Ha_CWB - Haskell Concurrency Workbench, sort of ...

José Manuel Serra Marques (35839)
Carlos Manuel Gomes Vilhena (35812)
Departamento de Informática :: Universidade do Minho

July 25, 2006

Abstract

Para a criação do Haskell Concurrency Workbench HA_CWB foi necessário dividir esse projecto em duas fases:

- Animação de processos através de grafos de transição - nesta fase, no âmbito da disciplina de Métodos de Programação 4 e principal objectivo para o projecto da mesma disciplina, foram definidos processos em Haskell, e um conjunto de módulos também em Haskell para a manipulação de processos (concretamente avaliação de processos) e criação de grafos de tansição.
- Cálculo de processos - segunda fase do projecto, ainda não abordada, que visa a interacção entre processos, a criação de digramas de sincronização de processos, bem como verificar Bissimulação e Equivalência Estrita, Equivalência Observacional e Igualdade de Processos. Para além disso, será ainda objectivo do projecto incluir um *parser* para a linguagem Π-*Calculus*.
  Nesta fase, será ainda incluído um modo gráfico, para facilmente a ferramenta poder ser usada. O modo gráfico será criado em *WX-Haskell*, e toda a ferramenta criada estará por de trás do motor gráfico.

Contents

1 Objectivos

Nesta secção vamos falar um pouco sobre a razão que nos levou a lançar este projecto também iremos abordar mais detelhadamente os objectivos que temos para o nosso trabalho.
A ideia deste pojecto, surgiu-nos quando foi propos-to na disciplina de MP4 a utilização de uma excelente ferramenta (CWB) que visa a animação de processos e a formulação de vários tipo de provas sobre os mesmos. Mas apesar de ser uma boa ferramenta, tem o grande defeito de ser muito difícil (para não dizer impossível) de instalar. Para ajudar ainda mais, existe na Universidade do Minho uma grande cultura para a utilização da poderosa linguagem de programação `Haskell`, e foi então que nos surgiu a ideia da construção de uma ferramenta em `Haskell`permitindo assim tirar todas as vantagens desta linguagem de programação.
Passando agora mais concretamente aos nossos objectivos que temos para este projecto, é nosso "sonho" que a nossa ferramenta tenha pelo menos as mesmas funcionalidades do CWB, com uma interface um pouco mais polida e ainda adicionar mais funcionalidade ao nível da interação do utilizador com o resultado da computação realizada sobre a lista de processos inicial.
Actualmente a ferramenta lê de um ficheiro, um conjunto de processos inicial e depois de seleccionado oo processo pelo qual deseja iniciar, é gerado o grafo de transições desse processo para um limite desejado, sendo de seguida esse resultado exportado para um ficheiro `GraphViz`. O `GraphViz` foi escolhido como forma de mostrar o resultado pois é muito simples a geração da representação do grafo de transições de precessos arbitrariamente complexos, sendo também muito fácil a visualização por parte do utilizador do respectivo resultado.
Como objectivos futuros, ou seja, funcionalidades para próximas versões da ferramenta, estamos empenhados em adicionar o reconhecimento da linguagem de CCS com anotação bem como a possibilidade da realização de tarefas bem mais interessantes sobre os processos como sejam a prova de simulações, bi-simulações, equivalências observacionais, etc.

2 Software Utilizado

Aqui vamos enunciar as três principais ferramentas de software utilizadas no desenvolvimento do trabalho. Na última secção falaremos resumindamente de outras ferramentas que nos ajudaram a levar o trabalho a "bom porto".
Como compilador principal da linguagem `Haskell`, utilizamos a ferramenta GHC.
Na construção do interpretador de ficheiros utilizamos a ferramenta Alex como *lexer* e a ferramenta Happy como gerador do *parser*.
O Alex gera a lista de *tokens* reconhecidos no texto de entrada e o Happy interpreta esses *tokens* de acordo com a linguagem especificada.

2.1 GHC

A ferramenta GHC foi escolhida para este trabalho pois ela pemite-nos compilar o trabalho e não apenas interpreta-lo. Isto é tanto ao mais importante pois é nosso objectivo gerar uma ferramenta autónoma e não apenas desenvolver um conjunto de bibliotecas para serem posteriormente interpretadas num qualquer interpretador de `Haskell`.
Assim sendo, na fase de desenvolvimento do trabalho utilizamos o interpretador GHCi, o que nos permitiu ir monitorizando e corrigindo os erros e quando quisemos gerar a ferramenta **hacwb** usamos o compilador GHC.
Mais informações em `http://www.haskell.org/ghc/`.

2.2 Alex

A ferramenta Alex pretende ser um gerador de analisadores léxicos para a linguagem `Haskell` em que cada *token* é descrito sobe a forma de expressão regular. A melhor maneira de a descrever é dizer que Alex está para o `Haskell` como o Flex está para o `C/C++`.
Mais informações em `http://www.haskell.org/alex/`.

2.3 Happy

Happy pretende ser gerador de *parsers* também para a linguagem `Haskell`, ou seja, dada uma determinada linguagem na forma **BNF** (BackusNaur form), o Happy constroi o respectivo *parser*. A melhor maneira para o descrever é dizer que o Happy está para o `Haskell` como o YACC/Bison está para o `C/C++`.
De seguida apresenta-se a linguagem utilizada no para este trabalho e as respectivas acções semânticas.

```
Root :: { LstProcesses String }
Root : PROCNAME '=' Process ';' Root  { ($1, $3):$5 }
     | {- empty -} { [] }

Process :: {  Process String }
Process : Rests Proc_Body { if $1==[] then $2 else (New $1 $2) }
     | {- empty -} { Zero }

Rests :: { [String] }
Rests : NEW '{' More_Rests '}' { $3 }
      | {- empty -} { [] }
More_Rests : ACT ',' More_Rests { $1:$3 }
           | ACT { [$1] }

Proc_Body :: { Process String }
Proc_Body : Pro '+' Pro { Or $1 $3 }
        | Pro '|' Pro { Paralel $1 $3 }
        | Pro { $1 }
Pro : Pro_Complex { $1 }
    | P { $1 }
Pro_Complex : '(' Proc_Body ')'{ $2 }
P : ACT '.' P { Dot (Var $1) $3 }
  | '~' ACT '.' P { Dot (Comp $2) $4 }
  | '0' { Zero }
  | PROCNAME {NameP $1}
```

De notar que a cada produção da gramática está associada a respectiva acção semântica. Como se pode ver acima, o resultado final é a lista de processos contidos no ficheiro de entrada, no tipo de dados que utilizamos no nosso trabalho.
Mais informações em `http://www.haskell.org/happy/`.

2.4 Outras

Outra ferramenta que utilizamos no nosso trabalho foi a Make que vem com todas as distribuições `Unix`, permitindo desta forma automatizar o processo de criação da ferramenta. Para isso basta apenas fazer "make hacwb" e o executável é gerado automáticamente.

3 HaCWB

Nesta secção, será explicado o objectivo desta ferramenta, bem como o modo de utilização. Todas as opções permitidas serão explicadas, bem como o modo de funcionamento.

3.1 Descrição da Ferramenta

Como já foi dito anteriormente, neste momento esta ferramenta apenas manipula processos e permite criar grafos de transição. Assim, recebendo como parâmetro de entrada um ficheiro em CCS contendo processos, o *parser* lê-os, e coloca-os numa estrutura de dados intermédia, mais concretamente, uma lista de pares da forma (Nome do Processo , Processo), para posterior manipulação.

3.2 Utilização do HaCWB

Para melhor compreender e utilizar o hacwb, passamos à descrição de todas as opções disponíveis, e ainda o seu modo de funcionamento. Serão apresentados ainda alguns breves exemplos de utilização.

- *-h* - Opção de ajuda. Quando incluida nas opções, é imprimido um texto de ajuda, com todas as opções da ferramenta.
  Segue-se um exemplo da invocação da ferramenta com esta opção:

```
/src/bin$ ./hacwb -h

HaCWB - A Haskell tool to manipulate processes.
Usage: hacwb options ...

List of options:

  -H, -h                     --help
                            output a brief help message
  -L file_in, -l file_in     --load=file_in
                            specify input file
  -S file_out, -s file_out   --save=file_out
                            save transition graph
  -G, -g                     --graph
                            generate transition graph
  -F, -f                     --save
                            generate multiple files
                            with transition graphs
```

```
For more information see README file.
```

- *-g* - Para criar grafos de transição a partir de um ficheiro de *input*, é necessário incluir esta opção na ferramenta.

- *-l* - Esta opção requer argumento. Assim, dado um ficheiro de entrada em CCS, o hacwb faz o *parsing* do ficheiro, e grava para uma estrutura de dados intermédia os processos lidos.
  Para invocar o hacwb com esta opção: *hacwb -g -l"fin"*.
  Note-se que invocando desta maneira o hacwb, todo o *output* será redireccionado para o monitor.
  De seguida, mostra-se um exemplo de um ficheiro de *input*, aceite pelo hacwb:

```
P1 = (c.d.0 + w.0);

P2 = new {a} (a.0 | ~a.b.0);
```

- *-s* - Para guardar os grafos de transição criados apenas num ficheiro, basta invocar a ferramenta com esta opção seguida de um argumento (nome do ficheiro de saída sem extenção), para que todos os grafos de transição possam ser guardados (codificados em GraphViz), para posteriormente o utilizador vizualizar.
  Para invocar o hacwb com esta opção: *hacwb -g -l"fin" -s"fout"*.
  Após invocar o hacwb com o ficheiro de entrada referido no ponto anterior, e depois de gerar o respectivo *postscript* através da ferramenta *dot* incluida no GraphViz, obtemos os seguintes grafos de transição:

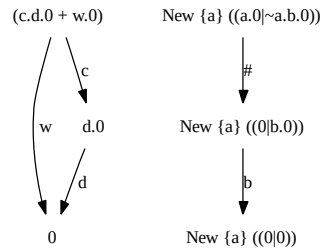

Figure 1: Grafo de transição referente ao ficheiro de entrada apresentado anteriormente.

- *-f* - Por fim, para guardar cada processo lido do ficheiro de entrada independentemente, utiliza-se a opção -f. Desta forma, sao criados n

ficheiros (com n número de processos, e em que o nome do ficheiro é o inteiro respectivo).
Para invocar o hacwb com esta opção: *hacwb -g -l"fin" -f*.
O resultado desta invocação, para i ficheiro de entrada anterior, são 4 ficheiros *dot*, com os grafos de transição separados por ficheiros.

3.3 Estruturas de dados criadas para definir Processos

Aqui, apresentam-se as estruturas de dados mais importantes criadas ao longo da realização deste trabalho prático, para definir processos e grafos de transição.
Em primeiro ligar é necessário definir processos. Para isso foi criado o tipo *Process*, com todos os construtores necessários. Um desses construtores, mais concretamente o consrutor *NameP*, foi criado para dar um nome a um processo, para assim permitir que processos de invoquem mutuamente. Segue-se o tipo de dados:

```
data Process a = Or (Process a)  (Process a)
               | Paralel (Process a) (Process a)
               | Dot (Action a) (Process a)
               | Zero
               | New [a] (Process a)
               | NameP String
```

Depois de definir processos, torna-se necessário definir Acções. Tomamos como acções a acção de sincronização *Tau*, bem como as variáveis e os seus complementares:

```
data Action a = Var a
              | Comp a
              | Tau
```

Quando lêmos um ficheiro de entrada, gravamos o conteúdo do ficheiro CCS numa estrutura intermédia, chamada *LstProcesses*, definida mais a baixo, para atribuir a cada processo um nome:

```
type LstProcesses a = [(String, Process a)]
```

Por fim, para definir grafos de transição foi definido um tipo de dados, similar a *Rose Tree*, contendo no nodo o processo e de seguida uma lista de todas as possíveis derivações:

```
data TransG a p = Node p [(a,TransG a p)]
```

Para além destas estruturas de dados, foram criadas mais estruturas auxiliares, das quais destacamos a estrutura de estado, utilizada na avaliação de processos (possívelmente infinitos):

```
type Stt a = (Int, Path a, LstTrees a)
```

Em que:

```
type Path a = [(Action a, Process a)]

type LstTrees a = [(Process a, Path a)]
```

4 TODO

Para que o Haskell Concurrency Workbench esteja finalizado, é necessário ainda desenvolver alguns pontos. Segue-se uma lista resumida do trabalho futuro proposto:

- Melhoramento a nível do código - Trata-se de um objectivo a curto prazo. Pretende-se melhorar o código criado, de forma a tornar os algoritmos mais eficientes de forma mais elegante.
- Cálculo de Processos - trata-se da verdadeira essência do HaCWB. A introdução deste *item* vai permitir a interacção de processos (criação de diagramas de sincronização) e ainda todo o tipo de verificações:
  - Bissimulação e Equivalência Estrita;
  - Equivalência Observacional;
  - Igualdade de Processos.
- Como já foi referido, será criado um *parser* para $\Pi-Calculus$, juntando-se assim ao parser de CCS.
- Por fim, para facilitar a utilização desta ferramenta, será incluído um modo gráfico a ser construído utilizando a biblioteca *WX-Haskell*.

5 Conclusão

Chegamos à secção mais interessante do projecto que é aquela em que contamos a nossa experência no decorrer deste projecto.
A fase inicial foi um pouco estranha pois, como nenhum de nós tem muita experiência neste campo, a base do trabalho estava a mudar constantemente o que não nos permitia ter alguma coisa em concreto a funcionar. Depois a base do trabalho começou a estabilizar, permitindo assim começar a pensar mais estruturadamente o que contribui para que os resultados começacem a aparecer o que começou a dar mais alento.
O desenvolvimento deste trabalho bem como a pesquisa de tecnologias associadas ao mesmo consumiu muito tempo, prolongando assim para a época de exames o que fez com que desacelerasse o seu desenvolvimento o que fez com que não fossem incluidas algumas funcionalidades (GUI, CSS anotado) mas que irão ser incluídas em versões futuras.
Para finalizar, resta dizer que foi muito interessante poder "fazer" alguma coisa mais com a teoria que aprendemos na aulas ao longo do semestre, funcionando como uma motivação extra.