Fundamentos do 'Bounded Model Checking' de Programas

Projeto "CROSS"- Refª PTDC/EIA-CCO/108995/2008

BI4-2011_PTDC/EIA-CCO/108995/2008_UMINHO

(de 31 de março de 2012 a 30 de junho de 2012)

(José João Peixoto Pereira)

Resumo

O presente relatório técnico expressa a existência de uma dissertação de mestrado como resultado da contribuição da bolsa de investigação de referência BI4-2011_PTDC/EIA-CCO/108995/2008_UMINHO

Introdução

A bolsa de investigação referida permitiu a familiarização do método conhecido por Bounded Model Checking (BMC) e a construção de uma ferramenta de aplicação do mesmo método a programas de software de modalidade imperativa. O desenvolvimento desta ferramenta necessitou da adaptação de conhecimentos de compilação e satisfiabilidade lógica.

O projecto foi estendido, concluindo com o estudo da correcção e rimeiros passos no sentido da formalização do método BMC.

Desenvolvimento

Como resultado, encontra-se em fase de término a elaboração de uma dissertação, para discussão e aprovação ao grau de mestre do investigador José João Peixoto Pereira, intitulada ``Bounded Model Checking para uma linguagem imperativa simples'', com data de entrega prevista para finais de Julho.

Nesta dissertação formaliza-se uma linguagem imperativa simples com expressões inteiras e booleanas. Recorre-se à semântica operacional para estudar e interpretar o significado dos vários comandos da linguagem, em particular o resultado da aplicação de um comando nulo; o resultado expresso no estado da computação por parte da aplicação de uma atribuição; o significado da aplicação sequêncial de comandos; o significado da aplicação de condicionais e ciclos; bem como as compatibilidades e eligibilidades exercidas pela substituição de dois comandos.

Define-se uma noção de correcção de programas como a avaliação de um estado indicativo de erro. Estuda-se a manutenção de propriedades dos programa ao longo das consecutivas transformações aplicadas pelo método.

Aborda-se o BMC, do ponto de vista histórico, como surgiu, de onde surgiu, e como duas áreas distintas desenvolvem um mesmo método para solucionar um problema de duas perspectivas diferentes.

Explica-se e procura-se formalizar transformações as transformações do BMC. Seguindo para a prova de correcção e completude de uma das transformações, a transformação de um programa normalizado pelo método BMC numa fórmula lógica de primeira ordem, solucionável por um método de decisão.

Conclusão

Contamos poder providenciar o resultado final desta dissertação brevemente, sendo apresentado em anexo a versão corrente.

José João Peixoto Pereira

Conteúdo

Capítulo 1

Introdução

A verificação formal é o elemento primordial para o desenvolvimento de sistemas. De acordo com Harry Foster [12], em média, a procura de 'bugs' em sistemas digitais chega a consumir mais de 60 por cento do esforço de desenvolvimento nos *circuitos integrados de aplicação específica* (ASIC) e *Sistemas integrados em chips* (SoC). Claramente, este é um tema que a indústria deve enfrentar, e algumas organizações já o têm feito. Em [1] afirma-se que as impresas que adotaram uma metodologia baseada em afirmações de verificação, i.e., **Assertion-Based Verification (ABV)**, experiênciaram uma significativa redução no tempo de simulação e de depuração, uma melhoria de até cerca de 50 por cento. A **verificação formal de sistemas de software** é o acto de provar a correcção dos algoritmos presentes em sistemas de software com respeito a **especificações formais** ou **propriedades**, usando métodos formais matemáticos. Os processos de verificação consistem em validar aspectos estruturais (estáticos) e comportamentais (dinâmicos). Os aspectos estáticos envolvem análises detalhadas ao código, enquanto que os aspectos dinâmicos estão intimamente ligados a baterias de testes que são aplicados à execução do sistema. A crescente complexidade evidenciada nos mais recentes sistemas tem salientado a importância de aplicar técnicas de verificação formal na indústria de hardware [15, 13]. Grande parte dos maiores fabricantes de sistemas digitais tem procurado aplicar sistemas de verificação formal.

A detecção de erros é fundamental. Intuitivamente, um erro é algo indesejável num sistema. Indesejável pois acarreta custos temporais, económicos e biológicos, e em sistemas críticos a ocorrência de erros é extremamente indesejável, sendo que se implementam mecanismos **fail-safe** para prevenir a falha do sistema. Contudo pouco estudo formal se encontra sobre a noção *erro* e a teoria do erro, sendo que a maioria do investimento em estudos recore a práticas de detecção de erro, empiricamente, através de testes e seus derivados. Enquanto na industria do hardware o desenvolvimento de técnicas automaticas de verficação tem levado à adopção generalizada destas técnicas, o mesmo não sucede ainda na área do software, em que as técnicas predominantes são baseadas tipicamente em provas dedutivas, [**?**] com base em asserções (em particular invariantes de ciclo) fornecidas por engenheiros de software e na utilização de demonstradores de teoremas de difícil domínio.

Nos últimos anos tem-se evidênciado um desenvolvimento notável em métodos automáticos de verificação.

Uma mais valia reside em não precisar de invariantes para relizar a verificação.

O sucesso dos SMT tem potênciado estes métodos de verificação, mas constatamos que não há trabalho formal realizado no sentido da correcção do método.

Verificamos a inexistência de trabalhos formais na área. Embora se encontrem divérsos artigos sobre os sucessos da técnica **Bounded Model Checking** e diversas aplicações práticas, pouco trabalho é realizado na verificação formal do processo.

CBMC é uma ferramenta de **Bounded Model Checking** para programas escritos na linguagem ANSI-C e C++, que permite verifivar limites em `arrays`, `buffer overflows`, `pointer safety`, verificação de excepções e afirmações especificadas pelo programador. O processo de veri-

ficação do CBMC transforma o programa, desdobrando ciclos whiles em sequências equivalentes, convertendo o programa numa equação que é validada por um procedimento de decisão como **SAT solvers**.

Diversos artigos salientam o sucesso deste método [5, 14, 6].

Capítulo 2

A linguagem imperativa base com asserts - $\mathcal{W}hile_{\textbf{assert}}$

Este capítulo procura definir uma linguagem de programação que será utilizada ao longo da tese. Esta linguagem é **simples**, **imperativa** e **básica** pelo que apenas procura representar os principais mecanismos de progamação. A esta linguagem dá-se a denominação **Simple Imperative Language** e, embora seja **simples**, será suficientemente poderosa para podermos raciocinar sobre as transformações que ocorrem no processo **Bounded Model Checking**.

Neste capítulo são definidos todos os objectos que nos permitirão raciocinar sobre a validade do método **Bounded Model Checking** e as transformações em que assenta. Abordar-se-ão questões de *sintaxe* da linguagem, bem como as questões de *semântica*. Por um lado, a *sintaxe* é a área que estuda a forma das frases de uma linguagem. A *sintaxe* é definida por uma gramática, normalmente uma gramática normalizada pelo formalismo **Backus Naur Form** (BNF), ou uma variante desta. Por outro lado, a *semântica* é a área que estuda a forma de expressar o significado dessas frases numa linguagem, sendo ambas necessárias para definir e raciocinar sobre uma linguagem.

A linguagem **Simple Imperative Language**, procurará exprimir **ciclos** com o comando **while**(b)**do** c_w, **condicionais** com o comando **if**(b)**then** c_t **else** c_f, **atribuições** com o comando $x := e$, as **asserções** com o comando **assert**(b), **nop**'s (no operation) com o comando **skip**, e finalmente a construção típica das linguagens imperativas, evidenciada pela **composição** de commandos com o comando c_i **;** c_j.

2.1 Aspectos sintáticos da linguagem básica

A *sintaxe* de uma linguagem de programação procura especificar a estrutura e a forma de organizar os constituintes da linguagem (as palavras, os chamados `tokens` da linguagem). Geralmente expressos em texto, `strings` ou frases da linguagem.

A *sintaxe* procura descrever um conjunto de regras, conhecidas como produções da linguagem, que definem possíveis construções de uma frase da linguagem. Estas regras descrevem a possível organização dos elementos da linguagem (`tokens`), e organizam a estrutura em classes sintáticas. A *sintaxe* diz-se ambígua quando a *sintaxe* permite mais do que uma forma de construção da mesma frase da linguagem.

Como dito anteriormente, a *sintaxe* é definida por uma gramática, normalmente numa gramática normalizada em **Backus Naur Form** ou uma variante desta. Iremos exprimir a nossa *sintaxe* numa variante desta meta-notação. Uma vez que não nos iremos preocupar com questões como *"o que é um número"* e *"como deve ser um número representado"*, iremos apresentar a nossa *sintaxe* em termos de uma *sintaxe abstracta*.

As sequências de símbolos permitidos numa frase sintaticamente correta do programa é definido pela seguinte *sintaxe abstracta*.

2.1.1 Expressões inteiras

A linguagem é **simples**, pelo que iremos manipular objectos conhecidos e pouco complexos. Precisamos de definir as variáveis do programa que irão representar números inteiros. As variáveis serão os objectos que a linguagem manipula.

Convenção 2.1.1. *Assumimos a existência de variáveis* $x_0, x_1, \ldots, x_n, \ldots$.
Esta colecção de variáveis é numerável e notada por $\mathcal{V}_{Integer}$. *Para raciocinar sobre os habitantes, usamos as letras* $x, x', y, y', \ldots$ *como meta-variáveis sobre* $\mathcal{V}_{Integer}$.

As variáveis de $\mathcal{V}_{Integer}$, no contexto da avaliação de expressões, assumirão valores em $\mathbb{Z}$. Habitualmente usamos $z, z', z_0, \ldots$ como meta-variáveis sobre $\mathbb{Z}$

Por forma a fornecer algum poder à linguagem, é permitido a elaboração de **expressões inteiras**, com recurso a operações sobre os números e sobre as variáveis. Introduzimos a noção de número inteiro, variável de números inteiros, conforme defínido pela Definição 2.1.1 (pág. 6) . Introduzimos, também, a noção de negativo, bem como as operações binárias de soma, multiplicação, divisão usual e a operação que calcula o resto da divisão. A interpretação será a usual, conforme expresso porteriormente na Definição 2.2.2 (pág. 11) .

Definição 2.1.2. *A notação $\mathcal{E}xpression_{Integer}$ representará a classe das* **expressões inteiras**. *A classe sintática $\mathcal{E}xpression_{Integer}$ é definida indutivamente do seguinte modo:*

$\mathcal{E}xpression_{Integer}\ e$	$::=$	z	$se\ z \in \mathbb{Z}$
	$\mid$	x	$se\ x \in \mathcal{V}_{Integer}$
	$\mid$	$-e$	
	$\mid$	$e_i \square e_j$	$, \square \in \{ +,\ -,\ \times,\ \div,\quad \text{mod}\ \}$

Tabela 2.1: Classe das **expressões inteiras**

Habitualmente usamos $e, e', e_i, e_j, e_1, \ldots$ como meta-variáveis sobre $\mathcal{E}xpression_{Integer}$.

Variáveis presentes nas expressões inteiras

Recorremos à função FV para obter os elementos de $\mathcal{V}_{Integer}$ presentes numa expressão inteira.

Definição 2.1.3. *A função $FV : \mathcal{E}xpression_{Integer} \to \mathcal{P}(\ \mathcal{V}_{Integer}\ )$ é definida recursivamente por:*

$$\begin{array}{rcll} FV(\ z\ ) & = & \emptyset & se\ z \in \mathbb{Z} \\ FV(\ x\ ) & = & \{\ x\ \} & \\ FV(\ -e\ ) & = & FV(\ e\ ) & \\ FV(\ e_i \square e_j\ ) & = & FV(\ e_i\ ) \cup FV(\ e_j\ ) & se \square \in \{ +,\ -,\ \times,\ \div,\quad \text{mod}\ \} \end{array}$$

Exemplo 1. $FV(\ 3 + x_0 \times y_3 - 4 \div 96\ ) = \{\ x_0,\ y_3\ \}$

2.1.2 Expressões booleanas

Esta subsecção procura formalizar a forma de expressar expressões boleanas, que nos permitirão racícionar sobre a verdade e a falsidade de afirmações. Estas definições são necessárias para exprimir ciclos, condicionais e mesmo afirmações dentro dos programas, além de que o método de **Bounded Model Checking**, por si transforma um programa numa fórmula lógica, uma expressão, que procura modelar todo o comportamento expresso pelo programa. As expressões booleanas são necessáriamente relevantes na construção destas fórmulas.

Precisamos de definir uma notação para os valores de verdade, representado pelo símbolo $\top$ e falso, representado pelo símbolo $\bot$.

Definição 2.1.4.
Usamos o símbolo $\top$ para denotar o **valor lógico verdadeiro**.
Usamos o símbolo $\bot$ para denotar o **valor lógico falso**.
Denotaremos o conjunto dos valores lógicos, por $\mathbb{B}$. Por vezes, chamaremos **valores booleanos** *aos valores lógicos.*

A seguinte classe sintática de expressões booleanas corresponde a valores de verdade presentes em $\mathbb{B}$, a negação desses valores, introduz também a representação de algumas operações lógicas como a conjunção lógica $\wedge$, a conjunção lógica $\vee$ e a implicação lógica $\rightarrow$. A implicação lógica não é comum nas linguagens de programação usual, contudo optamos por introduzir a noção, evitando injecções deste símbolo na linguagem, e a respectiva formalização de novas classes e interpretações associadas, quando for realizado estudo da expressão lógica resultante do processo **Bounded Model Checking**.Introduzimos também a noção de comparações, que são o mecanismo fundamental para a construção de expressões booleanas a partir dos valores das variáveis, i.e., do estudo do programa.¡

A avaliação desta classe sintática é definida posteriormente na semântica, tendo usualmente a interpretação comum dos símbolos de comparação expressos, conforme apresentado na Definição 2.2.2 (pág. 11) .

Definição 2.1.5. *A notação $\mathcal{E}xpression_{Boolean}$ representará a classe das* **expressões booleanas**. *A classe sintática $\mathcal{E}xpression_{Boolean}$ é definida indutivamente do seguinte modo:*

$$
\begin{array}{rcll}
\mathcal{E}xpression_{Boolean}\ b & ::= & \top & \\
 & | & \bot & \\
 & | & e_i \,\square\, e_j & ,\square \in \{\ \leq, <, >, \geq, =, \neq\ \} \\
 & | & \neg\, b & \\
 & | & b_i \,\square\, b_j & ,\square \in \{\ \wedge, \vee, \rightarrow\ \}
\end{array}
$$

Tabela 2.2: Classe das **expressões booleanas**

Habitualmente usamos $b, b', b_i, b_j, b_1, \ldots$ como meta-variáveis sobre $\mathcal{E}xpression_{Boolean}$.

Variáveis presentes nas expressões booleanas

Recorremos à função FV para obter os elementos de $\mathcal{V}_{Integer}$ presentes numa expressão booleana.

Definição 2.1.6. *A função $FV : \mathcal{E}xpression_{Boolean} \rightarrow \mathcal{P}(\mathcal{V}_{Integer})$ é definida recursivamente por:*

$$
\begin{array}{rcll}
FV(\ \top\ ) & = & \emptyset & \\
FV(\ \bot\ ) & = & \emptyset & \\
FV(\ e_i \,\square\, e_j\ ) & = & FV(\ e_i\ ) \cup FV(\ e_j\ ) & ,\square \in \{\ \leq, <, >, \geq, =, \neq\ \} \\
FV(\ \neg\, b\ ) & = & FV(\ b\ ) & \\
FV(\ b_i \,\square\, b_j\ ) & = & FV(\ b_i\ ) \cup FV(\ b_j\ ) & ,\square \in \{\ \wedge, \vee, \rightarrow\ \}
\end{array}
$$

Exemplo 2. $FV(\ \top \rightarrow \bot \vee y_a - 4 \geq x_b \div 96\ ) = \{\ y_a,\ x_b\ \}$

2.1.3 Comandos da linguagem

A seguinte classe sintática corresponde aos comandos presentes na linguagem. Conforme indicado na introdução, esta classe indica a ordem e sequênciação de simbolos que exprimem os ciclos, os condicionais, os mecanismos de modificação de variávies, de afirmação de propriedades do programa, etc.

O comando nulo **skip** representa uma "não operação". Pode ser considerado o elemento neutro, pois não efectua qualquer tipo de intervenção na computação estudada. A sua existência, embora não afecte os estados da computação, permite programar condicionais em que apenas um dos ramos tem conteúdo computacional.

O comando de atribuição $x \ := \ e$, permite manipular variáveis do programa, onde estas representam o valor da avaliação da expressão e no estado em que esta atribuição é realizada. Este mecanismo de substituição é o único que altera o estado. O comando de composição c_i **;** c_j, permite sequênciar um conjunto de passos de computação, simulando a execução temporal das linguagens imperativas. O comando condicional **if**$(\ b\ )$**then** c_t **else** c_f, permite racícionar em termos de casos, onde a condição b rege a decisão de computar uma das duas possibilidades, c_t ou c_f.

Definição 2.1.7. *A notação* $\mathcal{C}$**ommand** *representará a classe de comandos. A classe sintática* $\mathcal{C}$**ommand** *é definida indutivamente do seguinte modo:*

$\mathcal{C}$**ommand** c	::=	**skip**	
	|	**assert**(b)	
	|	$x := e$	se $x \in \mathcal{V}_{Integer}$
	|	c_i **;** c_j	
	|	**if**(b)**then** c_t **else** c_f	
	|	**while**(b)**do** c_w	

Tabela 2.3: Classe das expressões de comandos

A uma expressão de $\mathcal{C}$**ommand** *chamaremos de comando. Habitualmente usamos* $c, c', c_i, c_j, c_1, \ldots$ *como meta-variáveis sobre* $\mathcal{C}$**ommand**.

Variáveis presentes nas expressões de comandos

Recorremos à função FV para obter os elementos de $\mathcal{V}_{Integer}$ presentes num comando.

Definição 2.1.8. *A função* $FV : \mathcal{C}\textbf{ommand} \to \mathcal{P}(\ \mathcal{V}_{Integer}\ )$ *é definida recursivamente por:*

$$\begin{array}{rcl} FV(\ \textbf{\textit{skip}}\ ) & = & \emptyset \\ FV(\ \textbf{\textit{assert}}(\ b\ )\ ) & = & FV(\ b\ ) \\ FV(\ x := e\ ) & = & \{\ x\ \} \cup FV(\ e\ ) \\ FV(\ c_i\ \textbf{;}\ c_j\ ) & = & FV(\ c_i\ ) \cup FV(\ c_j\ ) \\ FV(\ \textbf{\textit{if}}(\ b\ )\textbf{\textit{then}}\ c_t\ \textbf{\textit{else}}\ c_f\ ) & = & FV(\ b\ ) \cup FV(\ c_t\ ) \cup FV(\ c_f\ ) \\ FV(\ \textbf{\textit{while}}(\ b\ )\textbf{\textit{do}}\ c_w\ ) & = & FV(\ b\ ) \cup FV(\ c_w\ ) \end{array}$$

Exemplo 3. *Seja* p *o programa em* $\mathcal{C}$**ommand** ,
tal que $p = x_7 := 1 + e_a\ \textbf{;}\ x_7 := 1 \times e_b\ \textbf{;}\ x_7 := 1 \mod e_c\ \textbf{;}\ x_7 := 1 \div e_d\ \textbf{;}\ x_7 := x_3 \times x_7$,
$FV(\ p\ ) = \{\ x_7,\ x_3,\ e_a,\ e_b,\ e_c,\ e_d\ \}$

Variáveis atribuídas nas expressões de comandos

Recorremos à função **assign** para obter os elementos de $\mathcal{V}_{Integer}$ que são atribuídos um valor nos comandos.

Definição 2.1.9. *A função* $\textbf{\textit{assign}} : \mathcal{C}\textbf{ommand} \to \mathcal{P}(\ \mathcal{V}_{Integer}\ )$ *é definida recursivamente por:*

$$\begin{array}{rcl} \textbf{\textit{assign}}(\ \textbf{\textit{skip}}\ ) & = & \emptyset \\ \textbf{\textit{assign}}(\ \textbf{\textit{assert}}(\ b\ )\ ) & = & \emptyset \\ \textbf{\textit{assign}}(\ x := e\ ) & = & \{\ x\ \} \\ \textbf{\textit{assign}}(\ c_i\ \textbf{;}\ c_j\ ) & = & \textbf{\textit{assign}}(\ c_i\ ) \cup \textbf{\textit{assign}}(\ c_j\ ) \\ \textbf{\textit{assign}}(\ \textbf{\textit{if}}(\ b\ )\textbf{\textit{then}}\ c_t\ \textbf{\textit{else}}\ c_f\ ) & = & \textbf{\textit{assign}}(\ c_t\ ) \cup \textbf{\textit{assign}}(\ c_f\ ) \\ \textbf{\textit{assign}}(\ \textbf{\textit{while}}(\ b\ )\textbf{\textit{do}}\ c_w\ ) & = & \textbf{\textit{assign}}(\ c_w\ ) \end{array}$$

Exemplo 4. *Seja* p *o programa em* $\mathcal{C}$**ommand** , $x := 1\ \textbf{;}\ x := 1\ \textbf{;}\ x := 1\ \textbf{;}\ x := 1\ \textbf{;}\ x := 1$
$\textbf{\textit{assign}}(\ p\ ) = \{\ x\ \}$

2.2 A semântica operacional

A semântica operacional pode ser classificada em semântica operacional estrutural e semântica natural. Por um lado, a semântica operacional estrutural, também conhecida como **small-step semantics**, define o significado individual de cada passo da computação. Por outro, a semântica natural, também conhecida como **big-step semantics**, define o significado geral das computações que terminam, i.e., o que deve acontecer no final da computação.

A semântica operacional é uma forma rigorosa de formalizar o significado de uma computação. Descreve a forma como uma frase válida da linguagem deve ser interpretada, em sequências de passos computacionais. Estas sequências reflectem o significado da frase, o significado do programa. Cada expressão de uma linguagem, como **if**(b)**then** c_t **else** c_f, tem um intuito, um significado, um resultado desejado, i.e., procura exprimir algo. A semântica operacional é uma ferramenta essencial para a construção, vericação e validação de uma linguagem, por providenciar uma **descrição formal** do seu comportamento.

Uma alternativa à semântica operacional é a semântica denotacional que pode ser considerada como a interpretação matemática de programas, uma vez que os programas são traduzidos para um domínio semântico. Permite **racícionar** sobre o programa. Ambas as semânticas (semântica operacional e semântica denotacional) são convenientes para a descrição completa de uma linguagem.

2.2.1 Estados

Necessitaremos de definir a Noção de Estado. Como foi dito, as semânticas procuram representar o significado da execução de um programa. Os programas são executados em máquinas finitas que recorrem à manipulação de memória e recursos disponíveis ao sistema de computação. Para representar o significado da computação iremos representar o estado dos objectos que são manipulados pelo programa. Consideraremos que um estado de execução do programa s, contém a toda a informação gerida por um programa, i.e., o valor de todas as variáveis que um programa manipula.

Definição 2.2.1. *Chamaremos* **estado** *a uma função do tipo* $\mathcal{V}_{Integer} \to \mathbb{Z}$.
O conjunto de todos os estados possíveis será denotado por Σ. *Usamos as letras* $s, s', s_i, s_1, \ldots$ *como meta-variáveis sobre* Σ.

Aos elementos de Σ chamamos de estados e consideraremos os estados como funções. Considerar um estado como uma função que armazena informação associanda a variáveis, permite obter o valor das variáveis pela simples aplicação do estado à variável cujos valores se podem conhecer em determinado estado.

Além de aceder aos valores das variáveis armazenadas nos estados, precisamos de modificar o valor das variáveis, pelo que é necessário definir uma função conhecida por "override". Para já tome-se a definição seguinte.

Notação 1. *Para* $s \in \Sigma$, $x \in \mathcal{V}_{Integer}$ *e* $v \in \mathbb{Z}$

$$s\begin{pmatrix} x \\ v \end{pmatrix} \text{ é o estado } s \text{ onde a variável } x \text{ passa a tomar o valor de } v.$$

$$s\begin{pmatrix} x \\ v \end{pmatrix}(y) = \begin{cases} v & \text{se } x = y \\ s(y) & \text{se } x \neq y \end{cases}$$

O resultado desta substituição é por si um estado, onde a variável substítuida toma o valor do número inteiro v. Tome-se o seguinte exemplo, onde independentemente do valor tomado pela variável x, o resultado da substituição é o mesmo estado, onde essa variável toma o valor da última substituição.

Exemplo 5. $s\begin{pmatrix} x_0 \\ 2 \end{pmatrix}\begin{pmatrix} x_0 \\ 3 \end{pmatrix} = s\begin{pmatrix} x_0 \\ 3 \end{pmatrix}$

2.2.2 Avaliação de expressões inteiras

Nesta subsecção abordamos a avaliação de expressões inteiras. Raciocinamos sobre o valor da expressão de acordo com um determinado estado, i.e., dado um estado, o que se pode dizer sobre a espressão nesse estado. A avaliação da mesma expressão em estados diferentes poderá produzir resultados diferentes.

Por definição, um estado avalia um número inteiro para o valor da sua representação. O símbolo 1 significa o valor 1, o símbolo 2 significa o valor 2 e assim consecutivamente.

O estado avalia uma variável para o valor que é armazenado no estado para essa variável. Isto resulta do valor obtido pela aplicação do estado à variável.

A negação inteira é a negação usual dos numeros inteiros e consequêntes operadores apresentados. Desta forma avaliamos expressões e calculamos valores numéricos, no estilo usual do números inteiros. A Definição 2.2.2 (pág. 11) formaliza este conceito.

Definição 2.2.2. *A relação de avaliação de expressões inteiras,* $[\![\, . \,]\!] : \mathcal{E}xpression_{Integer} \to \Sigma \to \mathbb{Z}$ *é definida recursivamente por:*

$$
\begin{array}{rcll}
[\![z]\!]_s & = & z & , se\ z \in \mathbb{Z} \\
[\![x]\!]_s & = & s(x) & , se\ x \in \mathcal{V}_{Integer} \\
[\![-e]\!]_s & = & -([\![e]\!]_s) & \\
[\![e_1 \,\square\, e_2]\!]_s & = & [\![e_1]\!]_s \,\square\, [\![e_2]\!]_s & , para\ \square \in \{ +, -, \times \} \\
[\![e_1 \,\square\, e_2]\!]_s & = & \begin{cases} [\![e_1]\!]_s \,\square\, [\![e_2]\!]_s & se\ [\![e_2]\!]_s \neq 0 \\ 0 & de\ outro\ modo \end{cases} & , para\ \square \in \{ div, mod \}
\end{array}
$$

As expressões são interpretadas no estilo denotacional.

De imediato, surge a propriedade expressa pelo Lema 2.2.3 (pág. 11) . Intuitivamente, afirma que para quaisquer dois estados, se todas as variáveis presentes numa expressão de $\mathcal{E}xpression_{Integer}$ **têm atribuido o mesmo valor** em ambos os estados, i.e., a aplicação desses dois estados às variáveis presentes na expressão resulta no mesmo valor, então o resultado de **avaliar a expressão** nesses dois estádos é igual.

Lema 2.2.3. *Para quaisquer* $e \in \mathcal{E}xpression_{Integer}$

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV(e),\ s(y) = s'(y)) \Rightarrow [\![e]\!]_s = [\![e]\!]_{s'}$$

Demonstração. Por indução estrutural em $\mathcal{E}xpression_{Integer}$.

i) Para qualquer $z \in \mathbb{Z}$,

Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV(z),\ s(y) = s'(y)) \Rightarrow [\![z]\!]_s = [\![z]\!]_{s'}$$

Pela Convenção 2.2.2 (pág. 11) , $[\![z]\!]_s = z = [\![z]\!]_{s'}$.

ii) Para qualquer $x \in \mathcal{V}_{Integer}$,

Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV(x),\ s(y) = s'(y)) \Rightarrow [\![x]\!]_s = [\![x]\!]_{s'}$$

Pela Convenção 2.2.2 (pág. 11) , tendo em conta que $FV(x) = \{ x \}$, $s(x) = s'(x)$.

iii) Para qualquer $e \in \mathcal{E}xpression_{Integer}$,

Por hipótese de indução temos,

$$HI: \quad \forall s,\ s' \in \Sigma,\ (\forall y \in FV(\ e\ ),\ s(\ y\ ) \ =\ s'(\ y\ )) \Rightarrow [\![\ e\ ]\!]_s \ =\ [\![\ e\ ]\!]_{s'}$$

Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV(\ -e\ ),\ s(\ y\ ) \ =\ s'(\ y\ )) \Rightarrow [\![\ -e\ ]\!]_s \ =\ [\![\ -e\ ]\!]_{s'}$$

Pela Convenção 2.2.2 (pág. 11) , $[\![\ -e\ ]\!]_s = -(\ [\![\ e\ ]\!]_s\ )$.
Segue da hipótese e de $FV(\ -e\ ) = FV(\ e\ )$.

iv) Para quaisquer $e_i,\ e_j \in \mathcal{E}xpression_{Integer}$ e $\square \in \{\ +,\ -,\ \times,\ \div,\quad \text{mod}\ \}$

Por hipótese de indução temos,

$$HI_i: \quad \forall s_i,\ {s'}_i \in \Sigma,\ (\forall y_i \in FV(\ e_i\ ),\ s_i(\ y_i\ ) \ =\ {s'}_i(\ y_i\ )) \Rightarrow [\![\ e_i\ ]\!]_{s_i} \ =\ [\![\ e_i\ ]\!]_{{s'}_i}$$

$$HI_j: \quad \forall s_j,\ {s'}_j \in \Sigma,\ (\forall y_j \in FV(\ e_j\ ),\ s_j(\ y_j\ ) \ =\ {s'}_j(\ y_j\ )) \Rightarrow [\![\ e_j\ ]\!]_{s_j} \ =\ [\![\ e_j\ ]\!]_{{s'}_j}$$

Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV(\ e_i \square e_j\ ),\ s(\ y\ ) \ =\ s'(\ y\ )) \Rightarrow [\![\ e_i \square e_j\ ]\!]_s \ =\ [\![\ e_i \square e_j\ ]\!]_{s'}$$

Pela Convenção 2.2.2 (pág. 11) ,
$\forall\ e_1,\ e_2 \in \mathcal{E}xpression_{Integer},\ \forall\ s_0 \in \Sigma:\ [\![\ e_1 \square e_2\ ]\!]_{s_0} = [\![\ e_1\ ]\!]_{s_0} \square [\![\ e_2\ ]\!]_{s_0}$.

Basta mostrar que $[\![\ e_i\ ]\!]_s \square [\![\ e_j\ ]\!]_s = [\![\ e_i\ ]\!]_{s'} \square [\![\ e_j\ ]\!]_{s'}$
Pelas hipóteses HI_i e HI_j, tomando $s_i = s_j = s$ e $s'_i = s'_j = s'$,
e tendo em conta que $FV(\ e_i \square e_j\ ) = FV(\ e_i\ ) \cup FV(\ e_j\ )$.

□

2.2.3 Avaliação de expressões booleanas

Nesta subsecção, abordamos a avaliação de expressões booleanas.Estados diferentes proderão produzir resultados diferentes, tal como acontece na avaliação de expressões inteiras.

Por definição, qualquer estado avalia os valores de verdade $\top$ e $\bot$ para os seus valores respectivos. As operações lógicas são avaliadas na sua forma usual, bem como as operações de comparação.

Definição 2.2.4. *A relação de avaliação de expressões booleanas,* $[\![\, . \,]\!] : \mathcal{E}xpression_{Boolean} \to \Sigma \to \mathbb{B}$ *é definida recursivamente por:*

$$
\begin{array}{rcl}
[\![\top]\!]_s & = & \top \\
[\![\bot]\!]_s & = & \bot \\
[\![e_1 \,\square\, e_2]\!]_s & = & \begin{cases} \top & se\ [\![e_1]\!]_s \,\square\, [\![e_2]\!]_s \text{[1]} \\ \bot & caso\ contrário \end{cases} \\
[\![\neg\, b]\!]_s & = & \begin{cases} \top & se\ [\![b]\!]_s = \bot \\ \bot & caso\ contrário \end{cases} \\
[\![b_1 \wedge b_2]\!]_s & = & \begin{cases} \bot & se\ [\![b_1]\!]_s = \bot \\ [\![b_2]\!]_s & caso\ contrário \end{cases} \\
[\![b_1 \vee b_2]\!]_s & = & \begin{cases} \top & se\ [\![b_1]\!]_s = \top \\ [\![b_2]\!]_s & caso\ contrário \end{cases} \\
[\![b_1 \to b_2]\!]_s & = & \begin{cases} \bot & se\ ([\![b_1]\!]_s = \top) \wedge ([\![b_2]\!]_s = \bot) \\ \top & caso\ contrário \end{cases} \\
[\![b_1 \to b_2]\!]_s & = & \begin{cases} \top & se\ [\![b_1]\!]_s = \bot \\ [\![b_2]\!]_s & caso\ contrário \end{cases}
\end{array}
$$

Notação 2. *Dado um estado s e dada uma expressão booleana b,*
Usaremos a notação $s \models b$ para expressar que $[\![b]\!]_s = \top$.
Usaremos a notação $s \not\models b$ para expressar que $[\![b]\!]_s = \bot$.

As expressões são interpretadas no estilo denotacional.

De imediato, surge a propriedade expressa pelo Lema 2.2.5 (pág. 13) , análoga à propriedade expressa pelo Lema 2.2.3 (pág. 11) para expressões inteiras. Intuitivamente, afirma que para quaisquer dois estados, se todas as variáveis presentes numa expressão de $\mathcal{E}xpression_{Boolean}$ **têm atribuido o mesmo valor**, i.e., a aplicação desses dois estados às variáveis presentes na expressão resulta no mesmo valor, então o resultado de **avaliar a expressão** nesses dois estádos é igual.

Lema 2.2.5. *Para quaisquer $b \in \mathcal{E}xpression_{Boolean}$*

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV(\ b\ ),\ s(\ y\ ) \ =\ s'(\ y\ )) \Rightarrow [\![b]\!]_s \ \Leftrightarrow\ [\![b]\!]_{s'}$$

Demonstração. Por indução estrutural em $\mathcal{E}xpression_{Boolean}$.

i)
Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV(\ \top\ ),\ s(\ y\ ) \ =\ s'(\ y\ )) \Rightarrow [\![\top]\!]_s \ \Leftrightarrow\ [\![\top]\!]_{s'}$$

[1] Nesta expressão o símbolo $\square$ significa a respectiva operação binária em $\mathbb{Z}$ com a interpretação usual.

Pela Convenção 2.2.4 (pág. 13) , $[\![\top]\!]_s \Leftrightarrow \top \Leftrightarrow [\![\top]\!]_{s'}$.

ii)
Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV(\ \bot\ ),\ s(\ y\ )\ =\ s'(\ y\ )) \Rightarrow [\![\ \bot\ ]\!]_s\ \Leftrightarrow\ [\![\ \bot\ ]\!]_{s'}$$

Pela Convenção 2.2.4 (pág. 13) , $[\![\bot]\!]_s \Leftrightarrow \bot \Leftrightarrow [\![\bot]\!]_{s'}$.

iii) Para quaisquer $e_i,\ e_j \in \mathcal{E}xpression_{Integer}$ e $\square \in \{\ \leq, <, >, \geq, =, \neq\ \}$,

Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV(\ e_i \square e_j\ ),\ s(\ y\ )\ =\ s'(\ y\ )) \Rightarrow [\![\ e_i \square e_j\ ]\!]_s\ \Leftrightarrow\ [\![\ e_i \square e_j\ ]\!]_{s'}$$

Pela Convenção 2.2.4 (pág. 13) ,
$\forall\ e_1,\ e_2 \in \mathcal{E}xpression_{Integer},\ \forall\ s_0 \in \Sigma :\ [\![\ e_1 \square e_2\ ]\!]_{s_0} = [\![\ e_1\ ]\!]_{s_0} \square [\![\ e_2\ ]\!]_{s_0}$.

Basta mostrar que $[\![\ e_i\ ]\!]_s \square [\![\ e_j\ ]\!]_s = [\![\ e_i\ ]\!]_{s'} \square [\![\ e_j\ ]\!]_{s'}$
Pelo Lema 2.2.3 (pág. 11) ,
tomando $\forall s_i,\ s'_i \in \Sigma,\ (\forall y_i \in FV(\ e_i\ ),\ s_i(\ y_i\ )\ =\ s'_i(\ y_i\ )) \Rightarrow [\![\ e_i\ ]\!]_{s_i}\ =\ [\![\ e_i\ ]\!]_{s'_i}$.
tomando $\forall s_j,\ s'_j \in \Sigma,\ (\forall y_j \in FV(\ e_j\ ),\ s_j(\ y_j\ )\ =\ s'_j(\ y_j\ )) \Rightarrow [\![\ e_j\ ]\!]_{s_j}\ =\ [\![\ e_j\ ]\!]_{s'_j}$.
e tomando em conta que $FV(\ e_i \square e_j\ ) = FV(\ e_i\ ) \cup FV(\ e_j\ )$.

iv) Para qualquer $b \in \mathcal{E}xpression_{Boolean}$,

Por hipótese de indução temos,

$$HI:\ \ \forall s,\ s' \in \Sigma,\ (\forall y \in FV(\ b\ ),\ s(\ y\ )\ =\ s'(\ y\ )) \Rightarrow [\![\ b\ ]\!]_s\ \Leftrightarrow\ [\![\ b\ ]\!]_{s'}$$

Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV(\ \neg\ b\ ),\ s(\ y\ )\ =\ s'(\ y\ )) \Rightarrow [\![\ \neg\ b\ ]\!]_s\ \Leftrightarrow\ [\![\ \neg\ b\ ]\!]_{s'}$$

Pela Convenção 2.2.4 (pág. 13) , $[\![\ \neg\ b\ ]\!]_s = \neg\ (\ [\![\ b\ ]\!]_s\ )$.
Segue da hipótese e de $FV(\ /b\ ) = FV(\ b\ )$.

v) Para quaisquer $b_i,\ b_j \in \mathcal{E}xpression_{Boolean}$ e $\square \in \{\ \wedge, \vee, \rightarrow\ \}$,

Por hipótese de indução temos,

$$HI_i:\ \ \forall s_i,\ s'_i \in \Sigma,\ (\forall y_i \in FV(\ b_i\ ),\ s_i(\ y_i\ )\ =\ s'_i(\ y_i\ )) \Rightarrow [\![\ b_i\ ]\!]_{s_i}\ \Leftrightarrow\ [\![\ b_i\ ]\!]_{s'_i}$$

$$HI_j:\ \ \forall s_j,\ s'_j \in \Sigma,\ (\forall y_j \in FV(\ b_j\ ),\ s_j(\ y_j\ )\ =\ s'_j(\ y_j\ )) \Rightarrow [\![\ b_j\ ]\!]_{s_j}\ \Leftrightarrow\ [\![\ b_j\ ]\!]_{s'_j}$$

Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV(\ b_i \square b_j\ ),\ s(\ y\ )\ =\ s'(\ y\ )) \Rightarrow [\![\ b_i \square b_j\ ]\!]_s\ \Leftrightarrow\ [\![\ b_i \square b_j\ ]\!]_{s'}$$

Pela Convenção 2.2.4 (pág. 13) ,

$$\forall b_i,\ b_j \in \mathcal{E}xpression_{Boolean},\ \forall\ s_0 \in \Sigma :\ [\![\ b_i \square b_j\ ]\!]_{s_0} = [\![\ b_i\ ]\!]_{s_0} \square [\![\ b_j\ ]\!]_{s_0}.$$

Basta mostrar que $[\![\ b_i\ ]\!]_s \square [\![\ b_j\ ]\!]_s = [\![\ b_i\ ]\!]_{s'} \square [\![\ b_j\ ]\!]_{s'}$
Pelas hipóteses HI_i e HI_j, tomando $s_i = s_j = s$ e $s'_i = s'_j = s'$.
e por $FV(\ b_i \square b_j\ ) = FV(\ b_i\ ) \cup FV(\ b_j\ )$.

□

2.2.4 Estados com erro

Para estudar o objecto $\mathcal{T}_3$, Definição 3.3.8 (pág. 53) , e propriedades de safety Definição 4.2.18 (pág. 72) , necessitaremos de uma forma de exprimir a ocorrência de um erro, uma condição indesejável. Um erro não é um estado, e para o propósito do nosso estudo, não é possível recuperar de um erro. Pelo que um erro é algo diferente de um estado que optamos por descrever como um símbolo, um objecto. O estado de erro não possui qualquer informação sobre as variáveis manipuladas pelo programa. Pode ser considerado como um estado nulo, presente apenas para simbolizar algo que correu mal, uma excepção abstracta. Um erro resulta de uma afirmação que não é verdadeira. Indicando que o programa falha a sua especificação em pelo menos um ponto, é a intuição deste objecto. Note-se que não estudamos a ocorrência de mais do que uma falha na verificação, uma vez que ocorrido um erro, tem de se decidir como é a progressão dos estados relativamente ao erro ocorrido. Este complexidade acrescida, deixamos ao leitor para futura investigação.

Encaramos o estado como uma função total. O programa trabalha sobre todas as variáveis do conjunto finito $\mathcal{V}_{Integer}$, onde estas estão inicializadas por um valor padrão. O programa resume-se a manipular estas variáveis, não se preocupando com a geração nem com a eliminação de variávies.

Definição 2.2.6. *Assumimos a existência de um único* **estado de erro**, *que denotamos por* erro. *O conjunto de todos os elementos de* Σ, *ao qual adicionamos um elemento* erro, *i.e.,* $\Sigma_{\text{erro}} = (\Sigma + \{ \text{ erro } \})$ *denotamos por* Σ_{erro}.

O objecto *erro* é associado como o elemento único que representa o incumprimento do programa face a, pelo menos, uma das suas afirmações.

O estado particular *erro*, é único e representa todos erros ocorridos no programa, no caso particular deste estudo, todos os erros ocorridos no programa também são um e um só, visto que decorrem da não validação de uma expressão booleana, $s \not\models e$ Definição 2.2.4 (pág. 13) . Precisamos de exprimir este comportamento, Regra de inferência 4.8 (pág. 72) .

A ocorrência do estado de erro é um produto gerado pela relação $\rightsquigarrow$, para exprimir o significado de executar erros nos programas, conforme

Nesta linha de raciocínio, não faz sentido exprimir as seguintes ideias:

O estado de erro não informa sobre variáveis, i.e., para $s = erro$, $s(x)$

O estado de erro permite a substituição do valor das variáveis, i.e., para $s = erro$, $s \begin{pmatrix} x \\ z \end{pmatrix}$

O estado de erro não valida uma expressão, i.e., para $s = erro$, $s \models z$, $s \models b$

A preocupação centra-se com as questões de lifting do conjunto de estados, onde decidimos usar a Notação 2.2.6 (pág. 16) para indicar que um elemento (o *erro*) é adicionado ao conjunto, na relação $\rightsquigarrow$. i.e., temos um conjunto de estados a que adicionamos um elemento, o elemento que indica um erro, que nada deve ocorrer da computação.

Existem sistemas formais que raciocínam sobre injecções, mas consideramos que não faz parte do âmbito do projecto, entrar em detalhes nesse sentido. Este trabalho procura-se focar na correcção do processo **Bounded Model Checking**.

2.2.5 Relação de avaliação $\rightsquigarrow$

Definida a noção de estado de um programa, Subsecção 2.2.1 (pág. 10) , necessitaremos de definir a relação de avaliação entre estados que procuramos modelar com esta linguagem de programação. A relação ternária $\rightsquigarrow$, expressa o significado de computar as construções presentes na definição 2.1.7 (pág. 9) . $\rightsquigarrow$ relaciona um comando $c \in \mathcal{C}\textbf{ommand}$ e um estado $s \in \Sigma$ arbitrários, com o resultado da computação de c em s, em semântica *big-step*.Conforme expresso pela propriedade de `safety` presente na Definição 4.2.18 (pág. 72) . A relação representa a alteração de um estado s arbitrário para o estado s' (exprimindo o resultado de uma computação bem sucedida) ou um *erro*, os possíveis resultados da execução do comando $c \in \mathcal{C}\textbf{ommand}$.

Definição 2.2.7. *A relação $\rightsquigarrow$ será representada por:*

$$\forall c \in \mathcal{C}\textbf{ommand},\ s \in \Sigma,\ s' \in \Sigma_{\text{erro}}, (\ c,\ s\ ) \rightsquigarrow s'$$

*A relação de avaliação $\rightsquigarrow$ é uma relação ternária entre $\mathcal{C}$**ommand**, estados e estados com erro, ou seja, $\rightsquigarrow \subseteq \mathcal{C}\textbf{ommand} \times \Sigma \times \Sigma_{\text{erro}}$. A relação $\rightsquigarrow$ é definida indutivamente pelas regras presentes na Tabela 2.4 (pág. 18) .*

De seguida analisaremos as alterações de estados produzidos por cada uma das regras.
N.B.: Em $(\ c,\ s\ ) \rightsquigarrow s'$, s é necessáriamente um estado que não é *erro*.

Regra SKIP A aplicação do comando **skip** não produz qualquer alteração ao estado, sendo que no final da computação o estado final é exactamente o estado inicial, e vice versa. Este comando permite "transitar" para o mesmo estado de imediato.

$$\frac{}{(\ \textbf{skip},\ s\ ) \rightsquigarrow s}\ \textbf{SKIP}$$

Regra ASSERT A aplicação do comando **assert**$(\ b\ )$ permite, num determinado "ponto" de execução, afirmar propriedades que se espera que o estado do programa satisfaça nesse "ponto" de execução. O comando **assert**$(\ b\ )$ não deve alterar o estado corrente de execução caso a afirmação seja avaliada como verdadeira no presente estado. Este comando também permite "transitar" para o mesmo estado, na condição de que a avaliação da expressão booleana b seja verdadeira.

$$\frac{s \models b}{(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s}\ \textbf{ASSERT}$$

Regra ASSERT - Error Por outro lado, este comando, deve indicar que o programa entra no incumprimnto da asserção/afirmação, transitando para o estado de erro.

$$\frac{s \not\models b}{(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{ASSERT - Error}$$

$$\frac{}{(\ \mathbf{skip},\ s\ ) \rightsquigarrow s}\ \textbf{SKIP}$$

$$\frac{s \models b}{(\ \mathbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s}\ \textbf{ASSERT}$$

$$\frac{s \not\models b}{(\ \mathbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{ASSERT - Error}$$

$$\frac{}{(\ x\ :=\ e,\ s\ ) \rightsquigarrow s\begin{pmatrix} x \\ s \models e \end{pmatrix}}\ \textbf{ASSIGN}$$

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow \mathit{erro}}{(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{COMPOSITION - Break}$$

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow s'' \quad s'' \neq \mathit{erro} \quad (\ c_j,\ s''\ ) \rightsquigarrow s'}{(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow s'}\ \textbf{COMPOSITION}$$

$$\frac{s \models b \quad (\ c_t,\ s\ ) \rightsquigarrow s'}{(\ \mathbf{if}(\ b\ )\mathbf{then}\ c_t\ \mathbf{else}\ c_f,\ s\ ) \rightsquigarrow s'}\ \textbf{IF - True}$$

$$\frac{s \not\models b \quad (\ c_f,\ s\ ) \rightsquigarrow s'}{(\ \mathbf{if}(\ b\ )\mathbf{then}\ c_t\ \mathbf{else}\ c_f,\ s\ ) \rightsquigarrow s'}\ \textbf{IF - False}$$

$$\frac{s \not\models b}{(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow s}\ \textbf{WHILE - False}$$

$$\frac{s \models b \quad (\ c_w,\ s\ ) \rightsquigarrow \mathit{erro}}{(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{WHILE - Break}$$

$$\frac{s \models b \quad (\ c_w,\ s\ ) \rightsquigarrow s'' \quad (\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s''\ ) \rightsquigarrow s'}{(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow s'}\ \textbf{WHILE - True}$$

Tabela 2.4: Regras da semântica operacional

Regra ASSIGN A aplicação do comando $x := e$ permite, num determinado

O mecanismo de manipulação de variáveis é essencialmente uma substituição, onde o valor tomado pela variável passa a ser o valor expresso pela expressão no estado em que é avaliada esta manipulação.

$$\frac{}{(x := e,\ s) \rightsquigarrow s \begin{pmatrix} x \\ s \models e \end{pmatrix}}\ \textbf{ASSIGN}$$

Regra COMPOSITION O comando c_i **;** c_j permite a composição de comandos, realizando a (avaliação/entrelaçamento) dos estados. Este comando representa o poder imperativo dos programas onde o estado inicial de cada comando é providenciado pelo resultado da computaçã do comando prévio. Existe assim uma conservação da computação através de estados de computação.

$$\frac{(c_i,\ s) \rightsquigarrow s'' \quad s'' \neq \mathit{erro} \quad (c_j,\ s'') \rightsquigarrow s'}{(c_i \textbf{ ; } c_j,\ s) \rightsquigarrow s'}\ \textbf{COMPOSITION}$$

Regra COMPOSITION - Break

$$\frac{(c_i,\ s) \rightsquigarrow \mathit{erro}}{(c_i \textbf{ ; } c_j,\ s) \rightsquigarrow \mathit{erro}}\ \textbf{COMPOSITION - Break}$$

Se ocorre um erro na computação do comando antecedente, todo o programa passa ao estado de erro, sendo desnecessário qualquer tipo de avaliação posterior. Neste estudo não é possível recuperar de um erro, o programa não verificou uma afirmação presente na sua especificação.
Regra IF - True A aplicação do comando **if**(b)**then** c_t **else** c_f permite, num determinado

$$\frac{s \models b \quad (c_t,\ s) \rightsquigarrow s'}{(\textbf{if}(b)\textbf{then } c_t \textbf{ else } c_f,\ s) \rightsquigarrow s'}\ \textbf{IF - True}$$

Regra IF - False No caso em que o estado não valida a avaliação da expressões booleanas b, este comando transita pelo resultado da computação c_f.

$$\frac{s \not\models b \quad (c_f,\ s) \rightsquigarrow s'}{(\textbf{if}(b)\textbf{then } c_t \textbf{ else } c_f,\ s) \rightsquigarrow s'}\ \textbf{IF - False}$$

A aplicação do comando **while**(b)**do** c_w procura exprimir ciclos num determinado "ponto" da computação. Procura-se exprimir que o conteúdo de um ciclo é executado um determinado número de vezes, mais ainda o número de execuções do ciclo é definido por uma condição, neste caso expressa na classe $\mathcal{E}xpression_{Boolean}$.

Regra WHILE - False Se a condição for falsa, i.e., o estado de execução não validar a condição, nada mais é executado e é retornado o estado prévio, com as computações executadas previamente.

$$\frac{s \not\models b}{(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow s}\ \mathbf{WHILE\ -\ False}$$

Regra WHILE - Break Caso a condição seja validada no estado de execução, queremos indicar que o conteúdo do ciclo é executado, e o resultado dessa computação é transmitido às próximas computações. Assim se $s \models b$ e $(\ c_w,\ s\ ) \rightsquigarrow s''$, caso o resultado da computação do comando c_w, seja *erro*, devemos indicar que, independentemente de posteriores computações, o programa termina no estado de erro.

$$\frac{s \models b \quad (\ c_w,\ s\ ) \rightsquigarrow \mathit{erro}}{(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow \mathit{erro}}\ \mathbf{WHILE\ -\ Break}$$

Regra WHILE - True Caso contrário, o resultado da computação do conteúdo do ciclo s'', expresso por $(\ c_w,\ s\ ) \rightsquigarrow s''$, deve ser transmitido ao próximo passo de computação, que se espera ser outra iteração do ciclo, se b ainda se verificar em s''.

$$\frac{s \models b \quad (\ c_w,\ s\ ) \rightsquigarrow s'' \quad (\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s''\ ) \rightsquigarrow s'}{(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow s'}\ \mathbf{WHILE\ -\ True}$$

Note-se que se $(\ c,\ s\ ) \rightsquigarrow s'$, então existe uma árvore de derivação na semântica operacional do comando c com o estado inicial s e o resultado s'.

2.2.6 Propriedades elementares da relação de avaliação

De imediato, surge a propriedade expressa pelo Lema 2.2.8 (pág. 21) . Intuitivamente, afirma que para quaisquer dois estados, o valor das variáveis que não são manipuladas pelo comando, não se alteram depois da computação, permanecendo iguais.

Lema 2.2.8. *Para qualquer* $c \in \mathcal{C}$**ommand**,

$$\forall s, s' \in \Sigma,\ (\ c,\ s\ ) \rightsquigarrow s' \wedge s' \neq \text{erro} \Rightarrow \forall y \notin \textbf{\textit{assign}}(\ c\ ),\ s(\ y\ )\ =\ s'(\ y\ )$$

Demonstração. Por indução estrutural na relação $\rightsquigarrow$.

Caso Regra SKIP:
Para qualquer $s \in \Sigma$,

Queremos mostrar que

$$\begin{array}{c}(\ \textbf{skip},\ s\ ) \rightsquigarrow s \wedge s \neq \textit{erro} \\ \Downarrow \\ \forall y \notin \textbf{assign}(\ \textbf{skip}\ ),\ s(\ y\ )\ =\ s(\ y\ )\end{array}$$

Neste caso, pela suposição $(\ \textbf{skip},\ s\ ) \rightsquigarrow s'$ e $s' \neq \textit{erro}$.
sabemos que $(\ \textbf{skip},\ s\ ) \rightsquigarrow s$.

Pretendemos mostrar que para todo $y \notin \textbf{assign}(\ \textbf{skip}\ ),\ s(\ y\ ) = s'(\ y\ )$.

Como $(\ \textbf{skip},\ s\ ) \rightsquigarrow s$, temos que $s' = s$, logo para todo $y \notin \textbf{assign}(\ \textbf{skip}\ ),\ s(\ y\ ) = s'(\ y\ )$.

Caso Regra ASSERT:
Para quaisquer $b \in \mathcal{E}xpression_{Boolean}$ e $s \in \Sigma$,

Por hipótese de indução temos,

$$s \models b$$

Queremos mostrar que

$$\begin{array}{c}(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s \wedge s \neq \textit{erro} \\ \Downarrow \\ \forall y \notin \textbf{assign}(\ \textbf{assert}(\ b\ )\ ),\ s(\ y\ )\ =\ s(\ y\ )\end{array}$$

Neste caso, pela suposição $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s'$ e $s' \neq \textit{erro}$.
sabemos que $s \models b$ e $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s$.

Pretendemos mostrar que para todo $y \notin \textbf{assign}(\ \textbf{assert}(\ b\ )\ ),\ s(\ y\ ) = s'(\ y\ )$.

Como $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s$, temos que $s' = s$, logo para todo $y \notin \textbf{assign}(\ \textbf{assert}(\ b\ )\ ),\ s(\ y\ ) = s'(\ y\ )$.

Caso Regra ASSERT - Error:
Para quaisquer $b \in \mathcal{E}xpression_{Boolean}$ e $s \in \Sigma$,

Por hipótese de indução temos,

$$s \not\models b$$

Queremos mostrar que

$$\begin{array}{c}(\textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow \mathit{erro} \wedge \mathit{erro} \neq \mathit{erro} \\ \Downarrow \\ \forall y \notin \textbf{assign}(\ \textbf{assert}(\ b\ )\ ),\ s(\ y\ )\ =\ \mathit{erro}(\ y\ )\end{array}$$

Neste caso, pela suposição $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s'$ e $s' \neq \mathit{erro}$,
sabemos que $s \not\models b$ e $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow \mathit{erro}$.

Pretendemos mostrar que para todo $y \notin \textbf{assign}(\ \textbf{assert}(\ b\ )\ ),\ s(\ y\ ) = s'(\ y\ )$.

Sabemos contudo que a aplicação desta regra, pelas suposições, nunca ocorre, pois $s' = \mathit{erro}$.

Caso Regra ASSIGN:
Para quaisquer $x \in \mathcal{V}_{Integer}$, $e \in \mathcal{E}xpression_{Integer}$ e $s \in \Sigma$,

Queremos mostrar que

$$\begin{array}{c}(x\ :=\ e,\ s\ ) \rightsquigarrow s\begin{pmatrix} x \\ e \end{pmatrix} \wedge s\begin{pmatrix} x \\ e \end{pmatrix} \neq \mathit{erro} \\ \Downarrow \\ \forall y \notin \textbf{assign}(\ x\ :=\ e\ ),\ s(\ y\ )\ =\ s\begin{pmatrix} x \\ e \end{pmatrix}(\ y\ )\end{array}$$

Neste caso, pela suposição $(\ x\ :=\ e,\ s\ ) \rightsquigarrow s'$ e $s' \neq \mathit{erro}$,
sabemos que $(\ c_i,\ s\ ) \rightsquigarrow s\begin{pmatrix} x \\ e \end{pmatrix}$.

Pretendemos mostrar que para todo $y \notin \textbf{assign}(\ x\ :=\ e\ ),\ s(\ y\ ) = s'(\ y\ )$.

Apenas o valor da variável x é alterado, pela definição de substituição, mantendo os restantes,
pelo que $\forall\ y \notin \textbf{assign}(\ x\ :=\ e\ ),\ s(\ y\ ) = s\begin{pmatrix} x \\ e \end{pmatrix}(\ y\ )$ e $y \neq\ x$

Caso Regra COMPOSITION - Break:
Para quaisquer $s \in \Sigma$,

Por hipótese de indução temos,

$$HI_i : \begin{array}{c} (\ c_i,\ s\ ) \rightsquigarrow \mathit{erro} \wedge \mathit{erro} \neq \mathit{erro} \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\ c_i\ ),\ s(\ y\ ) \ =\ \mathit{erro}(\ y\ ) \end{array}$$

Queremos mostrar que

$$\begin{array}{c} (\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow \mathit{erro} \wedge \mathit{erro} \neq \mathit{erro} \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\ c_i\ ;\ c_j\ ),\ s(\ y\ ) \ =\ \mathit{erro}(\ y\ ) \end{array}$$

Neste caso, pela suposição $(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow s'$ e $s' \neq \mathit{erro}$,
sabemos que $(\ c_i,\ s\ ) \rightsquigarrow \mathit{erro}$.

Pretendemos mostrar que para todo $y \notin \mathbf{assign}(\ c_i\ ;\ c_j\ ),\ s(\ y\ ) = s'(\ y\ )$.

Sabemos contudo que a aplicação desta regra, pelas suposições, nunca ocorre, pois $s' = \mathit{erro}$.

Caso Regra COMPOSITION:
Para quaisquer $s \in \Sigma$ e $s' \in \Sigma_{\mathit{erro}}$,

Por hipótese de indução temos,

$$\exists\ s'' \in \Sigma :,\ s'' \neq \mathit{erro},\ \begin{cases} HI_i : & \begin{array}{c} (\ c_i,\ s\ ) \rightsquigarrow s'' \wedge s'' \neq \mathit{erro} \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\ c_i\ ),\ s(\ y\ ) \ =\ s''(\ y\ ) \end{array} \\ \\ HI_j : & \begin{array}{c} (\ c_j,\ s''\ ) \rightsquigarrow s' \wedge s' \neq \mathit{erro} \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\ c_j\ ),\ s''(\ y\ ) \ =\ s'(\ y\ ) \end{array} \end{cases}$$

Queremos mostrar que

$$\begin{array}{c} (\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow s' \wedge s' \neq \mathit{erro} \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\ c_i\ ;\ c_j\ ),\ s(\ y\ ) \ =\ s'(\ y\ ) \end{array}$$

Neste caso, pela suposição $(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow s'$ e $s' \neq \mathit{erro}$,
sabemos que existe $s'' \neq \mathit{erro}$ tal, que $(\ c_i,\ s\ ) \rightsquigarrow s''$ e $(\ c_j,\ s''\ ) \rightsquigarrow s'$.

Pretendemos mostrar que para todo $y \notin \mathbf{assign}(\ c_i\ ;\ c_j\ ),\ s(\ y\ ) = s'(\ y\ )$.

Seja $y \notin \mathbf{assign}(\ c_i\ ;\ c_j\ )$. Então:
i) como $\mathbf{assign}(\ c_i\ ;\ c_j\ ) = \mathbf{assign}(\ c_i\ ) \cup \mathbf{assign}(\ c_j\ )$ 2.1.9 (pág. 9) ,
temos $y \notin \mathbf{assign}(\ c_i\ )$ e $y \notin \mathbf{assign}(\ c_j\ )$,
e pela HI_i relativa à avaliação $(\ c_i,\ s\ ) \rightsquigarrow s''$, segue $s(\ y\ ) = s''(\ y\ )$;

ii) pela HI_j relativa à avaliação $(c_j,\ s'') \rightsquigarrow s'$, segue $s''(y) = s'(y)$.

Consequentemente, $s(y) = s'(y)$.

□

Demonstração. (Continuação)
Caso Regra IF - True:
Para quaisquer $b \in \mathcal{E}xpression_{Boolean}$, $s \in \Sigma$ e $s' \in \Sigma_{erro}$,

Por hipótese de indução temos,

$$HI_b : \quad s \models b$$

$$HI_t : \quad \begin{array}{c} (c_t,\ s) \rightsquigarrow s' \wedge s' \neq erro \\ \Downarrow \\ \forall y \notin \mathbf{assign}(c_t),\ s(y) \ = \ s'(y) \end{array}$$

Queremos mostrar que

$$\begin{array}{c} (\mathbf{if}(b)\mathbf{then}\ c_t\ \mathbf{else}\ c_f,\ s) \rightsquigarrow s' \wedge s' \neq erro \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\mathbf{if}(b)\mathbf{then}\ c_t\ \mathbf{else}\ c_f),\ s(y) \ = \ s'(y) \end{array}$$

Neste caso, pela suposição $(\mathbf{if}(b)\mathbf{then}\ c_t\ \mathbf{else}\ c_f,\ s) \rightsquigarrow s'$ e $s' \neq erro$,
sabemos que $s \models b$ e $(c_t,\ s) \rightsquigarrow s'$.

Pretendemos mostrar que para todo $y \notin \mathbf{assign}(\mathbf{if}(b)\mathbf{then}\ c_t\ \mathbf{else}\ c_f),\ s(y) = s'(y)$.

Seja $y \notin \mathbf{assign}(\mathbf{if}(b)\mathbf{then}\ c_t\ \mathbf{else}\ c_f)$. Então:
i) como $\mathbf{assign}(\mathbf{if}(b)\mathbf{then}\ c_t\ \mathbf{else}\ c_f) = \mathbf{assign}(c_t) \cup \mathbf{assign}(c_f)$ 2.1.9 (pág. 9) ,
temos $y \notin \mathbf{assign}(c_t)$ e $y \notin \mathbf{assign}(c_f)$,

ii) pela HI_t relativa à avaliação $(c_t,\ s) \rightsquigarrow s'$, segue $s(y) = s'(y)$.

Caso Regra IF - False:
Para quaisquer $b \in \mathcal{E}xpression_{Boolean}$, $s \in \Sigma$ e $s' \in \Sigma_{erro}$,

Por hipótese de indução temos,

$$HI_b : \quad s \not\models b$$

$$HI_f : \quad \begin{array}{c} (c_f,\ s) \rightsquigarrow s' \wedge s' \neq erro \\ \Downarrow \\ \forall y \notin \mathbf{assign}(c_f),\ s(y) \ = \ s'(y) \end{array}$$

Queremos mostrar que

$$\begin{array}{c} (\mathbf{if}(b)\mathbf{then}\ c_t\ \mathbf{else}\ c_f,\ s) \rightsquigarrow s' \wedge s' \neq erro \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\mathbf{if}(b)\mathbf{then}\ c_t\ \mathbf{else}\ c_f),\ s(y) \ = \ s'(y) \end{array}$$

Neste caso, pela suposição $(\mathbf{if}(b)\mathbf{then}\ c_t\ \mathbf{else}\ c_f,\ s) \rightsquigarrow s'$ e $s' \neq erro$,
sabemos que $s \not\models b$ e $(c_f,\ s) \rightsquigarrow s'$.

Pretendemos mostrar que para todo $y \notin \mathbf{assign}(\ \mathbf{if}(\ b\ )\mathbf{then}\ c_t\ \mathbf{else}\ c_f\ ),\ s(\ y\ ) = s'(\ y\ )$.

Seja $y \notin \mathbf{assign}(\ \mathbf{if}(\ b\ )\mathbf{then}\ c_t\ \mathbf{else}\ c_f\ )$. Então:
i) como $\mathbf{assign}(\ \mathbf{if}(\ b\ )\mathbf{then}\ c_t\ \mathbf{else}\ c_f\ ) = \mathbf{assign}(\ c_t\ ) \cup \mathbf{assign}(\ c_f\ )$ 2.1.9 (pág. 9) ,
temos $y \notin \mathbf{assign}(\ c_t\ )$ e $y \notin \mathbf{assign}(\ c_f\ )$,

ii) pela HI_f relativa à avaliação $(\ c_f,\ s\ ) \rightsquigarrow s'$, segue $s(\ y\ ) = s'(\ y\ )$.

Caso Regra WHILE - False:
Para quaisquer $b \in \mathcal{E}xpression_{Boolean}$, $s \in \Sigma$,

Por hipótese de indução temos,

$$HI_b: \quad s \not\models b$$

Queremos mostrar que

$$\begin{array}{c}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow s \wedge s \neq \mathit{erro} \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w\ ),\ s(\ y\ )\ =\ s(\ y\ )\end{array}$$

Neste caso, pela suposição $(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow s'$ e $s' \neq \mathit{erro}$,
sabemos que $s \not\models b$.

Como $(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow s$, temos que $s' = s$,
logo para todo $y \notin \mathbf{assign}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w\ ),\ s(\ y\ ) = s'(\ y\ )$.

Caso Regra WHILE - Break:
Para quaisquer $b \in \mathcal{E}xpression_{Boolean}$ e $s \in \Sigma$,

Por hipótese de indução temos,

$$\begin{array}{ll} HI_b: & s \models b \\ \\ HI_w: & \begin{array}{c}(\ c_w,\ s\ ) \rightsquigarrow \mathit{erro} \wedge \mathit{erro} \neq \mathit{erro} \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\ c_w\ ),\ s(\ y\ )\ =\ \mathit{erro}(\ y\ )\end{array}\end{array}$$

Queremos mostrar que

$$\begin{array}{c}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow \mathit{erro} \wedge \mathit{erro} \neq \mathit{erro} \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w\ ),\ s(\ y\ )\ =\ \mathit{erro}(\ y\ )\end{array}$$

Neste caso, pela suposição $(\ c_w,\ s\ ) \rightsquigarrow s'$ e $s' \neq \mathit{erro}$,
sabemos que $s \models b$ e $(\ c_w,\ s\ ) \rightsquigarrow \mathit{erro}$.

Pretendemos mostrar que para todo $y \notin \mathbf{assign}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w\ ),\ s(\ y\ ) = s'(\ y\ )$.

Sabemos contudo que a aplicação desta regra, pelas suposições, nunca ocorre.

Caso Regra WHILE - True:
Para quaisquer $b \in \mathcal{E}xpression_{Boolean}$, $s \in \Sigma$ e $s' \in \Sigma_{erro}$,

Por hipótese de indução temos,

$HI_b : s \models b$

$$\exists\ s'' \in \Sigma,\ s'' \neq erro, \begin{cases} HI_w : & \begin{array}{c}(c_w,\ s) \rightsquigarrow s'' \wedge s'' \neq erro \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\ c_w\ ),\ s(\ y\ ) = s''(\ y\ )\end{array} \\ e & \\ HI_r : & \begin{array}{c}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s''\ ) \rightsquigarrow s' \wedge s' \neq erro \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w\ ),\ s''(\ y\ ) = s'(\ y\ )\end{array} \end{cases}$$

Queremos mostrar que

$$\begin{array}{c}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow s' \wedge s' \neq erro \\ \Downarrow \\ \forall y \notin \mathbf{assign}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w\ ),\ s(\ y\ ) = s'(\ y\ )\end{array}$$

Neste caso, pela suposição $(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s\ ) \rightsquigarrow s'$ e $s' \neq erro$,
sabemos que $s \models b$ e existe $s'' \neq erro$ tal, que $(\ c_w,\ s\ ) \rightsquigarrow s''$ e $(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s''\ ) \rightsquigarrow s'$.

Pretendemos mostrar que para todo $y \notin \mathbf{assign}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w\ ),\ s(\ y\ ) = s'(\ y\ )$.

Seja $y \notin \mathbf{assign}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w\ )$. Então:

i) como $\mathbf{assign}(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w\ ) = \mathbf{assign}(\ c_w\ )$ 2.1.9 (pág. 9) , temos $y \notin \mathbf{assign}(\ c_w\ )$, e pela HI_w relativa à avaliação $(\ c_w,\ s\ ) \rightsquigarrow s''$, segue $s(\ y\ ) = s''(\ y\ )$;

ii) pela HI_r relativa à avaliação $(\ \mathbf{while}(\ b\ )\mathbf{do}\ c_w,\ s''\ ) \rightsquigarrow s'$, segue $s''(\ y\ ) = s'(\ y\ )$.

Consequentemente, $s(\ y\ ) = s'(\ y\ )$.

□

Teorema 2.2.9. *Unicidade da relação* $\rightsquigarrow$

$$\forall s, s', s'' \in \Sigma,\ \forall c \in \mathcal{C}\textbf{ommand},\ (\ c,\ s\ ) \rightsquigarrow s' \wedge (\ c,\ s\ ) \rightsquigarrow s'' \Rightarrow s' = s''$$

2.2.7 Equivalência de programas

Necessitaremos de racícionar sobre a equivalência entre dois comandos. Consideraremos que dois programas são equivalentes se efectuarem a mesma alteração a um dado estado. A equivalência de programas será expresso pela seguinte relação:

Definição 2.2.10. *Para qualquer* $c_1,\ c_2 \in \mathcal{C}\textbf{ommand}$,
A relação $\approx$ *sobre* $\mathcal{C}\textbf{ommand}$ *é definida por:*

$$c_1 \approx c_2 \text{ se } (\ c_1,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ c_2,\ s\ ) \rightsquigarrow s',$$

para qualquer $s, s' \in \Sigma$ *e* $c_1, c_2 \in \mathcal{C}\textbf{ommand}$

Vamos mostrar que a relação definida é uma relação de equivalência. De acordo com [Livro], uma relação R sobre um conjunto A é uma **relação de equivalência** sse R é reflexiva, simétrica e transitiva. Uma relação R sobre um conjunto A é uma **relação reflexiva** sse para todo $a \in A, aRa$. Uma relação R sobre um conjunto A é uma **relação simétrica** sse para quaisquer $a_0, a_1 \in A, a_0Ra_1$ e a_1Ra_0. Uma relação R sobre um conjunto A é uma **relação transitiva** sse para quaisquer $a, b, c \in A$, Se aRb e bRc então aRc.

Teorema 2.2.11. *A relação* $\approx$ *é uma relação de equivalência.*

Demonstração. Queremos mostrar que a relação $\approx$ é uma relação de equivalência.
Basta mostrar que

a) A relação $\approx$ é reflexiva, pelo Lema 2.2.12 (pág. 27)

b) A relação $\approx$ é simétrica, pelo Lema 2.2.13 (pág. 28)

c) A relação $\approx$ é transitiva, pelo Lema 2.2.14 (pág. 28)

□

Reflexividade da relação $\approx$

A relação $\approx$ é uma relação reflexiva sobre o conjunto $\mathcal{C}\textbf{ommand}$. Para quaisquer elemento do conjunto $\mathcal{C}\textbf{ommand}$, o elemento está relacionado com ele próprio na relação $\approx$.

Esta propriedade da relação $\approx$ expressa-se da seguinte forma.

Lema 2.2.12. *Para qualquer* $c \in \mathcal{C}\textbf{ommand}$

$$c \approx c$$

Demonstração.
Para qualquer $c \in \mathcal{C}\textbf{ommand}$
Queremos mostrar que
$c \approx c$, i.e., para quaisquer $s,\ s' \in \Sigma$, $(\ c,\ s\ ) \rightsquigarrow s'$ sse $(\ c,\ s\ ) \rightsquigarrow s'$

Ora LHS = RHS.

□

Simetria da relação $\approx$

A relação $\approx$ é uma relação simétrica sobre o conjunto $\mathcal{C}$**ommand**. Para quaisquer dois elementos c_1 e c_2 do conjunto $\mathcal{C}$**ommand**, se c_1 está relacionado com c_2 na relação $\approx$, então c_2 também está relacionado com c_1 na relação $\approx$.

Esta propriedade da relação $\approx$ expressa-se da seguinte forma.

Lema 2.2.13. *Para quaisquer* $c_1,\ c_2 \in \mathcal{C}$**ommand**

$$c_1 \approx c_2 \Rightarrow c_2 \approx c_1$$

Demonstração.
Sentido $\Rightarrow$

Suponhamos que

$c_1 \approx c_2$, i.e., para quaisquer $s,\ s' \in \Sigma$, $(\ c_1,\ s\ ) \rightsquigarrow s'$ sse $(\ c_2,\ s\ ) \rightsquigarrow s'$

Queremos mostrar que

$c_2 \approx c_1$, i.e., para quaisquer $s,\ s' \in \Sigma$, $(\ c_2,\ s\ ) \rightsquigarrow s'$ sse $(\ c_1,\ s\ ) \rightsquigarrow s'$

Pela hipótese, $(\ c_1,\ s\ ) \rightsquigarrow s'$ sse $(\ c_2,\ s\ ) \rightsquigarrow s'$.
Logo, $(\ c_1,\ s\ ) \rightsquigarrow s'$ sse $(\ c_1,\ s\ ) \rightsquigarrow s'$.

Sentido $\Leftarrow$

Suponhamos que

$c_2 \approx c_1$, i.e., para quaisquer $s,\ s' \in \Sigma$, $(\ c_2,\ s\ ) \rightsquigarrow s'$ sse $(\ c_1,\ s\ ) \rightsquigarrow s'$

Queremos mostrar que

$c_1 \approx c_2$, i.e., para quaisquer $s,\ s' \in \Sigma$, $(\ c_1,\ s\ ) \rightsquigarrow s'$ sse $(\ c_2,\ s\ ) \rightsquigarrow s'$

Pela hipótese, $(\ c_2,\ s\ ) \rightsquigarrow s'$ sse $(\ c_1,\ s\ ) \rightsquigarrow s'$.
Logo, $(\ c_2,\ s\ ) \rightsquigarrow s'$ sse $(\ c_2,\ s\ ) \rightsquigarrow s'$.

□

Transitividade da relação $\approx$

A relação $\approx$ é uma relação transitiva sobre o conjunto $\mathcal{C}$**ommand**. Para quaisquer três elementos c_1, c_2 e c_3 do conjunto $\mathcal{C}$**ommand**, se c_1 está relacionado com c_2 e c_2 por sua vêz, está relacionado com c_3 na relação $\approx$, então c_1 também está relacionado com c_3 na relação $\approx$.

Esta propriedade da relação $\approx$ expressa-se da seguinte forma.

Lema 2.2.14. *Para quaisquer* $c_1,\ c_2,\ c_3 \in \mathcal{C}$**ommand**

$$(\ c_1 \approx c_2\ ) \wedge (\ c_2 \approx c_3\ ) \Leftrightarrow c_1 \approx c_3$$

Demonstração.
Suponhamos que

HI_a: $c_1 \approx c_2$, i.e., para quaisquer $s_a,\ s'_a \in \Sigma$, $(\ c_1,\ s_a\ ) \rightsquigarrow s'_a$ sse $(\ c_2,\ s_a\ ) \rightsquigarrow s'_a$

HI_b: $c_2 \approx c_3$, i.e., para quaisquer $s_b,\ s'_b \in \Sigma$, $(\ c_2,\ s_b\ ) \rightsquigarrow s'_b$ sse $(\ c_3,\ s_b\ ) \rightsquigarrow s'_b$

Queremos mostrar que

$c_1 \approx c_3$, i.e., para quaisquer $s,\ s' \in \Sigma$, $(\ c_1,\ s\ ) \rightsquigarrow s'$ sse $(\ c_3,\ s\ ) \rightsquigarrow s'$

Da hipótese HI_a, em particular, tomando $s_a = s$ e $s_a = s'$,
Da hipótese HI_b, em particular, tomando $s_b = s$ e $s_b = s'$,
Logo, $(\ c_1,\ s\ ) \rightsquigarrow s'$ sse $(\ c_3,\ s\ ) \rightsquigarrow s'$

□

Provar Compatibilidade

Provar Congruência

Propriedade E: Fecho de compatibilidade sobre a composição, da relação $\approx$

Lema 2.2.15. *Para quaisquer* $c_1,\ c_2,\ c_3,\ c_4 \in \mathcal{C}$**ommand**

$$(\ c_1\ \approx\ c_2\ ) \wedge (\ c_3\ \approx\ c_4\ ) \Rightarrow (\ c_1\ ;c_3\ \approx\ c_2\ ;c_4\ )$$

Demonstração.
Sentido $\Rightarrow$:

Suponhamos que
HI_a:$c_1\ \approx\ c_2$, i.e., para quaisquer $s_a,\ s'_a \in \Sigma$, $(\ c_1,\ s_a\ ) \rightsquigarrow s'_a$ sse $(\ c_2,\ s_a\ ) \rightsquigarrow s'_a$
HI_b:$c_3\ \approx\ c_4$, i.e., para quaisquer $s_b,\ s'_b \in \Sigma$, $(\ c_3,\ s_b\ ) \rightsquigarrow s'_b$ sse $(\ c_4,\ s_b\ ) \rightsquigarrow s'_b$
Queremos mostrar que
$c_1\ ;c_3\ \approx\ c_2\ ;c_4$, i.e., para quaisquer $s,\ s' \in \Sigma$, $(\ c_1\ ;c_3,\ s\ ) \rightsquigarrow s'$ sse $(\ c_2\ ;c_4,\ s\ ) \rightsquigarrow s'$

Suponhamos que $(\ c_1\ ;c_3,\ s\ ) \rightsquigarrow s'$.
Pela Regra de derivação 4.10 (pág. 72) , da relação $\rightsquigarrow$, existe s'' tal que $(\ c_1,\ s\ ) \rightsquigarrow s''$ e $(\ c_3,\ s''\ ) \rightsquigarrow s'$.
Logo, das hipóteses surge, pela Definição 2.2.10 (pág. 27) , que $(\ c_2,\ s\ ) \rightsquigarrow s''$ e $(\ c_4,\ s''\ ) \rightsquigarrow s'$.
Daqui, pela Regra de derivação 4.10 (pág. 72) da relação $\rightsquigarrow$, conclui-se $(\ c_2\ ;c_4,\ s\ ) \rightsquigarrow s'$.

□

O inverso já não se verifica.

Propriedade F da relação $\approx$

Lema 2.2.16. *Para quaisquer* $b \in \mathcal{E}xpression_{Boolean}$ *e* $c_1,\ c_2 \in \mathcal{C}$**ommand**

$$\begin{array}{rl} \textit{Se} & FV(\ b\ ) \cap \textbf{\textit{assign}}(\ c_1\ ) = \emptyset \\ \textit{Então} & \textbf{\textit{if}}(\ b\ )\ \textbf{\textit{then}}\ c_1\ ;c_2\ \approx\ \textbf{\textit{if}}(\ b\ )\ \textbf{\textit{then}}\ c_1\ ;\textbf{\textit{if}}(\ b\ )\ \textbf{\textit{then}}\ c_2 \end{array}$$

Propriedade G da relação $\approx$

Proposição 2.2.17. *Para quaisquer* $c \in \mathcal{C}$**ommand** *e* $s,\ s' \in \Sigma$

$$(\ \textbf{\textit{skip}},\ s\ ) \rightsquigarrow s' \textit{ sse } (\ \textbf{\textit{skip ; skip}},\ s\ ) \rightsquigarrow s'$$

Demonstração.
Por um lado, pela Regra de derivação 4.7 (pág. 72) da relação $\rightsquigarrow$ temos que $s = s'$, i.e. (**skip**, s) $\rightsquigarrow s$.
Por outro lado, existe s'' tal que (**skip**, s) $\rightsquigarrow s''$ e (**skip**, s'') $\rightsquigarrow s'$.
Pela mesma ordem de raciocínio $s = s'' = s'$, pelo que (**skip ; skip**, s) $\rightsquigarrow s$

□

Propriedade H da relação $\approx$

Lema 2.2.18. *Para quaisquer* $b \in \mathcal{E}xpression_{Boolean}$ *e* $c_t,\ c_f \in \mathcal{C}$**ommand**

$$\textbf{\textit{if}}(\ b\ )\textbf{\textit{then}}\ c_t\ \textbf{\textit{else}}\ c_f\ \approx\ \textbf{\textit{if}}(\ b\ )\textbf{\textit{then}}\ c_t\ \textbf{\textit{else skip ; if}}(\ \neg b\ )\textbf{\textit{then}}\ c_f\ \textbf{\textit{else skip}}$$

2.3 Correcção de programas

,

Durante a execução do programa, queremos declarar o que não deve acontecer, ou de igual forma, o que deve sempre acontecer. Necessitamos de uma forma de representar propriedades de "safety" nos comandos.

Para qualquer $c \in \mathcal{C}\mathbf{ommand}$, usaremos a propriedade *safe*(c), para indicar que um programa c executa sem erros. A noção de erro, neste contexto é a execução dum programa que não satisfaz todas as afirmações presentes na sua específicação, i.e., *safe*(c) indica que o programa só transita por estados que verificam as afirmações expressadas pela específicação. Desta forma sabemos que um comando satisfaz todas as suas especificações /afirmações.

Em particular, queremos racícionar sobre as propriedades de safety de um comando, em relação a um estado em particular, e posteriormente em relação a todos os estados.

Assim, a noção expressa em *safe*(c), indica que um programa satisfaz esta propriedade, se não existir nenhum triplo que resultou de uma transição para *erro*, i.e., não existe nenhum triplo da forma ($c,\ s,\ erro$).

Racícionar sobre a propriedade *safe*(c) é racícionar exactamente sobre a definição da relação $\rightsquigarrow$, que parece subtil.

Definição 2.3.1.

$$\text{safe}(\ c,\ s\ ) \Leftrightarrow_{def} (\ c,\ s\ ) \not\rightsquigarrow \text{erro}$$

Definição 2.3.2.

$$\text{safe}(\ c\ ) \Leftrightarrow_{def} \forall\ s \in \Sigma :\ \text{safe}(\ c,\ s\ )$$

Nota 1. *Observe-se que* $(\ c,\ s\ ) \not\rightsquigarrow \text{erro} \Leftrightarrow_{def} \neg(\ (\ c,\ s\ ) \rightsquigarrow \text{erro}\ ) \Leftrightarrow (\ (\ c,\ s\ ) \rightsquigarrow \text{erro} \Rightarrow \bot\ )$
i.e. o triplo $(\ c,\ s,\ \text{erro}\ ) \notin \rightsquigarrow$, *representando que não existe triplo na relação com a forma* $(\ c,\ s,\ \text{erro}\ )$

Propriedades

De imediato surge a propriedade expressa pelo Lema 2.3.3 (pág. 32) . Intuitivamente, afirma que, para quaisquer dois estados $s_a,\ s_b \in \Sigma$, se todas as variáveis presentes numa expressão $b' \in \mathcal{E}xpression_{Boolean}$ **não são atribuidas** pelo comando $c \in \mathcal{C}\mathbf{ommand}$, i.e., se o comando não manipula as variáveis livres da espressão, mas s_b é resultado da transição por s_a, se c é safe, então avaliar b' no estado s_a terá o mesmo resultado de avaliar b' em s_b, e vice-versa.

Lema 2.3.3.

$$\begin{array}{c}(\ FV(\ b'\ ) \cap \boldsymbol{assign}(\ c\ ) = \emptyset\ ) \wedge (\ c,\ s_a\ ) \rightsquigarrow s_b \wedge \text{safe}(\ c,\ s\ ) \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{array}$$

Demonstração.
Por indução estrutural na classe $\mathcal{C}\mathbf{ommand}$

i) Queremos mostrar que para quaisquer $b' \in \mathcal{E}xpression_{Boolean}$ e $s_a,\ s_b \in \Sigma$

$$\begin{array}{c}(\ FV(\ b'\ ) \cap \mathbf{assign}(\ \mathbf{skip}\ ) = \emptyset\ ) \wedge (\ \mathbf{skip},\ s_a\ ) \rightsquigarrow s_b \wedge \mathit{safe}(\ c,\ s\ ) \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{array}$$

Pela Regra de inferência 4.7 (pág. 72) , conluímos que $s_a = s_b,\ \therefore s_a \models b' = s_b \models b'$

ii) Queremos mostrar que para quaisquer $b' \in \mathcal{E}xpression_{Boolean}$ e $s_a,\ s_b \in \Sigma$

$$\begin{array}{c}(FV(\ b'\ ) \cap \textbf{assign}(\ \textbf{assert}(\ b\ )\ ) = \emptyset\ ) \wedge (\ \textbf{assert}(\ b\ ),\ s_a\ ) \rightsquigarrow s_b \wedge \mathit{safe}(\ c,\ s\ ) \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{array}$$

Pela suposição *safe*(**assert**(b)), sabemos que $s_a \models b'$.
Pela Regra de inferência 4.8 (pág. 72) , sabemos que $s_a = s_b,\ \therefore s_a \models b' = s_b \models b'$

iii) Queremos mostrar que para quaisquer $b' \in \mathcal{E}xpression_{Boolean}$ e $s_a,\ s_b \in \Sigma$

$$\begin{array}{c}(FV(\ b'\ ) \cap \textbf{assign}(\ x\ :=\ e\ ) = \emptyset\ ) \wedge (\ x\ :=\ e,\ s_a\ ) \rightsquigarrow s_b \wedge \mathit{safe}(\ c,\ s\ ) \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{array}$$

Pela Regra de inferência 4.9 (pág. 72) , sabemos que $s_b = s_a \begin{pmatrix} x \\ e \end{pmatrix}$.
Basta mostrar que $(\ s_a \models b'\ ) = (\ s_a \begin{pmatrix} x \\ e \end{pmatrix} \models b'\ )$

Pela suposição, ($FV(\ b'\ ) \cap \textbf{assign}(\ x\ :=\ e\ ) = \emptyset$),
logo qualquer variável em b' em s_a não é manipulada, e assim, $s_a \models b' = s_b \models b'$.

iv) Suponhamos que para quaisquer $b' \in \mathcal{E}xpression_{Boolean}$ e $s_a,\ s_b \in \Sigma$, $\exists\ s_e \in \Sigma_{erro}$

$$HI_i\text{:}\quad \begin{array}{c}(FV(\ b'\ ) \cap \textbf{assign}(\ c_i\ ) = \emptyset\ ) \wedge (\ c_i,\ s_a\ ) \rightsquigarrow s_e \wedge \mathit{safe}(\ c,\ s\ ) \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_e \models b'\ )\end{array}$$

$$HI_j\text{:}\quad \begin{array}{c}(FV(\ b'\ ) \cap \textbf{assign}(\ c_j\ ) = \emptyset\ ) \wedge (\ c_j,\ s_e\ ) \rightsquigarrow s_b \wedge \mathit{safe}(\ c,\ s\ ) \\ \Downarrow \\ (\ s_e \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{array}$$

Queremos mostrar que

$$\begin{array}{c}(FV(\ b'\ ) \cap \textbf{assign}(\ \textbf{assert}(\ b\ )\ ) = \emptyset\ ) \wedge (\ \textbf{assert}(\ b\ ),\ s_a\ ) \rightsquigarrow s_b \wedge \mathit{safe}(\ c,\ s\ ) \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{array}$$

Pela suposição *safe*(**assert**(b)), sabemos que $s_e \neq \mathit{erro}$, i.e., $s_e \in \Sigma$.
Pelas hipóteses de indução HI_i e HI_j, provamos que $s_a \models b' = s_e \models b' = s_b \models b'$

v) Suponhamos que para quaisquer $b' \in \mathcal{E}xpression_{Boolean}$ e $s_a,\ s_b \in \Sigma$

$$HI_t\text{:}\quad \begin{array}{c}(FV(\ b'\ ) \cap \textbf{assign}(\ c_t\ ) = \emptyset\ ) \wedge (\ c_t,\ s_a\ ) \rightsquigarrow s_b \wedge \mathit{safe}(\ c,\ s\ ) \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{array}$$

$$HI_f\text{:}\quad \begin{array}{c}(FV(\ b'\ ) \cap \textbf{assign}(\ c_f\ ) = \emptyset\ ) \wedge (\ c_f,\ s_a\ ) \rightsquigarrow s_b \wedge \mathit{safe}(\ c,\ s\ ) \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{array}$$

Queremos mostrar que

$$(FV(b') \cap \mathbf{assign}(\mathbf{if}(b)\mathbf{then}\ c_t\ \mathbf{else}\ c_f) = \emptyset) \wedge (\mathbf{if}(b)\mathbf{then}\ c_t\ \mathbf{else}\ c_f,\ s_a) \rightsquigarrow s_b \wedge \mathit{safe}(c,\ s) \Downarrow (s_a \models b') \Leftrightarrow (s_b \models b')$$

Pela Regra de inferência **??** (pág. **??**) ,
caso $s_a \models b'$: Segue da hipótese de indução HI_t.
caso $s_a \not\models b'$: Segue da hipótese de indução HI_f.

vi) Suponhamos que para quaisquer $b' \in \mathcal{E}xpression_{Boolean}$ e $s_a,\ s_b \in \Sigma$, $\exists\ s_e \in \Sigma_{erro}$

$$HI_w\text{:}\quad (FV(b') \cap \mathbf{assign}(c_w) = \emptyset) \wedge (c_w,\ s_a) \rightsquigarrow s_e \wedge \mathit{safe}(c,\ s) \Downarrow (s_a \models b') \Leftrightarrow (s_e \models b')$$

$$HI_r\text{:}\quad (FV(b') \cap \mathbf{assign}(\mathbf{while}(b)\mathbf{do}\ c_w) = \emptyset) \wedge (\mathbf{while}(b)\mathbf{do}\ c_w,\ s_e) \rightsquigarrow s_b \wedge \mathit{safe}(c,\ s) \Downarrow (s_e \models b') \Leftrightarrow (s_b \models b')$$

Queremos mostrar que

$$(FV(b') \cap \mathbf{assign}(\mathbf{while}(b)\mathbf{do}\ c_w) = \emptyset) \wedge (\mathbf{while}(b)\mathbf{do}\ c_w,\ s_a) \rightsquigarrow s_b \wedge \mathit{safe}(c,\ s) \Downarrow (s_a \models b') \Leftrightarrow (s_b \models b')$$

Pela Regra de inferência **??** (pág. **??**) ,
Caso $s_a \not\models b'$: conluímos que $s_a = s_b$, $\therefore s_a \models b' = s_b \models b'$.

Caso $s_a \models b'$: Pela suposição $\mathit{safe}(\ \mathbf{while}(b)\mathbf{do}\ c_w\ )$, sabemos que $s_e \neq \mathit{erro}$, i.e., $s_e \in \Sigma$.
Segue das hipóteses HI_w e HI_r.

□

Surge, também, a propriedade intuitiva, expressa pelo Lema 2.3.3 (pág. 32) , que afirma que se dois comandos são equivalentes, então se se provar que um satifaz a condição $\mathit{safe}(\ c\ )$, o outro também satisfará.

Lema 2.3.4.

$$c_1 \approx c_2 \Rightarrow \text{safe}(\ c_1\ )\ \mathit{sse}\ \text{safe}(\ c_2\ )$$

Demonstração.
Suponhamos que $c_1 \approx c_2$, i.e., $(c_1,\ s) \rightsquigarrow s' \Leftrightarrow (c_2,\ s) \rightsquigarrow s'$
Queremos mostrar que

$$\models \mathit{safe}(\ c_1\ )\ \mathit{sse}\ \models \mathit{safe}(\ c_2\ )$$

No sentido $\Rightarrow$:
Supondo $\mathit{safe}(\ c_1\ )$, i.e., $\forall\ s \neq \mathit{erro},\ (c_1,\ s) \not\rightsquigarrow \mathit{erro}$
e $c_1 \approx c_2$, i.e., $(c_1,\ s) \rightsquigarrow s' \Leftrightarrow (c_2,\ s) \rightsquigarrow s'$
Queremos mostrar que $\mathit{safe}(\ c_2\ )$, i.e., $\forall\ s \neq \mathit{erro},\ (c_2,\ s) \not\rightsquigarrow \mathit{erro}$

Pela Hipótese $c_1 \approx c_2$, $\therefore \forall\ s \neq \mathit{erro},\ (c_2,\ s) \not\rightsquigarrow \mathit{erro}$.

No sentido $\Leftarrow$:
Supondo $\mathit{safe}(\ c_2\ )$, i.e., $\forall\ s \neq \mathit{erro},\ (c_2,\ s) \not\rightsquigarrow \mathit{erro}$

e $c_1 \approx c_2$, i.e., $(c_1, s) \rightsquigarrow s' \Leftrightarrow (c_2, s) \rightsquigarrow s'$
Queremos mostrar que *safe*(c_1), i.e., $\forall\ s \neq erro,\ (c_1, s) \not\rightsquigarrow erro$

Pela Hipótese $c_1 \approx c_2$, $\therefore \forall\ s \neq erro,\ (c_1, s) \not\rightsquigarrow erro$.

□

2.4 Classes Especiais de Programas

2.4.1 Comandos Single Assignment

O single assignment é uma propriedade que afirma que todas as variáveis de um programa, no nosso caso de um comando, são atribuídas uma e uma só vez. Neste sentido, as variáveis são imutáveis. Os comandos **Single Assignment** procuram exprimir esta propriedade em comandos. O método de conversão de um comando para a forma **Single Assignment** é efectuada pela transformação 2 $\mathcal{T}_2(c)$ expressa na secção X.

A classe $\mathcal{SA}$ implementa uma representação intermédia dos comandos $\mathcal{C}$**ommand** onde se garantem as propriedades do **Single Assignment**.

Definimos um comando **Single Assignment** da seguinte forma:

Definição 2.4.1. *A notação $\mathcal{SA}$ representará a classe das expressões de comandos single assignment. A classe sintática $\mathcal{SA}$ é definida indutivamente do seguinte modo:*

$\mathcal{SA}\ c$	::=	**skip**	
	|	**assert**(b)	
	|	$x := e$	, se $x \in \mathcal{V}_{Integer}$
	|	c_i **;** c_j	, se **assign**$(c_i) \cap$ **assign**$(c_j) = \emptyset$
	|		e $FV(c_i) \cap$ **assign**$(c_j) = \emptyset$
	|	**if**(b)**then** c_t **else** c_f	, se **assign**$(c_t) \cap$ **assign**$(c_f) = \emptyset$
	|		e $FV(b) \cap ($ **assign**$(c_t) \cup$ **assign**$(c_f)) = \emptyset$
	|	**while**(b)**do** c_w	, se $FV(b) \cap$ **assign**$(c_w) = \emptyset$

Tabela 2.5: Class of Commands

A classe $\mathcal{SA}$ reflecte comandos da classe $\mathcal{C}$**ommand** cujas variáveis não possuem mais do que uma unica atribuição.

Assim, os comandos **skip**, **assert**(b) e $x := e$ são por si **Single Assignment**, e portanto pertencem à classe $\mathcal{SA}$.

A composição de comandos c_i **;** c_j expressa no entanto duas condições, as variáveis que são atribuidas em c_i e em c_j nunca são iguais, i.e., não pode haver atribuições às mesmas variáveis (**assign**$(c_i) \cap$ **assign**$(c_f) = \emptyset$) e ($FV(b) \cap ($ **assign**$(c_t) \cup$ **assign**$(c_f)) = \emptyset$) que indica que as atribuições às variáveis de c_j nunca podem ser a efectuadas a variáveis que já ocorreram em c_i.

O comando condicional **if**(b)**then** c_t **else** c_f expressa duas condições, as variáveis que são atribuidas no ramo c_t e no ramo c_f nunca são iguais, i.e., não pode haver atribuições às mesmas variáveis (**assign**$(c_t) \cap$**assign**$(c_f) = \emptyset$)(staticSA) e ($FV(b) \cap ($ **assign**$(c_t) \cup$**assign**$(c_f)) = \emptyset$) que indica que as atribuições às variáveis do ramo c_t e do ramo c_f nunca podem ser a efectuadas a variáveis que já ocorreram na condição b.

O comando cíclico **while**(b)**do** c_w $FV(b) \cap$ **assign**$(c_w) = \emptyset$

Observe-se que a classe $\mathcal{SA}$ não necessita tomar em consideração os ciclos while, estando presente para completude, pois para o interesse deste estudo, o **Single Assignment** é uma propriedade de uma representação intermédia, entre as transformações $\mathcal{T}_2$ e $\mathcal{T}_3$

Desta forma garantimos que todas as variáveis do comando, não possuem mais do que uma atribuição.

Tome-se o exemplo da seguinte figura onde o programa 2.1a (pág. 36) resulta na single assignment form expressa pela Figura 2.1b (pág. 36) .

Assume-se que todas as variáveis foram inicializadas com um valor. Assim cada nova atribuição gera uma nova versão da mesma variável onde o indíce 0 representa a versão da variável no estado inicial.

$$\begin{array}{l} a = 0; \\ b = b; \\ b = a + b; \\ c = c + 8; \\ d = a + b - f \end{array}$$

(a) Sequência de comandos

$$\begin{array}{l} a_1 = 0; \\ b_1 = b_0; \\ b_2 = a_1 + b_1; \\ c_1 = c_0 + 8; \\ d_1 = a_1 + b_3 - f_0 \end{array}$$

(b) Resultado da transformação $\mathcal{T}_2(\ c\ )$

Propriedades Single Assignment

2.4.2 Expressões Conditional Normal Form

Queremos referir nos ao resultado produzido pelo objecto $\mathcal{T}_3$, que simbolizará a terceira transformação do **Bounded Model Checking** que iremos estudar. Esta transformação irá produzir um subconjunto da classe $\mathcal{C}$**ommand** cujos elementos têm uma estrutura muito própria.

Ao conjunto de elementos que são produzidos pela transformação $\mathcal{T}_3$, na forma de **if**$(\ b\ )$**then** c_t **else** c_f iremos atribuir ao conjunto $\mathcal{C}nf$, confome definido pela Definição 2.4.2 (pág. 37) .

Por comodidade, e visto que a forma normal produz uma sequência de `if`'s cujo termo falso é o **skip**, iremos simplificar a notação e exprimir estes comandos das **Expressões Conditional Normal Form** , por um `if` que expressa o termo que varia. Conforme apresentado de seguida.

Notação 3. *Para quaisquer* $b \in \mathcal{E}xpression_{Boolean}$ *e* $c \in \mathcal{C}$**ommand**,
Denota-se ***if***$(\ b\ )$ ***then*** c *como a abreviatura* ***if***$(\ b\ )$***then*** c ***else skip***

O método **Bounded Model Checking** exige uma representação intermédia, onde todos o comandos básicos do programa são normalizados na forma $b' \rightarrow\ c$ representado por **if**$(\ b'\ )$ **then** c, onde c são comandos básicos ($c \in \{$ **skip**, **assert**$(\ b\ )$, $x\ :=\ e\ \}$) e $b' \in \mathcal{E}xpression_{Boolean}$ é a representação de um caminho de decisão dos condicionais presentes no programa. I.e., procura-se exprimir que se o traço de execução seguiu os condicionais que geraram o caminho b', então os comandos básicos foram executados.

assert$(\ g == h\ )$;
if$(x == 0)$then{
    $g\ :=\ h + x$;
    assert$(\ (g - h) == x\ )$
}else{
    if(g+h == x)then{
        assert$(\ (x - g - h) == 0\ )$
    }else{
        assert$(\ \bot\ )$
    }
};

(c) programa representado na classe $\mathcal{C}$**ommand**

if $(\top)$ then **assert**$(\ g == h\ )$;
if $(\top \wedge (\ x == 0\ ))$ then $g\ :=\ h + x$;
if $(\top \wedge (\ x == 0\ ))$ then **assert**$(\ (g - h) == x\ )$;
if $(\top \wedge \neg(\ x == 0\ ) \wedge (\ g + h == x\ ))$ then
    assert$(\ (x - g - h) == 0\ )$;
if $(\top \wedge \neg(\ x == 0\ ) \wedge \neg(\ g + h == x\ ))$ then
    assert$(\ \bot\ )$;

(d) programa representado na classe $\mathcal{C}nf$

Desta forma pretendemos representar que a afirmação **assert**$(\ \bot\ )$ só será considerada se a negação dos condição dos condicionais ($x == 0$) e ($g + h == x$) for satisfeita. Surge então a necessidade de definir esta representação intermédia de comandos.

Definição 2.4.2. *A classe $\mathcal{C}nf$ representará o conjunto das* ***Expressões*** **Conditional Normal Form** *. Interpretação usual, em estilo denotacional. As expressões de* **Conditional Normal Form** *são comandos com uma forma especial.*
A classe sintática $\mathcal{C}nf$ é definida recursivamente do seguinte modo:

$\mathcal{C}nf\ c$	::=	**if**(b') **then skip**	*se* $b' \in \mathcal{E}xpression_{Boolean}$
	|	**if**(b') **then assert**(b)	*se* $b' \in \mathcal{E}xpression_{Boolean}$
	|	**if**(b') **then** $x := e$	*se* $b' \in \mathcal{E}xpression_{Boolean}$ e $x \in \mathcal{V}_{Integer}$
	|	c_i **;** c_j	

Tabela 2.6: Classe das **Expressões Conditional Normal Form**

Esta avaliação das espressões de $\mathcal{C}$**ommand** não permite racícionar sobre o conjunto de estados quando é gerado um erro pelo programa. Necessitamos de definir a função $\mathbf{Eval}_{cnf}$ que avaliará o resultado da execução de um programa, independentemente das afirmações presentes serem verdadeiras ou falsas. Este mecanismo irá permitir racicíonar sobre o estádo das variáveis quando o programa não é válido, i.e., transita para *erro*. Assim, o resultado desta função será o resultante da computação de todo o programa. Note-se que não é o estado exactamente anterior à computação que originou o elemento *erro*.

Definição 2.4.3. *A função* $\boldsymbol{Eval}_{cnf}$: **Conditional Normal Form** $\rightarrow \Sigma \rightarrow \Sigma$ *é definida recursivamente por:*

$$
\begin{aligned}
\boldsymbol{Eval}_{cnf}(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then skip}},\ s\ ) &= s \\
\boldsymbol{Eval}_{cnf}(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then assert}}(\ b\ ),\ s\ ) &= s \\
\boldsymbol{Eval}_{cnf}(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then}}\ x := e,\ s\ ) &= \begin{cases} s\begin{pmatrix} x \\ e \end{pmatrix} & s \models b' \\ s & s \not\models b' \end{cases} \\
\boldsymbol{Eval}_{cnf}(\ c_i\ ;\ c_j,\ s\ ) &= \boldsymbol{Eval}_{cnf}(\ c_2,\ \boldsymbol{Eval}_{cnf}(\ c_1,\ s\ )\ )
\end{aligned}
$$

Propriedades para a Representação Intermédia $\mathcal{C}nf$

Capítulo 3

Bounded Model Checking

Neste capítulo apresentamos uma introdução histórica sobre **Bounded Model Checking**. Discutimos a sua origem, de onde surgiu, quais foram as necessidades que levaram a este tipo de investigação, em que ponto se encontra actualmente, as abordagens concorrentes, e abordamos algumas ferramentas que implementam esta técnica para verificar sistemas de Software.

É uma técnica de verificação formal de software que surgiu de uma outra área de investigação. Parece-nos que este facto evidencia a importância do trabalho deste tipo de técnicas visto que surgiu como uma necessidade de uma área que não está normalmente ligada à área de verificação formal.

Essencialmente o **Bounded Model Checking** é um processo de dois passos. Em primeiro lugar um modelo de software é extraido do código fonte, e é codificado numa fórmula lógica o comportamento sequencial do sistema de transições deste modelo, sobre um conjunto finito de estados (esta limitação está na base da designação **Bounded Model Checking**). Seguidamente, no segundo passo, a fórmula lógica produzida no primeiro passo é avaliada por um processo de decisão, usualmente **SAT solver**, por forma a obter uma interpretação lógica que refute a validade do software (providenciando um contra-exemplo ao modelo de software) ou concluia que a fórmula não é satisfazivel, indicando que o modelo é válido até ao limite aplicado sobre o conjunto dos caminhos de execução. Além do limite nada se poderá concluir, uma vez que o método **Bounded Model Checking** carece de **completude**

Este capítulo está dividido em duas secções, na primeira aborda-se o **Model Checking** e como evoluiu para **Symbolic Model Checking** com a utilização de **Binary Decision Diagrams** para verificar modelos, em particular de *hardware*. Em seguida, a segunda secção aborda a técnica **Bounded Model Checking** que evoluiu separadamente para colmatar as deficiências das técnicas abordadas, concluindo com a constatação que a técnica **Bounded Model Checking** aplicada à verificação de *hardware* é uma ideia que também é aplicável à verificação do *software*.

3.1 Model Checking

Na área da verificação, os sistemas de transição de estados modelam de forma efectiva as especificações ou o `design` de diversos projectos de *hardware*. Em especial especificações de harware digital.[art02] As técnicas automáticas de verificação formal de sistemas de transição de estados finitos desenvolveram-se ao ponto em que as empresas de design de chips e *hardware* as têm integrado nos processos de garantia de qualidade do *hardware*. As técnicas de **Model Checking** têm sido as mais adoptadas.(art01[11, 13]).

As técnicas de **Model Checking** permitem verificar de forma algorítmica os complexos sistemas de transição de estados.(art02.9.10.14.29) A designação **Model Checking** refere-se a um conjunto de algoritmos e técnicas de verificação de propriedades em sistemas de transição de estados recorrendo à exploração do espaço de estados. Para determinar se o design obedece à especificação do comportamento pretendido, efectuam-se buscas nos grafos de transição associados aos modelos.(art02[9, 10, 14, 29])

Estes algoritmos conseguem executar uma verificação exaustiva do espaço de estados e conseguem-no fazer de forma automática. Por este motivo, atraíram um elevado interesse por parte da indústria de *hardware*. No entanto, o **Model Checking** é afectado pelo problema da explosão do número de estados que necessitam de ser verificados. Este problema resulta do facto do número de estados, num sistema, crescer exponencialmente com o número de componentes presentes nesse sistema. Muita investigação tem sido dedicada a atenuar este problema.(art02)

As propriedades provadas pelo **Model Checking**, são geralmente classificadas como propriedades de **safety**, declarando o que não deve acontecer (ou equivalentemente, o que deve acontecer sempre) ou como propriedades de **liveness**, propriedades que declaram o que deve acontecer, eventualmente.(art01) Os projectos de *hardware* a serem verificados são modelados por máquinas de estados finitas, e as suas especificações são formalizadas por propriedades em lógica temporal, que é um formalismo para se raciocinar sobre a ordenação temporal de eventos no tempo, sem introduzir explicitamente o elemento tempo. Os estados alcançáveis no sistema de transição de estados são percorridos, a fim de verificar as propriedades modeladas na especificação. Quando uma propriedade não é verificada, é gerado de forma automática um contra-exemplo, na forma de uma sequência de estados, que representa o "raciocínio" que o método constroí para encontrar a falha do modelo, relativamnete à especificação.(art01)

Neste sentido, um contra-exemplo para propriedades `safety` será uma sequência de estados, onde o último estado contradiz a propriedade procurada. Um contra-exemplo para propriedades `liveness`, será algo semelhante a uma sequência de estados até um estado que verifica a propriedade desejada. Um ciclo infinito representa um caminho que nunca atinge o estado desejado na especificação.(art01)

O termo **Model Checking** foi adotado por Clarke e Emerson (art01.[11]) nos anos oitenta. Nos primórdios, os algoritmos de **Model Checking** enumeravam de forma explícita os estados alcançáveis pelo sistema, a fim de verificar a correcção de uma especificação. A capacidade dos programas que utilizavam técnicas de **Model Checking** para verificar projectos ficava assim restringida a uns poucos milhões de estados, visto que o número de estados cresce exponencialmente com o número de variáveis. As implementações iniciais não eram capazes de lidar com projetos de complexidade à escala industrial.(art01) Mesmo os mais avançados algoritmos de **Model Checking** explícito são incapazes de verificar todas as propriedades desejadas no sistema em tempo útil, devido ao problema da explosão dos estados. Contudo, não é possível saber se a especificação de um sistema é correcta ou mesmo completa, não é possível saber se o que se especifica traduz efectivamente o significado que se pretende dar ao sistema. Conclui-se assim que não existe o chamado 'sistema correcto', sendo apenas possível verificar se o sistema satisfaz a especificação dada.

Assim, o **Model Checking** é frequentemente utilizado para encontrar uma falsificação na especificação do sistema, procurando erros lógicos, em vez de provar que eles não existem.

As ferramentas **Model Checking** são muitas vezes capazes de encontrar erros que dificilmente seriam encontrados pela simulação. Os simuladores apenas consideram um número restrito de estados enquanto as ferramentas de **Model Checking** consideram de forma exaustiva todos os

possíveis comportamentos do sistema. Por outro lado, o recurso à lógica temporal clarifica algumas ambiguidades que podem ter sido criadas inadvertidamente pela especificação.

As ferramentas de **Model Checking** não são, por muitos, consideradas como alternativas, mas antes como um complemento essencial aos métodos tradicionais de teste e simulação de sistemas. O facto de se tratar de um método automático, não dependendo de interações complexas com o utilizador para a construção de provas; a geração dos contra exemplos gerados pelo caminho de estados percorrido pelo método; e o facto de se focar em sistemas finitos devido ao recurso a modelos de sistemas de transição de estados; e o recurso a variantes da lógica temporal para a especificação das propriedades da especificação, caracterizam o conceito de **Model Checking** e explicam a razão da sua vasta utilização. Em resumo, o **Model Checking** é uma técnica algorítmica para a verificação de propriedades temporais em sistemas de estados finitos.

Symbolic Model Checking com recurso a Binary Decision Diagrams

Conforme dito anteriormente, as primeiras implementações das técnicas **Model Checking** no início da década de 80 recorriam à representação explícita de grafos de transição dos sistemas de transição de estados que procuravam explorar de forma exaustiva o universo de estados, com recurso a técnicas tão eficientes quanto possíveis de travessia de grafos. Por este motivo, estas técnicas tornam-se inviáveis e inadequados para a maioria das aplicações industriais, pois limitavam projetos de *hardware* a circuitos com cerca de 20 componentes. O universo de estados tornava-se demasiado vasto para ser explorado, em tempo útil.

Técnicas que utilizam a exploração de representações simbolicas do universo de estados de um sistema de transição, surgiram por volta de 1990 (art02[8, 15, 27]). O primeiro avanço no sentido da utilização mais ampla destas técnicas de verificação por parte da indústria foi realizado aquando da introdução do **Symbolic Model Checking** (art01[9, 15]). A técninca **Symbolic Model Checking**, representa e manipula conjuntos integrais de estados implicitamente, através de funções booleanas.

Os **Binary Decision Diagrams** são mais eficientes a lidar com funções booleanas que representam conjuntos de estados ao invés de manter e manipular uma lista extensiva e explícita de estados, como é realizado no **Model Checking** explícito. Em particular, (art01[8]) uma representação em grafo de funções booleanas permite uma manipulação eficiente de fórmulas booleanas através dos conhecidos **Reduced Ordered Binary Decision Diagrams**. O **Symbolic Model Checking** realiza através de **Binary Decision Diagrams**, a travessia do grafo de transição de estados utilizando um algoritmo de pesquisa em largura (`breadth first search`) art02[6]. Os **Binary Decision Diagrams** mantêm as propriedades das funções características dos conjuntos de estados, e permitem a computação das transições entre conjuntos de vários estados, em vez de computar cada estado individualmente.

Explicamos de forma abstracta o funcionamento de uma técnica de **Model Checking** baseada em **Binary Decision Diagrams**.

1 Um conjunto de estados iniciais é representado como árvore binária de decisão.

2 O conjunto de estados alcançáveis a partir dos estados no conjunto inicial são adicionados à representação na árvore binária de decisão. Chamemos-lhe a **representação dos estados**.

3 O conjunto de novos estados alcançáveis a partir da **representação dos estados**, é intersectado com o conjunto que representa todos os estados que validam a negação da propriedade que se procura validar. Caso a intersecção não seja o conjunto vazio $\emptyset$, um erro é detectado.

4 Os pontos 2 e 3 são repetidos até que **não haja novos estados alcançáveis**, ou **seja detectado um erro**.

Caso **não haja novos estados alcançáveis**, a propriedade é verificada pelo argumento que não existem estados alcançáveis que contradigam a propriedade. Caso **seja detectado um erro**, um contra-exemplo é gerado.

Note-se que a terminação do processo é garantida, uma vez que há apenas um número finito de estados, nos sistemas de transição de estados.

Deste a sua introdução na década de 90, **Binary Decision Diagrams** têm sido adotadas pela indústria nos processos de garantia de várias empresas produtoras de *hardware*. A principal desvantagem do método **Binary Decision Diagrams** reside, tal como o método explícito, no crescimento exponencial dos estados a serem verificados, com o número de componentes presente no sistema. Infelizmente, é um método cuja quantidade de memória disponível restringe o tamanho dos circuitos que podem ser verificados de forma eficiente. Apesar disto, sistemas finitos de estados, tais como os circuitos em série e os protocolos de comunicação, têm sido verificados formalmente com sucesso na última década atrávés de técnicas de **Symbolic Model Checking** com recursos a **Binary Decision Diagrams**.

De acordo com (art01[9]), a combinação de **Symbolic Model Checking** com **Binary Decision Diagrams** [20, 15] permite verificar eficientemente sistemas com mais de 10^{20} estados. Pela

primeira vez, um número significativo de sistemas de *hardware* no mercado, puderam ser verificados eficientemente por estes métodos, o que levou à sua crescente adopção. Empresas como a Intel e a IBM desenvolveram a sua própria ferramenta de **Model Checking**, inicialmente com projectos experimentais, e posteriormente integrando-a nos seus processos de garantia de qualidade de *hardware*.

3.2 Bounded Model Checking

O poder crescente dos conhecidos *propositional SAT solvers* veio potenciar o sucesso da técnica que passamos a descrever. O Bounded model checking (BMC) é uma das técnicas de verificação mais utilizadas na industria dos semicondutores, para verificar circuitos lógicos. Em vez de recorrer a técnicas de manipulação de **Binary Decision Diagrams**, como é o caso do sucesso verificado pelo **Symbolic Model Checking**, o **Bounded Model Checking**, recorre a ferramentas de decisão como os **SAT solvers** e **Satisfiability Modulo Theories solvers**.

A combinação de técnicas de **Model Checking**, com ferramentas de decisão (SAT-solvers) permite aos métodos conhecidos por **Bounded Model Checking**, efectuar uma exploração do espaço de estados de forma muito rápida.[art02][art01] Para algumas classes de problemas, verifica-se uma performance superior relativamente a técnicas anteriores ao **Bounded Model Checking**.

O método foi introduzido pela primeira vez em 1999 por Biere et al. art01[4], como uma técnica complementar à aplicação de **Symbolic Model Checking** com recurso a **Binary Decision Diagrams**. O método deve o seu nome **Bounded** ao facto de verficar o modelo apenas até um determinado número de estados. O projecto de hardware é modelado por um sistema de transição de estados, que representa o conjunto das transições e estados possíveis do projecto de *hardware*. Este é "desenrolado" em k iterações e conjugado com uma propriedade de forma a construir uma fórmula proposicional. A fórmula é posteriormente avaliada por meio de um **procedimento de decisão**, usualmente SAT. Se a fórmula é satisfeita, então existe uma sequência de estados que refuta a propriedade. Se não existir valoração que satisfaça a fórmula gerada, a técnica é inconclusiva um vez que poderão existir sequências de estados com mais de k estados que refutem a propriedade.

Assim, o método **Bounded Model Checking** não permite provar ou negar propriedades sobre o projecto de *hardware*, uma vez que não é completo.

Apesar disto, este método tem se demonstrado útil uma vez que diversas inconsistências têm sido idêntificados, que de outro modo passariam despercebidos. Por permitir encontrar muitos erros lógicos em sistemas com alguma complexidade, que não podem ser tratados por meio de técnicas concorrentes, o **Bounded Model Checking** tem sido acolhido pela indústria, como uma técnica complementar às técnicas de **Model Checking**, baseadas em **Binary Decision Diagrams**. Diversas publicações que efectuam esta comparação têm apoiado este ponto de vista. A motivação original que levou ao desenvolvimento da técnica de **Bounded Model Checking**, prendeu-se com uma tentativa de aproveitar o recente sucesso da satisfiabilidade **SAT** de resolver fórmulas booleanas para realizar **Model Checking**. Durante os últimos anos tem-se observado um enorme aumento no poder de raciocínio e decisão de **SAT solvers**. Os **SAT solvers** modernos conseguem lidar com centenas de milhares de variáveis, e milhões de cláusulas. **Symbolic Model Checking** com **Binary Decision Diagrams**, por outro lado, pode apenas verificar sistemas com não mais do que algumas centenas de ciclos.

Resumindo, o principal conceito associado ao **Bounded Model Checking**, como explicado anteriormente, é a consideração de um traço de execução finito que pode ser uma testemunha/contra-exemplo na verificação de uma propriedade de "**safety**". O método limita o tamanho do traço de execução, procurando aumentar progressivamente o limite, à procura de testemunhas em traços mais extensos. O processo de verificação utilizando o **Bounded Model Checking**, consiste em procurar um contra-exemplo à especificação, dentro das computações do sistema até um limite máximo de $k \in \mathbb{N}$ iterações. Este limite máximo de iterações é gradualmete aumentado até ser encontrado um contra-exemplo à especificação, a verificação exceder a sua decisão em tempo não útil, ou ser atingido o **Completeness Threshold** do sistema. O **Completeness Threshold** é um limite superior de iterações que o sistema deve verificar até garantir que as fórmulas **Bounded Model Checking** são verdadeiras. VijayKeynote[84].

Os **SAT solvers** são usualmente aplicados neste método porque é possivel reduzir, de forma eficiente, a um **Propositional Satisfiability Problem** o problema da explosão de estados do método **Bounded Model Checking**, e, deste modo, solucionar com ferramentas de decisão como **SAT solvers** e mesmo **Satisfiability Modulo Theories solvers**, pois os procedimentos **Propositional Satisfiability Problem** não sofrem do problema da explosão de estados evidênciado

pelos métodos baseados em **Binary Decision Diagrams**. Os **SAT solvers** modernos, lidam fácilmente com **Propositional Satisfiability Problem** de elevada complexidade.

Surge então um novo tipo de técnicas de **Model Checking**, o **Bounded Model Checking** com recurso a ferramentas de decisão (**SAT**) art02[2-4]. O método permite verificar propriedades **safety** (verificando se um determinado é alcançável) e **liveness** (analisando ciclos para determinar a eventualidade das propriedades).

O **Bounded Model Checking** é muito atraente para o uso industrial, devido à sua robustez, a sua superior capacidade de verificação e o facto de ser um método automático de verificação. Ferramentas de decisão como GRASP [33], SATO [39], e o algoritmo Stalmarck [35], que raramente necessitam de espaço exponencial, têm contribuido para este sucesso. As desvantagens, no entanto, devem-se ao facto do método carecer de integridade e os tipos de propriedades que podem ser verificados actualmente são muito limitadas. Alguns destes inconvenientes têm sido abordados em trabalhos mais recentes, como visto em [32, 37, 1], onde foram alcansados resultados encorajadores.

Com esta proposta, o problema de complexidade do **Model Checking** não é solucionado pelo **Bounded Model Checking**, por este também se basear num procedimento exponencial, o que limita a sua capacidade. Contudo, a aplicação empírica do método tem mostrado que o **Bounded Model Checking** resolve muitos casos que não podem ser resolvidos por técnicas baseadas em **Binary Decision Diagrams**, e o inverso também se verifica. Há problemas que são solucionados de forma mais efficiente por técnicas baseadas em **Binary Decision Diagrams**. Outra desvantagem reflete-se no facto do **Bounded Model Checking** ser incapaz de provar a ausência de erros. Por este motivo as técnicas **Bounded Model Checking** não substituem nenhuma ferramenta de verificação automática, juntando-se ao arsenal de ferramentas utilizadas pela indústria.

O **Bounded Model Checking** visa resolver os mesmos problemas que as técnicas de **Symbolic Model Checking** tradicional baseado em **Binary Decision Diagrams**. O **Bounded Model Checking** possui duas caracteristicas: 1) é necessário fornecer um limite ao número de estados a serem explorados. Se o limite não for suficientemente alto, o método é incompleto. 2) utiliza técnicas **Propositional Satisfiability Problem** em vez de **Binary Decision Diagrams**.

Empiricamente tem se verificado que se o limite for suficientemente pequeno (dependendo do modelo e do **SAT solvers**), o método **Bounded Model Checking** supera as técnicas baseadas em **Binary Decision Diagrams**. Resultados empiricos também mostraram que há pouca correlação entre os problemas que são difíceis (`hard`) para **Propositional Satisfiability Problem** e os problemas que são difíceis (`hard`) para as técnicas baseadas em **Binary Decision Diagrams**. Verifica-se que problemas da classe `hard` para **Binary Decision Diagrams**, muitas vezes são decidiveis por **SAT**.

Por art01[27], acreditamos que caso se configure os **SAT solvers** para tirar partido da estrutura original das fórmulas resultantes do process **Bounded Model Checking**, o método melhora significativamente[27].

Uma pesquisa publicada pela Intel art01[14] mostraram que o **Bounded Model Checking** tem vantagens tanto na capacidade de verificação como na produtividade dos resultados relativamente a ferramentas **Symbolic Model Checking** baseadas em **Binary Decision Diagrams**, quando aplicada à modelação de sistemas dos conhecidos $Pentium - 4^{TM}$. As vantagens de produtividade resultam de que, normalmente, as técnicas baseadas em **Binary Decision Diagrams**, para optimizar o seu desempenho, requerem algum nível de orientação manual.

De acordo com art01[18], uma abordagem semelhante foi tomada para resolver os clássicos `planning problem` na área da Inteligência Artificial [18]. Os clássicos `planning problem` procuram encontrar uma sequêcia de passos a fim de realizar uma tarefa ("por exemplo, posição cubos um sobre o outro em tamanho decrescente sob certas restrições sobre os estados intermediários"). Tal como no **Bounded Model Checking**, encontrar um plano é equivalente a encontrar caminhos até um determinado limite máximo de passos. Os planos possiveis são expressos numa instância de **Propositional Satisfiability Problem**, sendo decididos em tempo polinomial. Comparado com o **Model Checking**, os `planning problems` determinísticos preocupam-se apenas com propriedades de *safety* simples: como alcançar um determinado estado.

BMC aplicado na verificação de software

Os sistemas de computação consistem em complexos aglomerados de hardware e software. Assegurar a correcção do software tem se verificado um desafio mais complexo do que garantir a correcção do hardware aplicado.[VijayKeynote].

Existem numerosas ferramentas para procurar falhas funcionais nas arquitecturas dos waffers de silicone, no entanto poucas são as ferramentas que asseguram qualidade no software.

O custo de garantir a ausência de erros no software é tão elevada que as ferramentas desenvolvidas raramente garantem a correcção total do software.

Os algoritmos de **Model Checking** examinam de forma exaustiva todos os estados alcansáveis pelo modelo de software.

Com a evolução do crescente poder de computação, o tamanho e a complexidade dos programas de software tem crescido ao ponto de se tornar difícil de assegurar a correcção e a qualidade dos produtos de software. Especialmente nas áreas da comunicação e transportes, a correcção do software toma uma importância agravada. É comum encontrar-se erros de software que resultam em enormes perdas financeiras e até mesmo perdas de vidas.

A ideia subjacente ao BMC, técnica aplicada à verificação de hardware, pode também ser aplicada a sistemas de software. Segundo **??** (pág. **??**) a forma mais linear de realizar esta transição resume-se em considerar todo o programa como uma relação de transição.

O sistema de software ou mesmo o programa de software, neste caso, também é visto como um sistemas de transição de estados a ser analisado.

Um sistema de transição é o modelo do software que procuramos verificar e consiste de uma relação de transição , um conkunto de estados iniciais e um conjunto de estados que codificam a localização de comandos e variáveis do programa bem como os seus valores ao logo da sua execução.

Neste processo[VijayKeynote], cada instrução mais básica é traduzida numa fórmula na forma Single assignment VijayKeynote[2], os operadores aritméticos são convertidos em cirquitos lógicos equivalentes, arrys e apontadores são considerados como memórias em hardware através do construtor “case split” com todos os possiveis valores de memória.

Desenrolar o sistemas de transição de estados com k passos permite explorar todos os caminhos possiveis dentor do sistemas de transição de estados com comprimento k ou menos. O custo de desenrolar o sistemas de transição de estados em k passos, resulta no tamanho k vezes o tamanho do programa. Para grandes programas este cálculo torna-se excessivo pelo que têm surgido propostas de optimização.

Desta relaçao de transição é gerda uma fórmula que codifica o sistemas de transição de estados e que é possivel avaliar por meio de um **procedimento de decisão** baseado em teorias lógicas, como são exemplo, arrays, bit-vectors, lógicas de aritmética de inteiros, lógicas de aritmética de reais.

Mais uma vez, caso a fórmula seja validada pelo **procedimento de decisão** é imediato a extracção da valoração que satisfaz a fórmula. Esta valoração fornece o contra exemplo lógico pelo qual o programa não satisfaz a especificação. Caso a fórmula não seja satisfazivel, não existindo valoração que a torne verdadeira, podemos concluir que não há estado que não satisfaça a especificação, em pelo menos k passos.

Ao processo também falha **Completude** uma vez que procura violações nas propriedades até um dado limite de execução, não sendo possível garantir a falha de erros por completo. Erros posteriores ao limite dado não são idêntificados, contudo permite provar propridades de safety e liveness até um determinado ponto.

nothing

Existe no entanto um ponto, tal que se garantirmos certas propriedades até esse ponto, podemos garantir que os pontos posteriores serão estados já verificados. O ponto que garante esta propriedade é chamado de completeness threshold VijayKeynote[84]

nothing

Contudo, aproximações ao completeness threshold são suficientes uma vez que encontrar este ponto concretamente é tão difícil quanto realizar o próprio **Model Checking**.

Uma forma de obter esta aproximação, em termos de verficação de software, resume-se a determinar o **high-level worst-case execution time (WCET)**[VijayKeynote,chp16]

O WCET é o limite ao número máximo de iterações de ciclo. Existem ferramentas que calculam o WCET através de análise sintática às estruturas ciclicas. Caso o resultado destas ferramentas não seja adequado, é possivel aplicar um processo iterativo onde o limite e constantemente aumentado chp16[KCY03, CKL04].

nothing

nothing

O método de encontrar WCET apenas é aplicável a programas que possuem um limite de run-time, que se verifica em aplicações embebidas, onde há uma maior necessidade de verificação do software.

Ferramentas de Verificação de Software com BMC

Existem algumas ferramentas de verificação de software baseadas em **Bounded Model Checking**. A primeira implementação data de 2000 onde Currie et al. chp16[CHR00] implementaram uma procura simbólica em profundidade e limitada.

Outra das primeira implementações do método BMC é apresentado pela ferramenta cmbc desenvolvida na CMu. Procura simular uma vasta gama de arquitecturas para testar os programas de software a ser verificados. Providencia suporte para memórias little e big-Endian bem como suporte a header files para linux, windows e Mac-Os x. A ferramenta além de implementar técnicas de desenrolar ciclos, também implementa bit-flattening, ou bit-blasting para decidir fórmulas de bit-vector.

As formulas geradas por esta ferramenta são traduziveis para muitos dos standards conhecidos, sendo a única ferramenta que suporta SystemC, SpecV e C++.

A IBM desenvolveu uma versão do Cbc para verificar programas concorrentes.

Desenvolvido pela NEC Research, o F-Soft chp16[ISGG05] é a única ferramenta que desenrola por completo todo o sistemas de transição de estados subjacente ao programa. Esta ferramenta fornece um SAT solver próprio que foi optimizado aos problemas de decisão gerados pelo método **Bounded Model Checking**.

Têm surgido várias versões destas ferramentas, como a conhecida versão do Cbmc de Armando et al. que gera um problema de desição para SMt's em aritmética de inteiros linear. A aplicação de constraint solvers nos problemas de decisão gerados por BMC também é estudado por VijayKeynote[40]

nothing

A ferramenta Saturn também é digna de se notar pela sua escalabilidade VijayKeynote[108]. Foi aplicada ao kernell do LINUX para detectar NULL-pointer dereferences e conve ções de locks na API, demonstrando assim que a técnica aplicada no Saturn é escalável o suficiente para analizar todo o kernel do sistema operativo linux.

A ferramenta EXE VijayKeynote[109] aplica simulação simbolica para detectar erros no código de sistemas de ficheiros, usando um modelo de memória de baixo nível

O BMC é uma técnica excepcional para detectar erros de software. Outras técnicas serão mais apropriadas para provar propriedades sobre software.

3.3 Transformações BMC de programas $\mathcal{W}hile_{\mathbf{assert}}$

3.3.1 Transformação 1 - $\mathcal{T}_1(\ c,\ k\ )$

A transformação $\mathcal{T}_1$ procura transformar um qualquer comando que contenha ciclos num comando equivalente onde os ciclos são desenrolados até um limite k. Esta transformação reduz o espaço de comandos as ser analisado, procuranto substituir todas as ocorrências do comando **while**(b)**do** c_w por uma construção equivalente à custa de outros elementos presentes na classe $\mathcal{C}$**ommand**. Um programa sem ciclos, não sofre qualquer tipo de alteração, como é o caso do programa expresso na Figura 3.1 (pág. 48) .

```
if(x > 3)then{
    assert(y > 0)
}else{
    x = 4
}
```

Figura 3.1: Instância de um programa sem ciclos

Tome-se o programa representado na Figura 3.2a (pág. 49) , existem difersas formas de implementar esta transformção. Pode-se desenrolar o ciclo mais profundo k vezes (Figura 3.2d (pág. 49)), pode-se desenrolar o ciclo menos profundo k vezes (Figura 3.2c (pág. 49)), ou todas as ocorrências (Figura 3.2b (pág. 49)).

Esta é a transformação que melhor expressa o problema da explosão dos estados para este tipo de técnicas.

Definição 3.3.1. *Para quaisquer* $k \in \mathbb{N}$ *e* $c \in \mathcal{C}$**ommand**,
A transformação $\mathcal{T}_1$ *é definida recursivamente do seguinte modo:*

$$\begin{array}{rcl} \mathcal{T}_1(\ \textbf{\textit{skip}},\ k\ ) & \coloneqq & \textbf{\textit{skip}} \\ \mathcal{T}_1(\ x \coloneqq e,\ k\ ) & \coloneqq & x \coloneqq e \\ \mathcal{T}_1(\ \textbf{\textit{assert}}(\ b\ ),\ k\ ) & \coloneqq & \textbf{\textit{assert}}(\ b\ ) \\ \mathcal{T}_1(\ c_i\ ;\ c_j,\ k\ ) & \coloneqq & \mathcal{T}_1(\ c_i,\ k\ )\ ;\ \mathcal{T}_1(\ c_j,\ k\ ) \\ \mathcal{T}_1(\ \textbf{\textit{if}}(\ b\ )\textbf{\textit{then}}\ c_t\ \textbf{\textit{else}}\ c_f,\ k\ ) & \coloneqq & \textbf{\textit{if}}(\ b\ )\textbf{\textit{then}}\ \mathcal{T}_1(\ c_t,\ k\ )\ \textbf{\textit{else}}\ \mathcal{T}_1(\ c_f,\ k\ ) \\ \mathcal{T}_1(\ \textbf{\textit{while}}(\ b\ )\textbf{\textit{do}}\ c_w,\ k\ ) & \coloneqq & \textbf{\textit{construtor}}(\ k,\ \textbf{\textit{while}}(\ b\ )\textbf{\textit{do}}\ c_w,\ k\ ) \end{array} \tag{3.1}$$

Definição 3.3.2. *Para quaisquer* $i, k \in \mathbb{N}$ *e* $c \in \mathcal{C}$**ommand**,
A função ***construtor*** *é definida recursivamente do seguinte modo:*

$$\begin{array}{rcl} \textbf{\textit{construtor}} & : & \mathbb{N} \times \mathcal{C}\textbf{ommand} \times \mathbb{N} \to \mathcal{C}\textbf{ommand} \\ \textbf{\textit{construtor}}(\ 0,\ \textbf{\textit{while}}(\ b\ )\textbf{\textit{do}}\ c_w,\ k\ ) & \coloneqq & \textbf{\textit{assert}}(\ \neg\ b\ ) \\ \textbf{\textit{construtor}}(\ i,\ \textbf{\textit{while}}(\ b\ )\textbf{\textit{do}}\ c_w,\ k\ ) & \coloneqq & \textbf{\textit{if}}(\ b\ )\ \textbf{\textit{then}} \\ & & \quad \mathcal{T}_1(\ c_w,\ k\ )\ \textbf{\textit{;}} \\ & & \quad \textbf{\textit{construtor}}(\ i-1,\ \textbf{\textit{while}}(\ b\ )\textbf{\textit{do}}\ c_w,\ k\ ) \end{array} \tag{3.2}$$

Esta definição expande todos os ciclos (internos e externos) $k+1$ iterações.

3.3.2 Transformação 2 - $\mathcal{T}_2(\ c\ )$

A transformação $\mathcal{T}_2$ procura transformar um qualquer comando e atribuir-lhe propriedades **Single Assignment** na forma **Expressões Conditional Normal Form** . Esta transfomação atribuí novas versões das mesmas variáveis, pelo que, normaliza o comando com atribuições que são únicas.

Definição 3.3.3. *Para qualquer* $c \in \mathcal{C}$**ommand**,
Representa-se c respeita as propriedades **Single Assignment** *com* sa(c).

Neste projecto optamos pelo **Dynamic Single Assignment** em vez do **Static Single Assignment**. A diferença reside na forma de unificar as variáveis após a ocorrência de um condicional.

O **Dynamic Single Assignment** unifica as variáveis dentro de cada ramo, gerando uma sequência de atribuições para uma noava versão da variável, de nomenclatura superior a qualquer um dos ramos do condicional. O **Static Single Assignment** unifica as variáveis exactamente a sequir à ocorrência do condicional.

```
while(x > 3)do{
    while(y < 5)do{
        x = x − 1;
        y = y + 1
    }
}
```

(a) Programa com ciclos aninhados

```
if(x > 3)then{
    if(y < 5)then{
        x = x − 1;
        y = y + 1;
        if(y < 5)then{
            x = x − 1;
            y = y + 1;
            if(y < 5)then{
                assert(⊥) // end(y) for k(x) = 2
            }else{skip} // k(y) = 3
        }else{skip} // k(y) = 2
    }else{skip}; // k(y) = 1
    if(x > 3)then{
        if(y < 5)then{
            x = x − 1;
            y = y + 1;
            if(y < 5)then{
                x = x − 1;
                y = y + 1;
                if(y < 5)then{
                    assert(⊥) // end(y) for k(x) = 2
                }else{skip} // k(y) = 3
            }else{skip} // k(y) = 2
        }else{skip}; // k(y) = 1
        if(x > 3)then{
            assert(⊥) // end(x)
        }else{skip} // k(x) = 3
    }else{skip} // k(x) = 2
}else{skip} // k(x) = 1
```

(b) Expansão de todas as ocorrências de ciclos

```
while(x > 3)do{
    if(y < 5)then{
        x = x − 1;
        y = y + 1;
        if(y < 5)then{
            x = x − 1;
            y = y + 1;
            if(y < 5)then{
                assert(⊥) // end(y)
            }else{skip} // k(y) = 3
        }else{skip} // k(y) = 2
    }else{skip}; // k(y) = 1
}
```

(c) Expansão dos ciclos mais internos

```
if(x > 3)then{
    while(y < 5)do{
        x = x − 1
        y = y + 1
    };
    if(x > 3)then{
        while(y < 5)do{
            x = x − 1
            y = y + 1
        };
        if(x > 3)then{
            assert(⊥) // end(x)
        }else{skip} // k(x) = 3
    }else{skip} // k(x) = 2
}else{skip} // k(x) = 1
```

(d) Expansão dos ciclos mais externos

Assim é adicionado uma seqência de comandos adicional entre o if e o consequênte. Esta sequência de condicionais renomeia todas as variáveis alteradas para os valores calculados quer num ou noutro ramo, dependendo da condição inicial.

Note-se que é necessário um cuidado quanto ao algoritmo de atribuição das versões uma vez que a versão não pode ocorrer em ambos os contextos. Após a execução dos camos, os contextos têm de ser distintos.

$$(\textbf{if}(b)\textbf{then } c_t \textbf{ else } c_f) \text{ ; comand}$$
$$\downarrow$$
$$(\textbf{if}(b)\textbf{then } c_t \textbf{ else } c_f) \text{ ; } \textit{sequência de normalização}\text{; } \textit{normaliza}(\text{ comand })$$

Ambas requerem o conhecimento da evolução das versões em cada ramo da computação. A garantia é dada porque a variável inicial em b nunca foi alterada e as invocações posteriores são sempre a chamadas normalizadas.

Note-se a ocorrência de condicionais aninhados e a necessidade de manipular as variáveis mais internas após o aninhamento.

Definição 3.3.4. *Para quaisquer* $c \in \mathcal{C}\textbf{ommand}\backslash\{\ \textbf{\textit{while}}(b)\textbf{\textit{do}}\, c_w \ \}$,
A transformação $\mathcal{T}_2$ *é definida pela aplicação da função* ***aux*** *a todas as variáveis do programa.*

Definição 3.3.5. *Para quaisquer* $c \in \mathcal{C}\textbf{ommand}\backslash\{\ \textbf{\textit{while}}(b)\textbf{\textit{do}}\, c_w \ \}$,
A função ***aux*** *é definida recursivamente do seguinte modo:*

$$\begin{array}{rcll}
\textbf{\textit{aux}}(v,\ \textbf{\textit{skip}}) & = & (v,\ \textbf{\textit{skip}}) & \\
\\
\textbf{\textit{aux}}(v,\ \textbf{\textit{assert}}(b)) & = & (v,\ \textbf{\textit{assert}}(\textbf{\textit{subs}}(b,\ v))) & \\
\\
\textbf{\textit{aux}}(v,\ x := e) & = & (v,\ x := \textbf{\textit{subs}}(e,\ v)) & ,\textit{se } x \notin [v] \\
\textbf{\textit{aux}}(v,\ x := e) & = & (v',\ v' := \textbf{\textit{subs}}(e,\ v)) & ,\textit{se } x \in [v] \\
\\
\textbf{\textit{aux}}(v_0,\ c_i \textbf{ ; } c_j) & = & (v_2,\ c_i^\flat \textbf{ ; } c_j^\flat) & \textit{onde}(v_1,\ c_i^\flat) = \textbf{\textit{aux}}(v_0,\ c_i) \\
 & & & \textit{onde}(v_2,\ c_j^\flat) = \textbf{\textit{aux}}(v_1,\ c_j)
\end{array}$$

onde

Definição 3.3.6. *Define-se* ***subs***$(e,\ v)$ *como a expressão onde ocorre a substituição de todas as ocorências das variáveis que são* **versões** *de* v*, por* v *em* e*.*

$$\begin{array}{rcll}
\textbf{\textit{subs}}((e_1 \square e_2,\ ,\ ) v) & = & \textbf{\textit{subs}}(e_1,\ v) \square \textbf{\textit{subs}}(e_2,\ v) & \\
\textbf{\textit{subs}}(n,\ v) & = & n & \\
\textbf{\textit{subs}}(x,\ v) & = & x & ,\textit{onde } x \notin [v] \\
\textbf{\textit{subs}}(x,\ v) & = & v & ,\textit{onde } x \in [v]
\end{array}$$

3.3.3 Transformação 3 - $\mathcal{T}_3(c,\ b')$

A transformação $\mathcal{T}_3$ procura transformar um comando com propriedades **Single Assignment** na forma **Expressões Conditional Normal Form** . Esta transfomação normaliza o comando, eliminando os condicionais representados pelo comando **if**(b)**then** c_t **else** c_f. é então criado uma comando fórmula lógica que acomula a profundidade das atribuições e asserções, e as escolhas através do seguinte mecanismo:

Definição 3.3.7. *Para quaisquer* $b \in \mathcal{E}xpression_{Boolean}$ *e* $c \in \mathcal{C}\textbf{ommand}$: sa(c),
A transformação $\mathcal{T}_3$ *é definida recursivamente do seguinte modo:*

$$\begin{array}{rcl}
\mathcal{T}_3 & : & \mathcal{C}\textbf{ommand} \rightarrow \mathcal{C}nf \\
\mathcal{T}_3(\textbf{\textit{skip}},\ b') & = & \textbf{\textit{if}}(b')\ \textbf{\textit{then skip}} \\
\mathcal{T}_3(\textbf{\textit{assert}}(b),\ b') & = & \textbf{\textit{if}}(b')\ \textbf{\textit{then assert}}(b) \\
\mathcal{T}_3(x := e,\ b') & = & \textbf{\textit{if}}(b')\ \textbf{\textit{then}}\, x := e \\
\mathcal{T}_3(c_i \textbf{ ; } c_j,\ b') & = & \mathcal{T}_3(c_i,\ b') \textbf{ ; } \mathcal{T}_3(c_j,\ b') \\
\mathcal{T}_3(\textbf{\textit{if}}(b)\textbf{\textit{then}}\, c_t \textbf{\textit{ else}}\, c_f,\ b') & = & \mathcal{T}_3(c_t,\ b' \wedge b) \textbf{ ; } \mathcal{T}_3(c_f,\ b' \wedge \neg\, b)
\end{array}$$

```
if(x ≡ x)then{
    w:=w;
    if(i ≡ i)then{
        w:=w;
        if(a ≡ a)then{
            w:=w
        }else{
            w:=w
        }
    }else{
        w:=w;
        if(b ≡ b)then{
            w:=w
        }else{
            w:=w
        }
    };
}else{
    w:=w;
    if(j ≡ j)then{
        w:=w;
        if(c ≡ c)then{
            w:=w
        }else{
            w:=w
        }
    }else{
        w:=w;
        if(d ≡ d)then{
            w:=w
        }else{
            w:=w
        };
    };
};
assert(y > 0)
```

Figura 3.2: Instância de um programa sem ciclos

```
if(a > 0)then{
    if(t > 0)then{
        if(i > 0)then{
            assert(p == q)
        }else{
            assert(q == p)
        }
    }else{
        if(j > 0)then{
            assert(r == s)
        }else{
            assert(s == r)
        }
    }
}else{
    if(f > 0)then{
            assert(x == y)
    }else{
            assert(y == x)
    }
}
```

Figura 3.3: Instância de um programa com ciclos aninhados

```
if(    (a > 0) and     (t > 0) and     (i > 0) )then{assert(p == q)}
if(    (a > 0) and     (t > 0) and not(i > 0) )then{assert(q == p)}
if(    (a > 0) and not(t > 0) and     (j > 0) )then{assert(r == s)}
if(    (a > 0) and not(t > 0) and not(j > 0) )then{assert(s == r)}
if(not(a > 0) and     (f > 0)                )then{assert(x == y)}
if(not(a > 0) and not(f > 0)                )then{assert(y == x)}
```

Figura 3.4: Resultado de uma transformação

Definição 3.3.8. *Para quaisquer* $b \in \mathcal{E}xpression_{Boolean}$ *e* $c \in \mathcal{C}\mathbf{ommand}:$ sa(c), *A transformação* $\mathcal{T}_3$ *é definida recursivamente do seguinte modo:*

$$
\begin{array}{rcl}
\mathcal{T}_3 & : & \mathcal{C}\mathbf{ommand} \rightarrow \mathcal{C}nf \\
\mathcal{T}_3(\ \textbf{\textit{skip}},\ b'\ ) & = & \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then skip}} \\
\mathcal{T}_3(\ \textbf{\textit{assert}}(\ b\ ),\ b'\ ) & = & \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then assert}}(\ b\ ) \\
\mathcal{T}_3(\ x\ :=\ e,\ b'\ ) & = & \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then}}\ x\ :=\ e \\
\mathcal{T}_3(\ c_i\ ;\ c_j,\ b'\ ) & = & \mathcal{T}_3(\ c_i,\ b'\ )\ ;\ \mathcal{T}_3(\ c_j,\ b'\ ) \\
\mathcal{T}_3(\ \textbf{\textit{if}}(\ b\ )\textbf{\textit{then}}\ c_t\ \textbf{\textit{else}}\ c_f,\ b'\ ) & = & \mathcal{T}_3(\ c_t,\ b' \wedge\ b\ )\ ;\ \mathcal{T}_3(\ c_f,\ b' \wedge \neg\ b\ )
\end{array}
$$

3.3.4 Transformação 4 - $\mathcal{T}_4(\ c\ )$

A transformação $\mathcal{T}_4$ procura transformar um comando na forma **Expressões Conditional Normal Form** , com propriedades **Single Assignment**, numa representação lógica. Procurando exprimir que as afirmações presentes do programa são consequência das atribuições realizadas no comando.

Definição 3.3.9. *Para quaisquer* $b \in \mathcal{E}xpression_{Boolean}$ *e* $c \in \mathcal{C}\mathbf{ommand}:$ sa(c), *A transformação* $\mathcal{T}_3$ *é definida recursivamente do seguinte modo:*

$$
\begin{array}{rcl}
\mathcal{T}_4 & : & \mathcal{C}nf \rightarrow \mathcal{E}xpression_{Boolean} \\
\mathcal{T}_4(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then skip}}\ ) & = & (\ \emptyset,\ \emptyset\ ) \\
\mathcal{T}_4(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then assert}}(\ b\ )\ ) & = & (\ \emptyset,\ \{\ b' \rightarrow\ b\ \}\ ) \\
\mathcal{T}_4(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then}}\ x\ :=\ e\ ) & = & (\ \{\ b' \rightarrow (\ x = e\ )\ \},\ \emptyset\ ) \\
\mathcal{T}_4(\ c_i\ ;\ c_j\ ) & = & (\ \mathcal{C}_{c_i} \cup \mathcal{C}_{c_j},\ \mathcal{P}_{c_i} \cup \mathcal{P}_{c_j}\ )
\end{array}
\qquad
\begin{array}{l}
onde\ \ (\ \mathcal{C}_{c_i},\ \mathcal{P}_{c_i}\ ) = \mathcal{T}_4(\ c_i\ ) \\
\qquad\quad\ (\ \mathcal{C}_{c_j},\ \mathcal{P}_{c_j}\ ) = \mathcal{T}_4(\ c_j\ )
\end{array}
$$

Capítulo 4

Correcção e Completude de transformações BMC para programas SA

4.1 Transformação 3 - $\mathcal{T}_3$

Assume-se que na sequência de transformações BMC, a transformação $\mathcal{T}_1$ substituiu todas as ocorrências do comando **while**(b)**do** c_w e a transformação $\mathcal{T}_2$ adicionou propriedades **Single Assignment** ao comando. Queremos provar a correcção de $\mathcal{T}_3$.

Teorema 4.1.1. *Para qualquer* $c \in \mathcal{C}$**ommand** *e* $b' \in \mathcal{E}xpression_{Boolean}$

$$\text{safe}(\ c\ ) \ sse\ \text{safe}(\ \mathcal{T}_3(\ c,\ b'\ )\ ) \tag{4.1}$$

Demonstração. Queremos mostrar que para qualquer $c \in \mathcal{C}$**ommand** e $b' \in \mathcal{E}xpression_{Boolean}$

$$safe(\ c\ )\ sse\ safe(\ \mathcal{T}_3(\ c,\ b'\ )\ ) \tag{4.2}$$

Pelo lema 2.3.4 (pág. 34) , basta mostrar que

$$c\ \approx\ \mathcal{T}_3(\ c,\ b'\ ) \tag{4.3}$$

Pelo lema 4.1.3 (pág. 58) . □

Teorema 4.1.2. *Para qualquer* $c \in \mathcal{C}\textbf{ommand}$ *e* $b' \in \mathcal{E}xpression_{Boolean}$

$$\forall \psi \in \mathcal{E}xpression_{Boolean},\ \text{safe}(\ s,\ c\ ) \Rightarrow \text{safe}(\ \mathcal{T}_3(\ s,\ b'\ ),\ c\ ) \tag{4.4}$$

Demonstração. Por indução estrutural no comando c.

Caso $c = \textbf{skip}$,

Queremos mostrar que

$$\forall \psi \in \mathcal{E}xpression_{Boolean},\ safe(\ \textbf{skip},\ s\ ) \Rightarrow safe(\ \mathcal{T}_3(\ \textbf{skip},\ b'\ ),\ s\ )$$

Pela definição da transformação, queremos mostrar que $safe(\ \textbf{if}(\ b'\ )\ \textbf{then skip},\ s\ )$.
Caso $s \not\models b'$: pela definição da relação $\rightsquigarrow$, $(\ \textbf{skip},\ s\ ) \rightsquigarrow s$.
Caso $s \models b'$: pela definição da relação $\rightsquigarrow$, $(\ \textbf{skip},\ s\ ) \rightsquigarrow s$.

Caso $c = \textbf{assert}(\ b\ )$,
Para qualquer $b \in \mathcal{E}xpression_{Boolean}$,

Queremos mostrar que

$$\forall \psi \in \mathcal{E}xpression_{Boolean},\ safe(\ \textbf{assert}(\ b\ ),\ s\ ) \Rightarrow safe(\ \mathcal{T}_3(\ \textbf{assert}(\ b\ ),\ b'\ ),\ s\ )$$

Pela definição da transformação, queremos mostrar que $safe(\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ ),\ s\ )$.
Caso $s \not\models b'$: pela definição da relação $\rightsquigarrow$, $(\ \textbf{skip},\ s\ ) \rightsquigarrow s$.
Caso $s \models b'$: pela definição da relação $\rightsquigarrow$, $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s'$.
Como, por suposição, $safe(\ \textbf{assert}(\ b\ ),\ s\ )$, $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s$.

Caso $c = x\ :=\ e$,
Para quaisquer $x \in \mathcal{V}_{Integer}$ e $e \in \mathcal{E}xpression_{Integer}$,

Queremos mostrar que

$$\forall \psi \in \mathcal{E}xpression_{Boolean},\ safe(\ x\ :=\ e,\ s\ ) \Rightarrow safe(\ \mathcal{T}_3(\ x\ :=\ e,\ b'\ ),\ s\ )$$

Pela definição da transformação, queremos mostrar que $safe(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ )$.
Caso $s \not\models b'$: pela definição da relação $\rightsquigarrow$, $(\ \textbf{skip},\ s\ ) \rightsquigarrow s$.
Caso $s \models b'$: pela definição da relação $\rightsquigarrow$, $(\ x\ :=\ e,\ s\ ) \rightsquigarrow s \begin{pmatrix} x \\ e \end{pmatrix}$.

Caso $c = c_i \textbf{ ; } c_j$,
Para quaisquer $c_i,\ c_j \in \mathcal{C}\textbf{ommand}$,

Por hipótese de indução temos,

$$HI_i: \quad \forall \psi \in \mathcal{E}xpression_{Boolean},\ safe(\ c_i,\ s\ ) \Rightarrow safe(\ \mathcal{T}_3(\ c_i,\ b'\ ),\ s\ )$$

$$HI_j: \quad \forall \psi \in \mathcal{E}xpression_{Boolean},\ safe(\ c_j,\ s\ ) \Rightarrow safe(\ \mathcal{T}_3(\ c_j,\ b'\ ),\ s\ )$$

Queremos mostrar que

$$\forall \psi \in \mathcal{E}xpression_{Boolean},\ safe(\ c_i \textbf{ ; } c_j,\ s\ ) \Rightarrow safe(\ \mathcal{T}_3(\ c_i \textbf{ ; } c_j,\ b'\ ),\ s\ )$$

Pela definição da transformação, queremos mostrar que $safe(\ \mathcal{T}_3(\ c_i,\ b'\ )\textbf{ ; }\mathcal{T}_3(\ c_j,\ b'\ ),\ s\ )$.
Pela definição da relação $\rightsquigarrow$, $\exists\ s_e \in \Sigma,\ (\ \mathcal{T}_3(\ c_i,\ b'\ ),\ s\ ) \rightsquigarrow s_e \wedge (\ \mathcal{T}_3(\ c_j,\ b'\ ),\ s_e\ ) \rightsquigarrow s'$.
Pelas hipóteses.

Caso $c = \textbf{if}(\ b\ )\textbf{then}\ c_t\ \textbf{else}\ c_f$,
Para quaisquer $b \in \mathcal{E}xpression_{Boolean},\ c_t,\ c_f \in \mathcal{C}\textbf{ommand}$,

Por hipótese de indução temos,

$$HI_t: \quad \forall \psi \in \mathcal{E}xpression_{Boolean},\ safe(\ c_t,\ s\ ) \Rightarrow safe(\ \mathcal{T}_3(\ c_t,\ b'\ ),\ s\ )$$

$$HI_f: \quad \forall \psi \in \mathcal{E}xpression_{Boolean},\ safe(\ c_f,\ s\ ) \Rightarrow safe(\ \mathcal{T}_3(\ c_f,\ b'\ ),\ s\ )$$

Queremos mostrar que

$$\forall \psi \in \mathcal{E}xpression_{Boolean},\ safe(\ \textbf{if}(\ b\ )\textbf{then}\ c_t\ \textbf{else}\ c_f,\ s\ ) \Rightarrow safe(\ \mathcal{T}_3(\ \textbf{if}(\ b\ )\textbf{then}\ c_t\ \textbf{else}\ c_f,\ b'\ ),\ s\ )$$

Pela definição da transformação, queremos mostrar que
$safe(\ \mathcal{T}_3(\ c_t,\ b' \wedge\ b\ )\textbf{ ; }\mathcal{T}_3(\ c_f,\ b' \wedge \neg\ b\ ),\ s\ )$.
i.e.,
$(\ \mathcal{T}_3(\ c_t,\ b' \wedge\ b\ )\textbf{ ; }\mathcal{T}_3(\ c_f,\ b' \wedge \neg\ b\ ),\ s\ ) \not\rightsquigarrow erro$

Pela SO, temos de mostrar que, $\exists\ s'' \in \Sigma,\ (\ \mathcal{T}_3(\ c_t,\ b' \wedge\ b\ ),\ s\ ) \rightsquigarrow s'' \wedge (\ \mathcal{T}_3(\ c_t,\ b' \wedge \neg\ b\ ),\ s''\ ) \rightsquigarrow s'$ e $s' \neq erro$
Independentemente da avaliação de $s \models b'$ e $s \models b$, sabemos que $s \models b' \neq s \models b'$, i.e., $[\![\ b'\ ]\!]_s = \neg [\![\ b'\ ]\!]_s$

Caso $s \not\models b'$: Pelo lema X, $safe(\ \mathcal{T}_3(\ c_t,\ b' \wedge\ b\ ),\ s\ )$ e $safe(\ \mathcal{T}_3(\ c_f,\ b' \wedge \neg\ b\ ),\ s\ )$
e por consequente $safe(\ \mathcal{T}_3(\ c_t,\ b' \wedge\ b\ )\textbf{ ; }\mathcal{T}_3(\ c_f,\ b' \wedge \neg\ b\ ),\ s\ )$.

Caso $c = \textbf{while}(\ b\ )\textbf{do}\ c_w$,
Para quaisquer $b \in \mathcal{E}xpression_{Boolean},\ c_w \in \mathcal{C}\textbf{ommand}$,

Por hipótese de indução temos,

$$HI_w: \quad \forall \psi \in \mathcal{E}xpression_{Boolean},\ safe(\ c_w,\ s\ ) \Rightarrow safe(\ \mathcal{T}_3(\ c_w,\ b'\ ),\ s\ )$$

Queremos mostrar que

$$\forall \psi \in \mathcal{E}xpression_{Boolean},\ safe(\ \textbf{while}(\ b\ )\textbf{do}\ c_w,\ s\ ) \Rightarrow safe(\ \mathcal{T}_3(\ \textbf{while}(\ b\ )\textbf{do}\ c_w,\ b'\ ),\ s\ )$$

□

Lema 4.1.3. *Para qualquer* $c \in \mathcal{C}\textbf{ommand}$ *e* $b' \in \mathcal{E}xpression_{Boolean}$

$$c \approx \mathcal{T}_3(\ c,\ b'\ ) \tag{4.5}$$

Demonstração.
Queremos mostrar que para qualquer $c \in \mathcal{C}\textbf{ommand}$, $b' \in \mathcal{E}xpression_{Boolean}$ e $s, s' \in \Sigma$
$c \approx \mathcal{T}_3(\ c,\ b'\ )$, i.e., $(\ c,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \mathcal{T}_3(\ c,\ b'\ ),\ s\ ) \rightsquigarrow s'$
Por indução estrutural na classe $\mathcal{C}\textbf{ommand}$

caso **skip**:
Queremos mostrar que $(\ \textbf{skip},\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \mathcal{T}_3(\ \textbf{skip},\ b'\ ),\ s\ ) \rightsquigarrow s'$
Pela Definição 3.3.8 (pág. 53) , basta mostrar: $(\ \textbf{skip},\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \textbf{if}(\ b'\ )\ \textbf{then skip},\ s\ ) \rightsquigarrow s'$

Caso $s \not\models b'$: basta mostrar: $(\ \textbf{skip},\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \textbf{skip},\ s\ ) \rightsquigarrow s'$
Caso $s \models b'$: basta mostrar: $(\ \textbf{skip},\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \textbf{skip},\ s\ ) \rightsquigarrow s'$
Ambos, pela definição da relação de transição.
caso **assert**$(\ b\ )$:
Queremos mostrar que $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \mathcal{T}_3(\ \textbf{assert}(\ b\ ),\ b'\ ),\ s\ ) \rightsquigarrow s'$
Pela Definição 3.3.8 (pág. 53) , basta mostrar: $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ ),\ s\ ) \rightsquigarrow s'$

Caso $s \models b'$: basta mostrar: $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s'$
pela definição da relação de transição.
Caso $s \not\models b'$: basta mostrar: $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \textbf{skip},\ s\ ) \rightsquigarrow s'$
Caso $s \models b$: $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s \Leftrightarrow (\ \textbf{skip},\ s\ ) \rightsquigarrow s$
Caso $s \not\models b$: $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow \mathit{erro} \Leftrightarrow (\ \textbf{skip},\ s\ ) \rightsquigarrow s$
caso $x := e$:
Queremos mostrar que $(\ x := e,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \mathcal{T}_3(\ x := e,\ b'\ ),\ s\ ) \rightsquigarrow s'$
Pela Definição 3.3.8 (pág. 53) , basta mostrar: $(\ x := e,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \textbf{if}(\ b'\ )\ \textbf{then}\ x := e,\ s\ ) \rightsquigarrow s'$
Caso $s \models b'$: basta mostrar: $(\ x := e,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ x := e,\ s\ ) \rightsquigarrow s'$
pela definição da relação de transição.
Caso $s \not\models b'$: basta mostrar: $(\ x := e,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \textbf{skip},\ s\ ) \rightsquigarrow s'$

caso $c_i\ ;\ c_j$:
Queremos mostrar que $(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \mathcal{T}_3(\ c_i\ ;\ c_j,\ b'\ ),\ s\ ) \rightsquigarrow s'$
Pela Definição 3.3.8 (pág. 53) , basta mostrar: $(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow s'$
Por Definição 2.4.2 (pág. 37) .

caso $\textbf{if}(\ b\ )\textbf{then}\ c_t\ \textbf{else}\ c_f$:
Suponhamos que
HI_t: $(\ c_t,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \mathcal{T}_3(\ c_t,\ b'\ ),\ s\ ) \rightsquigarrow s'$
HI_f: $(\ c_f,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \mathcal{T}_3(\ c_f,\ b'\ ),\ s\ ) \rightsquigarrow s'$
Queremos mostrar que
$(\ \textbf{if}(\ b\ )\textbf{then}\ c_t\ \textbf{else}\ c_f,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \mathcal{T}_3(\ \textbf{if}(\ b\ )\textbf{then}\ c_t\ \textbf{else}\ c_f,\ b'\ ),\ s\ ) \rightsquigarrow s'$
Pela Definição 3.3.8 (pág. 53) , basta mostrar:
$(\ x := e,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ \mathcal{T}_3(\ c_t,\ b' \wedge\ b\ )\ ;\ \mathcal{T}_3(\ c_f,\ b' \wedge \neg\ b\ ),\ s\ ) \rightsquigarrow s'$

Pela Regra de inferência 4.10 (pág. 72) , sabemos que existe s'' tal que,
$(\ \mathcal{T}_3(\ c_t,\ b' \wedge\ b\ ),\ s\ ) \rightsquigarrow s''$ e $(\ \mathcal{T}_3(\ c_f,\ b' \wedge \neg\ b\ ),\ s''\ ) \rightsquigarrow s'$
Note-se que $s \models b' \wedge\ b \neq s \models b' \wedge \neg\ b$
Caso $s \models b' \wedge\ b = \top$: $s \models b' \wedge \neg\ b = \bot$ e $s'' = s'$.
Pela HI_t.
Caso $s \models b' \wedge\ b = \bot$: $s = s''$ e $s \models b' \wedge \neg\ b = \top$.
Pela HI_f.

□

4.2 Transformação 4 - $\mathcal{T}_4$

Nesta secção provaremos a correcção e a completude da transformação $\mathcal{T}_4$. Mais concretamente, provaremos que um comando $\mathcal{C}nf$ é safe se e só se a fórmula que lhe é associada pela transformação $\mathcal{T}_4$ é uma fórmula válida.

4.2.1 Teorema da Correcção

Para tornarmos a exposição mais concisa, definem-se as seguintes convenções desta notação.

Convenção 4.2.1. *Dado um comando $c \in \mathcal{C}nf$, $\mathcal{C}_c$ e $\mathcal{P}_c$ denotam respectivamente a 1^a e a 2^a componentes de $\mathcal{T}_4(\ c\ )$ (o resultado de aplicar a transformação $\mathcal{T}_4(\ c\ )$ a c).*

Convenção 4.2.2. *Seja $\mathcal{S}$ um conjunto, convencionamos que $\bigwedge \mathcal{S}$ denota a conjunção lógica de todos os elementos do conjunto $\mathcal{S}$, assumindo em particular, que $\bigwedge \emptyset = \top$.*

Note-se no seguinte lema, que em particular $(c,\ s) \not\rightsquigarrow \mathit{erro}$.

Lema 4.2.3. *Lema 4, Cor.PropIII vEval*

$$\forall s \in \Sigma,\ \forall c \in \mathcal{C}nf,\ \text{safe}(\ c,\ s\ ) \Rightarrow (\ c,\ s\ ) \rightsquigarrow \boldsymbol{Eval}_{cnf}(\ c,\ s\ )$$

Demonstração. Por indução estrutural em $\mathcal{C}nf$

i) Caso $c = \textbf{if}(\ b'\ )\ \textbf{then skip}$.
Por um lado, pela definição da função $\textbf{Eval}_{cnf}$, $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then skip},\ s\ ) = s$.
Por outro, pela semântica operacional,
Caso $s \models b'$: $(\ \textbf{if}(\ b'\ )\ \textbf{then skip},\ s\ ) \rightsquigarrow s$ pois $(\ \textbf{skip},\ s\ ) \rightsquigarrow s$.
Caso $s \not\models b'$: $(\ \textbf{if}(\ b'\ )\ \textbf{then skip},\ s\ ) \rightsquigarrow s$ pois $(\ \textbf{skip},\ s\ ) \rightsquigarrow s$.

ii) Caso $c = \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ )$.
Por um lado, pela definição da função $\textbf{Eval}_{cnf}$, $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ ),\ s\ ) = s$.
Por outro, pela semântica operacional,
Caso $s \models b'$:
$(\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ ),\ s\ ) \rightsquigarrow s$ pois $(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s$ e também pela hipótese *safe*$(\ \textbf{assert}(\ b\ ),\ s\ )$, i.e., $(\ \textbf{assert}(\ b\ ),\ s\ ) \not\rightsquigarrow \mathit{erro}$.
Caso $s \not\models b'$:
$(\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ ),\ s\ ) \rightsquigarrow s$ pois $(\ \textbf{skip},\ s\ ) \rightsquigarrow s$.

iii) Caso $c = \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e$.
Por um lado, pela semântica operacional,
Caso $s \models b'$:
Temos $(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ ) \rightsquigarrow s \begin{pmatrix} x \\ e \end{pmatrix}$ pois $(\ x\ :=\ e,\ s\ ) \rightsquigarrow s \begin{pmatrix} x \\ e \end{pmatrix}$.
Pelo outro, neste caso a definição de $\textbf{Eval}_{cnf}$ revela que $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ ) = s \begin{pmatrix} x \\ e \end{pmatrix}$.
Caso $s \not\models b'$:
Temos $(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ ) \rightsquigarrow s$ pois $(\ \textbf{skip},\ s\ ) \rightsquigarrow s$.
Pelo outro, neste caso a definição de $\textbf{Eval}_{cnf}$ revela que $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ ) = s$.

iv) Caso $c = c_i\ ;\ c_j$.
Por hipótese de indução temos,

$$HI_i : \forall s \in \Sigma,\ \forall c_i \in \mathcal{C}nf,\ \mathit{safe}(\ c_i,\ s\ ) \Rightarrow (\ c_i,\ s\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_i,\ s\ )$$

$$HI_j : \forall s \in \Sigma,\ \forall c_j \in \mathcal{C}nf,\ \mathit{safe}(\ c_j,\ s\ ) \Rightarrow (\ c_j,\ s\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_j,\ s\ )$$

Queremos mostrar que

$$\forall s \in \Sigma,\ \forall c \in \mathcal{C}nf,\ \mathit{safe}(\ c,\ s\ ) \Rightarrow (\ c,\ s\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c,\ s\ )$$

Pela definição da função $\textbf{Eval}_{cnf}$, $\textbf{Eval}_{cnf}(\ c_i\ ;\ c_j,\ s\ ) = \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )$.

O Lema 4.2.15 (pág. 71) afirma:

$$\forall\ s \in \Sigma,\ \forall c_i, c_j \in \mathcal{C}\textbf{ommand},\ \mathit{safe}(\ c_i\ ;\ c_j,\ s\ ) \Leftrightarrow \begin{pmatrix} (\ c_i,\ s\ ) \not\rightsquigarrow \mathit{erro} \\ \wedge \\ \forall\ s'' \in \Sigma,\ (\ c_i,\ s\ ) \rightsquigarrow s'' \Rightarrow (\ c_j,\ s''\ ) \not\rightsquigarrow \mathit{erro} \end{pmatrix}$$

Por HI_i e pelo Lema 4.2.15 (pág. 71) temos $(c_i,\ s) \rightsquigarrow \mathbf{Eval}_{cnf}(c_i,\ s)$ e $safe(c_j,\ \mathbf{Eval}_{cnf}(c_i,\ s))$.
Tomando $s'' = \mathbf{Eval}_{cnf}(c_i,\ s)$, por HI_j temos $(c_j,\ \mathbf{Eval}_{cnf}(c_i,\ s)) \rightsquigarrow \mathbf{Eval}_{cnf}(c_j,\ \mathbf{Eval}_{cnf}(c_i,\ s))$.
Como pretendiamos mostrar.

□

Note-se que o seguinte lema é um caso particular do Lema 4.2.3 (pág. 60) , tomando $s' = \mathbf{Eval}_{cnf}(c,\ s)$. Neste caso sabemos mais, sabemos que o s' resultante é sempre $\mathbf{Eval}_{cnf}(c,\ s)$ e sabemos que $s' \neq erro$.

Lema 4.2.4. *Lema 4, Cor.PropIII vS*

$$\forall s \in \Sigma,\ \forall c \in \mathcal{C}nf,\ \text{safe}(c,\ s) \Rightarrow \exists s' \in \Sigma,\ (c,\ s) \rightsquigarrow s'$$

Demonstração. Por indução estrutural em $\mathcal{C}nf$

i) Caso $c = \textbf{if}(b')\ \textbf{then skip}$.
Tome-se $s' = s$
Pela Notação 4 (pág. 72) ,
Caso $s \models b'$: Pelas únicas regras de inferência aplicáveis 4.11 (pág. 72) e 4.7 (pág. 72) , $(\textbf{skip},\ s) \rightsquigarrow s$.
Caso $s \not\models b'$: Pelas únicas regras de inferência aplicáveis 4.12 (pág. 72) e 4.7 (pág. 72) , $(\textbf{skip},\ s) \rightsquigarrow s$.

ii) Caso $c = \textbf{if}(b')\ \textbf{then assert}(b)$.
Tome-se $s' = s$ pois
Pela Notação 4 (pág. 72) ,
Caso $s \models b'$: Pela regra de inferência 4.11 (pág. 72) , $(\textbf{assert}(b),\ s) \rightsquigarrow s'$,
Caso $s \models b$: como $safe(\textbf{if}(b')\ \textbf{then assert}(b),\ s)$, pelo Lema 4.2.13 (pág. 71) ,
segue que $(\textbf{assert}(b),\ s) \rightsquigarrow s' \therefore s' = s \therefore s' \neq erro$.

Caso $s \not\models b$: Pela suposição sabemos que este caso não ocorre pois $(\textbf{if}(b')\ \textbf{then assert}(b),\ s) \not\rightsquigarrow erro$.

Caso $s \not\models b'$: Pela regra de inferência 4.12 (pág. 72) , $(\textbf{skip},\ s) \rightsquigarrow s'$,
Pela Regra de inferência 4.7 (pág. 72) , $s' = s \therefore s' \neq erro$.

iii) Caso $c = \textbf{if}(b')\ \textbf{then}\ x\ :=\ e$.
Tome-se $\begin{cases} s' = s\begin{pmatrix} x \\ e \end{pmatrix} & , se\ s \models b' \\ s' = s & , se\ s \not\models b' \end{cases}$
Pela Notação 4 (pág. 72) ,
Caso $s \models b'$: Pela regra de inferência 4.11 (pág. 72) , $(x\ :=\ e,\ s) \rightsquigarrow s'$,
Pela única regra de inferência aplicável 4.9 (pág. 72) , $(x\ :=\ e,\ s) \rightsquigarrow s\begin{pmatrix} x \\ e \end{pmatrix}$
Note-se $s\begin{pmatrix} x \\ e \end{pmatrix} \neq erro$.

Caso $s \not\models b'$: Pela regra de inferência 4.12 (pág. 72) , $(\textbf{skip},\ s) \rightsquigarrow s'$,
Pela única regra de inferência aplicável 4.7 (pág. 72) , $s' = s \therefore s' \neq erro$.

iv) Caso $c = c_i \ ; \ c_j$.
Por hipótese de indução temos,

$$HI_i : \forall s \in \Sigma,\ \forall c_i \in \mathcal{C}nf,\ safe(\ c_i,\ s\ ) \Rightarrow \exists s' \in \Sigma,\ (\ c_i,\ s\ ) \rightsquigarrow s'$$

$$HI_j : \forall s \in \Sigma,\ \forall c_j \in \mathcal{C}nf,\ safe(\ c_j,\ s\ ) \Rightarrow \exists s' \in \Sigma,\ (\ c_j,\ s\ ) \rightsquigarrow s'$$

Queremos mostrar que

$$\forall s \in \Sigma,\ \forall c \in \mathcal{C}nf,\ safe(\ c,\ s\ ) \Rightarrow \exists s' \in \Sigma,\ (\ c,\ s\ ) \rightsquigarrow s'$$

Suponhamos que $safe(\ c_i \ ; \ c_j,\ s\ )$, pelo Lema 4.2.15 (pág. 71) temos:

1) $\forall\ s \in \Sigma,\ \forall c_i, c_j \in \mathcal{C}\textbf{ommand},\ safe(\ c_i \ ; \ c_j,\ s\ ) \Leftrightarrow \left(\begin{array}{c} (\ c_i,\ s\ ) \not\rightsquigarrow erro \\ \wedge \\ \forall\ s'' \in \Sigma,\ (\ c_i,\ s\ ) \rightsquigarrow s'' \Rightarrow (\ c_j,\ s''\ ) \not\rightsquigarrow erro \end{array} \right)$

Por HI_i e pelo Lema 4.2.15 (pág. 71) temos $(\ c_i,\ s\ ) \rightsquigarrow s''$ e $safe(\ c_j,\ s''\ )$.
Por HI_j temos $(\ c_j,\ s''\ ) \rightsquigarrow s'$ e $safe(\ c_i \ ; \ c_j,\ s\ )$.
Como pretendiamos mostrar.

□

Lema 4.2.5. *Lema 5*

$$\forall s \in \Sigma,\ \forall s' \in \Sigma_{\text{erro}},\ \forall c \in \mathcal{C}nf,\ (\ s \models \wedge\mathcal{C}_c\ ) \wedge (\ c,\ s\ ) \rightsquigarrow s' \wedge (\ s' \neq \text{erro}\ ) \Rightarrow s = s'$$

Demonstração. Por indução estrutural em $\mathcal{C}nf$

i) Caso $c =$ **if**$(\ b'\ )$ **then skip**.
Caso i) do Lema 4.2.4 (pág. 61)

ii) Caso $c =$ **if**$(\ b'\ )$ **then assert**$(\ b\ )$.
Caso ii) do Lema 4.2.4 (pág. 61)

iii) Caso $c =$ **if**$(\ b'\ )$ **then** $x\ :=\ e$.
Por Definição 4.2.19 (pág. 72) , $\wedge\mathcal{C}_c = b' \rightarrow (\ x = e\ )$.

Assim, $s \models b' \rightarrow (\ x = e\ )$, ou seja,

$$(\ s \models b'\ ) \Rightarrow [\![\ x\ ]\!]_s = [\![\ e\ ]\!]_s \tag{4.6}$$

Caso $s \not\models b'$: $s' = s$.
Recorde-se que $c =$ **if**$(\ b'\ )$**then** $x\ :=\ e$ **else skip**,
e pela semântica operacional temos de ter $(\ \textbf{skip},\ s\ ) \rightsquigarrow s$.
Caso $s \models b'$: $s' = s \begin{pmatrix} x \\ e \end{pmatrix}$

Pela semântica operacional temos que ter, $(\ x\ :=\ e,\ s\ ) \rightsquigarrow s'$ logo $s' = s \begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}$,
mas por (4.6), $s(\ x\ ) = [\![\ e\ ]\!]_s$ e portanto $s' = s$.

iv) Caso $c = c_i\ ;\ c_j$. Suponhamos que

$$HI_i : \forall s \in \Sigma,\ \forall s' \in \Sigma_{erro},\ \forall c_i \in \mathcal{C}nf,\ (\ s \models \wedge\mathcal{C}_{c_i}\ ) \wedge (\ c_i,\ s\ ) \rightsquigarrow s' \wedge (\ s' \neq erro\ ) \Rightarrow s = s'$$

$$HI_j : \forall s \in \Sigma,\ \forall s' \in \Sigma_{erro},\ \forall c_j \in \mathcal{C}nf,\ (\ s \models \wedge\mathcal{C}_{c_j}\ ) \wedge (\ c_j,\ s\ ) \rightsquigarrow s' \wedge (\ s' \neq erro\ ) \Rightarrow s = s'$$

Queremos mostrar que

$$\forall s \in \Sigma,\ \forall s' \in \Sigma_{erro},\ \forall c \in \mathcal{C}nf,\ (\ s \models \wedge\mathcal{C}_c\ ) \wedge (\ c,\ s\ ) \rightsquigarrow s' \wedge (\ s' \neq erro\ ) \Rightarrow s = s'$$

1) Por Definição 4.2.19 (pág. 72) , $\wedge\mathcal{C}_{c_i\ ;\ c_j} = \wedge(\ \mathcal{C}_{c_i} \cup \mathcal{C}_{c_j}\ )$, logo, de $s \models \mathcal{C}_c$, temos $(\ s \models \mathcal{C}_{c_i}\ ) \wedge (\ s \models \mathcal{C}_{c_j}\ )$.

2) De $(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow s' \wedge (\ s' \neq erro\ )$ então $\exists\ s'' \in \Sigma :\ (\ c_i,\ s\ ) \rightsquigarrow s'' \wedge (\ c_j,\ s''\ ) \rightsquigarrow s'$.

3) Pela HI_i, de $s \models \wedge\mathcal{C}_{c_i}$ e $(\ c_i,\ s\ ) \rightsquigarrow s''$, segue $s = s''$

4) Como $s = s''$ e $s \models \wedge\mathcal{C}_{c_j}$, $s'' \models \wedge\mathcal{C}_{c_j}$

5) Pela HI_j, de $s'' \models \wedge\mathcal{C}_{c_j}$ e $(\ c_j,\ s''\ ) \rightsquigarrow s'$, segue $s'' = s'$

6) De $s = s''$ e $s'' = s'$, segue $s = s'$

□

Deverá ser consequência do Lema 4.2.5 (pág. 63) e do Lema do determinismo **??** (pág. **??**)

Lema 4.2.6. *Lema 5 eval*

$$\forall s \in \Sigma,\ \forall s' \in \Sigma_{\text{erro}},\ \forall c \in \mathcal{C}nf,\ (\ s \models \wedge\mathcal{C}_c\ ) \wedge (\ c,\ s\ ) \rightsquigarrow s' \wedge (\ s' \neq \text{erro}\ ) \Rightarrow s = \boldsymbol{Eval}_{cnf}(\ c,\ s\ )$$

Demonstração. Por indução estrutural em $\mathcal{C}nf$

i) Caso $c =$ **if**$(\ b'\ )$ **then skip**.
Como **Eval**$_{cnf}(\ $**if**$(\ b'\ )$ **then skip**$,\ s\ ) = s$, basta mostrar que $s' = s$.
Caso i) do Lema 4.2.4 (pág. 61)
ii) Caso $c =$ **if**$(\ b'\ )$ **then assert**$(\ b\ )$.
Como **Eval**$_{cnf}(\ $**if**$(\ b'\ )$ **then assert**$(\ b\ )\ ,\ s\ ) = s$, basta mostrar que $s' = s$.
Caso ii) do Lema 4.2.4 (pág. 61)
iii) Caso $c =$ **if**$(\ b'\ )$ **then** $x\ :=\ e$.
Se $s \models b'$:

Por um lado, **Eval**$_{cnf}(\ $**if**$(\ b'\ )$ **then** $x\ :=\ e,\ s\ ) = s \begin{pmatrix} x \\ e \end{pmatrix}$
Pelo outro, pela regra de inferência 4.11 (pág. 72) , $(\ x\ :=\ e,\ s\ ) \rightsquigarrow s'$,
Pela única regra de inferência aplicável 4.9 (pág. 72) , $(\ x\ :=\ e,\ s\ ) \rightsquigarrow s \begin{pmatrix} x \\ e \end{pmatrix}$.
Logo (**if**$(\ b'\ )$ **then** $x\ :=\ e,\ s\ ) \rightsquigarrow$ **Eval**$_{cnf}(\ $**if**$(\ b'\ )$ **then** $x\ :=\ e,\ s\ )$ se $s \models b'$.

Se $s \not\models b'$:
Por um lado, **Eval**$_{cnf}(\ $**if**$(\ b'\ )$ **then** $x\ :=\ e,\ s\ ) = s$.
Pelo outro, pela regra de inferência 4.12 (pág. 72) , (**skip**$,\ s\ ) \rightsquigarrow s$.
Logo (**if**$(\ b'\ )$ **then** $x\ :=\ e,\ s\ ) \rightsquigarrow$ **Eval**$_{cnf}(\ $**if**$(\ b'\ )$ **then** $x\ :=\ e,\ s\ )$ se $s \not\models b'$.

iv) Caso $c = c_i$ **;** c_j.
Por hipótese de indução temos,

$$HI_i : \forall s \in \Sigma,\ \forall s' \in \Sigma_{erro},\ \forall c_i \in \mathcal{C}nf,\ (\ s \models \wedge\mathcal{C}_{c_i}\ ) \wedge (\ c_i,\ s\ ) \rightsquigarrow s' \wedge (\ s' \neq erro\ ) \Rightarrow s = \textbf{Eval}_{cnf}(\ c_i,\ s\ )$$

$$HI_j : \forall s \in \Sigma,\ \forall s' \in \Sigma_{erro},\ \forall c_j \in \mathcal{C}nf,\ (\ s \models \wedge\mathcal{C}_{c_j}\ ) \wedge (\ c_j,\ s\ ) \rightsquigarrow s' \wedge (\ s' \neq erro\ ) \Rightarrow s = \textbf{Eval}_{cnf}(\ c_j,\ s\ )$$

Queremos mostrar que

$$\forall s \in \Sigma,\ \forall s' \in \Sigma_{erro},\ \forall c \in \mathcal{C}nf,\ (\ s \models \wedge\mathcal{C}_c\ ) \wedge (\ c,\ s\ ) \rightsquigarrow s' \wedge (\ s' \neq erro\ ) \Rightarrow s = \textbf{Eval}_{cnf}(\ c,\ s\ )$$

Por um lado, **Eval**$_{cnf}(\ c_i$ **;** $c_j,\ s\ ) =$ **Eval**$_{cnf}(\ c_j,$ **Eval**$_{cnf}(\ c_i,\ s\ )\ )$.
Pelo outro, como $safe(\ c_i$ **;** $c_j\ )$,
pelo Lema 4.2.15 (pág. 71), então:
1) $(\ c_i,\ s\ ) \not\rightsquigarrow erro$, i.e., $safe(\ c_i,\ s\ )$
2) $\forall\ s' \in \Sigma,\ (\ c_i,\ s\ ) \rightsquigarrow s' \Rightarrow (\ c_j,\ s'\ ) \not\rightsquigarrow erro$, i.e., $safe(\ c_j,\ s'\ )$
e pelo Lema **??** (pág. **??**), então:
3) $\exists\ s'' \in \Sigma :\ (\ c_i,\ s\ ) \rightsquigarrow s'' \wedge (\ c_j,\ s''\ ) \rightsquigarrow s'$
Por HI_i, $s =$ **Eval**$_{cnf}(\ c_i,\ s\ )$
Por 3) $(\ c_i,\ s\ ) \rightsquigarrow s''$.
Por HI_j, $s'' =$ **Eval**$_{cnf}(\ c_j,\ s''\ )$.

Falta mostrar que $s'' = s$.

Pelo lema anterior (Lema 4.2.5 (pág. 63)) e HI_i[?]

□

Teorema 4.2.7. *Terorema I*

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ \text{safe}(\ c,\ s\ ) \Rightarrow s \models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$$

Demonstração. Por indução estrutural nos comandos $c \in \mathcal{C}nf$

i) Caso $c =$ **if**$(\ b'\ )$ **then skip**.
Por definição, $\mathcal{C}_c = \emptyset$ e $\mathcal{P}_c = \emptyset$.
Assim, temos que garantir $s \models \top \rightarrow \top$ e tal é verdade por $\top \rightarrow \top$ ser tautologia.

ii) Caso $c =$ **if**$(\ b'\ )$ **then assert**$(\ b\ )$.
Por definição, $\mathcal{C}_c = \emptyset$ e $\mathcal{P}_c = \{\ b' \rightarrow\ b\ \}$.
Assim, temos que mostrar $s \models b' \rightarrow\ b$.

Caso $s \models b'$:
Como, por hipótese, *safe*(**if**$(\ b'\ )$ **then assert**$(\ b\ )$, $s\ )$ temos que ter $s \models\ b$.
De outro modo teríamos (**if**$(\ b'\ )$ **then assert**$(\ b\ )$, $s\ ) \rightsquigarrow$ *erro*, pois (**assert**$(\ b\ )$, $s\ ) \rightsquigarrow$ *erro* se $s \not\models b$, e contrariando a suposição *safe*(**if**$(\ b'\ )$ **then assert**$(\ b\ )$, $s\ )$.
Caso $s \not\models b'$:
De imediato temos $s \models b' \rightarrow\ b$.

iii) Caso $c =$ **if**$(\ b'\ )$ **then** $x\ :=\ e$.
Por definição, $\mathcal{C}_c = \{\ b' \rightarrow (\ x = e\ )\ \}$ e $\mathcal{P}_c = \emptyset$ e portanto,
temos que mostrar $s \models (\ b' \rightarrow (\ x = e\ )\ ) \rightarrow \top$.
Ora, isto é verdade porque o consequênte da implicação é uma tautologia.

iv) Caso $c = c_i$ **;** c_j.
Por definição, $\bigwedge \mathcal{C}_c = \bigwedge \mathcal{C}_{c_i} \cup \bigwedge \mathcal{C}_{c_j}$ e $\bigwedge \mathcal{P}_c = \bigwedge \mathcal{P}_{c_i} \cup \bigwedge \mathcal{P}_{c_j}$
Como hipóteses de indução temos,

$$HI_i : \forall\ s \in\ cs,\ \mathit{safe}(\ c_i,\ s\ ) \Rightarrow s \models \bigwedge \mathcal{C}_{c_i} \rightarrow \bigwedge \mathcal{P}_{c_i}$$

$$HI_j : \forall\ s \in\ cs,\ \mathit{safe}(\ c_j,\ s\ ) \Rightarrow s \models \bigwedge \mathcal{C}_{c_j} \rightarrow \bigwedge \mathcal{P}_{c_j}$$

Queremos mostrar que,

$$\forall\ s \in\ cs,\ \mathit{safe}(\ c_j,\ s\ ) \Rightarrow s \models \bigwedge \mathcal{C}_{c_j} \rightarrow \bigwedge \mathcal{P}_{c_j}$$

Seja $s \in \Sigma$ tal que *safe*$(\ c_i$ **;** $c_j,\ s\ )$.

Suponhamos que $s \models \bigwedge(\ \mathcal{C}_{c_i} \cup \mathcal{C}_{c_j}\ )$. Temos que mostar que $s \models \wedge(\ \mathcal{P}_{c_i} \cup \mathcal{P}_{c_j}\ )$.
Como *safe*$(\ c_i$ **;** $c_j\ )$, pelo Lema 4.2.15 (pág. 71), então:
1) *safe*$(\ c_i,\ s\ )$
2) $\forall\ s' \in \Sigma,$ *safe*$(\ c_j,\ s'\ )$
a) Ora, por *safe*$(\ c_i,\ s\ )$,
e por HI_i, sabemos que $s \models \bigwedge \mathcal{C}_{c_i} \rightarrow \bigwedge \mathcal{P}_{c_i}$.
Como tal, dado que $s \models \wedge \mathcal{C}_{c_i}$, $s \models \wedge \mathcal{P}_{c_i}$.

b) Falta mostrar $s \models \wedge \mathcal{P}_{c_j}$.
Uma vez que *safe*$(\ c_i,\ s\ )$, pelo Lema 4.2.3 (pág. 60), $(\ c_i,\ s\ ) \rightsquigarrow$ **Eval**$_{cnf}(\ c_i,\ s\ )$
Por 2) *safe*$(\ c_j,$ **Eval**$_{cnf}(\ c_i,\ s\ )\ )$
E por HI_j, **Eval**$_{cnf}(\ c_i,\ s\ ) \models \bigwedge \mathcal{C}_{c_j} \rightarrow \bigwedge \mathcal{P}_{c_j}$.

Uma vez que $s \models \wedge\mathcal{C}_{c_i}$ e $(\ c_i,\ s\ ) \rightsquigarrow \mathbf{Eval}_{cnf}(\ c_i,\ s\ )$ com $\mathbf{Eval}_{cnf}(\ c_i,\ s\ ) \neq \mathit{erro}$,
pelo Lema 4.2.5 (pág. 63), $s = \mathbf{Eval}_{cnf}(\ c_i,\ s\ )$.
Logo, como $s \models \wedge\mathcal{C}_{c_j}$, $\mathbf{Eval}_{cnf}(\ c_i,\ s\ ) \models \wedge\mathcal{C}_{c_j}$.
Portanto, $\mathbf{Eval}_{cnf}(\ c_i,\ s\ ) \models \wedge\mathcal{P}_{c_j}$ e sendo $\mathbf{Eval}_{cnf}(\ c_i,\ s\ ) = s$, $s \models \wedge\mathcal{P}_{c_j}$, como pretendido.
□

Teorema 4.2.8. *Correcção/Completude de* $\mathcal{T}_4$

$$\forall c \in \mathcal{C}nf,\ \text{safe}(\ c\ ) \Rightarrow \models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$$

Demonstração. Suponhamos que *safe*(c), i.e., suponhamos que $\forall\ s \in \Sigma :\ \mathit{safe}(\ c,\ s\ )$.
Queremos mostrar que $\models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$, i.e., $\forall\ s \in \Sigma :\ s \models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$.

Seja $s \in \Sigma$. Pela suposição, sabemos *safe*($c,\ s$).
Logo, pelo Teorema 4.2.7 (pág. 66), segue $s \models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$, como pretendido. □

Avaliação de elementos da classe $\mathcal{C}$ommand

Definição 4.2.9. *A função parcial* $\boldsymbol{Eval}_c : \mathcal{C}\textbf{ommand} \to \Sigma \to \Sigma$ *é definida recursivamente por:*

$$
\begin{array}{rcl}
\boldsymbol{Eval}_c(\ \textbf{\textit{skip}},\ s\ ) & = & s \\
\boldsymbol{Eval}_c(\ \textbf{\textit{assert}}(\ b\ ),\ s\ ) & = & s \\
\boldsymbol{Eval}_c(\ x\ :=\ e,\ s\ ) & = & s\begin{pmatrix} x \\ e \end{pmatrix} \\
\boldsymbol{Eval}_c(\ c_i\ \textbf{;}\ c_j,\ s\ ) & = & \boldsymbol{Eval}_c(\ c_j,\ \boldsymbol{Eval}_c(\ c_i,\ s\ )\ ) \\
\boldsymbol{Eval}_c(\ \textbf{\textit{if}}(\ b\ )\textbf{\textit{then}}\ c_t\ \textbf{\textit{else}}\ c_f,\ s\ ) & = & \begin{cases} \boldsymbol{Eval}_c(\ c_t,\ s\ ) & , s \models b \\ \boldsymbol{Eval}_c(\ c_f,\ s\ ) & , s \not\models b \end{cases} \\
\boldsymbol{Eval}_c(\ \textbf{\textit{while}}(\ b\ )\textbf{\textit{do}}\ c_w,\ s\ ) & = & \begin{cases} s & , s \not\models b \\ \boldsymbol{Eval}_c(\ \textbf{\textit{while}}(\ b\ )\textbf{\textit{do}}\ c_w,\ \boldsymbol{Eval}_c(\ c_w,\ s\ )\ ) & , s \models b \end{cases}
\end{array}
$$

onde $(\ \textbf{\textit{while}}(\ b\ )\textbf{\textit{do}}\ c_w,\ s\ )\downarrow$ *não origina computação infinita.*

O comando **skip**, semânticamente não realiza qualquer alteração ao estado. A sua avaliação deve resultar no estado inicial.

O comando **assert**$(\ b\ )$, procurando penas afirmar propriedades sobre o estado, semânticamente não realiza qualquer alteração ao estado. A sua avaliação deve resultar no estado inicial.

O comando $x\ :=\ e$, atribui o valor de uma expressão inteira a uma variável. A sua avaliação deve resultar no estado inicial, onde o valor da variável x contém o valor $[\![\ e\ ]\!]_s$. Isto é melhor expresso pela função $s\begin{pmatrix} x \\ e \end{pmatrix}$.

O comando c_i **;** c_j, resulta na avaliação consecutiva de comandos. surge naturalmente que o comando c_j deve ser avaliado com o estado resultante da avaliação de c_i ao estado inicial s.

O comando **if**$(\ b\ )$**then** c_t **else** c_f é um condicional, logo deve exprimir a avaliação do comando c_t a s caso $s \models b$, ou exclusivamente, a avaliação do comando c_f a s caso $s \not\models b$,

Caso a computação termine, o comando **while**$(\ b\ )$**do** c_w exprime um ciclo que deve ter o mesmo comportamento que o comando **skip** quando $s \not\models b$ e quando $s \models b$, o corpo do ciclo deve ser avaliado com o estado prévio s e só depois se deve racícionar sobre a avaliação da próxima iteração do comando **while**$(\ b\ )$**do** c_w sobre $\textbf{Eval}_c(\ c_w,\ s\ )$.

Avaliação de elementos da classe $\mathcal{C}nf$

Definição 4.2.10. *A função* $\boldsymbol{Eval}_{cnf} : \mathcal{C}nf \to \Sigma \to \Sigma$ *é definida recursivamente por:*

$$
\begin{array}{rcl}
\boldsymbol{Eval}_{cnf}(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then skip}},\ s\ ) & = & s \\
\boldsymbol{Eval}_{cnf}(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then assert}}(\ b\ ),\ s\ ) & = & s \\
\boldsymbol{Eval}_{cnf}(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then}}\ x\ :=\ e,\ s\ ) & = & \begin{cases} s\begin{pmatrix} x \\ e \end{pmatrix} & s \models b' \\ s & s \not\models b' \end{cases} \\
\boldsymbol{Eval}_{cnf}(\ c_i\ \textbf{;}\ c_j,\ s\ ) & = & \boldsymbol{Eval}_{cnf}(\ c_j,\ \boldsymbol{Eval}_{cnf}(\ c_i,\ s\ )\ )
\end{array}
$$

Equivalência entre definições

Lema 4.2.11. *Para quaisquer* $c \in \mathcal{C}nf$ *e* $s \in \Sigma$,

$$\boldsymbol{Eval}_{cnf}(\ c,\ s\ ) = \boldsymbol{Eval}_{c}(\ c,\ s\ )$$

Demonstração.
i) $c = \textbf{if}(\ b'\ )\ \textbf{then skip}$
Caso $s \models b'$:
Por um lado $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then skip},\ s\ ) = s$
e por outro $\textbf{Eval}_{c}(\ \textbf{if}(\ b'\ )\ \textbf{then skip},\ s\ ) = \textbf{Eval}_{c}(\ \textbf{skip},\ s\ ) = s$
Caso $s \not\models b'$:
Por um lado $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then skip},\ s\ ) = s$
e por outro$\textbf{Eval}_{c}(\ \textbf{if}(\ b'\ )\ \textbf{then skip},\ s\ ) = \textbf{Eval}_{c}(\ \textbf{skip},\ s\ ) = s$

ii) $c = \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ )$
Caso $s \models b'$:
Por um lado $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ ),\ s\ ) = s$
e por outro$\textbf{Eval}_{c}(\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ ),\ s\ ) = \textbf{Eval}_{c}(\ \textbf{assert}(\ b\ ),\ s\ ) = s$
Caso $s \not\models b'$:
Por um lado $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ ),\ s\ ) = s$
e por outro $\textbf{Eval}_{c}(\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ ),\ s\ ) = \textbf{Eval}_{c}(\ \textbf{skip},\ s\ ) = s$

iii) $c = \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e$
Caso $s \models b'$:
Por um lado $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ ) = s\begin{pmatrix} x \\ e \end{pmatrix}$,
e por outro $\textbf{Eval}_{c}(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ ) = \textbf{Eval}_{c}(\ x\ :=\ e,\ s\ ) = s\begin{pmatrix} x \\ e \end{pmatrix}$.
Caso $s \not\models b'$:
Por um lado $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ ) = s$,
e por outro $\textbf{Eval}_{c}(\ \textbf{if}(\ b'\ )\ \textbf{then skip},\ s\ ) = \textbf{Eval}_{c}(\ \textbf{skip},\ s\ ) = s$

iv) $c = c_i\ ;\ c_j$
Por um lado $\textbf{Eval}_{cnf}(\ c_i\ ;\ c_j,\ s\ ) = \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )$,
e por outro $\textbf{Eval}_{c}(\ c_i\ ;\ c_j,\ s\ ) = \textbf{Eval}_{c}(\ c_j,\ \textbf{Eval}_{c}(\ c_i,\ s\ )\ )$.
Por HI_i temos que para qualquer s, $\textbf{Eval}_{cnf}(\ c_i,\ s\ ) = \textbf{Eval}_{c}(\ c_i,\ s\ )$.
Por HI_j temos que para qualquer s, $\textbf{Eval}_{cnf}(\ c_j,\ s\ ) = \textbf{Eval}_{c}(\ c_j,\ s\ )$.
Segue das hipóteses.

□

4.2.2 Lemas auxiliares

Lema 4.2.12.

$$\forall\ s \in \Sigma,\ \forall b' \in \mathcal{E}xpression_{Boolean},\ \text{safe}(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then skip}},\ s\ )$$

Demonstração.
Queremos mostrar que para qualquer $s \in \Sigma$, (**if**(b') **then skip**, s) $\not\rightsquigarrow$ *erro*.
Pela semântica operacional, quer $s \models b'$ ou $s \not\models b'$, **if**(b') **then skip** $\approx$ **skip**.
Como $s \in \Sigma$ temos $s \neq$ *erro*, e por (**skip**, s) $\rightsquigarrow s$, obtemos o que pretendiamos mostrar.

□

Lema 4.2.13.

$$\forall\ s \in \Sigma,\ \forall b', b \in \mathcal{E}xpression_{Boolean},\ \text{safe}(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then assert}}(\ b\ ),\ s\ ) \Leftrightarrow \begin{pmatrix} s \not\models b' \\ \vee \\ s \models b' \wedge\ b \end{pmatrix}$$

Demonstração.
a) Pela Definição 4.2.17 (pág. 72) ,
de *safe*(**if**(b') **then assert**(b), s) temos $\forall\ s \in \Sigma :$ (**if**(b') **then assert**(b), s) $\not\rightsquigarrow$ *erro*
b) Pelas regras de inferência 4.8 (pág. 72) e 4.7 (pág. 72) , de (**if**(b') **then assert**(b), s) $\not\rightsquigarrow$ *erro* segue

$$\forall\ s \in \Sigma :\ \begin{pmatrix} s \not\models b' \\ \vee \\ s \models b' \wedge\ b \end{pmatrix}$$

□

Lema 4.2.14.

$$\forall\ s \in \Sigma,\ \forall b' \in \mathcal{E}xpression_{Boolean},\ \forall\ x \in \mathcal{V}_{Integer},\ \forall\ e \in \mathcal{E}xpression_{Integer},\ \text{safe}(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then}}\ x\ :=\ e,\ s\ )$$

Demonstração.
Queremos mostrar que para qualquer $s \in \Sigma$, (**if**(b') **then** $x\ :=\ e$, s) $\not\rightsquigarrow$ *erro*.
if(b') **then** $x\ :=\ e$ abrevia **if**(b')**then** $x\ :=\ e$ **else skip**.
Dado $s \in \Sigma$, (**if**(b') **then** $x\ :=\ e,\ s\ ) \rightsquigarrow s' \Rightarrow$ ($x\ :=\ e,\ s\ ) \rightsquigarrow s'$ ou (**skip**, s) $\rightsquigarrow s'$.
E atendendo às regras que definem $\rightsquigarrow$, s' é necessariamente diferente do estado *erro*.

□

Lema 4.2.15.

$$\forall\ s \in \Sigma,\ \forall c_i, c_j \in \mathcal{C}\textbf{ommand},\ \text{safe}(\ c_i\ ;\ c_j,\ s\ ) \Leftrightarrow \begin{pmatrix} (\ c_i,\ s\ ) \not\rightsquigarrow \text{erro} \\ \wedge \\ \forall\ s'' \in \Sigma,\ (\ c_i,\ s\ ) \rightsquigarrow s'' \Rightarrow (\ c_j,\ s''\ ) \not\rightsquigarrow \text{erro} \end{pmatrix}$$

Demonstração.
a) Pela Definição 4.2.17 (pág. 72) , de *safe*(c_i **;** c_j, s) temos $\forall\ s \in \Sigma :$ (c_i **;** c_j, s) $\not\rightsquigarrow$ *erro*
b) Pela Regra de inferência 4.10 (pág. 72) , de (c_i **;** c_j, s) $\not\rightsquigarrow$ *erro* segue

$$\forall\ s \in \Sigma :\ \begin{pmatrix} (\ c_i,\ s\ ) \not\rightsquigarrow \textit{erro} \\ \wedge \\ \forall\ s' \in \Sigma :\ (\ c_i,\ s\ ) \rightsquigarrow s' \Rightarrow (\ c_j,\ s'\ ) \not\rightsquigarrow \textit{erro} \end{pmatrix}$$

□

Definição 4.2.16. $\wedge\emptyset =_{def} \top$

Definição 4.2.17.

$$\text{safe}(\ c,\ s\ ) \Leftrightarrow_{def} (\ c,\ s\ ) \not\rightsquigarrow \text{erro}$$

Definição 4.2.18.

$$\text{safe}(\ c\ ) \Leftrightarrow_{def} \forall\ s \in \Sigma :\ \text{safe}(\ c,\ s\ )$$

Definição 4.2.19. *Para quaisquer* $b \in \mathcal{E}xpression_{Boolean}$ *e* $c \in \mathcal{C}$**ommand** : $\text{sa}(\ c\ )$, *A transformação* $\mathcal{T}_3$ *é definida recursivamente do seguinte modo:*

$$\begin{array}{rcl} \mathcal{T}_4 & : & \mathcal{C}nf \rightarrow \mathcal{E}xpression_{Boolean} \\ \mathcal{T}_4(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then skip}}\ ) & = & (\ \emptyset,\ \emptyset\ ) \\ \mathcal{T}_4(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then assert}}(\ b\ )\ ) & = & (\ \emptyset,\ \{\ b' \rightarrow\ b\ \}\ ) \\ \mathcal{T}_4(\ \textbf{\textit{if}}(\ b'\ )\ \textbf{\textit{then}}\ x\ :=\ e\ ) & = & (\ \{\ b' \rightarrow (\ x = e\ )\ \},\ \emptyset\ ) \\ \mathcal{T}_4(\ c_i\ \textbf{;}\ c_j\ ) & = & (\ \mathcal{C}_{c_i} \cup \mathcal{C}_{c_j},\ \mathcal{P}_{c_i} \cup \mathcal{P}_{c_j}\ ) \end{array} \quad \textit{onde} \quad \begin{array}{l} (\ \mathcal{C},\ \mathcal{P}\ ) = \mathcal{T}_4(\ ) \\ (\ \mathcal{C}_{c_j},\ \mathcal{P}_{c_j}\ ) = \mathcal{T}_4(\ c_j\ ) \end{array}$$

Notação 4. *Para quaisquer* $b \in \mathcal{E}xpression_{Boolean}$ *e* $c \in \mathcal{C}$**ommand**, *Denota-se* ***if***$(\ b\ )$ ***then*** c *como a abreviação* ***if***$(\ b\ )$***then*** c ***else skip***

$$\frac{}{(\ \textbf{skip},\ s\ ) \rightsquigarrow s}\ \textbf{SKIP} \tag{4.7}$$

$$\frac{s \models b}{(\ \textbf{assert}(\ b\ ),\ s\ ) \rightsquigarrow s}\ \textbf{ASSERT} \tag{4.8}$$

$$\frac{}{(\ x\ :=\ e,\ s\ ) \rightsquigarrow s \begin{pmatrix} x \\ s \models e \end{pmatrix}}\ \textbf{ASSIGN} \tag{4.9}$$

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow s'' \quad s'' \neq \mathit{erro} \quad (\ c_j,\ s''\ ) \rightsquigarrow s'}{(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow s'}\ \textbf{COMPOSITION} \tag{4.10}$$

$$\frac{s \models b \quad (\ c_t,\ s\ ) \rightsquigarrow s'}{(\ \textbf{if}(\ b\ )\textbf{then}\ c_t\ \textbf{else}\ c_f,\ s\ ) \rightsquigarrow s'}\ \textbf{IF - True} \tag{4.11}$$

$$\frac{s \not\models b \quad (\ c_f,\ s\ ) \rightsquigarrow s'}{(\ \textbf{if}(\ b\ )\textbf{then}\ c_t\ \textbf{else}\ c_f,\ s\ ) \rightsquigarrow s'}\ \textbf{IF - False} \tag{4.12}$$

Lema 4.2.20. *[Sugestão]*
Para quaisquer $c \in \mathcal{C}nf$,

$$(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow s' \wedge s' \neq \text{erro} \Rightarrow \exists\ s'' \in \Sigma :\ (\ c_i,\ s\ ) \rightsquigarrow s'' \wedge (\ c_j,\ s''\ ) \rightsquigarrow s'$$

4.2.3 Teorema da Completude

Teorema 4.2.21. *(Teorema Completude)*

$$\forall c \in \mathcal{C}nf,\ \models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c \Rightarrow \text{safe}(\ c\ )$$

Demonstração. Mostraremos o contra-recíproco da implicação.
Suponhamos que $\neg$*safe*(c), i.e., $\exists\ s \in \Sigma,\ (\ c,\ s\ ) \rightsquigarrow$ *erro*.

Logo, pela Proposição 2 4.2.25 (pág. 77) , temos

$$(\ c,\ \mathbf{Eval}_{cnf}(\ c,\ s\ )\ ) \rightsquigarrow \mathit{erro} \tag{4.13}$$

Pela Proposição 1 i 4.2.22 (pág. 74) , temos

$$\mathbf{Eval}_{cnf}(\ c,\ s\ ) \models \bigwedge \mathcal{C}_c \tag{4.14}$$

Logo de (4.13) e (4.14), pela Proposição 3 4.2.24 (pág. 76) ,

$$\mathbf{Eval}_{cnf}(\ c,\ s\ ) \not\models \wedge \mathcal{P}_c \tag{4.15}$$

De (4.14) e (4.15) segue que $\not\models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$, como pretendido.

□

Proposição 4.2.22. *Proposição 1 i*

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ \boldsymbol{Eval}_{cnf}(\ c,\ s\ ) \models \bigwedge \mathcal{C}_c$$

Demonstração. Por indução estrutural em $c \in \mathcal{C}nf$.

i) Caso $c =$ **if**$(\ b'\ )$ **then skip**.
Pela Definição 4.2.19 (pág. 72) de $\mathcal{T}_4$, $\mathcal{C}_c = \emptyset$.
Assim, pela Definição 4.2.16 (pág. 71) , $\bigwedge \mathcal{C}_c = \top$, pelo que, trivialmente, temos $\models \bigwedge \mathcal{C}_c$.

ii) Caso $c =$ **if**$(\ b'\ )$ **then assert**$(\ b\ )$.
Pela Definição 4.2.19 (pág. 72) de $\mathcal{T}_4$, $\mathcal{C}_c = \emptyset$, e tal como no caso anterior,
$\bigwedge \mathcal{C}_c = \top$, e consequentemente, $\models \bigwedge \mathcal{C}_c$.

iii) Caso $c =$ **if**$(\ b'\ )$ **then** $x\ :=\ e$.
Pela Definição 4.2.19 (pág. 72) de $\mathcal{T}_4$, $\mathcal{C}_c = \{\ b' \rightarrow (\ x = e\ )\ \}$.
Caso $s \models b'$: $\mathbf{Eval}_{cnf}(\ c,\ s\ ) = s \begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}$, e como tal

$$s \begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix} \models (\ x = e\ )\ sse\ [\![\ e\ ]\!]_s = [\![\ e\ ]\!]_{s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}} \tag{4.16}$$

Como $x \notin\ e$, $[\![\ e\ ]\!]_{s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}} = [\![\ e\ ]\!]_s$.

Caso $s \not\models b'$:
$s \models b' \rightarrow (\ x = e\ )$ independente de $s \models (\ x = e\ )$ ou $s \not\models (\ x = e\ )$.
(O antecedente da implicação é falso).
Como $\mathbf{Eval}_{cnf}(\ c,\ s\ ) = s$,
também $\mathbf{Eval}_{cnf}(\ c,\ s\ ) \not\models b'$ e $\mathbf{Eval}_{cnf}(\ c,\ s\ ) \models b' \rightarrow (\ x = e\ )$.

iv) Caso $c = c_i\ ;\ c_j$.
Por hipótese de indução temos,

$$\begin{array}{ll} HI_i: & \forall\ s \in \Sigma,\ \mathbf{Eval}_{cnf}(\ c_i,\ s\ ) \models \bigwedge \mathcal{C}_{c_i} \\ HI_j: & \forall\ s \in \Sigma,\ \mathbf{Eval}_{cnf}(\ c_j,\ s\ ) \models \bigwedge \mathcal{C}_{c_j} \end{array}$$

Por definição, $\mathbf{Eval}_{cnf}(\ c_i\ ;\ c_j,\ s\ ) = \mathbf{Eval}_{cnf}(\ c_j,\ \mathbf{Eval}_{cnf}(\ c_i,\ s\ )\ )$.
Queremos então mostrar que $\mathbf{Eval}_{cnf}(\ c_j,\ \mathbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_i} \bigwedge \mathcal{C}_{c_j}$.

i) Pela HI_j, temos $\mathbf{Eval}_{cnf}(\ c_j,\ \mathbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_j}$.

ii)Falta provar que $\mathbf{Eval}_{cnf}(\ c_j,\ \mathbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_i}$
Dado que $\mathbf{assign}(\ c_j\ ) \cap FV(\ \bigwedge \mathcal{C}_{c_i}\ ) = \emptyset$.
Então $\forall\ x \in FV(\ \bigwedge \mathcal{C}_{c_i}\ )$, $\mathbf{Eval}_{cnf}(\ c_i,\ s\ )(\ x\ ) = \mathbf{Eval}_{cnf}(\ c_j,\ \mathbf{Eval}_{cnf}(\ c_i,\ s\ )\ )(\ x\ )$. Logo,[por lema já visto] $\mathbf{Eval}_{cnf}(\ c_j,\ \mathbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_i}$ sse $\mathbf{Eval}_{cnf}(\ c_i,\ s\ ) \models \bigwedge \mathcal{C}_{c_i}$.

Logo, como $\mathbf{Eval}_{cnf}(\ c_i,\ s\ ) \models \bigwedge \mathcal{C}_{c_i}$, também $\mathbf{Eval}_{cnf}(\ c_j,\ \mathbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_i}$.

□

Proposição 4.2.23. *Proposição 1 ii*

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ \forall x \in \mathcal{V}_{Integer},\ \boldsymbol{Eval}_{cnf}(\ c,\ s\ )(\ x\ ) \neq s(\ x\ ) \Rightarrow x \in \boldsymbol{assign}(\ c\ )$$

Demonstração. Por indução estrutural em $c \in \mathcal{C}nf$.

i) Caso $c = \textbf{if}(\ b'\ )\ \textbf{then skip}$.
Como $\textbf{Eval}_{cnf}(\ c,\ s\ ) = s$, o antecedente da implicação que pretendemos provar se torna falso, e como consequência a propriedade é uma verdade vacuosa.

ii) Caso $c = \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ )$.
Como $\textbf{Eval}_{cnf}(\ c,\ s\ ) = s$, e tal como no caso anterior,
a propriedade torna-se uma verdade vacuosa.

iii) Caso $c = \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e$.
Por definição, $\textbf{assign}(\ c\ ) = \textbf{assign}(\ x\ :=\ e\ ) \cup \textbf{assign}(\ \textbf{skip}\ ) = \{\ x\ \}$.

iv) Caso $c = c_i\ ;\ c_j$.
Por hipótese de indução temos,

$$\begin{array}{ll} HI_i: & \forall\ s \in \Sigma,\ \forall\ x \in \mathcal{V}_{Integer},\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )(\ x\ ) \neq s(\ x\ ) \Rightarrow x \in \textbf{assign}(\ c_i\ ) \\ HI_j: & \forall\ s \in \Sigma,\ \forall\ x \in \mathcal{V}_{Integer},\ \textbf{Eval}_{cnf}(\ c_j,\ s\ )(\ x\ ) \neq s(\ x\ ) \Rightarrow x \in \textbf{assign}(\ c_j\ ) \end{array}$$

Queremos mostrar que,

$$\textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )(\ x\ ) \neq\ s(\ x\ ) \Rightarrow\ x \in \textbf{assign}(\ c_i\ ) \cup \textbf{assign}(\ c_j\ )$$

Caso $\textbf{Eval}_{cnf}(\ c_i,\ s\ )(\ x\ ) \neq\ s(\ x\ )$. Pela HI_i temos $x \in \textbf{assign}(\ c_i\ )$ logo $x \in \textbf{assign}(\ c_i\ ;\ c_j\ )$.

Caso $\textbf{Eval}_{cnf}(\ c_i,\ s\ )(\ x\ ) = s(\ x\ )$.
Sabemos que $\textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )(\ x\ ) \neq\ s(\ x\ )$, pela hipótese inicial.
Tome-se $\textbf{Eval}_{cnf}(\ c_i,\ s\ )$, por HI_j
Se $\textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )(\ x\ ) \neq \textbf{Eval}_{cnf}(\ c_i,\ s\ )(\ x\ )$, então $x \in \textbf{assign}(\ c_j\ )$

□

Proposição 4.2.24. *Proposição 3*

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ (\ c,\ s\ ) \rightsquigarrow \text{erro} \wedge (\ s \models \bigwedge \mathcal{C}_c\ ) \Rightarrow s \not\models \bigwedge \mathcal{P}_c$$

Demonstração. Por indução em $(\ c,\ s\ ) \rightsquigarrow$ *erro*.

Pela semântica operacional, as unicas regras que permitem a transição para o estado *erro* são as regras **CNF - ASSERT - Error**, **CNF - COMPOSITION - Break** e **CNF - COMPOSITION**, sendo os restantes casos demonstrados por vacuosidade.

Caso **CNF - ASSERT - Error**,
Suponhamos que $(\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ ),\ s\ ) \rightsquigarrow$ *erro* e $s \models \bigwedge \mathcal{C}_{\textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ )}$.
Queremos mostrar que $s \not\models \bigwedge \mathcal{P}_{\textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ )}$.

Sabemos então que $\bigwedge \mathcal{C}_{\textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ )} = \top$ e $\bigwedge \mathcal{P}_{\textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ )} = b' \rightarrow\ b$.
Basta mostrar que $s \not\models b' \rightarrow\ b$, que é verdade só no caso onde se aplica a regra **CNF - ASSERT - Error**, sendo que $s \models b'$ e $s \not\models b$.

Caso **CNF - COMPOSITION - Break**,

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow \mathit{erro}}{(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{CNF - COMPOSITION - Break} \tag{4.17}$$

Sabemos então que, por suposição
1) $(\ c_i,\ s\ ) \rightsquigarrow$ *erro*,
2) $s \models \bigwedge \mathcal{C}_{c_i} \bigwedge \mathcal{C}_{c_j}$, i.e., $s \models \bigwedge \mathcal{C}_{c_i}$ e $s \models \bigwedge \mathcal{C}_{c_j}$
sabemos ainda, por HI, que
3) $(\ c_i,\ s\ ) \rightsquigarrow \mathit{erro} \wedge (\ s \models \bigwedge \mathcal{C}_{c_i}\ ) \Rightarrow s \not\models \bigwedge \mathcal{P}_{c_i}$.

Queremos mostrar que $s \not\models \mathcal{P}_{c_i\ ;\ c_j}$. i.e., $s \not\models \bigwedge \mathcal{P}_{c_i} \bigwedge \mathcal{P}_{c_j}$,

Ora, de 1), 2) e 3) temos $s \not\models \bigwedge \mathcal{P}_{c_i}$ e disto, também $s \not\models \bigwedge \mathcal{P}_{c_i} \bigwedge \mathcal{P}_{c_j}$.

Caso **CNF - COMPOSITION**.

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow s'' \quad s'' \neq \mathit{erro} \quad (\ c_j,\ s''\ ) \rightsquigarrow s'}{(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow s'}\ \textbf{CNF - COMPOSITION} \tag{4.18}$$

Sabemos então que, por suposição
1) Existe $s'' \in \Sigma$ tal que $(\ c_i,\ s\ ) \rightsquigarrow s''$ e $(\ c_j,\ s''\ ) \rightsquigarrow$ *erro*,
2) $s \models \bigwedge \mathcal{C}_{c_i} \bigwedge \mathcal{C}_{c_j}$, i.e., $s \models \bigwedge \mathcal{C}_{c_i}$ e $s \models \bigwedge \mathcal{C}_{c_j}$
sabemos ainda, por HI, que
HI_i) $(\ c_i,\ s\ ) \rightsquigarrow \mathit{erro} \wedge (\ s \models \bigwedge \mathcal{C}_{c_i}\ ) \Rightarrow s \not\models \bigwedge \mathcal{P}_{c_i}$,
HI_j) $(\ c_j,\ s\ ) \rightsquigarrow \mathit{erro} \wedge (\ s \models \bigwedge \mathcal{C}_{c_j}\ ) \Rightarrow s \not\models \bigwedge \mathcal{P}_{c_j}$.

Queremos mostrar que $s \not\models \mathcal{P}_{c_i\ ;\ c_j}$. i.e., $s \not\models \bigwedge \mathcal{P}_{c_i} \bigwedge \mathcal{P}_{c_j}$,

Tomando $s \in \Sigma$, sabemos que $s'' \neq$ *erro*, pois caso contrário aplicar-se ia a regra **CNF - COMPOSITION - Break**.
Pelo Lema 5 sabemos que $s'' = s$.

Ora, de 1) e 2), tomando $s'' = s$, por HI_j temos $s \not\models \bigwedge \mathcal{P}_{c_j}$ e disto, também $s \not\models \bigwedge \mathcal{P}_{c_i} \bigwedge \mathcal{P}_{c_j}$. $\square$

Proposição 4.2.25. *Proposição 2 i*

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma, \quad \begin{array}{c} (\ c,\ s\ ) \rightsquigarrow \text{erro} \\ \Downarrow \\ (\ c,\ \boldsymbol{Eval}_{cnf}(\ c,\ s\ )\ ) \rightsquigarrow \text{erro} \end{array}$$

Demonstração. Por indução em $(\ c,\ s\ ) \rightsquigarrow s'$.

Seja $\mathcal{X}$ o seguinte conjunto de comandos $\{$ **if**$(\ b'\ )$ **then skip**, **if**$(\ b'\ )$ **then assert**$(\ b\ )\ \} \subseteq \mathcal{C}nf$,

a prova dos casos $\left\{ \begin{array}{l} \textbf{CNF - SKIP} \\ \textbf{CNF - SKIP - False} \\ \\ \textbf{CNF - ASSERT} \\ \textbf{CNF - ASSERT - Error} \\ \textbf{CNF - ASSERT - False} \end{array} \right\}$ e **CNF - ASSIGN - False** são semelhantes, onde $c \in \mathcal{X}$,

por definição, $\textbf{Eval}_{cnf}(\ c,\ s\ ) = s$.

Pela hipótese é imediato que $(\ c,\ \textbf{Eval}_{cnf}(\ c,\ s\ )\ ) \rightsquigarrow s$.

Caso **CNF - ASSIGN**,

1) Pela Definição 4.2.10 (pág. 69) , como $s \models b'$,

temos $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ ) = s \begin{pmatrix} x \\ e \end{pmatrix}$.

2) Por $x \notin FV(\ b'\ )$, $(\ s \models b'\ ) \Rightarrow (\ \textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ ) \models b'\ )$.

3) Falta mostrar que $\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ ) \begin{pmatrix} x \\ [\![\ e\ ]\!]_{\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ )} \end{pmatrix} = s \begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}$

Basta mostrar que $[\![\ e\ ]\!]_{\textbf{Eval}_{cnf}(\ \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\ )} = [\![\ e\ ]\!]_s$

Tal deve seguir de $x \notin FV(\ e\ )$ (Lema A ?? (pág. ??)) que é garantido de $c \in \mathcal{SA}$.

Caso **CNF - COMPOSITION - Break**:

1) HI garante $(\ c_i,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) \rightsquigarrow \textit{erro}$

2) Como $c_i\ ;\ c_j \in \mathcal{SA}$, $FV(\ c_i\ ) \cap \textbf{assign}(\ c_j\ ) = \emptyset$, de 1),

segue pelo Lema A ?? (pág. ??) $(\ c_i,\ \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow \textit{erro}$,

onde $\textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) = \textbf{Eval}_{cnf}(\ c_i,\ s\ )$.

Caso **CNF - COMPOSITION**:

1) HI_2 garante $(\ c_j,\ \textbf{Eval}_{cnf}(\ c_j,\ s''\ )\ ) \rightsquigarrow s'$

2) HI_1 garante $(\ c_i,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) \rightsquigarrow s''$ e $s'' \neq \textit{erro}$.

Como $FV(\ c_i\ ) \cap \textbf{assign}(\ c_j\ ) = \emptyset$ por Lema A ?? (pág. ??),

e $s'' \neq \textit{erro}$ temos $(\ c_i,\ \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_j,\ s''\ )$.

Logo $\dfrac{(\ c_i,\ \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_j,\ s''\ ) \quad (\ c_j,\ \textbf{Eval}_{cnf}(\ c_j,\ s''\ )\ ) \rightsquigarrow s'}{(\ c_i\ ;\ c_j,\ \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow s'}$ **CNF - COMPOSI**

onde $\textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) = \textbf{Eval}_{cnf}(\ c_i\ ;\ c_j,\ s\ )$

□

Proposição 4.2.26. *Proposição 2 ii*

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma, \quad \begin{array}{c} (\ c,\ s\ ) \rightsquigarrow \boldsymbol{Eval}_{cnf}(\ c,\ s\ ) \\ \Downarrow \\ (\ c,\ \boldsymbol{Eval}_{cnf}(\ c,\ s\ )\ ) \rightsquigarrow \boldsymbol{Eval}_{cnf}(\ c,\ s\ ) \end{array}$$

Demonstração. Por indução estrutural em $c \in \mathcal{C}nf$.

i) Caso $c = \textbf{if}(\ b'\ )\ \textbf{then skip}$.
Pela Definição $\textbf{Eval}_{cnf}(\ c,\ s\ ) = s$.

ii) Caso $c = \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ )$.
Pela Definição $\textbf{Eval}_{cnf}(\ c,\ s\ ) = s$.

iii) Caso $c = \textbf{if}(\ b'\ )\ \textbf{then}\ x\ :=\ e$.
Caso $b' \not\models s$.
Pela Definição $\textbf{Eval}_{cnf}(\ c,\ s\ ) = s$.

Caso $b' \models s$.
Pela Definição $\textbf{Eval}_{cnf}(\ c,\ s\ ) = s \begin{pmatrix} x \\ e \end{pmatrix}$.
Por ser SA, $s \begin{pmatrix} x \\ e \end{pmatrix} = s$

iv) Caso $c = c_i\ ;\ c_j$.
Por hipótese de indução temos,

$$HI_i: \quad \forall\ s \in \Sigma, \quad \begin{array}{c} (\ c_i,\ s\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_i,\ s\ ) \\ \Downarrow \\ (\ c_i,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_i,\ s\ ) \end{array}$$

$$HI_j: \quad \forall\ s \in \Sigma, \quad \begin{array}{c} (\ c_j,\ s\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_j,\ s\ ) \\ \Downarrow \\ (\ c_j,\ \textbf{Eval}_{cnf}(\ c_j,\ s\ )\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_j,\ s\ ) \end{array}$$

Supondo que $(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )$.
Queremos então mostrar que $(\ c_i\ ;\ c_j,\ \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )$.

Pela semântica operacional, $\exists\ s'',\ (\ c_i,\ s\ ) \rightsquigarrow s'' \wedge (\ c_j,\ s''\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )$.

Tome-se $s'' = \textbf{Eval}_{cnf}(\ c_i,\ s\ )$, pela HI_j,

$$\begin{array}{c} (\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) \\ \Downarrow \\ (\ c_j,\ \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow \textbf{Eval}_{cnf}(\ c_j,\ \textbf{Eval}_{cnf}(\ c_i,\ s\ )\ ) \end{array}$$

Pela HI_i. □

Lema 4.2.27. *Lema I*

$$\forall s_1, s_2 \in \Sigma,\ \forall c \in \mathcal{C}nf, \quad \begin{array}{c}(c,\ s_1) \rightsquigarrow \text{erro} \wedge (\forall\ x \in FV(\ c\ ) :\ s_1(\ x\ ) = s_2(\ x\ )\ ) \\ \Downarrow \\ (c,\ s_2) \rightsquigarrow \text{erro}\end{array}$$

Demonstração. Por indução em $c \in \mathcal{C}nf$.

Caso **CNF - ASSERT - Error**.
Sabemos então que $c = \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ )$, $(\ c,\ s\ ) \rightsquigarrow \textit{erro}$, $s_1 \models b'$ e $s_1 \not\models\ b$.
Sabemos também por suposição $\forall\ x \in FV(\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ )\ ) :\ s_1(\ x\ ) = s_2(\ x\ )$

Queremos mostrar que $(\ \textbf{if}(\ b'\ )\ \textbf{then assert}(\ b\ ),\ s_2\ ) \rightsquigarrow \textit{erro}$
Pela semântica operacional, isto só ocorre se $s_2 \models b'$ e $s_2 \not\models\ b$
De 1) pela semântica operacional sabemos que $s_1 \models b'$ e $s_1 \not\models\ b$
Por 2)

Caso **CNF - COMPOSITION - Break**.

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow \textit{erro}}{(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow \textit{erro}}\ \textbf{CNF - COMPOSITION - Break} \tag{4.19}$$

Sabemos então que, por suposição
1) $(\ c_i,\ s_1\ ) \rightsquigarrow \textit{erro}$
2) $\forall\ x \in FV(\ c_i\ \textbf{;}\ c_j\ ) :\ s_1(\ x\ ) = s_2(\ x\ )$
Sabemos também por hipótese de indução que,
3) $$\begin{array}{c}(c_i,\ s_1) \rightsquigarrow \textit{erro} \wedge (\forall\ x \in FV(\ c_i\ ) :\ s_1(\ x\ ) = s_2(\ x\ )\ ) \\ \Downarrow \\ (c_i,\ s_2) \rightsquigarrow \textit{erro}\end{array}$$

Queremos mostrar que $(\ c_i\ \textbf{;}\ c_j,\ s_2\ ) \rightsquigarrow \textit{erro}$
Basta mostrar que $(\ c_i,\ s_2\ ) \rightsquigarrow \textit{erro}$.

De 1), 2) e 3).

Caso **CNF - COMPOSITION**.

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow s'' \quad s'' \neq \mathit{erro} \quad (\ c_j,\ s''\ ) \rightsquigarrow s'}{(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow s'}\ \textbf{CNF - COMPOSITION} \tag{4.20}$$

Sabemos então que, por suposição
1) $(\ c_i,\ s_1\ ) \rightsquigarrow s''$, $s'' \neq \mathit{erro}$ e $(\ c_j,\ s''\ ) \rightsquigarrow \mathit{erro}$
2) $\forall\ x \in FV(\ c_i\ ;\ c_j\ ) :\ s_1(\ x\ ) = s_2(\ x\ )$
Sabemos também por hipótese de indução que,

$$HI_i\text{: } \forall\ s_a, s_b \in \Sigma,\ \begin{array}{c}(\ c_i,\ s_a\ ) \rightsquigarrow \mathit{erro} \wedge (\ \forall\ x \in FV(\ c_i\ ) :\ s_a(\ x\ ) = s_b(\ x\ )\ ) \\ \Downarrow \\ (\ c_i,\ s_b\ ) \rightsquigarrow \mathit{erro}\end{array}$$

$$HI_j\text{: } \forall\ s_a, s_b \in \Sigma,\ \begin{array}{c}(\ c_j,\ s_a\ ) \rightsquigarrow \mathit{erro} \wedge (\ \forall\ x \in FV(\ c_j\ ) :\ s_a(\ x\ ) = s_b(\ x\ )\ ) \\ \Downarrow \\ (\ c_j,\ s_b\ ) \rightsquigarrow \mathit{erro}\end{array}$$

Queremos mostrar que $(\ c_i\ ;\ c_j,\ s_2\ ) \rightsquigarrow \mathit{erro}$
Pela semântica operacional, basta mostrar que,
$(\ c_i,\ s_2\ ) \rightsquigarrow s''$ $\qquad$ $s'' \neq \mathit{erro}$ $\qquad$ $(\ c_j,\ s''\ ) \rightsquigarrow \mathit{erro}$

□

Lema 4.2.28. *Lema II*

$$\forall s_1, s_2 \in \Sigma,\ \forall c \in \mathcal{C}nf, \quad \begin{array}{c}(\forall\ x \in FV(\ c\ ):\ s_1(\ x\ ) = s_2(\ x\ )\ ) \\ \Downarrow \\ (\forall\ x \in FV(\ c\ ):\ \boldsymbol{Eval}_{cnf}(\ c,\ s_1\ )(\ x\ ) = \boldsymbol{Eval}_{cnf}(\ c,\ s_2\ )(\ x\ )\ )\end{array}$$

Demonstração. Por indução em $c \in \mathcal{C}nf$.

Seja $\mathcal{X}$ o seguinte conjunto de comandos $s_e t$**if**$(\ b'\ )$ **then skip**, **if**$(\ b'\ )$ **then assert**$(\ b\ )s_u bseteq\mathcal{C}nf$, onde $c \in \mathcal{X}$,

por definição, para qualquer $s \in \Sigma$, $\mathbf{Eval}_{cnf}(\ c,\ s\ ) = s$.

Tomando $s_1, s_2 \in \Sigma$ pela hipótese é imediato que $s_1(\ x\ ) = s_2(\ x\ )$.

Caso $c =$ **if**$(\ b'\ )$ **then** $x\ :=\ e$,

Caso $s_1 \not\models b'$, por hipótese $s_2 \not\models b'$ e vice versa pois $\forall\ x \in FV(\ c\ ),\ FV(\ b'\ )s_u bseteqFV(\ c\ )$.

Por definição, $\mathbf{Eval}_{cnf}(\ c,\ s_1\ ) = s_1$ e $\mathbf{Eval}_{cnf}(\ c,\ s_2\ ) = s_2$.

Segue da hipótese.

Caso $s_1 \models b'$, por hipótese $s_2 \models b'$ e vice versa.

Por definição, $\mathbf{Eval}_{cnf}(\ c,\ s_1\ ) = s_1 \begin{pmatrix} x \\ [\![\ e\ ]\!]_{s_1} \end{pmatrix}$ e $\mathbf{Eval}_{cnf}(\ c,\ s_2\ ) = s_2 \begin{pmatrix} x \\ [\![\ e\ ]\!]_{s_2} \end{pmatrix}$.

Basta mostrar que $[\![\ e\ ]\!]_{s_1} = [\![\ e\ ]\!]_{s_2}$

Pelo lema Como

Caso $c = c_i\ ;\ c_j$,

Suponhamos que $\forall\ x \in FV(\ c_i\ ;\ c_j\ ):\ s_1(\ x\ ) = s_2(\ x\ )$

i.e., $\forall\ x \in FV(\ c_i\ ) \cup FV(\ c_j\ ):\ s_1(\ x\ ) = s_2(\ x\ )$

Queremos mostrar que $\mathbf{Eval}_{cnf}(\ c_i\ ;\ c_j,\ s_1\ )(\ x\ ) = \mathbf{Eval}_{cnf}(\ c_i\ ;\ c_j,\ s_2\ )(\ x\ )$

i.e., $\mathbf{Eval}_{cnf}(\ c_j,\ \mathbf{Eval}_{cnf}(\ c_i,\ s_1\ )\ )(\ x\ ) = \mathbf{Eval}_{cnf}(\ c_j,\ \mathbf{Eval}_{cnf}(\ c_i,\ s_2\ )\ )(\ x\ )$

Por indução obtemos as seguintes hipóteses,

$$HI_i\text{:} \quad \begin{array}{c}(\forall\ x \in FV(\ c_i\ ):\ s_{i1}(\ x\ ) = s_{i2}(\ x\ )\ ) \\ \Downarrow \\ (\forall\ x \in FV(\ c_i\ ):\ \mathbf{Eval}_{cnf}(\ c_i,\ s_{i1}\ )(\ x\ ) = \mathbf{Eval}_{cnf}(\ c_i,\ s_{i2}\ )(\ x\ )\ )\end{array}$$

$$HI_j\text{:} \quad \begin{array}{c}(\forall\ x \in FV(\ c_j\ ):\ s_{j1}(\ x\ ) = s_{j2}(\ x\ )\ ) \\ \Downarrow \\ (\forall\ x \in FV(\ c_j\ ):\ \mathbf{Eval}_{cnf}(\ c_j,\ s_{j1}\ )(\ x\ ) = \mathbf{Eval}_{cnf}(\ c_j,\ s_{j2}\ )(\ x\ )\ )\end{array}$$

Tome-se s_1 e s_2 da suposição, por HI_i, $\mathbf{Eval}_{cnf}(\ c_i,\ s_1\ ) = \mathbf{Eval}_{cnf}(\ c_i,\ s_2\ )$.

Tome-se $\mathbf{Eval}_{cnf}(\ c_i,\ s_1\ )$ e $\mathbf{Eval}_{cnf}(\ c_i,\ s_2\ )$, por HI_j,

$$\mathbf{Eval}_{cnf}(\ c_j,\ \mathbf{Eval}_{cnf}(\ c_i,\ s_1\ )\ ) = \mathbf{Eval}_{cnf}(\ c_j,\ \mathbf{Eval}_{cnf}(\ c_i,\ s_2\ )\ )$$

□

Capítulo 5

Implementação de uma Ferramenta Bounded Model Checking

O desenvolvimento deste projecto envolveu tecnologias de diversas áreas. Foi necessário construir um sistema para reconhecer sequências de símbolos da linguagem válidas (**palavras** e **palavras reservadas**); construir um sistema para reconhecer a estrutura da linguagem e realizar a transição para o sistema Haskell. (**a gramática da linguagem**); construir a ferramenta e um sistema de documentação apropriado.

Embora a transição entre estes sistemas, seja linear, envolvendo transformações lineares até à geração de condições de verificação, e após a geração de condições de verificação, trata-se efectivamente de um projecto complexo. Em torno do nosso objectivo está um sistema similar aos sistemas de compilação de programas.

A implementação de uma ferramenta para a linguagem de programação desenvolvida envolve três (3) fases de desenvolvimento. Denotem-se por front-end, middle-end e back-end.

No front-end está envolvida toda a computação referente à transformação de uma frase válida da linguagem de programação **Simple Imperative Language**, numa representação da estrutura sintáctica, útil para raciocinar em termos de geração de condições de verificação. Assim, é necessário reconhecer os construtores da linguagem, atribuir uma representação simbólica e transformar numa representação normalizada da mesma. Por se basear no formalismo λ-calculus, linguagem de programação pura Haskell é ideal para a formalização e computação desta representação normalizada, a qual iremos relacionar com **Abstract Syntax Tree**'s (**AST**'s).

O cerne do middle-end foca-se na travessia de uma representação normalizada da estrutura sintáctica dos procedimentos. Recorre-se à linguagem Haskell para formalizar uma normalização da estrutura sintáctica de um programa numa árvore. Em Haskell é então representado o algoritmo que efectua a verificação do software baseado em **Bounded Model Checking**.

O back-end do sistema efectua todo o esforço computacional de representação da informação gerada (em LaTeX e SMT-LIB v2) bem como a formatação tipográfica e execução da validação. Lembre-se que o resultado deste projecto envolve a validação da formula $\wedge\mathcal{C} \rightarrow \wedge\mathcal{P}$ referentes a um programa/procedimento.

Nesta parte iremos documentar aspectos envolvidos na implementação de uma ferramenta **Bounded Model Checking**, em particular a ferramenta para a linguagem de programação **Simple Imperative Language** definida no capítulo **??** (pág. **??**) . Aborda-se as ferramentas utilizadas, a instalação da ferramenta em sistemas UNIX, a sua utilização, e a apresentação das decisões tomadas ao longo do projecto.

Este capítulo foca-se nas tecnologias envolvidas no projecto de desenvolvimento, construção e estudo de uma ferramenta de verificação baseada na técnica **Bounded Model Checking** para

verificação de **Software**. Do princípio ao fim iremos passar pelas tecnologias envolvidas no parsing da linguagem de programação definida no Capítulo 2 (pág. 5) , na geração de condições de verificação, representação da informação, formatação e processamento dos resultados.

São tecnologias aplicadas em diferentes áreas de saber, em particular:

Processamento de Linguagens, envolvendo Lex e Yacc para formalizar computacionalmente a linguagem de programação simples **Simple Imperative Language**, e reconhecer frases válidas desta mesma linguagem.

Tipografia, recorrendo ao sistema tipográfico LaTeX 2_ε para formatar e apresentar os resultados obtidos numa forma gráfica apresentável.

Satisfiable Modulo Theories, envolvendo a ferramenta Z3 da **Microsoft Corporation** para a satisfazer as condições de verificação geradas.

Geração de condições de verificação, recorrendo à linguagem Haskell para definir e percorrer um sistema de garfos normalizado.

Scripting, envolvendo Perl e Makefile para realizar a comunicação entre as fases de desenvolvimento, a automação da instalação do sistema, automatização na realização e validação de testes, bem como a comunicação com os sistemas tipográficos da informação.

Após estudar o processo **Bounded Model Checking** para verificação de **Software**, podemos considerar que o processo se divide em 5 partes fundamentais, cinco algoritmos que se traduzem em funções (as funções estudadas), que se resumem a transformações dotadas de propriedades e que por sua vez, mantêm outras propriedades.

Sabendo que apenas um conjunto restrito de técnicas de construção de compiladores, produzem tradutores eficientes de uma variedade de linguagens e máquinas, decidimos construir uma ferramenta **Bounded Model Checking** para verificação de **Software**, com base nos nossos conhecimentos de construção de compiladores.

5.1 Conceitos

De forma abstracta, e reduzindo ao nosso objectivo, construimos um tradutor, a ferramenta `swbmc`.

Um compilador interpreta um programa escrito numa linguagem, **a linguagem fonte**, no nosso caso a linguagem dos comandos, e traduz este programa, para um programa equivalente, escrito numa outra linguagem, a **linguagem ojectivo**.

Como resultado da aplicação de compiladores, a codificação presente na representação do reconhecimento de uma linguagem por parte do compilador, indica se um programa está bem escrito na linguagem fonte, isto é se um método automático consegue extrair significado do programa. Por ser um processo automático, um compilador garante que o resultado está bem escrito, de acordo com as regras de transformação presentes no compilador, após percepção.

Existe uma vasta gama de compiladores, tipos de compiladores, tipos de linguagens (fonte e objectivo), e tipos de regras.

Regra geral, dependendo da forma como são construidos, ou o intuíto para o qual foram desenvolvidos, os compiladores são classificados como `single-pass`, `multi-pass`, `load-and-go`, `debugging` ou `optimizers` [2].

A compilação requer duas fases distintas, a fase de análise do programa na linguagem original, e a fase de síntese do programa na linguagem objectivo.

A parte referente à análise da linguagem, divide o programa em elementos e estrutura-os de forma a tomarem significado numa representação intermédia. A parte referente à síntese, constrói um programa na linguagem objectivo de acordo com a representação intermédia que codifica o significado do programa analisado na linguagem fonte, interpretado pelo compilador.

Destas duas partes, a síntese é a parte do compilador que, actualmente, requer as técnicas mais especializadas.

Durante a fase de análise da linguagem, as operações expressas pelo programa são determinadas, interpretadas e registadas numa estrutura hierárquica denominada por `árvore de sintaxe`. Alguns tipos de árvores, e nós usaremos algumas, representam as operações nas raízes e os operandos nas folhas desta `árvore de sintaxe`.

Muito desenvolvimento foi realizado na organização e desenvolvimento na construção de compiladores, desde o seu aparecimento no início da década de 1950. Muito do trabalho inicial foi realizado à custa de experimentação e implementação de conhecimentos empíricos, sendo só aprofundada e desenvolvida a teoria da compilação posteriormente. Ao longo dos anos 50 os compiladores eram considerados como programas de elevada dificuldade de implementação e segundo [Backus et. al 1957] o primeiro compilador da linguagem `FORTRAN` levaria aproximadamente 18 anos de trabalho por parte de uma só pessoa, para ser concretizado.

A análise de um programa subdivide-se me três partes:

> Análise linear, onde o programa é analisado como uma sequência de símbolos, que são agrupados de acordo com o significado que têm ao se associarem em grupo. A Análise linear, é também chamada de `análise léxica` ou `scanning`
>
> A Analise de hierarquia, onde estes agrupamentos são organizados hierarquicamente, de cordo com o significado que têm associado à sua posção relativa. A Análise hierarquica, tambem chamada de `análise sintática` ou `parsing`, usualmente expressa por definições recursivas
>
> E a Análise semântica, onde um grupo de testes são efectuados para certificar que a ordem pela qual estão posicionados, ou e sequência de evocações, produz algum significado semântico, i.e, se a frase/programa produz algum sentido.

Após a análise do código fonte, alguns compiladores, geram uma representação intermédia explícita que deve possuir duas propriedades, ser de fácil construção e de fácil tradução para a linguagem objectivo. Existem diversas formas de representação intermédia.

5.2 Análise léxica - Reconhecimento dos tokens da linguagem

Esta secção foca-se na apresentação das ferramentas Lex e Yacc para reconhecimento de linguagens. Estas ferramentas foram utilizadas para reconhecer e representar os tokens da linguagem de programação, bem como transformar a estrutura sintáctica da linguagem numa estrutura normalizada para efeitos de geração de condições de verificação.

Começemos pela análise léxica onde introduziremos a definição dos lexemas da linguagem definida na secção ??.

Lex é uma ferramenta de geração automática de **analisadores léxicos**. A ferramenta foi originalmente descrita formalmente por Mike Lesk (Figura **??**) e Eric Schmidt (Figura **??**).

Escolheu-se esta ferramenta por ser uma ferramenta já conhecida, sendo leccionada no departamento de Informática desta ilustre instituição no tempo em que esta dissertação foi desenvolvida. Foi escolhido devido à sua facilidade na geração do analisador léxico necessário ao sistema implementado. Por ser uma ferramenta "freeware" disponível em diversas distribuições dos sistemas operativos UNIX. É conhecido como o gerador de analisadores léxicos padrão.

Dada uma sequência de caracteres num ficheiro do sistema operativo, o analisador léxico gera uma sequência de objectos (tokens) representativos dos lexemas presentes nas palavras identificadas no ficheiro, de acordo com as especificações fornecidas à ferramenta Lex.

A ferramenta Lex baseia-se no reconhecimento de tokens a partir de expressões regulares, pelo que é conveniente explicitar os mesmos.

As **expressões regulares**, representam uma notação para definir linguagens regulares. As **expressões regulares** estão intimamente ligadas aos **automatos finitos não determinísticos** e podem ser considerados como uma notação alternativa à pesada notação dos **automatos finitos não determinísticos**. As **expressões regulares**, são estremamente úteis para reconhecer palavras das liguagens regulares. As **expressões regulares**, são capazes de definir todas as linguagens regulares e apenas estas.

As expressões regulares tornam-se assim uma importante ferramenta na identificação de tokens . Em termos de linguagens, um alfabeto **A** é um conjunto não vazio, cujos elementos são letras. Uma palavra **u** sobre um alfabeto **A** é uma sequência (possivelmente vazia) de letras de **A**. A palavra vazia denota-se por ε. O conjunto de todas as palavras de **A** denota-se por $\mathbf{A}^*$.

A **concatenação de palavras** é uma função sobre palavras de um alfabeto **A**

$$(\cdot) : \mathbf{A}^* \to \mathbf{A}^* \to \mathbf{A}^*$$
$$\mathbf{u} \cdot \mathbf{v} = a_1 a_2 ... a_n b_1 b_2 ... b_m$$

onde $\mathbf{u} \in \mathbf{A}^*$ e $\mathbf{u} = a_1 a_2 ... a_n$, para qualquer $n \in \mathbb{N}$, onde $a_i \in \mathbf{A}, 1 \leq i \leq n$, $\mathbf{v} \in \mathbf{A}^*$ e $\mathbf{v} = b_1 b_2 ... b_m$, para qualquer $m \in \mathbb{N}$, onde $b_i \in \mathbf{A}, 1 \leq i \leq m$.

Uma linguagem formal **L** sobre um alfabeto **A** é um subconjunto do conjunto de todas as palavras de **A**, i.e., $\mathbf{L} \subseteq \mathbf{A}^*$.

Seja **A** alfabeto e $\mathbf{L}, \mathbf{L_1}, \mathbf{L_2} \subseteq \mathbf{A}^*$, algumas operações sobre linguagens podem ser expressas por

$$\begin{aligned}
\mathbf{L_1} \cap \mathbf{L_2} &= \{\mathbf{u} \in \mathbf{A}^* : \mathbf{u} \in \mathbf{L_1} \text{ e } \mathbf{u} \in \mathbf{L_2}\} \\
\mathbf{L_1} \cup \mathbf{L_2} &= \{\mathbf{u} \in \mathbf{A}^* : \mathbf{u} \in \mathbf{L_1} \text{ ou } \mathbf{u} \in \mathbf{L_2}\} \\
\mathbf{L_1} \backslash \mathbf{L_2} &= \{\mathbf{u} \in \mathbf{A}^* : \mathbf{u} \in \mathbf{L_1} \text{ e } \mathbf{u} \notin \mathbf{L_2}\} \\
\mathbf{L_1} \cdot \mathbf{L_2} &= \{\mathbf{u} \in \mathbf{A}^* : \text{ existem } \mathbf{u_1} \in \mathbf{L_1} \text{ e existem } \mathbf{u_2} \in \mathbf{L_2} : \mathbf{u} = \mathbf{u_1} \cdot \mathbf{u_2}\}
\end{aligned}$$

Seja **A** alfabeto e $\mathbf{A}' = \mathbf{A} \cup \{\underline{+}, \underline{.}, (\_)^+, (\_)^*, \underline{\emptyset}, \underline{\varepsilon}, (,)\}$, o conjunto das expressões regulares sobre o alfabeto **A** ($\mathbf{RExp_A} \subseteq \mathbf{A}'^*$) é definido indutivamente por:

$$
\begin{array}{rll}
\underline{\emptyset} & \in \mathbf{REXP_A} & \\
\underline{\varepsilon} & \in \mathbf{REXP_A} & \\
a & \in \mathbf{REXP_A} & \text{, para qualquer } a \in \mathbf{A} \\
R_1 \underline{+} R_2 & \in \mathbf{REXP_A} & \text{, para qualquer } R_1, R_2 \in \mathbf{REXP_A} \\
R_1 \underline{\cdot} R_2 & \in \mathbf{REXP_A} & \text{, para qualquer } R_1, R_2 \in \mathbf{REXP_A} \\
R^+ & \in \mathbf{REXP_A} & \text{, para qualquer } R \in \mathbf{REXP_A} \\
R^* & \in \mathbf{REXP_A} & \text{, para qualquer } R \in \mathbf{REXP_A}
\end{array}
$$

Por último, a linguagem reconhecida por uma Expressão Regular Seja $\mathbf{R}, \mathbf{R}_1, \mathbf{R}_2$ expressões regulares sobre $\mathbf{A}$, i.e., $\mathbf{R}, \mathbf{R}_1, \mathbf{R}_2 \in \mathbf{REXP_A}$
Seja $\mathbf{A}' = \mathbf{A} \cup \{\underline{+}, \underline{\cdot}, (\_)^+, (\_)^*, \underline{\emptyset}, \underline{\varepsilon}, (,)\}$
A linguagem reconhecida por $\mathbf{R}$ é dada pela aplicação definida recursivamente por:

$$
\begin{array}{rll}
\mathcal{L} & : \mathbf{R} \to \mathbf{A}^* & \\
\mathcal{L}(\underline{\emptyset}) & = \emptyset & \\
\mathcal{L}(\underline{\varepsilon}) & = \{\varepsilon\} & \\
\mathcal{L}(a) & = \{a\} & \text{, para qualquer } a \in \mathbf{A} \\
\mathcal{L}(R_1 \underline{+} R_2) & = R_1 \cup R_2 & \text{, para qualquer } R_1, R_2 \in \mathbf{REXP_A} \\
\mathcal{L}(R_1 \underline{\cdot} R_2) & = (\mathcal{L}(R_1))\dot{(}\mathcal{L}(R_2)) & \text{, para qualquer } R_1, R_2 \in \mathbf{REXP_A} \\
\mathcal{L}(R^+) & = (\mathcal{L}(R)) \cdot (\mathcal{L}(R))^* & \text{, para qualquer } R \in \mathbf{REXP_A} \\
\mathcal{L}(R^*) & = (\mathcal{L}(R))^* & \text{, para qualquer } R \in \mathbf{REXP_A}
\end{array}
$$

Seguidamente apresentam-se os tokens da linguagem definida pelas palavras da linguagem. A linguagem de programação é rica em palavras reservadas, o que torna relativamente simples a definição da linguagem em termos de tokens. Algumas **expressões regulares** necessárias são sequências bem definidas de caracteres como é o caso das palavras "proc" para indicar a definição de procedimentos, "call" para indicar a invocação de procedimentos, "skip" representando o comando **skip**, "assert" representando a invocação do comando **assert**(b), ":=" representando o construtor do comando $x := e$, o símbolo ";" representando o construtor do comando c_i **;** c_j, "if" e *else* nessessários à construção dos comandos **if**(b)**then** c_t **else** c_f e **if**(b) **then** c, bem como as palavras reservadas "while" e "do" para representar o comando **while**(b)**do** c_w.

```
1  /* command expression tokens */
2  SKIP         "skip"
3  ASSERT       "assert"
4  ASSIGN       ":="
5  COMPOSITION  ";"
6  IF           "if"
7  ELSE         "else"
8  WHILE        "while"
9  DO           "do"
10 CALL         "call"
11 PROC         "proc"
```

Figura 5.1: Tokens: Comandos

As operações aritméticas sobre inteiros, abordadas pela linguagem, são expressas e idêntificadas pelos seguintes símbolos.

Ilustram-se os tokens referentes a valores e operadores da classe de expressões booleanas. As sequências de caracteres "true" e "false" para representar os valores lógicos $\top$ e $\bot$. O caracter "!" representa a negação lógica $\neg\ b$, "&" representa a conjunção lógica $b_1 \wedge\ b_2$, "—" representa a disjunção lógica $b_1 \vee\ b_2$, e , "-¿" representa a implicação lógica $b_1 \to\ b_2$. Os tokens identificados pelos caracteres "==", "!=", "¡", "¡=", "¿"e "¿=" representam os comparadores $=$, $\neq$, $<$, $\leq$, $>$ e $\geq$ respectivamente.

Outras **expressões regulares** no entanto não podem ser explicitadas como sequências de caracteres (os literais), por tomarem um vasto número de representações, i.e., representações

```
1  /* integer expression tokens */
2  I_Plus        "+"
3  I_Minus       "-"
4  I_Times       "*"
5  I_Div         "div"
6  I_Mod         "mod"
7  /*doc:end:lex_ExpInt*/
```

Figura 5.2: Tokens: Expressões de Inteiros

```
1  /* boolean expression tokens */
2  B_Top         "true"
3  B_Bottom      "false"
4  B_Not         "!"
5  B_And         "&"
6  B_Or          "|"
7  B_Then        "->"
8  B_Eq          "=="
9  B_Neq         "!="
10 B_Lt          "<"
11 B_Le          "<="
12 B_Gt          ">"
13 B_Ge          ">="
```

Figura 5.3: Tokens: Expressões Booleanas

numeráveis não contáveis. Tome-se o exemplo dos números inteiros, e a idêntificação de todas as variáveis do programa, expresso pela definição $?\mathcal{V}_{Integer}?$. A representação destas possíbilidades requer operações sobre as **expressões regulares**.

Assim, a detecção de caracteres em branco, representado pelo token WSPACES é uma classe de elementos, i.e., a classe dos espaços, tabulações, carriage returns, line feeds e newlines. O `token` LETTER representará a classe das letras reconhecidas pelo analisador léxico, abrangendo as letras do alfabeto latino, sem acentuação. Uma letra é um de (2×26) elementos. O `token` DIGIT representará a classe dos números reconhecidas pelo analisador léxico, abrangendo os caracteres 0,1,2,3,4,5,6,7,8 ou 9. Um dígito é um de 10 elementos.

```
1  /* non reservaded words (identifiers, numbers, etc.)*/
2  LETTER        [a-zA-Z]
3  DIGIT         [0-9]
4
5  NUMBER        {DIGIT}+
6  VARIABLE      [a-zA-Z_][a-zA-Z_0-9.]*
7
8  SOMETHING     [a-zA-Z0-9(){}+*\-]*
9  /* white spaces */
10 WSPACES       [ \t\r\f\n]
```

Figura 5.4: Classes simbólicas: White Spaces, letras e digitos

Definidas as classes de caracteres, definiram-se as expressões regulares que reconhecem os números e as variáveis. Um número é uma colecção de digitos com uma formatação apropriada e uma variável é um idêntificador próprio. Assim um número inteiro é representado pelo `token` NUMBER, representando uma sequência finita de 0 ou mais dígitos, e a definição usual de uma variável é representado pelo `token` VARIABLE, represetando um caracter que pode ser uma letra (LETTER) ou um `underscore`, seguido de uma sequência finita de zero ou mais letras,

underscores, digitos ou pontos.

```
1 NUMBER      {DIGIT}+
2 VARIABLE    [a-zA-Z_][a-zA-Z_0-9.]*
```

Figura 5.5: Expressões Regulares: Números e Identificadores de variáveis

Os tokens da Figura 5.5, expressos sobre expressões regulares, exprimem qualquer sequência (possivelmente vazia) de dígitos como um número inteiro, i.e., $NUMBER \in \mathbb{Z}$, e um identificador de uma variável como uma letra ou underscore possivelmente seguido de uma sequência de letras, dígitos, underscores ou pontos.

Ilustra-se o poder das expressões regulares no reconhecimento e representação de palavras.

5.3 Análise hierárquica

Definidos os tokens da linguagem, foi expressa a ordem pela qual os tokens devem ser organizados e estruturado por forma a tomarem significado na linguagem que definimos.

O Yacc é uma ferramenta genérica de formalização da estrutura de programas, vistos como uma frase de uma linguagem. Especificada a estrutura da linguagem é construído um compilador para a linguagem. A ferramenta Yacc codifica uma especificação em rotinas que processão frases de uma linguagem.

Especificada a estrutura da linguagem de comandos, recorremos neste projecto ao Yacc para construir uma **Gramática Tradutora**. Esta tradução permite abstrair os problemas inerentes ao reconhecimento da linguagem, e focar nos problemas centrais deste projecto (geração da condição de validação).

Uma árvore de sintaxe abstracta, **AST** é uma representação em árvore da **estrutura sintáctica abstracta** de um **programa** escrito numa **linguagem de programação**.

O objectivo desta fase centrou-se em mecanizar a representação de qualquer programa na linguagem de comandos em **árvores de sintaxe abstracta** na linguagem Haskell.

Introduzamos algumas noções sobre gramáticas.

Um símbolo terminal de uma linguagem $\mathbf{L}$ representa uma letra da linguagem $\mathbf{L}$. Um símbolo não terminal de uma linguagem $\mathbf{L}$ representa um construtor da linguagem $\mathbf{L}$. Uma gramática $\mathbf{G}$ que define a linguagem formal $\mathbf{L}$ é um tuplo $(\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathbf{S})$, onde:

$\mathcal{N}$ é um **conjunto finito** de **símbolos não terminais**.
$\mathcal{T}$ é o alfabeto da linguagem $\mathbf{L}$, um **conjunto finito** de **símbolos terminais**.
$\mathcal{R}$ é um **conjunto finito** de **produções**.
$\mathbf{S} \in \mathcal{N}$ é o símbolo inicial da gramática.
$\mathcal{N} \cap \mathcal{T} = \emptyset$

Seja $\mathbf{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathbf{S})$ a gramática que define a linguagem formal $\mathbf{L}$, uma produção de $\mathbf{G}$ é a relação $\rightarrow_{\mathbf{G}} \subseteq \mathcal{N} \times (\mathcal{N} \cup \mathcal{T})^*$ que a cada símbolo não terminal de $\mathcal{N}$ associa uma sequência de símbolos em $\mathcal{N} \cup \mathcal{T}$.

Uma sequência é uma função definida no conjunto dos números naturais, que a cada $n \in \mathbb{N}$ associa um elemento $a_n \in \mathbf{A}$

$$\begin{array}{c} \mathbb{N} \rightarrow \mathbf{A} \\ n \mapsto a_n \end{array}$$

Uma sequência é denotada por (a_n) ou alternativamente por $a_1 a_2 a_3 ... a_n ....$ Seja $\mathbf{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathbf{S})$ a gramática que define a linguagem formal $\mathbf{L}$, uma forma frasica de $\mathbf{G}$, $\mathbf{S}$ é definido por:

$\mathbf{S}$ é forma frasica sse $\alpha\beta\gamma$ são formas frasicas.

se $\beta \rightarrow_{\mathbf{G}} \delta \in \mathcal{R}$ então $\alpha\delta\gamma$ é uma forma frasica.

Uma linguagem formal **L** é um conjunto de palavras que possuem **forma frasica** constituída apenas por **símbolos terminais** numa gramática **G**.

Uma gramática diz-se linear à direita se todas as suas produções são da forma $A \rightarrow \alpha B$ ou $A \rightarrow \alpha$, onde $A, B \in \mathcal{N}$ e α é uma sequência de símbolos terminais.

Uma gramática diz-se dependente de contexto se todas as suas produções são da forma $A \rightarrow \alpha$, onde $A \in \mathcal{N}$ e α é uma sequência de símbolos não terminais

Uma gramática diz-se dependente de contexto se todas as suas produções são da forma $\alpha \rightarrow \beta$, onde α e β são sequências de símbolos terminais e não terminais, e o comprimento de $\alpha \leq$ comprimento de β.

Uma gramática diz-se sem restrições se não for **linear à direita**, **independente de contexto** ou **dependente de contexto**.

A Referência IEEE POSIX P1003.2 define as funcionalidades e os requisitos de ambas as ferramentas Lex and Yacc. Desenvolvido por Stephen C. Johnson em 1970 na empresa AT&T para o sistema operativo UNIX, a ferramenta Yacc gera `parsers` para linguagens baseadas em gramáticas escritas numa notação semelhante à BNF. Yacc é uma acrónimo para "Yet Another Compiler Compiler". Esta ferramenta requer um analizador lexico externo, no nosso caso o Lex, para produzir parsers. Os parsers gerados por esta ferramenta são definidos como LALR.

Os parsers LALR ou "Look-Ahead LR" são versões simplificadas do robusto LR parser, inventado por Donald Knuth em 1965. Significa "Left to Right, Rightmost derivation". O parser LR reconhece qualquer linguagem determinista, livre de contexto, em tempo linear finito.Contudo derivações orientadas à direita requerem quantidades epopeicas de memória, pelo que a sua implementação se tornou impraticável, devidos às restrições tecnológicas da altura. Dedicando-se a este problema, em 1969, Frank DeRemer propoz duas versões simplificadas do parser LR na sua tese de doutoramento "Practical Translators for LR(k) languages". Frank DeRemer sugeriu os conhecidos **Look-Ahead LR (LALR)** e o **Simple LR parser (SLR)**, implemenções que requerem um espaço de memória muito mais restrito, à custa da perda de poder de reconhecimento, i.e., a simplificação resulta em parsers com necessidádes reduzidas de memória , mas menor poder de análise. Embora seja um parser simplificado possui poder sufíciente para reconhecer os contrutores da nossa linguagem.

O LALR é mais poderoso que o SLR. Mais tarde em 1977 foram inventadas técnicas de optimização de memória para o **LR parser** mas muito à quem da poderosa alternativa fornecida pelos parseres LALR. Em 1979 foram anunciadas optimizações ao parser LALR por Frank DeRemer e Tom Pennello, aumentando a sua gestão da memória.

A riqueza a nível de tokens permite que a linguagem não seja rígida no sentido estrutural, i.e., é permitido fazer definições em qualquer parte do documento, em contras-te com linguagens como o C que exige uma estrutura documental.

A titulo de exemplo, na Figura 5.6 ilustra-se uma representação simbólica da estrutura dos programas escritos na linguagem C. A ordem das declarações é relevante, sendo boa prática a explicitação dos includes, as definições iniciais, declarações de variáveis e funções por ordem sequencial de invocação.

```
1 <includes>
2 <defines>
3 <variable declarations>
4 <function declarations>
```

Figura 5.6: Representação simbólica da estrutura em C

Como veremos mais à frente, na ferramenta `swbmc`, optamos por cada ficheiro ser constituido por uma sequência deordenada de procedimentos, cabendo ao middle-end a interpretação do

mesmo. Sendo cada procedimento uma peça de software a ferramenta validará um dado procedimento, sendo as dependências calculadas pelo próprio método através da

Esta foi uma opção que foi tomada por se tratar de um caso de estudo e permitir um maior grau de liberdade na programação, facilitando a construção pretendida por parte do programador e permitindo generalidade suficiente para possibilitar as modificações que um projecto académico necessitam. Cabe ao programador a implementação das boas práticas de programação nos seus documentos.

Por ser uma linguagem de programação simples, o reconhecimento dos tokens da linguagem torna-se relativamente simples. A linguagem não possui construtores complexos, como é o caso do aninhamento de blocos de código e a detecção da associatividade entre operadores. É maioritariamente definida à custa de palavras reservadas.

O formalismo normalizador Backus–Naur Form, **BNF** é uma técnica de notação para **gramáticas independentes de contexto**. Seja $\mathbf{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathbf{S})$ a gramática que define a linguagem formal **L**, o formalismo **BNF** representa formalmente o conjunto das produções $\mathcal{R}$ da gramática **G** para a linguagem **L**. Recorrendo a este formalismo, foram definidas as seguintes regras:

```
1  /* integer expression grammar */
2  exp_int
3          : NUMBER
4              {sprintf(buffer,"(Number %s)",$1);          $$=strdup(buffer);}
5
6          | VARIABLE
7              {sprintf(buffer,"(Variable %s)",$1);        $$=strdup(buffer);}
8
9          | VARIABLE '[' exp_int ']'
10             {sprintf(buffer,"(Array %s %s)",$1,$3);     $$=strdup(buffer);}
11
12         | VARIABLE arguments
13             {sprintf(buffer,"(Function %s %s)",$1,$2);  $$=strdup(buffer);}
14
15         | exp_int I_Plus  exp_int
16             {sprintf(buffer,"(I_Plus%s%s)",$1,$3);      $$=strdup(buffer);}
17
18         | exp_int I_Minus exp_int
19             {sprintf(buffer,"(I_Minus%s%s)",$1,$3);     $$=strdup(buffer);}
20
21         | exp_int I_Times exp_int
22             {sprintf(buffer,"(I_Times%s%s)",$1,$3);     $$=strdup(buffer);}
23
24         | exp_int I_Div   exp_int
25             {sprintf(buffer,"(I_Div%s%s)",$1,$3);       $$=strdup(buffer);}
26
27         | exp_int I_Mod   exp_int
28             {sprintf(buffer,"(I_Mod%s%s)",$1,$3);       $$=strdup(buffer);}
29         ;
```

Figura 5.7: Expressões de Inteiros

As produções gramaticais referêntes às expressões de inteiros, Figura **??** permitem idêntificar uma expressão inteira como um número, uma variável, ou as operações aritméticas sobre inteiros em notação infixe. Note-se que permitimos uma variável tome 3 comportamentos distintos. Embora não estudado, a linguagem permite um idêntificador comportar-se como uma variável, `array` ou *função*. Uma variável seguida de dos caracteres '[', uma expressão inteira e o caracter ']', indicando um acesso a uma possição de uma variável, uma variável seguida de uma lista de argumentos separados por ',' onde se pretende representar a aplicação de funções com os seus argumentos. Os argumentos de uma função são sempre explicitados, mesmo quando não há argumentos, os símbolos '(' e ')' expressam a aplicação de uma função. Note-se que poderiamos agrupar as produções referentes às variáveis, numa só produção, tornando os argumentos de funções e os

acessos à memoria das variáveis, opcionais. Contudo isso traria complexidade no tratamento e representação das variáveis na AST transportada para o Haskell. Optamos por

```
1 arguments
2         : '(' exp_int_list ')'
3             {sprintf(buffer,"[%s]",$2);                    $$=strdup(buffer);}
4         | '(' ')'
5             {sprintf(buffer,"[]");                         $$=strdup(buffer);}
6         ;
```

Figura 5.8: Argumentos dos identificadores segundo a filosofia aplicacional (λ-calculus)

(a) permite identificar a invocação a uma posição de um array, dada pela expressão.

(b) e (c) permite identificar a invocação de uma função com uma lista de expressões.

(d) permite identificar uma variável.

```
1 exp_int_list
2         : exp_int ',' exp_int_list
3             {sprintf(buffer,"%s,%s",$1,$3);                $$=strdup(buffer);}
4         | exp_int
5             {sprintf(buffer,"%s",$1);                      $$=strdup(buffer);}
6         ;
```

Figura 5.9: Definição recursiva à direita de uma lista não vazia de expressões inteiras

Representa uma lista não vazia de expressões de inteiros, separados pelo token ";". A lista de expressões inteiras aplica uma definição recursiva à direita.

As produções da gramática, presentes na Figura **??**, reconhecem os tokens B_Top e B_Bottom, a negação por B_Not seguido de uma expressão booleana, os operadores lógicos B_And, B_Or e B_Then em notação infix, e as operações de comparação B_Eq, B_Neq, B_Lt, B_Le, B_Gt e B_Ge também em notação infix.

As produções da gramática relativas aos comandos são semelhantes aos definidos na secção ??. O comando **skip** é reconhecido pelo token SKIP, o comando **assert**(b) é reconhecido pelo token ASSERT, seguido de uma expressão booleana delimitada pelos caracteres '(' e ')'.

O comando $x := e$ é reconhecido pelo posicionamento do token ASSIGN, após uma variável (reconhecida pelo token VARIABLE) e antes de uma expressão inteira. Desta forma relacionamos uma expressão inteira a uma variável, indicando uma atribuição cujo valor semântico é expresso na secção ??

O comando c_i **;** c_j é reconhecido pelo token COMPOSITION, relacionando dois comandos, em notação **infix**. Note-se que não se define a associatividade deste comando.

O comando **if**(b)**then** c_t **else** c_f é reconhecido pelo token IF seguido de uma expressão booleana, um comando que pretendemos dar significado, e a possível construção idêntificada pelo token ELSE. Note-se que caso não seja detectado a explicitação do ramo ELSE, este é implicitamnente constuido na **AST** com o comando **skip**, conforme ?if:then?.

O comando **while**(b)**do** c_w é reconhecido pelo token WHILE, seguido de uma expressão booleana, o token DO e um comando.

As frases desta linguagem, expressa por esta gramática, exigem a explicitação de blocos gramaticais, a fim de tornar o programa perceptivel ao programador, tentando tornar uma linguagem mais próxima da linguagem natural.

```
1  /* boolean expression grammar */
2  exp_bool
3          : B_Top
4              {sprintf(buffer,"(B_Top)");                  $$=strdup(buffer);}
5
6          | B_Bottom
7              {sprintf(buffer,"(B_Bottom)");               $$=strdup(buffer);}
8
9          | B_Not exp_bool
10             {sprintf(buffer,"(B_Not%s)"    , $2);        $$=strdup(buffer);}
11
12         | exp_bool B_And  exp_bool
13             {sprintf(buffer,"(B_And%s%s)"  , $1,$3);     $$=strdup(buffer);}
14
15         | exp_bool B_Or   exp_bool
16             {sprintf(buffer,"(B_Or%s%s)"   , $1,$3);     $$=strdup(buffer);}
17
18         | exp_bool B_Then exp_bool
19             {sprintf(buffer,"(B_Then%s%s)", $1,$3);      $$=strdup(buffer);}
20
21         | exp_int  B_Eq   exp_int
22             {sprintf(buffer,"(B_Eq%s%s)"   , $1,$3);     $$=strdup(buffer);}
23
24         | exp_int  B_Neq  exp_int
25             {sprintf(buffer,"(B_Neq%s%s)"  , $1,$3);     $$=strdup(buffer);}
26
27         | exp_int  B_Lt   exp_int
28             {sprintf(buffer,"(B_Lt%s%s)"   , $1,$3);     $$=strdup(buffer);}
29
30         | exp_int  B_Le   exp_int
31             {sprintf(buffer,"(B_Le%s%s)"   , $1,$3);     $$=strdup(buffer);}
32
33         | exp_int  B_Gt   exp_int
34             {sprintf(buffer,"(B_Gt%s%s)"   , $1,$3);     $$=strdup(buffer);}
35
36         | exp_int  B_Ge   exp_int
37             {sprintf(buffer,"(B_Ge%s%s)"   , $1,$3);     $$=strdup(buffer);}
38         ;
```

Figura 5.10: Expressões Booleanas

```
1  /* commands expression grammar */
2  command
3          : SKIP
4              {sprintf(buffer,"(Skip)");                    $$=strdup(buffer);}
5
6          | ASSERT '(' exp_bool ')'
7              {sprintf(buffer,"(Assert %s)", $3);           $$=strdup(buffer);}
8
9          | VARIABLE ASSIGN exp_int
10             {sprintf(buffer,"(Assign(%s)%s)", $1, $3);    $$=strdup(buffer);}
11
12         | command COMPOSITION command
13             {sprintf(buffer,"(Comp%s%s)", $1, $3);        $$=strdup(buffer);}
14
15         | IF     '(' exp_bool ')'     '{' command '}' if_else
16             {sprintf(buffer,"(If%s%s%s)", $3, $6, $8 ); $$=strdup(buffer);}
17
18         | WHILE '(' exp_bool ')' DO '{' command '}'
19             {sprintf(buffer,"(While%s%s)", $3, $7);       $$=strdup(buffer);}
20
21         | CALL VARIABLE
22             {sprintf(buffer,"(Call %s)", $2);             $$=strdup(buffer);}
23         ;
24
25 if_else
26         : ELSE '{' command '}'                           {$$=$3;}
27         |                                                {$$="(Skip)";}
28         ;
```

Figura 5.11: Comandos

As produções presentes na Figura **??** identificam a estrutura que reconhece um comando da linguagem. **Note-se** que a gramática permite ao programador exprimir construções if-then sem else. Contudo a linguagem de programação simples **Simple Imperative Language** não permite tal construção. A solução a este problema é trivial visto que a construção if-then pode ser reduzida à construção if-then-else considerando o comando skip na construção else.

```
1  /* procedures */
2  procedure
3      : PROC '(' VARIABLE ')' '{' command '}'
4              {sprintf(buffer,"(Proc %s %s)", $3, $6);$$=strdup(buffer);}
5      ;
```

Figura 5.12: Procedimentos

As produções presentes na Figura **??** ilustram um procedimento como uma sequência de tokens, noemadamente a palavra reservada "proc" seguido de uma variável que idêntifica o procedimento, o token "=" (açúcar sintáctico da programação) e o corpo do procedimento, resumido a um comando da linguagem.

Um **procedimento** é idêntificado pelo token PROC seguido por uma variável que idêntifica o procedimento/comando e um comando na linguagem definida. Temos assim uma sequência de comandos num ficheiro, idêntificados como procedimentos.

Finalmente, e para concluir os pontos importantes desta especificação, as produções presentes na Figura **??** ilustram a meta-estrutura que permite dinamizar a estrutura do documento. Assim, a raiz da **AST** é uma construção de declarações, i.e. o documento é seccionado e visto como uma lista de declarações que podem ser procedimentos, predicados, funções, axiomas, pré-condições ou

```
1 /* file root */
2 root
3         : root procedure      {printf("%s\n",$2);}
4         | procedure           {printf("%s\n",$1);}
5         |                     {;}
6         ;
```

Figura 5.13: Raíz da gramática

pós-condições. Após a construção da **AST** cabe ao Vcgen a interpretação da mesma.

Neste projecto, um ficheiro com procedimentos na linguagem definida, é representado por uma AST, cuja raíz pode representar uma sequência vazia, um procedimento, ou uma sequência de procedimentos.

A ferramenta YACC constrói uma representação textual do programa numa **AST**. A estrutura desta **AST** é uma implementada através de uma pequena **DSL** inspirada nas estruturas de dados nativas do **Haskell**, e "faz a ponte entre"/ permite a comunicação dos dois sistemas.

O uso do LEX e YACC, embora dispendioso, permite funcionalidades que seriam praticamente impossíveis de implementar em **Haskell** com os mesmo recursos.

Definição 5.3.1 (Abstract syntax tree)**.** *Abstract syntax tree **(AST)** is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in the sense that it does not represent every detail that appears in the real syntax.*

http://en.wikipedia.org/wiki/Abstract_syntax_tree

As palavras reservadas permitem dinamizar a estrutura do ficheiro. Quando identificadas, providenciam um contexto, restando apenas atribuir a sua semântica.

5.4 Haskell

Temos assim uma AST que representa programas na linguagem dos comandos e normaliza os ficheiros, obtendo a estrutura dos programas a validas. Cabe ao midle-end, onde está codificado o processo **Bounded Model Checking** para verificação de **Software**, extrair um modelo do programa/software e codificá-lo de forma automática numa espressão lógica a validar.

Haskell é uma linguagem de programação. Em particular, é uma linguagem puramente funcional, polimórfica de tipos estáticos. A linguagem foi dedicada a Haskell Brooks Curry, cujo trabalho realizado no campo da Lógica Matemática serve de fundamento às linguagens funcionais. Haskell baseia-se no sistema formal do λ-calculus.

5.4.1 Árvore de síntaxe abstracta dos comandos

De acordo com a especificação sintáctica da linguagem **??**, **??** e **??**, foram formalizadas na linguagem Haskell as estruturas de dados que seguidamente apresentamos. A interligação destas estruturas formaliza a **AST** que é percorrida para gerar as transformações do médodo **Bounded Model Checking** para verificação de **Software** e consequênte condição de validação. Define também a **DSL** que o middle-end aceita como input.

```
1  {-- Structure: ExpInt --}
2  data ExpInt      = Number     Int
3                   | Variable   Int String
4                   | Array      Int String ExpInt
5                   | Function   String [ExpInt]
6                   | Neg        ExpInt
7                   | I_Plus     ExpInt ExpInt
8                   | I_Minus    ExpInt ExpInt
9                   | I_Times    ExpInt ExpInt
10                  | I_Div      ExpInt ExpInt
11                  | I_Mod      ExpInt ExpInt
12     deriving Eq
```

Figura 5.14: Especificação da Classe de Expressões Inteiras

Na árvore de sintaxe definiu-se a representação de 5 (cinco) operações sobre inteiros binárias, uma unária e a representação das entidades. Números e variáveis são representados na sua forma textual, sendo que no âmbito deste projecto não é necessário atribuir significado para o cálculo das expressões.

Na primeira parte deste projecto foi desenvolvido um avaliador de expressões, chegando-se contudo à conclusão que o cálculo de expressões foi desnecessário. Este avaliador de expressões encontra-se apresentado na documentação do código. Conforme apresentado na Figura **??**. **Nota:**

Na árvore de sintaxe definiu-se também a representação de 3 (três) operações binárias sobre booleanos, 6 (seis) operações binárias sobre inteiros, uma operação unária sobre booleanos e a representação de entidades de verdade ($\top$ e $\bot$). Conforme apresentado na Figura **??**. Procura-se assim simbolizar os valores de verdade $\top$, $\bot$; as operações de negação, conjunção e disjunção lógicas; as operações de comparação: **igualdade**, **desigualdade**, **menor ou igual a**, **menor que**, **maior que** e **maior ou igual a**, sobre os inteiros.

```
1  {-- Structure: ExpBool --}
2  data ExpBool     = B_Top
3                   | B_Bottom
4                   | B_Not   ExpBool
5                   | B_And   ExpBool ExpBool
6                   | B_Or    ExpBool ExpBool
7                   | B_Then  ExpBool ExpBool
8                   | B_Eq    ExpInt  ExpInt
9                   | B_Neq   ExpInt  ExpInt
10                  | B_Le    ExpInt  ExpInt
11                  | B_Lt    ExpInt  ExpInt
12                  | B_Gt    ExpInt  ExpInt
13                  | B_Ge    ExpInt  ExpInt
14     deriving Eq
```

Figura 5.15: Especificação da Classe de Expressões Booleanas

A Figura **??** exprime a representação da classe dos Commandos na linguagem Haskell.

Um Skip é uma construção unitária, um Assignment relaciona uma variável a uma expressão inteira, a composição relaciona 2 (dois) comandos da linguagem, um If relaciona 2 (dois) comandos

da linguagem a uma condição booleana, enquanto que um While associa uma condição booleana a um invariante (que deve ser uma asserção) e um comando.

A invocação de procedimentos é representada pelo nodo CALL e respectiva identificação do procedimento.

```
{-- Structure: Command --}
data Command    = Skip
                | Assert ExpBool
                | Assign Int String  ExpInt
                | Comp               Command Command
                | If     ExpBool     Command Command
                | While  ExpBool     Command
                | Call   String
    deriving Eq
```

Figura 5.16: Especificação da Classe dos Comandos

5.4.2 Transformações

```
step0::[Procedure]->Command->Command
step0 l w@(Skip)         = (Skip)
step0 l w@(Assert b)     = (Assert b)
step0 l w@(Assign i v e) = (Assign i v e)
step0 l w@(Comp c1 c2)   = (Comp (step0 l c1) (step0 l c2))
step0 l w@(If b c1 c2)   = (If b (step0 l c1) (step0 l c2))
step0 l w@(While b c)    = (While  b          (step0 l c))
step0 l w@(Call  pName)
        |(length procedures == 1) =
                (step0 l (head procedures))
        |(length procedures < 1) =
                error("[ERROR]: procedure ["++pName++"] NOT found")
        |(length procedures > 1) =
                error("[ERROR]: multiple decalarations of procedure ["++pName++"]")
    where procedures = (getProcedures pName l)
```

Figura 5.17: Transformação 0

```
step1::Int->Command->Command
step1 n w@(Skip)         = (Skip)
step1 n w@(Assert b)     = (Assert b)
step1 n w@(Assign i v e) = (Assign i v e)
step1 n w@(Comp c1 c2)   = (Comp (step1 n c1) (step1 n c2))
step1 n w@(If b c1 c2)   = (If b (step1 n c1) (step1 n c2))
step1 n w@(While b c)    = (uwd (n,n) w)
    where
        uwd (n,0) w@(While b cw) = (Assert (B_Bottom));
        uwd (n,m) w@(While b cw) = (If b (Comp (step1 n cw) (uwd (n,m-1) w)) Skip)
```

Figura 5.18: Transformação 1

A Figura 5.21 (pág. 99) define a construção dos conjuntos $\mathcal{C}$ e $\mathcal{P}$ conforme Definição **??** (pág. **??**)

```
1      where (tb,cmd) = dSAF [] c
2
3  type Memory = [(String,Int)]
4
5  dSAF::Memory->Command->(Memory,Command)
6  dSAF lIn w@(Skip)           = (lIn, (Skip))
7  dSAF lIn w@(Assert bIn)     = (lIn, (Assert bOut ))
8      where
9          bOut = (substituteExp substituteExpBool bIn lIn)    -- for all lIn, update
               values
10
11 dSAF lIn w@(Assign i x eIn) = (lOut, (Assign n x eOut ))
12     where
13         eOut = (substituteExp substituteExpInt eIn lIn);    -- for all lIn, update
               values
14         lOut = update(lIn,x);                               -- apply succ function
15         n    = (get x lOut) {-- get value from update --}
16
17 dSAF lIn w@(Comp c1 c2)     = (lOut, (Comp cmd1 cmd2))
18     where
19         (lAux,cmd1) = (dSAF lIn c1);
20         (lOut,cmd2) = (dSAF lAux c2)
21
22 dSAF lIn w@(If b c1 c2)   = (lIn,
23         (If (boolean)                                       -- updated boolean
24             (foldl (\a b -> (Comp a b)) cmd1 lt)            -- join true
25             (foldl (\a b -> (Comp a b)) cmd2 lf)  )         -- join false
26     )
27     where
28         (boolean)     = (substituteExp substituteExpBool b lIn);
29         (lTrue,cmd1)  = (dSAF lIn c1);
30         (lFalse,cmd2) = (dSAF lIn c2);
31         (lt,lf)       = unzip(compile lTrue lFalse);
32         (lOut)        = (compose lTrue lFalse)            -- builds new table.
               Highest succ, win factor.
33
34 compose::Memory->Memory->Memory
35 compose m1 m2 = aux (m1++m2) []
36     where
37         aux []       acc = acc;
38         aux (h:t)    acc = aux t (aux2 h acc);
39
40         aux2 (x,i) []           = [(x,i)];-- nothing was found in this accumulator
41         aux2 (x,i) ((y,n):t)    |(x==y) = (y,(max i n)+1):t
```

Figura 5.19: Transformação 2

```
1  step3 p@(Assert b)       phi = (CNF_IF (phi) (Assert b))
2  step3 p@(Assign i str e) phi = (CNF_IF (phi) (Assign i str e))
3  step3 p@(Comp ci cj)     phi = (CNF_Comp (step3 ci phi)  (step3 cj phi))
4  step3 p@(If b ct cf)     phi = (CNF_Comp
5          (step3 ct (B_And phi b))
6          (step3 cf (B_And phi (B_Not b)))
7      )
```

Figura 5.20: Transformação 3

```
1     where
2         skim(CNF_IF phi p@(Skip)           ) = [];
3         skim(CNF_IF phi p@(Assert b)       ) = [(Right (B_Then phi b) )]; -- array
              and funcion normalization ignored
4         skim(CNF_IF phi p@(Assign i str e)) = [(Left  (B_Then phi (B_Eq (Variable i
              str) e ) ) )];
5         skim(CNF_Comp   ci cj              ) = (skim ci)++(skim cj)
6
7 setConjunction::[ExpBool]->ExpBool
```

Figura 5.21: Transformação 4

5.4.3 Ferramenta

Compilar a implementação ghc –make Main.hs -o engine.bin resulta numa ferramenta que recebe um e um só *bound* e multiplos ficheiros em estilo DSL. um ficheiro diz-se do estilo DSL se for reconhecido pela instancia de Read implementada (O parser gera este tipo de ficheiros). A ferramenta reconhece 3 argumentos de linha de comandos, -v -b e -f. Qualquer outro tipo de argumento que não seja reconhecido, as opções são disponibilizadas. Conforme Figura 5.22 (pág. 99) .

```
1  User C:Install\Directory\> ./engine.bin --something
2  engine.bin: unrecognized option '--something'
3  Usage: main [OPTION...]
4    -v                        --version                        show version number
5    -b 34                     --bound=34                       BMC unWinding bound
         limit
6    -f /Absolute/Path/file.dsl  --file=/Absolute/Path/file.dsl  input DSL file to
         read
7
8  User C:Install\Directory\> ./engine.bin --bound=4 --file=/Users/jj/Dropbox/gbmc/dsl
       /04.dsl --something
9  engine.bin: unrecognized option '--something'
10 Usage: main [OPTION...]
11   -v                        --version                        show version number
12   -b 34                     --bound=34                       BMC unWinding bound
         limit
13   -f /Absolute/Path/file.dsl  --file=/Absolute/Path/file.dsl  input DSL file to
         read
14
15 User C:Install\Directory\>
```

Figura 5.22: Argumentos reconhecíveis

A opção -v ou –version disponibilisa a informação sobre a implementação que está a ser executada. Várias invocações resultarão no mesmo resultado. Conforme Figura 5.23 (pág. 99) .

```
1 User C:Install\Directory\> ./engine.bin --bound=34 --version --version --version
2 Ferramenta Bounded Model Checking aplicado a software 2013
3 User C:Install\Directory\> engine.bin --version
4 Ferramenta Bounded Model Checking aplicado a software 2013
5 engine.bin: [error] please provide a bound
6 User C:Install\Directory\>
```

Figura 5.23: Ilustração de versão

A ferramenta necessitará e exigirá sempre um *bound*. Mais do que um *bound* gerará um erro pelo que será necessário executar a ferramenta a um dado ficheiro ou conjunto de ficheiros, tantas

vezes quantos os *bounds* necessários. Conforme Figura 5.24 (pág. 100) .

```
User C:Install\Directory\> ./engine.bin
engine.bin: [error] please provide a bound
User C:Install\Directory\> ./engine.bin --bound=4 --bound=5
engine.bin: [error] please provide just one bound
User C:Install\Directory\> ./engine.bin
engine.bin: [error] please provide a bound
User C:Install\Directory\> ./engine.bin --bound=7
User C:Install\Directory\>
```

Figura 5.24: Ilustração de *bounds*

Sem um ficheiro de input, a ferramenta não produz output. É possivel fornecer vários ficheiros de input, como consequência, o método é aplicado a todos os ficheiros fornecidos e é gerado 1 relatório para cada ficheiro.Conforme Figura 5.25 (pág. 100) .

```
User C:Install\Directory\> ./engine.bin --bound=7
User C:Install\Directory\> ./engine.bin --bound=7 --file=./input/file01.dsl
Successfully generated tex file: ./input/file01.dsl.tex
User C:Install\Directory\> ./engine.bin --bound=7 --file=./input/file01.dsl \
                                                  --file=./input/file02.dsl \
                                                  --file=./input/file03.dsl
Successfully generated tex file: ./input/file01.dsl.tex
Successfully generated tex file: ./input/file02.dsl.tex
Successfully generated tex file: ./input/file03.dsl.tex
User C:Install\Directory\>
```

Figura 5.25: Ilustração de input com ficheiros

É um processo iterativo, qualquer erro que suceda durante o processamento (erro de leitura, inexisência de ficheiro) não é tradado e os ficheiros criados/estados intermédios durante o processo não serão eliminados. Conforme Figura 5.26 (pág. 100) .

```
User C:Install\Directory\> ./engine.bin --bound=4   --file=./input/file01.dsl \
                       --file=./input/file02.dsl \
                       --file=./input/fileUnknown.dsl \
                       --file=./input/file04.dsl
engine.bin: ./input/fileUnknown.dsl: openFile: does not exist (No such file or
    directory)
User C:Install\Directory\>
```

Figura 5.26: Ilustração de erros gerados

Conforme expresso na gramática, Fig 5.13 (pág. 95) , cada ficheiro é considerado como uma sequência de procedimentos. A ferramenta swbmc analisa cada ficheiro e para cada procedimento cria um ficheiro ".smt" e introduz essa informaçao no relatório do ficheiro. Assim, um ficheiro com 1 procediemnto resultará num relatório ".tex" e 1 fórmula codificada no standard smt-liv v2 para o procedimento. Um ficheiro com 100 procedimentos resultará num relatório ".tex" e 100 formulas, uma para cada procedimento.

Note-se no entanto que o comando **call**(p) apenas poderá invocar procedimentos definidos dentro do ficheiro invocador. Caso contrário um erro é gerado pois o procedimento não é encontrado, nem está definido, como tal não se sabe qual o significado de aplicar o comando **call**(p). A ferramenta não realiza ligações entre ficheiros.

5.5 Z3

Z3 é um demonstrador de teoremas de alta performance, i.e., o **Z3** é um Satisfiability Modulo Theories (SMT) solver, desenvolvido pela **Microsoft Research**. É um verificador de *satisfiabilidade* automático de lógicas de primeira ordem tipadas. Além de trabalhar com quantificadores, reconhece as seguintes teorias lógicas por defeito:

- "equality over free (aka uninterpreted) function" e "predicate symbols";
- "real and integer arithmetic (with limited support for non-linear arithmetic)",
- "bit-vectors",
- "arrays",
- "tuple/records/enumeration types"
- "algebraic (recursive) data-types".

A ferramenta **Z3** fornece funcionalidades sobre a aritmética real ($\mathbb{R}$)e inteira ($\mathbb{Z}$), bit-arrays de tamanho fixo, arrays extensionais, uninterpreted functions, e quantificadores.

O **Z3** tem integrado um conjunto de ferramentas da **Microsoft Research**. As ferramentas de análise, verificação e teste de programas, incluem: Spec#/Boogie, Pex, Yogi, Vigilante, SLAM, F7, SAGE, VS3, FORMULA, e HAVOC. Além do formato nativo de sintaxe, o Z3 reconhece os formatos SMT-LIB, SMT-LIB *versão 2* e Simplify.

Neste projecto no entanto procura-se validar as condições de verificação geradas pelo Vcgen. Pelas demonstrações apresentadas na secção teórica (Parte I), se as condições de verificação geradas automáticas forem **válidas** os pressupostos apresentados serão também válidos.

Surge-nos assim a dificuldade apresentada pela ferramenta uma vez que os **SMT**'s procuram a existência de uma valoração que satisfaça a formula apresentada.

Contudo a solução deste problema é trivial como bem conhecido na lógica estudada no primeiro ano. Dada uma formula lógica de primeira ordem b' ao procuramos saber a sua validade, por redução ao absurdo (lógica clássica), se não existir uma valoração que satisfaça b', então b' é uma fórmula válida. Assim basta perguntar à ferramenta **SMT** se existe valoração que satisfaça $\neg b'$. Caso haja valoração que satisfaça $\neg b'$, então, por redução ao absurdo, b' não é uma formula válida.

No relatório apresentado, o sistema realiza esta transformação, pedindo ao **SMT** a satisfação da negação das condições de verificação.

Bibliografia

[1] Yael Abarbanel, Ilan Beer, Leonid Gluhovsky, Sharon Keidar, and Yaron Wolfsthal. Focs – automatic generation of simulation checkers from formal specifications. In E.Allen Emerson and AravindaPrasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 538–542. Springer Berlin Heidelberg, 2000. 3

[2] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977. 85

[3] Armin Biere, Alessandro Cimatti, Edmund Melson Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[4] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). *Handbook of Satisfiability*. IOS Press, 2009.

[5] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003. 4

[6] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. 4

[7] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52:74–84, November 2009.

[8] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[9] Edmund Melson Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving.

[10] Edmund Melson Clarke and Ernest Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Lecture Notes in Computer Science*, page 52–71, 1981.

[11] Edmund Melson Clarke, O.Grumberg, and D.Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[12] Harry Foster. Assertion-based verification: Industry myths to realities (invited tutorial). In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 5–10. Springer Berlin Heidelberg, 2008. 3

[13] J. Harrison. Formal verification at intel. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 45–54, 2003. 3

[14] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003. 4

[15] Zerksis D. Umrigar and Vijay Pitchumani. Formal verification of a real-time hardware design. In *Proceedings of the 20th Design Automation Conference*, DAC '83, pages 221–227, Piscataway, NJ, USA, 1983. IEEE Press. 3