintes estruturas:

```
for <var> in <lim1>..<lim2> [stepup|stepdown n]
                            [satisfying <expressão booleana>]

for <var> inArray <array1> [satisfying <expressão booleana>]
```

No primeiro exemplo, no caso de a opção utilizada ser `stepup`, o ciclo iniciar-se-á em `lim1` e irá até `lim2`. No caso de ser utilizado o `stepdown`, começará em `lim2` até `lim1`. Caso o *salto* do ciclo não seja especificado, é assumido `stepup 1`. A cláusula `satisfying` é opcional.

3.5 Comentários

Em `LISS`, os comentários respeitam as seguintes estruturas:

```
// comentário   <EOL>

-- comentário   <EOL>

/* comentário - linha 1
   comentário - linha 2
```

```
    ...
comentário - linha n   */
```

3.6 Conclusão

A sua simplicidade aliada aos mecanismos evoluídos que disponibiliza, faz do LISS uma linguagem de estudo bastante interessante. É esta a linguagem utilizada na construção dos programas verificados pela arquitectura LISSOM descrita no próximo capítulo.

Capítulo 4

LISSOM

O LISSOM[17, 22, 23] é um projecto para implementar uma arquitectura PCC (*Proof Carrying Code*).

As arquitecturas PCC tradicionais baseavam os seus esforços de certificação e validação dum programa, no seu *output*. No entanto, recentemente tem havido uma evolução no sentido da validação dum programa ser feita ao nível do seu código fonte.

O LISSOM assenta em dois princípios em relação ao PCC:

- basear a verificação dos programas no código-fonte;
- reutilizar ao máximo as ferramentas já existentes. Existe uma grande variedade de ferramentas para a validação de código fonte. Essas ferramentas têm já um interessante nível de desenvolvimento e são utilizadas por uma comunidade de utilizadores experientes.

4.1 Processo de prova no *LISSOM*

O processo de prova no LISSOM é constituído por duas partes.

A primeira parte diz respeito às técnicas e ferramentas utilizadas na especificação e na construção das demonstrações.

A segunda parte está relacionada com a validação das construções formais realizadas.

Estas são as bases da plataforma LISSOM, descritas no relatório do primeiro semestre do Daniel Martins[23]:

- uma linguagem de especificação de políticas baseada na linguagem de especificação do sistema COQ[3];
- um sistema de anotação de tipo JML[7] e um sistema de verificação de código fonte baseado na ferramenta WHY[4] e no sistema COQ;
- objectos de prova COQ como certificados;
- um compilador para a linguagem LISS com tradução de certificados;
- o verificador de provas do sistema COQ;
- uma máquina virtual com pilhas.

Desta forma, o trabalho para desenvolver a plataforma LISS, descrito no documento *Lissom, a Source Level Proof Carrying Code Platform*[17], está estruturado nos seguintes passos:

1. desenhar uma linguagem de anotação para a linguagem LISS;
2. extender a ferramenta WHY para contemplar esta linguagem de anotação, permitindo ao WHY gerar condições de verificação para o código fonte;
3. desenhar um sistema de prova para a linguagem LISS, integrado no COQ;
4. extender o compilador LISS de forma a ter capacidade para traduzir certificados;
5. desenhar um sistema de prova para a máquina virtual e a sua linguagem, integrado no COQ;
6. desenhar um gerador de condições de verificação para código compilado.

O projecto que desenvolvi, foca os pontos 1 e 2 descritos. Tem como objectivo apresentar uma linguagem de anotação (descrita no capítulo 5) e uma ferramenta para gerar condições de verificação a partir dum programa anotado (descrita no capítulo 6).

Estas duas funcionalidades são necessárias à implementação do módulo VCGEN, imprescindível na construção da plataforma PCC, implementada pelo LISSOM.

4.2 Geração de certificado ao nível do código fonte

Assim, o papel deste projecto foi fornecer ao LISSOM um gerador de certificados ao nível do código fonte e uma linguagem para anotar os programas LISS.

Para gerar o certificado de prova ao nível do código fonte, foi utilizada a ferramenta WHY[4].

A ideia inicial foi construir um módulo para o WHY que estendesse a ferramenta para suportar a linguagem LISS. Algo semelhante ao que foi feito com o Caduceus para a linguagem C. No entanto, como será explicado no capítulo 6, esta tarefa foi concluída construindo um compilador de LISS para C.

A linguagem de anotação, descrita no capítulo 5, é uma adaptação da linguagem JML[7]. No capítulo 6, descreve-se todo o trabalho desenvolvido na produção da ferramenta para gerar os certificados de prova, explicam-se as decisões tomadas e as alterações à ideia inicial.

4.3 Conclusão

Descreveu-se de forma resumida a plataforma LISSOM e o papel nela desempenhado por este projecto.

Para informação mais detalhada sobre a plataforma LISSOM, sugiro a consulta do projecto de primeiro semestre do Daniel Martins[23].

Capítulo 5

Linguagem de anotações lógicas para programas escritos em *LISS*

As anotações lógicas dos programas LISS são escritas numa linguagem definida para o efeito. Neste capítulo descreve-se essa linguagem de forma sumária.

O JML[7] (*Java Modeling Language*), uma linguagem especificamente criada para anotar programas Java, é utilizado para escrever as anotações de programas para serem verificados pela ferramenta Krakatoa[8].

Para anotar os programas a serem verificados pelo Caduceus[2], é utilizada uma linguagem muito parecida com o JML.

Uma vez que o CLISS traduz os programas LISS para C, era útil utilizar a mesma linguagem para comentar os programas LISS. Havia a possibilidade de ser necessário ajustá-la ligeiramente para se adaptar à linguagem LISS. No entanto, essa necessidade revelou-se mínima e todos os ajustamentos são feitos pelo CLISS.

De seguida, descreve-se a linguagem.

5.1 Linguagem de anotações utilizada

As anotações são inseridas em comentários no código fonte da forma `/*@...*/` ou numa só linha `//@...`.

A especificação duma função é feita exactamente antes da função e é

formada por três cláusulas opcionais: `requires` descreve as pré-condições, `assigns` descreve os *efeitos colaterais* e `ensures` descreve as pós-condições. É também possível, através da palavra-chave `invariant`, definir propriedades que se mantêm válidas, por exemplo, em cada passagem de um ciclo, ou definir propriedades para uma variável que se mantêm sempre que a variável é utilizada por uma função. Neste caso, informalmente, essas condições de invariância são adicionadas às pré-condições das funções que utilizam a variável.

5.1.1 Anotações

As anotações são predicados de lógica de primeira ordem construídos a partir de átomos e conectores como `&&`, `||`, `!`, etc. Os átomos são construídos a partir de expressões `C` livres de *efeitos colaterais*.

Desta forma, as anotações têm a seguinte estrutura:

```
/*@ requires <expressão lógica>
  @ assigns <variáveis>
  @ ensures <expressão lógica>
  @*/
subprogram ...
```

A cláusula `requires` descreve um estado da memória que tem que ser válido sempre que a função se inicia.

A cláusula `assigns` descreve um conjunto de localizações da memória que podem ser modificadas pela função. De outra forma, diz que qualquer endereço da memória já alocado que não seja mencionado nesta cláusula, não será alterado pela função.

A cláusula `ensures` descreve um estado da memória que tem de ser válido após a função terminar.

5.1.2 Predicados

Para além dos predicados definidos na linguagem de anotação, o utilizador pode definir outros predicados que pode depois utilizar nas anotações.

A estrutura para definir um predicado é a seguinte:

```
/*@ predicate <identificador>(<tipo> <argumento>, ...)
  @    {
```

```
  @       <instrução lógica>
  @    }
  @*/
```

O seguinte exemplo foi retirado do manual do Caduceus[13]:

```
31  typedef struct {
32     int balance;
33  } purse;
34
35  /*@ predicate purse_inv(purse *p)
36    @    { \valid(p) && p->balance >= 0 }
37    @*/
38
39  /*@ requires purse_inv(p) && s >= 0
40    @ ensures purse_inv(p) &&
41    @         p->balance == \old(p->balance) + s
42    @*/
43  void credit(purse *p,int s) {
44     p->balance = p->balance + s;
45  }
46
47  /*@ requires purse_inv(p) && 0 <= s <= p->balance
48    @ ensures purse_inv(p) &&
49    @         p->balance == \old(p->balance) - s
50    @*/
51  void withdraw(purse *p,int s) {
52     p->balance = p->balance - s;
53  }
```

5.1.3 Axiomas e funções lógicas

Tal como foi descrito para os predicados, é também possível introduzir novos axiomas e funções lógicas.

Esta é a estrutura utilizada na sua declaração:

```
/*@ logic <tipo> <função>(<tipo> <argumento>, ...)
                                  reads <localização>*/
```

```
/*@ axiom <identificador>:
  @   <instrução lógica> */
```

Exemplos relacionados podem ser encontrados no manual do Caduceus[13].

5.1.4 Invariância

As anotações de invariância são asserções que têm que ser válidas à entrada e à saída das funções. Têm a seguinte estrutura:

```
/*@ invariant identificador :
  @       <expressão lógica>
  @*/
```

5.1.5 Asserções intermediárias

Há asserções que podem ser colocadas em qualquer local do programa. Descrevem uma propriedade que é válida a partir do ponto onde é declarada. O Caduceus gera uma obrigação para provar a sua validade. Respeitam a seguinte estrutura:

```
//@ assert <expressão lógica>
```

É também possível dar especificações a blocos de instruções como ciclos. A sintaxe é a mesma da especificação de funções. A diferença destes para a anotação `assert`, é que estes se portam como uma *caixa negra*, o cálculo da pré-condição mais fraca não entra dentro destas instruções.

5.2 Conclusão

Este capítulo descreveu a linguagem que será utilizada para anotar os programas escritos em LISS.

No apêndice A está descrita uma lista das palavras-chave da linguagem de anotações, a sua sintaxe completa e a estrutura das anotações na linguagem C.

Para uma descrição da linguagem, mais profunda e acompanhada de exemplos, sugiro a consulta do manual do Caduceus[13].

Descrita esta linguagem, passa-se então, no capítulo seguinte, à descrição da ferramenta criada.

Capítulo 6

CLISS: compilador de *LISS* com anotações lógicas

A ideia inicial para construir um gerador de condições de verificação para o LISS, seria criar um módulo ou uma extensão do WHY[4], semelhante ao Caduceus[2]. Essa ferramenta permitiria receber os programas anotados e gerar as condições de verificação.

No entanto, esta abordagem do projecto levantava várias dificuldades, sobretudo na representação do modelo da memória no WHY. Tornava-se um trabalho imprevisível e sem garantias que pudesse ser terminado no semestre actual. Assim, optou-se por uma solução mais de acordo com os princípios do LISSOM, a reutilização de ferramentas existentes e já experimentadas. O objectivo tornou-se então construir um compilador de LISS para C. Depois de compilado o programa pode então ser verificado pelo Caduceus. O Caduceus gera condições de verificação que depois são demonstradas utilizando o sistema de prova COQ.

Desta forma, tornou-se vantajoso que a linguagem de anotação fosse muito semelhante à que era utilizada para anotar os programas verificados pelo Caduceus. Assim, o código gerado pelo compilador, guarda as anotações originais do programa LISS e acrescenta-lhe algumas outras necessárias para especificar partes de código criado para modelar algumas instruções LISS.

6.1 Desenvolvimento do *CLISS*

6.1.1 Gerador de compiladores *Coco/R*

O `Coco/R`[1, 24] é um gerador de compiladores que recebe uma gramática de atributos duma linguagem de programação e gera um *scanner* e um *parser* descendente recursivo para essa linguagem.

Para cada produção descrita na gramática, o `Coco/R` cria um método na classe *parser*.

O `Coco/R` gera também um *scanner* que lê uma *stream* de código e devolve uma *stream* de *tokens* para o parser.

O utilizador tem que criar a classe principal que chama o *parser*, tal como classes semânticas utilizadas pelas acções semânticas das produções descritas na gramática.

Para uma descrição detalhada do `Coco/R`, sugiro a consulta do manual[24].

6.1.2 Estrutura do desenvolvimento

O meu trabalho baseou-se, numa primeira fase, num compilador de `LISS` para uma máquina virtual, desenvolvido pela Daniela da Cruz, da Universidade do Minho. No entanto, as especificidades da tradução, bem como a modelação de tipos de dados e estruturas não existentes na linguagem `C`, implicavam que a própria descrição da gramática de atributos fosse feita de forma diferente. Obviamente, isto implicou também uma utilização diferente das restantes classes. Algumas classes deixaram de fazer sentido e foram removidas, outras foram criadas e as restantes foram reescritas para disponibilizarem os métodos que eram necessários ao novo compilador.

O compilador foi criado em `C#`, utilizando o `Coco/R` para gerar a classe *parser* e a classe *scanner*.

Foram criadas mais cinco classes para as acções semânticas do *parser*:

- `Liss.cs`, é a classe principal que lê os argumentos de entrada, recebe o ficheiro com o código fonte e chama o *scanner* e o *parser*;
- `SymbolTable.cs`, implementa uma tabela dos símbolos da linguagem e disponibiliza métodos para manipular a tabela;

- `Entry.cs`, implementa os objectos guardados na tabela de símbolos e disponibiliza métodos para manipular cada entrada;
- `VM_Code.cs`, disponibiliza métodos para gerarem código na linguagem alvo;
- `Resolver.cs`, disponibiliza métodos relacionados com resolução de conflitos.

6.1.3 Funcionamento interno

O ficheiro contendo o programa `LISS` é lido pela classe `Liss.cs`. Esta cria objectos das restantes classes e envia o ficheiro para a classe `Scanner.cs`, que cria uma *stream* de tokens e a envia à classe `Parser.cs`. Esta classe irá então produzir as acções semânticas previstas para cada *token*.

O compilador cria uma tabela de símbolos onde vão sendo guardadas e actualizadas as variáveis, funções e tipos de dados. Os símbolos são mantidos na tabela enquanto o alcance do subprograma onde foram declarados não termina. São guardadas várias características de um símbolo, o nome, o tipo, a classe, o endereço, os valores iniciais com que foi declarado, o tamanho e o nível em que aparecem no programa.

As acções semânticas do *parser* utilizam, na sua generalidade, os métodos da classe `SymbolTable.cs` para produzirem o programa alvo.

Também a classe `VM_Code.cs` é utilizada pelas classes `Parser.cs` e `SymbolTable.cs` para gerar código `C`.

6.1.4 Modelação de dados e respectivas operações

Apesar de ser uma linguagem simples, o `LISS` tem tipos de dados que não existem no `C`. Se os tipos de dados `Integer` ou `Array`, têm uma tradução mais óbvia, o mesmo não acontece com os tipos `Set`, `Sequence` e `Boolean`.

Assim, para estes últimos, foi necessário criar bibliotecas que definem estruturas capazes de modelar estes tipos de dados, bem como as operações que o `LISS` disponibiliza para eles.

liss_bool.h Foi definida a instrução de escrita, capaz de devolver valores do tipo `false` e `true`. O compilador trata da tradução dos valores utilizando inteiros para definir estes valores lógicos.

liss_list.h De forma a representar as sequências dinâmicas do LISS, foram definidas estruturas de listas ligadas da seguinte forma:

```
typedef int elem;

typedef struct node *link;

typedef struct node {
        elem value;
        link next;
                } listNode;

typedef struct ls {
        int size;
        link head;
        link last;
                } listst;

typedef listst * list;
```

Foram ainda definidas operações, capazes de representarem as funções (definidas no capítulo 3) que o LISS disponibiliza para sequências dinâmicas.

liss_set.h Definir um conjunto em C não é tão óbvio. A solução escolhida baseia-se na utilização de árvores binárias para representar a fórmula que define o conjunto.

Por exemplo, o conjunto LISS,

```
{ x | x > 10 || x >= -8 && x < 3 }
```

pode ser representado na estrutura em árvore ilustrada na imagem 6.1.

Esta é a estrutura definida:

```
54 typedef char * sElem;
55
56 typedef struct snode * set;
57
58 typedef struct snode {
```

```
59          sElem value;
60          set left;
61          set right;
62                  } tNode;
```

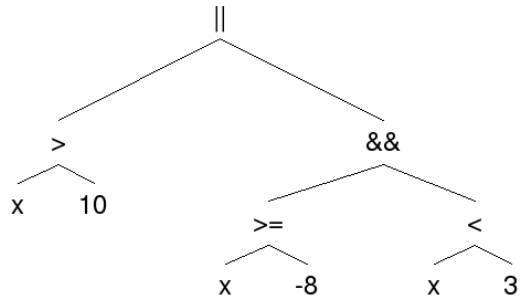

Figura 6.1: *Representação em árvore dum conjunto* LISS

Nesta biblioteca foram ainda definidas as funções (descritas no capítulo 3) que o LISS disponibiliza para sequências dinâmicas.

6.1.5 Pontos críticos na tradução dos programas *LISS*

Pontos críticos relacionados com a linguagem *LISS*

A especificidade de algumas construções da linguagem LISS e as diferenças significativas com a linguagem C, apresentaram diversos pontos críticos na tradução do LISS.

Como exemplo dessas dificuldades e com o objectivo de mostrar as soluções que foi necessário criar para cumprir os desafios que a tradução da linguagem foi apresentando, descreve-se o problema dos subprogramas, um dos principais pontos críticos na tradução dos programas LISS.

O LISS permite declarar os subprogramas juntamente com as restantes variáveis e aninhá-los uns dentro dos outros. Esta característica não é suportada pelas normas ANSI da linguagem C. Assim, foi necessário declarar as funções C fora do programa principal. Este facto levantava problemas quando as funções utilizavam variáveis declaradas fora da função, em qualquer nível anterior das funções aninhadas.

Uma solução possível, seria declarar todas as variáveis como sendo globais. No entanto, algo que foi preciso ter sempre presente durante a elaboração do projecto, é que a resultado do compilador não se destinava simplesmente a ser um programa `C`, mas sim um programa que pudesse ser verificado pelo `Caduceus`. Como o `Caduceus` não funciona com variáveis globais, esta não era na verdade, uma solução aceitável.

Assim, a solução implementada passou por adicionar aos argumentos das funções todas as variáveis que eram utilizadas dentro dela, mas que não eram lá declaradas. Estas variáveis passadas como argumento, tinham que ser, obviamente, passadas por referência para que tivessem um comportamento semelhante a variáveis globais. Estas variáveis eram guardadas num `ArrayList` na entrada correspondente à função, na tabela de símbolos. Assim, cada vez que uma função `f` era chamada por uma outra função `g`, eram acrescentadas aos argumentos da chamada da função `f`, as variáveis que `f` necessitava. Se as variáveis também não se encontrassem declaradas em `g`, eram também acrescentadas aos argumentos de `g`. E assim sucessivamente.

Ainda assim, sobrava um outro problema: as funções recursivas. Imaginemos o seguinte caso:

```
63    ...
64  subprogram f(x -> integer) :: boolean
65  {
66     Declarations
67        a := 4, res -> integer;
68
69     Statements
70        if ( z < ( a + x) ) Then
71        {
72           pred(x);
73           res = f(x);
74        }
75        Else
76        {
77           res = y + x;
78        }
79
80        return res;
```

```
}
   ...
```

Neste caso, as variáveis y e z são declaradas fora da função. Assim, quando traduzida para C, a função tem que receber as duas variáveis como argumentos. No entanto, a função vai ser lida e ao mesmo tempo vai ser convertida para C. No momento em que a função se chama a si própria, apenas a variável z tinha sido identificada e colocada na lista de variáveis a passar como argumento. Deste modo, o resultado seria o seguinte em C:

```
   ...
int f(int x, *z, *y)
{
   int a = 4, res;

   if (*z < (a + x))
   {
      x--;
      res = f(x, &(*z));
   }
   else
   {
      res = x + *y;
   }

   return res;
}
   ...
```

Como pode ser observado, existe um erro na chamada da função.

A solução passou por, sempre que uma função se chamava a si própria, os argumentos da chamada da função eram substituídos por um *string* de controlo e guardados num `ArrayList`. Após ter sido terminada a declaração da função, o código produzido era lido e a string de controlo era substituída pela declaração completa dos argumentos.

Especificidades e restrições do código *C* para o *Caduceus*

Outro dos pontos críticos, são as condicionantes impostas pelo `Caduceus`.

O Caduceus impõe algumas limitações ao código C, como por exemplo, o *casting* de ponteiros e a utilização da função malloc restringida a alguns casos. No entanto, nenhuma destas restrições interferia com o desenvolvimento deste projecto.

Ainda assim, existem algumas particularidades com o qual o Caduceus não lida bem. As bibliotecas C *standard*, não são suportadas pelo Caduceus. Este facto traz problemas à utilização das funções de escrita e leitura. A solução seria, em vez da referência à biblioteca stdio.h, introduzir apenas os cabeçalhos das funções. No entanto, uma vez que estas funções têm um número de argumentos variável, também não são suportadas pelo Caduceus.

Com a função malloc, o problema é ligeiramente diferente. O Caduceus tem o seu próprio suporte interno para a função. No entanto, este entra em conflito com qualquer declaração explícita no código, do protótipo da função. Assim, havia uma escolha a fazer: gerar um programa C utilizando, o malloc, que pudesse ser compilado por um compilador de C, ou gerar um programa que pudesse ser verificado pelo Caduceus.

A solução para estes dois problemas foi criar uma opção no compilador que permitisse gerar código para um compilador C ou código preparado para o Caduceus. Assim, o código gerado pelo compilador com a opção -caduceus produz código sem recurso a bibliotecas *standard*, com todas as instruções de leitura e escrita comentadas, de forma a não interferirem com o funcionamento normal do Caduceus. Foram ainda criadas versões das bibliotecas liss de onde foram removidas as referências a bibliotecas *standard*.

Na verdade, estas limitações vão de encontro à ideia de que é importante ter bons hábitos de programação, criando programas modulares e separando as funções de entrada e saída, das restantes.

6.2 Indicações de utilização

O CLISS corre na linha de comandos obedecendo à seguinte estrutura:

```
$ cliss.exe <ficheiro.liss>
              [ <directoria de destino> ] [ -caduceus ]
```

O resultado será um programa C. Este poderá depois ser utilizado pelo Caduceus da seguinte forma:

```
$ caduceus <ficheiro.c>

$ make -f ficheiro.makefile coq
```

Isto irá criar uma série de condições de verificação que podemos demonstrar utilizando o COQ.

6.3 Exemplo de utilização

Como exemplo básico de utilização do CLISS, vamos utilizar um exemplo do manual do Caduceus[13], de um programa para calcular o máximo de dois valores, desta vez escrito em LISS.

Este é o programa em LISS, anotado:

```
101 program calculoMax {
102    Declarations
103       a := 2, b := 4, c -> Integer;
104
105       /*@ ensures
106         @ \result >= x && \result >= y &&
107         @ \forall int z; z >= x && z >= y => z >= \result
108         @*/
109       SubProgram max(x -> integer; y -> integer) :: integer
110       {
111          Declarations
112             res -> integer;
113
114          Statements
115             if (x >= y) Then
116             {
117                res = x;
118             }
119             Else
120             {
121                res = y;
```

```
122             }
123
124          return res;
125       }
126
127    Statements
128       c = max(a,b);
129       writeln(c);
130 }
```

Fazendo:

```
$ ./cliss.exe max.liss -caduceus
```

o resultado é o ficheiro C:

```
/* THIS FILE WAS GENERATED TO BE USED IN CADUCEUS
 *
 */
#include "liss_caduceus_list.h"
#include "liss_caduceus_set.h"

 /*@ ensures
   @ \result >= x && \result >= y &&
   @ \forall int z; z >= x && z >= y => z >= \result
   @*/
int max(int x, int y)
{
   int res = 0;

   if ((x >= y))
   {
      res = x;
   }
   else
   {
      res = y;
   }
```

```
   return res;
}

void main()
{
   int a = 2, b = 4, c = 0;

   c = max(a,b);

   /* THIS CODE LINES WILL NOT BE READ BY CADUCEUS
   printf("%d", c);
   printf("\n");
   */
}
```

Após a utilização do Caduceus, basta utilizar o makefile para gerar as condições de verificação para o COQ e completar a demonstração utilizando o COQ. Como se trata dum exemplo simples, a demonstração fica completa sem ser necessária a interacção do utilizador.

As figuras 6.2, 6.3 e 6.4 ilustram todo o processo.

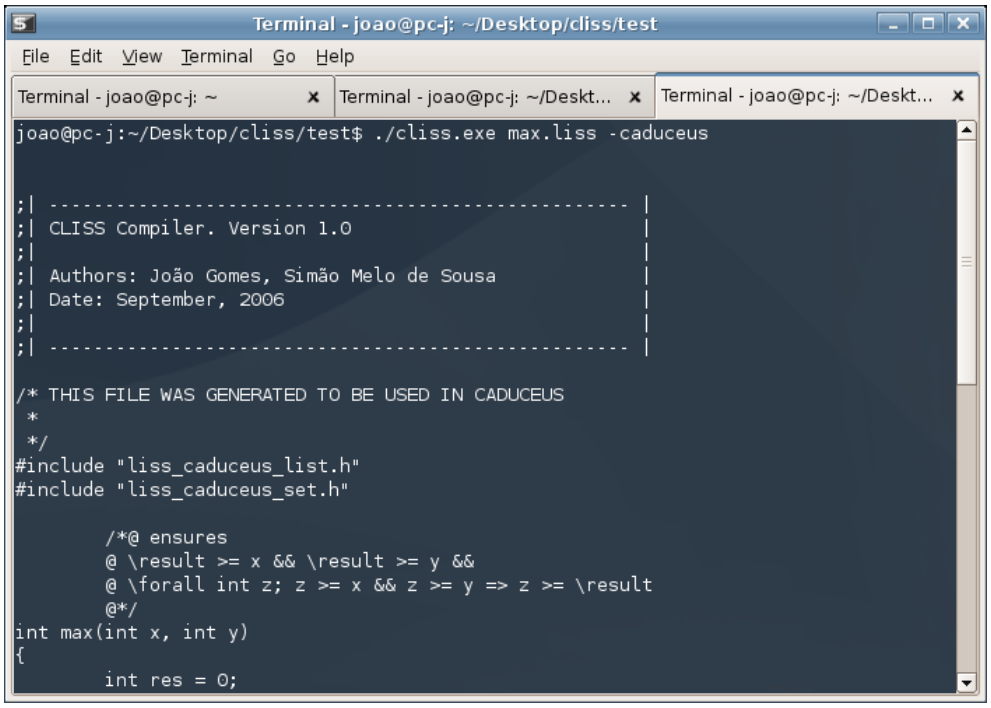

Figura 6.2: *Geração do ficheiro C anotado.*

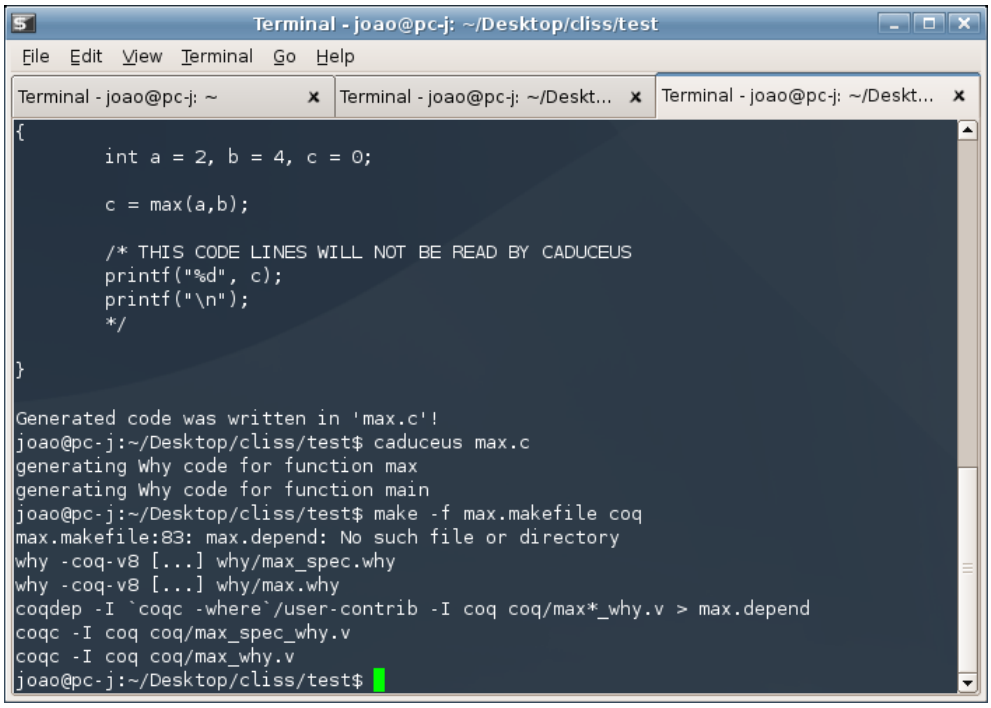

Figura 6.3: *Geração das condições de verificação.*

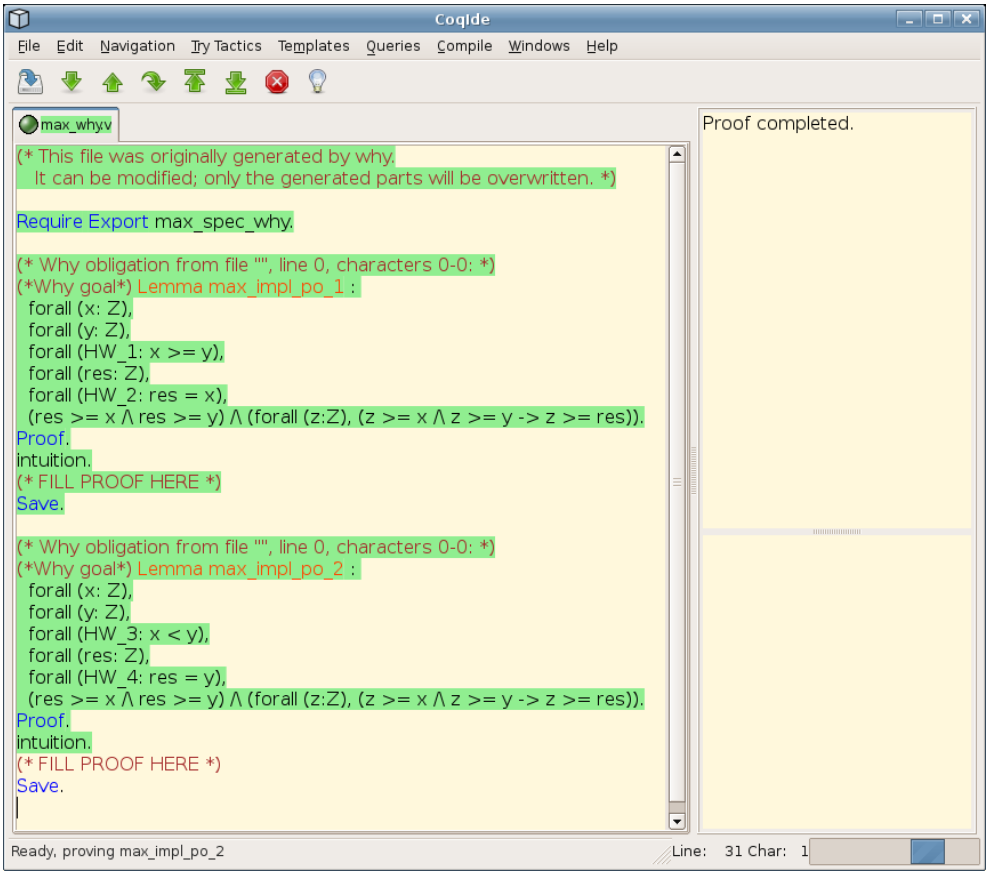

Figura 6.4: *Demonstração no* COQ.

Capítulo 7

Conclusão

Apresentam-se neste capítulo, os resultados do projecto, bem como as dificuldades e a aprendizagem obtida. Apresenta-se ainda o trabalho que se encontra em desenvolvimento e as ideias para o futuro.

7.1 Resultados

No final do projecto, os dois objectivos descritos no início foram cumpridos. Tal como está definido nos objectivos da plataforma `LISSOM`, privilegiaram-se as soluções que utilizassem ferramentas já existentes e experimentadas com bons resultados.

Foi apresentada uma linguagem de anotação adaptada da linguagem de anotação usada nos programas verificados pelo `Caduceus`.

Foi desenvolvido um compilador de `LISS` para `C`, que traduz o código `LISS`, de forma a que o resultado seja utilizado no `Caduceus` com o objectivo de gerar condições de verificação para o programa inicial, fazendo os ajustes necessários para o código `C`, nas anotações do programa.

No entanto, alguns pormenores carecem de um maior desenvolvimento e aperfeiçoamento.

7.2 Dificuldades

O projecto que me propus desenvolver, não era um projecto clássico de desenvolvimento de software. Era antes, constituído por uma importante

componente de investigação. Um dos passos, não só dessa investigação, mas do próprio projecto, era precisamente descobrir quais os problemas que iria enfrentar no desenvolvimento da ferramenta de verificação.

Assim, à medida que esse trabalho inicial de investigação foi apresentando resultados e mostrando os desafios que se impunham, foram-se fazendo escolhas. Essas escolhas não foram sempre no sentido da solução mais perfeita, mas sim de reaproveitar ferramentas existentes, de forma a apresentar resultados no final do semestre.

Dessa forma, as opções de estratégia tomadas, tiveram em conta, não só os conhecimentos científicos, mas também o contexto em que o projecto foi realizado (um projecto de final de curso) e os resultados que era possível obter dentro dos prazos previstos.

O facto de não existir nenhuma descrição completa, detalhada e rigorosa de todas as funcionalidades da linguagem `LISS`, foi também uma dificuldade que tornou o processo um pouco mais lento.

As diferenças sintácticas e semânticas entre as linguagens `C` e `LISS` fizeram do trabalho de geração de código, mais do que uma tradução, sendo necessário criar soluções de alguma complexidade. A estas diferenças entre as linguagens, juntaram-se algumas limitações que o `Caduceus` impõe ao código `C` e que tornam o processo de tradução com o objectivo de criar ficheiros de entrada para o Caduceus, bastante mais complicado.

7.3 Trabalho em Curso

Continua em curso o apuramento da linguagem de anotações adaptada, bem como os testes à ferramenta para identificar eventuais falhas ou insuficiências não detectadas.

7.4 Esforços futuros

Como esforço futuro, considero de grande importância a anotação das bibliotecas `liss` criadas para modelar os tipos de dados do `LISS` e as respectivas operações.

7.5 Considerações pessoais sobre o projecto

As áreas de Teoria da Computação e Compiladores são bastante trabalhosas, mas de um interesse bastante evidente.

A experiência de desenvolver um compilador, mostrou-me as dificuldades que tarefas, muito trabalhosas e nem sempre óbvias, apresentam. Mas despertou-me também o interesse em questões complexas e importantes, onde a criatividade de soluções desenvolvidas para problemas muito complexos, tornam aliciante o trabalho nesta área.

Termino este relatório com a sensação bastante evidente de que mais um semestre de trabalho teria apurado, de forma bastante significativa, a qualidade dos resultados obtidos.

Apêndice A

Estrutura sintáctica da linguagem de anotação

Os comentários são limitados por `(*` e `*)`.

As tabelas seguintes foram retiradas do manual do `Caduceus`[13].

A tabela A.1 contém a lista das palavras-chave que fazem parte da linguagem:

`\forall`	`\exists`	`int`	`float`	`decreases`
`\true`	`\false`	`if`	`then`	`else`
`invariant`	`variant`	`for`	`label`	`assert`
`requires`	`ensures`	`assigns`	`logic`	`axiom`
`predicate`	`\result`	`\old`	`\block_length`	`\null`
`reads`	`\valid`	`\valid_index`	`\valid_range`	`\fresh`
`\base_addr`	`\nothing`			

Tabela A.1: *Palavras chave da linguagem de anotação.*

A tabelas A.2 e A.3 contêm a sintaxe da linguagem de anotações.

A tabelas A.4 e A.5 contêm a estrutura das anotações no código `C`.

⟨*term*⟩	::==	⟨*constant*⟩
	|	⟨*term*⟩ ⟨*arith_op*⟩ ⟨*term*⟩
	|	- ⟨*term*⟩ | + ⟨*term*⟩
	|	* ⟨*term*⟩
	|	⟨*term*⟩ -> ⟨*identifier*⟩
	|	⟨*term*⟩ . ⟨*identifier*⟩
	|	⟨*identifier*⟩ (⟨*term*⟩$^{+}_{,}$)
	|	⟨*term*⟩ [⟨*term*⟩]
	|	(⟨*term*⟩)
	|	(⟨*logic_type*⟩) ⟨*term*⟩
	|	\old (⟨*term*⟩)
	|	\at (⟨*term*⟩ , ⟨*identifier*⟩)
	|	\block_length (⟨*term*⟩)
	|	\base_addr (⟨*term*⟩)
	|	\result
	|	\null
⟨*constant*⟩	::=	⟨*integer-constant*⟩ | ⟨*floating-point-constant*⟩
⟨*arith_op*⟩	::=	+ | - | * | / | %

Tabela A.2: *Sintaxe da linguagem de anotações.*

⟨*predicate*⟩	::=	`\true`
	|	`\false`
	|	⟨*identifier*⟩
	|	⟨*identifier*⟩ `(` ⟨*term*⟩$^{+}_{,}$ `)`
	|	⟨*term*⟩ ⟨*relation*⟩ ⟨*term*⟩ `[` ⟨*relation*⟩ ⟨*term*⟩ `]`
	|	⟨*predicate*⟩ `=>` ⟨*predicate*⟩
	|	⟨*predicate*⟩ `<=>` ⟨*predicate*⟩
	|	⟨*predicate*⟩ `||` ⟨*predicate*⟩
	|	⟨*predicate*⟩ `&&` ⟨*predicate*⟩
	|	`!` ⟨*predicate*⟩
	|	`if` ⟨*term*⟩ `then` ⟨*predicate*⟩ `else` ⟨*predicate*⟩
	|	`\forall` ⟨*logic_parameter*⟩$^{+}_{,}$ `;` ⟨*predicate*⟩
	|	`\exists` ⟨*logic_parameter*⟩$^{+}_{,}$ `;` ⟨*predicate*⟩
	|	`(` ⟨*predicate*⟩ `)`
	|	`\old (` ⟨*predicate*⟩ `)`
	|	`\at (` ⟨*predicate*⟩ `,` ⟨*identifier*⟩ `)`
	|	`\valid (` ⟨*term*⟩ `)`
	|	`\valid_index (` ⟨*term*⟩ `,` ⟨*term*⟩ `)`
	|	`\valid_range (` ⟨*term*⟩ `,` ⟨*term*⟩ `,` ⟨*term*⟩ `)`
	|	`\fresh (` ⟨*term*⟩ `)`
	|	⟨*identifier*⟩ `::` ⟨*predicate*⟩
⟨*relation*⟩	::=	`==` | `!=` | `<` | `<=` | `>` | `>=`

Tabela A.3: *Sintaxe da linguagem de anotações. (continuação)*

⟨*c_file*⟩	::=	⟨*declaration*⟩*
⟨*declaration*⟩	::=	⟨*spec*⟩ ⟨*type*⟩ ⟨*identifier*⟩ `(` ⟨*parameter*⟩$^{*}_{,}$ `)` `;`
	|	⟨*spec*⟩ ⟨*type*⟩ ⟨*identifier*⟩ `(` ⟨*parameter*⟩$^{*}_{,}$ `)` ⟨*block*⟩ `;`
	|	`/*@ logic` ⟨*logic_type*⟩ ⟨*identifier*⟩ `(` ⟨*logic_parameter*⟩$^{*}_{,}$ `)` [⟨*term*⟩ | `reads` ⟨*location*⟩$^{+}_{,}$] `*/`
	|	`/*@ predicate` ⟨*identifier*⟩ `(` ⟨*logic_parameter*⟩$^{*}_{,}$ `)` [⟨*predicate*⟩ | `reads` ⟨*location*⟩$^{+}_{,}$] `*/`
	|	`/*@ axiom` ⟨*identifier*⟩ `:` ⟨*predicate*⟩ `*/`
	|	`/*@ invariant` ⟨*identifier*⟩ `:` ⟨*predicate*⟩ `*/`
	|	`/*@ ghost` ⟨*logic_type*⟩ ⟨*identifier*⟩ [`=` ⟨*term*⟩] `*/`
⟨*assigned_locs*⟩	::=	⟨*location*⟩$^{+}_{,}$ | `\nothing`
⟨*spec*⟩	::=	`/*@` [`requires` ⟨*predicate*⟩] [`assigns` ⟨*assigned_locs*⟩] [`ensures` ⟨*predicate*⟩] [`decreases` ⟨*variant*⟩] `*/`
⟨*statement*⟩	::=	⟨*loop_annot*⟩ `while (` ⟨*expr*⟩ `)` ⟨*statement*⟩
	|	⟨*loop_annot*⟩ `do` ⟨*statement*⟩ `while (` ⟨*expr*⟩ `)`
	|	⟨*loop_annot*⟩ `for (` ⟨*statement*⟩ `;` ⟨*statement*⟩ `;` [⟨*expr*⟩] `)` ⟨*statement*⟩
	|	`/*@ assert` ⟨*predicate*⟩ `*/`
	|	`/*@ label` ⟨*identifier*⟩ `*/`
	|	⟨*spec*⟩ ⟨*statement*⟩
	|	`/*@ set` ⟨*identifier*⟩ `=` ⟨*term*⟩ `*/`
⟨*loop_annot*⟩	::=	`/*@` [`invariant` ⟨*predicate*⟩] [`loop_assigns` ⟨*assigned_locs*⟩] [`variant` ⟨*variant*⟩] `*/`
⟨*variant*⟩	::=	⟨*term*⟩ [`for` ⟨*identifier*⟩]

Tabela A.4: *Estrutura das anotações no código* C.

⟨*logic_type*⟩	::=	`void` | ⟨*sign*⟩ `char` | ⟨*sign*⟩ `short` | ⟨*sign*⟩ `int` | ⟨*sign*⟩ `long`
	|	⟨*sign*⟩ `long long` | `float` | `double` | `long double`
	|	⟨*logic_type*⟩ `*` | ⟨*identifier*⟩
⟨*logic_parameter*⟩	::=	⟨*logic_type*⟩ ⟨*identifier*⟩
⟨*sign*⟩	::=	(`signed` | `unsigned`)?
⟨*location*⟩	::=	⟨*term*⟩
	|	⟨*location*⟩ `.` ⟨*identifier*⟩
	|	⟨*location*⟩ `->` ⟨*identifier*⟩
	|	`*` ⟨*location*⟩
	|	⟨*location*⟩ `[` ⟨*term*⟩ `]`
	|	⟨*location*⟩ `[ .. ]` | ⟨*location*⟩ `[` ⟨*term*⟩ `.. ]`
	|	⟨*location*⟩ `[ ..` ⟨*term*⟩ `]` | ⟨*location*⟩ `[` ⟨*term*⟩ `..` ⟨*term*⟩ `]`

Tabela A.5: *Estrutura das anotações no código* C. *(continuação)*

Bibliografia

[1] Coco/R, a compiler generator. http://www.ssw.uni-linz.ac.at/Coco/.

[2] The Caduceus verification tool. http://why.liri.fr/caduceus.

[3] The Coq proof assistant. http://coq.inria.fr/.

[4] The *Why* verification tool. http://why.lri.fr/.

[5] The haRVey decision procedure. http://www.loria.fr/equipes/cassis/softwares/haRVey/.

[6] The HOL Light theorem prover. http://www.cl.cam.ac.uk/users/jrh/hol-light.

[7] The Java Moduling Language (JML). http://www.cs.iastate.edu/~leavens/JML/.

[8] The Krakatoa Tool for JML/Java Program Certification. http://www.lri.fr/~marche/krakatoa/.

[9] The Mizar project. http://mizar.uwb.edu.pl/.

[10] The ML Language. http://burks.bton.ac.uk/burks/language/ml/.

[11] The PVS Specification and Verification System. http://pvs.csl.sri.com.

[12] The Simplify decision procedure (part of ESC/Java). http://research.compaq.com/SRC/esc/simplify.

[13] Jean-Christophe Filliâtre and Claude March. The CADUCEUS verification tool for C programs. Tutorial and Reference Manual. `http://why.lri.fr/caduceus/manual/caduceus.ps`, January 2006.

[14] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification condition generator. `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.

[15] Jean-Christophe Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. `http://www.lri.fr/~filliatr/ftp/publis/jphd.ps.gz`, July 2003.

[16] Jean-Christophe Filliâtre and Claude Marché. Multi-Prover Verification of C Programs. `http://www.lri.fr/~filliatr/ftp/publis/caduceus.ps.gz`.

[17] J. Gomes, D. Martins, J.S. Pinto, and S. Melo de Sousa. Lissom, a Source Level Proof Carrying Code Platform. Proc. of the PCC workshop, LICS/FLOC 2006, 2006.

[18] João Gomes. Computer Programs Validation. Relatório de projecto de final de curso, February 2006.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. 12, October 1969.

[20] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. `http://www.cs.iastate.edu/~leavens/JML/documentation.shtml`, January 2006.

[21] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`, January 2006. Draft tutorial in PDF format.

[22] D. Martins and S. Melo de Sousa. LISSOM: Plataforma do Paradigma Proof Carrying Code para Execução Segura de Código Móvel. Proceedings of SINO'05, "Segurança Informática nas Organizações", 2005.

[23] Daniel Martins. LISSOM: Uma Arquitectura PCC ao nível fonte. Relatório de projecto de final de curso, Fevereiro 2006.

[24] Hanspeter Mössenböck. The Compiler Generator Coco/R, User Manual. `http://www.ssw.uni-linz.ac.at/Coco/Doc/UserManual.pdf`, 2006.

[25] Glynn Winskel. *The Formal Semantics of Programming Languages, An Introduction*. The MIT Press, 1993.