Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Linguagem de Especificação Leve Hoare-Separação para Java

Tiago Vieira Correia dos Santos (28023)

Lisboa
(2010)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Linguagem de Especificação Leve Hoare-Separação para Java

Tiago Vieira Correia dos Santos (28023)

Orientador: Prof. Doutor Luís Caires

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

Lisboa
(2010)

Júri:

Doutor Pedro Manuel Corrêa Calvente Barahona
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa (Presidente)

Doutora Maria Antónia Bacelar da Costa Lopes
Faculdade de Ciências, Universidade de Lisboa (Arguente)

Doutor Luís Manuel Marques da Costa Caires
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa (Orientador)

Para os meus pais, por tudo ...

Agradecimentos

Em primeiro lugar gostava de agradecer ao professor Luís Caires pela oportunidade que me deu em realizar a presente dissertação neste tema e pelo seu apoio e orientação ao longo do desenvolvimento desta tese.

Gostava de agradecer aos meus pais por todo o apoio incansável que me deram e continuam a dar ao longo da minha vida.

Por último agradeço também a todos os meus colegas e amigos, não mencionando nomes para não excluir ninguém e ainda a todos os professores que contribuíram ao longo destes anos para a minha formação académica.

Resumo

Esta tese tem como objectivo o desenvolvimento de uma linguagem de especificação leve para Java, e sua integração no processo de compilação. Este trabalho pretende assim, aproximar a verificação efectuada pelos sistemas de tipos de verificações lógicas mais informativas, através do uso de especificações leves em lógica proposicional. O processo de verificação é modular e baseado no cálculo de pré-condições mais fracas de Dijkstra, estendido para a linguagem orientada a objectos Java. Um aspecto distinto da nossa abordagem consiste numa técnica para lidar com *aliasing*, através da separação de propriedades puras de lineares, numa formulação em lógica de separação dual.

Nas últimas décadas, os temas da especificação, verificação e validação de software têm vindo a apresentar um papel muito importante no seu desenvolvimento, uma vez que garantem a sua correcção e ausência de erros de execução de forma estática, reduzindo custos de manutenção e desenvolvimento. O ano passado marcou o quadragésimo aniversário do artigo *An Axiomatic Basis for Computer Programming* de C.A.R. Hoare, que contribuiu para a revolução deste tema.

Recentemente, o uso de métodos formais para verificar propriedades de programas tem assistido a um impulso, com ferramentas e linguagens de programação (e.g. ESC/Java2, JACK, Spec#) que têm um grande poder expressivo e permitem a verificação estática de programas. Contudo, a sua maioria requer a interacção do utilizador e têm linguagens de especificação muito complexas, que são obstáculos à sua utilização.

Por outro lado, as linguagens de especificação leves, apesar de apresentarem menor expressividade, permitem ainda assim raciocinar sobre propriedades interessantes de um sistema, tornando o seu uso apelativo no desenvolvimento de software. Este trabalho pretende assim contribuir com um estudo inicial do uso de especificações leves, para provar a correcção de programas orientados a objectos de uma forma mais simples e intuitiva.

Palavras-chave: Especificações Leves, Análise Estática, Compilador Verificador, Lógica de Hoare, Lógica de Separação, Cálculo de pré-condições mais fracas

Abstract

The aim of this thesis is the development of a lightweight specification language for Java and its integration on the compilation process. This work thus intends to approach the verification carried out by type systems to more informative logical checks, by using lightweight specifications in propositional logic. The verification process is modular and based on Dijkstra weakest precondition calculus, extended to the Java object-oriented language. A distinguished aspect of our approach is a technique to handle aliasing through a separation of pure from linear properties, in a dual separation logic formulation.

Over the past decades, specification, verification and validation of software present a very important role in software development, since they guarantee correctness and the absence of runtime errors statically, reducing maintenance and development costs. Last year marked the fortieth anniversary of C.A.R. Hoare's paper *An Axiomatic Basis for Computer Programming*, that contributed to the revolution of this subject.

The use of formal methods for verifying program properties has witnessed an impulse recently, with tools and programming languages (e.g. ESC/Java2, JACK, Spec#) that have great expressiveness power and allow static verification of programs. However, most of them require user interaction and have very complex specification language, which are obstacles for their use.

Lightweight specification languages, on the other hand, thought presenting less expressiveness, still allow reasoning about interesting properties of a system with less effort, thus making its usage compelling in software development. This work thus intends to contribute with an initial study of lightweight specifications usage to prove correctness of object-oriented programs in a more simple and intuitive way.

Keywords: Lightweight Specifications, Static Analysis, Verifying Compiler, Hoare Logic, Separation Logic, Weakest Precondition Calculus

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

> *"...there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."*
>
> C.A.R. HOARE

O objectivo desta tese consiste no desenvolvimento de uma linguagem de especificação leve e sua integração no processo de compilação da linguagem Java, estendendo as verificações já efectuadas pelo sistema de tipos. Com esta finalidade, criamos uma linguagem de especificação leve baseada numa lógica monádica, que separa propriedades puras de lineares numa formulação em lógica de separação dual. Desenvolvemos ainda a lógica dual subjacente à linguagem de especificação. Estendemos o cálculo de pré-condições mais fracas proposto por Dijkstra para uma linguagem orientada a objectos, neste caso concreto o Java. Por último, desenvolvemos um protótipo que estende o compilador do Java com a linguagem de especificação e o cálculo desenvolvidos, permitindo a verificação formal de programas Java de acordo com a sua especificação. Pretendemos assim contribuir com um estudo inicial de como podem ser usadas as linguagens de especificação leves para raciocinar sobre programas orientados a objectos, provando a sua correcção.

Ao longo das últimas décadas, tem vindo a ser abordado os temas da especificação, verificação e validação de software. Estes apresentam um papel muito importante no ciclo de desenvolvimento de um software, tendo como finalidade a sua construção de uma forma correcta, podendo detectar erros antecipadamente, reduzindo custos de manutenção e de desenvolvimento do referido software. No ano passado, fez quarenta anos da publicação do artigo de C.A.R. Hoare que revolucionou este tema [Hoa69]. Marcando esta data, Hoare escreveu um novo artigo [Hoa09] onde faz uma análise retrospectiva de todo o desenvolvimento que ocorreu nesta área, e põe em evidência o

que poderá acontecer nos próximos anos.

A especificação define todas as características do software a ser implementado, e é um importante meio de comunicação entre os utilizadores e criadores de software. Para definir essas características, as especificações devem ser escritas formalmente, recorrendo a uma linguagem de especificação, sendo verificadas através de técnicas específicas de verificação e validação. Estas técnicas podem ser divididas em duas grandes áreas: análise estática e análise dinâmica.

Uma enorme vantagem da análise estática face à análise dinâmica consiste no facto de os erros poderem ser detectados sem ter de se executar o programa. Sendo assim um dos grandes objectivos, poder integrar estes mecanismos no processo de compilação de uma linguagem de programação, estendendo a análise estática já efectuada em muitos casos, como em linguagens orientadas a objectos tipificadas (e.g. Java), pelos sistemas de tipos, com novas especificações formais e precisas sobre o comportamento de um programa.

1.1 Motivação

O uso de métodos formais para verificar propriedades de programas é uma área de investigação muito antiga, que tem assistido a um novo impulso recentemente. As ferramentas (e.g. ESC/Java2 [FLL+02]) ou linguagens de programação (e.g. Spec# [BLS05]) existentes, que permitem a verificação de um programa de acordo com a sua especificação, têm uma linguagem de especificação bastante complexa, o que frequentemente, é um impedimento para a sua utilização devido ao esforço requerido, não só por parte dos utilizadores, mas também a nível computacional.

É então necessário utilizar linguagens de especificação leves ainda que apresentem um grau de expressividade mais reduzido, mas que no entanto, permitam raciocinar sobre propriedades interessantes e fáceis de especificar, de forma a tornar o uso de especificações nos programas prática comum, no âmbito de desenvolvimento de software.

Esta é a principal motivação desta tese, que consiste no desenvolvimento de uma linguagem de especificação leve para Java e sua integração no processo de compilação. Esta linguagem é baseada numa lógica monádica e a sua integração na linguagem Java, foi implementada recorrendo à ferramenta Polyglot [NCM03].

Como um exemplo motivador e bastante conhecido temos na Figura 1.1, uma classe `Buffer` com os métodos `write`, `read` e `dataSize` e um método que cria e utiliza uma instância desta classe (`use`).

```
1  class Buffer {
2      Buffer(int size);
3      void write(int value);
4      int read();
5      int dataSize();
6  }
7
8  void use() {
9      Buffer b = new Buffer(10);
10     b.write(2);
11     b.write(3);
12     b.read();
13     b.dataSize();
14 }
```

Figura 1.1: Declaração e Uso da Classe Buffer

Quando pretendemos escrever num buffer, este pode não ter espaço que lhe permita efectuar esta escrita. Sendo assim, podemos derivar uma propriedade que deve ser verificada aquando da escrita num buffer, que diz que o buffer não pode estar "cheio", quando o pretendemos fazer. Outra questão diz respeito às leituras efectuadas, pois se o buffer estiver "vazio" não contendo dados, deparamo-nos com a mesma questão. Posto isto, outra propriedade será ter como requisito de uma leitura de um buffer, que este não esteja "vazio". Com os sistemas de tipos tradicionais, este requisito não pode ser garantido, tendo de se recorrer a testes dinâmicos como por exemplo, guardar numa variável a representação do estado do buffer e verificar, de cada uma das vezes que os métodos fossem chamados, se o buffer se encontra num estado correcto.

A Figura 1.2 ilustra uma possível especificação desta situação, com a linguagem de especificação desenvolvida, no que diz respeito aos métodos `write` e `read` do buffer.

```
1  class Buffer {
2      ...
3      void write(int value)
4          requires + !full
5          ensures + !empty;
6      int read()
7          requires + !empty
8          ensures + !full;
9      ...
10 }
```

Figura 1.2: Possível Especificação da Classe Buffer

Muitos outros exemplos podem ser alvo de uma abordagem semelhante como o

caso de um ficheiro, ou de uma sessão FTP, que têm de seguir determinado protocolo. E ainda, outras situações mais simples que passam apenas por garantir que dada variável não seja nula evitando desreferenciações nulas, verificações estas que podem ser especificadas de uma forma simples e com um impacto reduzido no processo de compilação de uma linguagem.

1.2 Resultados Esperados

Esta tese tem como objectivo a criação de uma extensão ao compilador do Java com verificação de especificações leves, criando um compilador verificador, com recurso à ferramenta Polyglot.

Esta extensão irá permitir o uso de asserções na linguagem Java, fornecendo aos programadores uma forma de estes escreverem programas que são provados estarem correctos de acordo com as suas especificações, que podem ser expressas de uma forma menos complexa do que nas ferramentas existentes, não sendo necessária a intervenção do utilizador no processo de verificação.

Esta dissertação, dando início ao estudo do uso de especificações leves para provar a correcção de programas orientados a objectos e ao procurar o equilíbrio entre o poder expressivo e a complexidade das especificações, constitui uma base para uma melhor integração das especificações e verificações formais no desenvolvimento de software. A linguagem de especificação leve desenvolvida, permite ainda assim, raciocinar sobre propriedades interessantes de uma forma bastante intuitiva para o programador, ao usar uma formulação dual que separa objectos com estado, dos objectos imutáveis.

1.3 Contribuições

As contribuições desta dissertação são as seguintes:

- A principal contribuição deste trabalho é o desenvolvimento de uma linguagem de especificação leve para Java, de modo a tornar possível a verificação de programas Java de acordo com a sua especificação de uma forma simples (Secção 3.2).

- Desenvolvemos uma lógica de separação dual na qual se baseia a linguagem de especificação, que permite lidar com *aliasing* através da separação de propriedades puras e lineares (Secção 3.1). Esta formulação é inovadora e espera-se ao utilizar este tipo de separação, que de certa forma é uma característica inerente das linguagens de programação orientadas a objectos, pois podemos diferenciar

objectos com estado (lineares) de objectos imutáveis (puros), tornar as especificações mais intuitivas e simples para os programadores.

- Estendemos ainda o cálculo de pré-condições mais fracas proposto por Dijkstra para uma linguagem orientada a objectos, neste caso concreto o Java (Secção 3.3.1). Apesar de termos desenvolvido este cálculo para uma linguagem específica, julgamos que pode ser facilmente adaptado a qualquer outra linguagem orientada a objectos.

- Desenvolvemos um protótipo para esta linguagem (aprox. 11500 linhas de código) que verifica os programas automaticamente de acordo com a sua especificação, estendendo o sistema de tipos do Java (Capítulo 4).

- Foi efectuada a validação do protótipo desenvolvido recorrendo a um conjunto de exemplos (Capítulo 4).

1.4 Estrutura do Documento

Este documento está estruturado da seguinte forma:

- No Capítulo 2, descrevemos em detalhe o contexto na qual esta tese se insere, apresentando inicialmente, de uma forma resumida, uma visão histórica dos temas da especificação e verificação, destacando a sua importância no desenvolvimento de software. Posteriormente, enfoca-se na caracterização dos métodos formais e técnicas usadas na verificação de programas, culminando com uma análise de várias linguagens e ferramentas existentes que fornecem suporte à verificação de programas.

- No Capítulo 3, apresentamos a linguagem de especificação e lógica dual subjacente que foram desenvolvidas. Apresentamos também, o cálculo de pré-condições mais fracas desenvolvido para uma linguagem orientada a objectos, mais concretamente, para a linguagem Java.

- No Capítulo 4, explicamos a implementação de forma resumida, destacando os pontos de maior importância no desenvolvimento do protótipo, que estende a ferramenta Polyglot com a linguagem de especificação e cálculo desenvolvidos. Posteriormente, apresentamos exemplos validados pelo nosso protótipo e concluímos com uma análise comparativa com as linguagens e ferramentas descritas no Capítulo 2.

- Por último no Capítulo 5, apresentamos as considerações finais do trabalho realizado, culminando com uma análise de trabalho futuro ao desenvolvimento desta tese.

Capítulo 2

Especificação e Verificação Formal de Programas

Neste capítulo apresentamos o domínio em que esta dissertação está inserida. Iniciando com uma breve apresentação histórica dos temas da especificação e verificação formal de programas, onde são salientados os factos mais importantes que contribuíram para o seu aparecimento, explicando a necessidade de métodos formais mais leves, de forma a poder integrá-los totalmente no desenvolvimento de software sem constituir um grande peso para os programadores. Na Secção 2.1, apresentamos os conceitos inerentes a uma especificação formal e como se podem dividir as várias linguagens, de acordo com o modo de especificação de um sistema. Na Secção 2.2, são abordadas e agrupadas as técnicas de verificação existentes. Nas Subsecções 2.2.1, 2.2.2 e 2.2.3, apresentamos, respectivamente, a lógica de Hoare, a lógica de separação, e o cálculo de pré-condições mais fracas (*weakest preconditions*) proposto por Dijkstra, que constituem a base do processo de verificação desenvolvido nesta tese. Por último, na secção Secção 2.3, apresentamos algumas ferramentas e linguagens de programação existentes que fornecem suporte à verificação de programas no mesmo âmbito que o trabalho desenvolvido nesta dissertação.

Grande parte das vezes, os programas são documentados usando linguagem natural. Esta tem um carácter ambíguo em várias situações e pode levar a erros na documentação, não transmitindo a ideia correcta a quem a lê.

O uso de métodos formais é um meio de contornar este problema, pois estes são compostos por regras precisas de interpretação e verificação, permitindo obter uma qualidade de especificação dos programas mais elevada e a detecção antecipada de erros durante o ciclo de desenvolvimento de um software. Para além da sua correcção, estes permitem ainda verificar outras propriedades desejáveis dos programas, como por exemplo, equivalência de programas e sua terminação. O conceito de especificação

e verificação formal já existe há bastante tempo, e pode ser divido em três fases de desenvolvimento, como menciona Cliff Jones [Jon03]: Pré-Hoare, Axiomas de Hoare, e Pós-Hoare.

Este conceito revelou-se de bastante importância, quando nos finais da década de quarenta do século vinte, Alan Turing observou que raciocinar acerca de programas sequenciais tornava-se mais simples ao anotá-los em determinados pontos, com propriedades sobre estados do programa [Tur49].

Outros investigadores na área como Hoare, Floyd e Naur [Hoa69, Flo67, Nau66], nos finais da década de sessenta, propuseram técnicas axiomáticas para provar a consistência entre programas sequenciais e essas propriedades, denominadas de especificações.

Em 1975, Dijkstra propôs o uso de uma técnica formal denominada de cálculo de pré-condições mais fracas (*weakest precondition calculus*) [Dij75] para verificar se um programa era válido de acordo com essas especificações. Posteriormente, foram ainda propostas técnicas para exprimir propriedades específicas de outros programas, como por exemplo, programas concorrentes.

No início deste século Reynolds e O'Hearn [Rey02, OHRY01], estenderam a lógica originalmente proposta por Hoare, para programas com estruturas de dados partilhadas mutáveis, podendo raciocinar sobre programas na presença de múltiplas variáveis que apontam para a mesma célula de memória (*aliasing*). O cálculo de pré-condições mais fracas proposto por Dijkstra foi também estendido para suportar esta nova técnica [Rey02].

Até à actualidade, os temas da especificação e da verificação de programas permaneceram bastante activos, e nos últimos anos têm vindo a ser alvo de grande atenção. O próprio C.A.R. Hoare apresentou em 2003 [Hoa03], o problema da construção de um compilador verificador que garante a correcção de um programa sem ter de o executar, como sendo um dos maiores desafios em ciência da computação. Em 2007, Leino e Schulte [LS07] descreveram a componente de verificação de um compilador verificador para uma linguagem orientada a objectos concorrente. No ano passado, Hoare revisitando o seu passado, marcou os quarenta anos da publicação do seu trabalho [Hoa69] com a escrita de um artigo [Hoa09], onde discute os desenvolvimentos que ocorreram desde essa época, e o que poderá ainda acontecer. Apesar das suas vantagens, o uso de métodos formais, no início do seu aparecimento, não foi bem sucedido e não atingiu aplicações práticas na indústria, tendo proliferado o uso de testes para detectar erros nos programas, mesmo quando ocorreram situações de erros em aplicações críticas, que levaram à perda de vidas humanas e danos materiais.

O fenómeno que instigou o uso de verificações formais, por parte da indústria de

software, foi os ataques causados por *hackers* que levaram a perdas monetárias na ordem de biliões de dólares em cada ataque. Os atacantes exploram vulnerabilidades no código que muitas das vezes os testes não conseguem remover, como por exemplo, corridas (*race conditions*) [Hoa09]. A única forma suportável de detectar estes erros é através da análise automática do programa com recurso a métodos formais.

De forma a integrar estes métodos no ciclo de desenvolvimento de um software, a complexidade dos métodos formais é um factor muito importante. Para terem um grau de expressividade bastante elevado, a sua linguagem de especificação torna-se bastante complicada, voltando novamente a surgir algumas das dificuldades que se encontram presentes na linguagem natural. Casos de estudo reais, como o referido em [MS95], foram um exemplo de tal situação.

Um modo de contornar estes problemas passou por uma abordagem à especificação formal de uma forma mais leve [JJW96, AL99, Jac02]. O uso de especificações leves, apesar de apresentarem um grau de expressividade menor face aos métodos totalmente formais, permite ainda assim, raciocinar sobre propriedades interessantes do sistema que podem ser especificadas de forma mais simples, constituindo um ponto de partida para essa integração.

2.1 Especificações Formais

Uma especificação formal é a expressão, numa linguagem formal a um dado nível de abstracção, de uma colecção de propriedades que devem ser satisfeitas por um dado sistema. Esta definição é bastante geral pois aborda várias noções, dependendo do conceito de sistema, das propriedades de interesse, do nível de abstracção a ser considerado e do tipo de linguagem formal a ser usada.

Uma especificação é considerada formal, se for expressa numa linguagem composta por três componentes [Win90]: sintaxe, semântica e um conjunto de regras que permitem inferir determinada informação a partir da especificação. A sintaxe permite determinar se as frases presentes na especificação se encontram bem formadas de acordo com a sua gramática. No que diz respeito à semântica, esta atribui um significado preciso a essas construções gramaticais.

Quando nos referimos a especificação formal, é comum caracterizar as linguagens de especificação em três grupos [AP98, Win90]: orientadas a modelos (*model-oriented*), às propriedades (*property-oriented*) ou ao comportamento (*state-machine-oriented*).

No que diz respeito às orientadas a modelos, um sistema é especificado em termos de um modelo de estado e de operações sobre esse estado, e é formado usando construções matemáticas como conjuntos e sequências. Normalmente, o comportamento de

tal sistema é especificado em lógica de Hoare, usando pré-condições e pós-condições no estado observado das entidades. Exemplos de linguagens de especificação orientadas a modelos são, VDM, VDM++, Z, Object-Z, Z++, B-Method.

Quanto às orientadas a propriedades, estas podem ser ainda subdivididas em dois grupos: axiomáticas e algébricas. O método axiomático advém dos trabalhos de Hoare [Hoa69] em provar a correcção da implementação de tipos de dados abstractos, onde são usados predicados (pré-condições e pós-condições) em lógica de primeira ordem para especificar as suas operações. Exemplos conhecidos de linguagens que suportam este método são: a linguagem Anna que é a linguagem de especificação formal para a linguagem de programação Ada e também a família Larch. No método algébrico, os tipos de dados e processos são definidos como álgebras, e são usados axiomas equacionais para especificar propriedades do sistema. Exemplos são a família OBJ e também a família Larch.

Larch combina assim, o método axiomático e algébrico em duas camadas [Win90]. Uma primeira é chamada de especificação de interface e é semelhante à do Z e VDM. A segunda diz respeito à parte algébrica. Muitas linguagens de especificação mais recentes, como é o caso do JML (Java Modeling Language) [LBR99] que é uma linguagem de especificação formal para a linguagem de programação Java, basearam-se na aproximação seguida pela família Larch, no que diz respeito às especificações de interface. Já outras, como a linguagem de especificação formal OCL que estende o UML, baseiam-se e enquadram-se nas linguagens de especificação orientadas a modelos.

Por último, nas linguagens de especificação baseadas no comportamento, um sistema é especificado em termos das sequências de estados possíveis, em vez de tipos de dados. Um sistema é caracterizado por uma colecção de processos que podem ser executados por uma máquina abstracta. Exemplos bastante conhecidos do seu uso são as Petri Nets e a modelação através de cálculo de processos (álgebras de processos), como por exemplo, CSP, CCS, π-calculus e Spi-calculus.

2.2 Verificação

Existem várias técnicas de verificação de um programa, no entanto, esta pode ser dividida em duas categorias: análise dinâmica e análise estática. No caso da verificação através de análise dinâmica, é necessário que o programa alvo da análise seja executado com um número suficiente de testes de input, de forma a explorar, o mais possível o seu comportamento. A utilização de técnicas de teste de software, tais como cobertura de código, ajuda a garantir que um subconjunto do conjunto total dos vários comportamentos do programa tenha sido observado.

Quanto à análise estática, esta não envolve a execução do programa e pode ser realizada de forma manual ou automática, verificando formalmente propriedades do programa, como por exemplo: sintaxe correcta, parâmetros correctos nas funções, correcção de tipos, especificações sobre o estado do programa, etc.

Para verificar a correcção de um programa de acordo com a sua especificação de forma automática, existem na generalidade duas técnicas: *model checking* e demonstração de teoremas [CWA+96, Oui08]. Na primeira, as propriedades de um sistema são verificadas, geralmente, através de uma pesquisa exaustiva de todos os estados possíveis nos quais o sistema poderia entrar durante a sua execução, e a construção das fórmulas de especificação é baseada em variações da lógica temporal. A segunda, passa por produzir uma prova formal da correcção do programa, através das suas especificações e um conjunto de axiomas e regras de inferência, e os demonstradores de teoremas são baseados em variações da lógica de Hoare. Apesar de seguirem uma abordagem diferente à verificação, nos últimos anos, tem havido um esforço para as combinar, tentando juntar as suas melhores características.

Sendo o principal objectivo desta dissertação a criação de uma linguagem de especificação leve, sua integração na linguagem Java e posterior verificação do programa de acordo com a especificação em tempo de compilação, o nosso interesse foca-se na análise estática e enquadra-se na técnica de demonstração de teoremas. Na base do processo de verificação encontra-se a lógica de Hoare, lógica de separação e o cálculo de pré-condições mais fracas proposto por Dijkstra, que são apresentados nas secções seguintes.

2.2.1 Lógica de Hoare

Em [Flo67], Floyd forneceu uma base para a definição formal do significado de programas definidos numa linguagem de programação, estabelecendo uma forma rigorosa para provas sobre programas de computador, incluindo provas de correcção, equivalência e terminação. Como base da sua ideia, são atribuídas asserções às arestas de um fluxograma que representam as passagens de controlo possíveis entre os comandos de um programa, que são consideradas verdade sempre que arestas sejam atingidas durante a execução do programa. Com Floyd surgiu também a ideia de invariante de ciclo. Um invariante de ciclo é uma condição que deve ser verdadeira ao início da sua execução e que é preservado em cada iteração deste. Ou seja, o invariante pode ser quebrado durante a execução do corpo do ciclo, mas tem de ser restabelecido no fim de cada iteração. Sendo assim, à saída de um ciclo, o invariante e a condição de terminação do ciclo são ambas verdade.

P	::=		(Programa)
		skip	(Skip)
	|	$x := E$	(Afectação)
	|	**while** E **do** P	(While)
	|	**if** E **then** P **else** P	(If Else)
	|	**if** E **then** P	(If)
	|	P ; P	(Composição)
E	::=		(Expressões)
		E *bop* E	(Expressões Binárias)
	|	*uop* E	(Expressões Unárias)
	|	(E)	(Expressões entre Parênteses)
	|	x	(Identificador)
	|	n	(Número)
bop	::=		(Operadores Binários)
		+ | - | * | / | %	(Aritméticos)
	|	== | !=	(Igualdade)
	|	> | < | >= | <=	(Relacionais)
	|	&& | ||	(Condicionais)
uop	::=	- | !	(Operadores Unários)

Figura 2.1: Sintaxe SPL (*Simple Programming Language*)

C.A.R. Hoare em [Hoa69], inspirado no trabalho de Floyd, define um sistema formal composto por um conjunto de axiomas e regras de inferência, para raciocinar com rigor sobre a correcção dos programas de computador. Hoare propõe a verificação ao nível das linguagens de programação, ao invés de Floyd que propunha a verificação ao nível do programa. Na sua forma inicial, a lógica de Hoare apenas prova a correcção parcial dos programas e confina-se a uma linguagem de programação muito simples, como a que se encontra ilustrada na Figura 2.1. Posteriormente, a lógica de Hoare foi estendida, de modo a provar a correcção total, e foram também introduzidas novas regras de inferência para novas construções de várias linguagens de programação (e.g. regras para concorrência por Leslie Lamport [Lam80]).

A lógica de Hoare é usada para provar a correcção de programas, no que diz respeito a especificações expressas, como pré-condições e pós-condições definidas sobre a forma de asserções na lógica de predicados, sobre estados.

Um conceito fundamental da lógica de Hoare é o chamado Triplo de Hoare, que descreve como a execução de um pedaço de código muda o estado da computação de um programa.

Definição 2.1 (Triplo de Hoare)**.** Um Triplo de Hoare escreve-se na forma:

$$\{A\}\ P\ \{B\}$$

onde P é um programa (em SPL), e A e B são respectivamente, pré-condições e pós-condições.

$\{true\}$
if $x > y$ **then**
  $z := x$
else
  $z := y$
$\{(z = x \vee z = y) \wedge z \geq x \wedge z \geq y\}$

Figura 2.2: Exemplo de um Programa SPL e Triplo de Hoare

A Figura 2.2 exemplifica um programa escrito em SPL, que calcula o máximo entre dois números que se encontram guardados nas variáveis x e y, e cujo resultado desse máximo é afectado à variável z. A especificação do programa em questão também se encontra representada na figura, sendo esta composta pela pré-condição $true$ e pós-condição $(z = x \vee z = y) \wedge z \geq x \wedge z \geq y$, isto é, o programa quando começa no estado que satisfaz $true$[1], termina num estado em que a variável z contém o valor de x ou de y, nomeadamente, o maior destes dois.

Quando se verificam programas, há dois tipos de correcção:

- Correcção Total – O programa P quando começa num estado que satisfaz A (pré-condição) implica que o programa termina, e atinge sempre um estado que satisfaz B (pós-condição).
- Correcção Parcial – O programa P quando começa num estado que satisfaz A (pré-condição), se terminar, então, atinge um estado que satisfaz B (pós-condição).

Na Figura 2.3, encontram-se as regras de inferência e axiomas definidos por Hoare no seu sistema, que permite provar a correcção parcial de um programa em SPL. Como se pode observar, este é bastante simples, sendo composto por dois axiomas (**skip** e afectação) e quatro regras de inferência.

O primeiro axioma diz respeito ao comando vazio, cujo significado se traduz no facto desse comando não mudar o estado de um programa, ou seja, tudo o que se verifica antes do comando (**skip**) permanece válido após este.

[1] Qualquer estado satisfaz o estado $true$ e nenhum estado satisfaz o estado $false$

[skip] [afectação]

$$\frac{}{\{A\}\ \textbf{skip}\ \{A\}} \qquad \frac{}{\{A[x/E]\}\ x := E\ \{A\}}$$

[iteração - while]

$$\frac{\{I \wedge E\}\ P\ \{I\}}{\{I\}\ \textbf{while}\ E\ \textbf{do}\ P\ \{I \wedge \neg E\}}$$

[condicional - if else]

$$\frac{\{A \wedge E\}\ P\ \{B\} \quad \{A \wedge \neg E\}\ Q\ \{B\}}{\{A\}\ \textbf{if}\ E\ \textbf{then}\ P\ \textbf{else}\ Q\ \{B\}}$$

[composição] [dedução]

$$\frac{\{A\}\ P\ \{B\} \quad \{B\}\ Q\ \{C\}}{\{A\}\ P\ ;\ Q\ \{C\}} \qquad \frac{A' \Rightarrow A \quad \{A\}\ P\ \{B\} \quad B \Rightarrow B'}{\{A'\}\ P\ \{B'\}}$$

Figura 2.3: Sistema de Inferência da Lógica de Hoare

O segundo, axioma da afectação, dita que uma pós-condição A é garantida para um comando $x := E$, se a condição que resulta de substituir E por x em A ($A[x/E]$) é garantida como pré-condição.

No que diz respeito à regra de inferência da composição sequencial dos comandos de um programa, o seu significado traduz-se no facto de, se uma parte do programa (P) culmina num estado que satisfaz uma pós-condição (B) equivalente à pré-condição (B) da segunda parte do programa (Q), então a sua composição ($P; Q$), começando num estado que satisfaz a pré-condição da primeira parte do programa (A), irá terminar num estado que satisfaz a pós-condição da segunda parte do programa (C).

Quanto à regra condicional, esta pode ser explicada da seguinte forma: quando se executa um comando condicional, ou é executado o primeiro ramo (P), ou é executado o segundo (Q). Como tal, para estabelecer a pós-condição do comando condicional (B), tanto P como Q têm de terminar num estado que satisfaz B. Analogamente, podemos verificar que a pré-condição do comando condicional (A) tem de ser também pré-condição de ambos os ramos, P e Q. Finalmente, a escolha entre os dois ramos depende do valor de verdade da condição presente no comando condicional (E). Portanto, se esta for verdade, então o primeiro ramo irá ser escolhido, podendo assim assumir que E também é pré-condição de P, obtendo $\{A \wedge E\}\ P\ \{B\}$, e $\neg E$, pré-condição de Q, resultando em $\{A \wedge \neg E\}\ Q\ \{B\}$.

A regra da iteração usa a noção de invariante de ciclo (denotado como I na regra

de inferência), que é preservado pelas execuções do corpo do ciclo. Sendo assim, considerando que o corpo do ciclo (P) inicia e termina sempre num estado que verifica o invariante (I), então pode-se deduzir que ao fim de várias execuções de P (iterações do ciclo), este continua a ser verdade. Se o ciclo termina, então a condição de controlo tem de ser falsa, logo este facto faz parte da pós-condição do ciclo, $\{I \wedge \neg E\}$. Finalmente, pode-se observar que o corpo do ciclo só executa se essa condição for verdadeira, aparecendo, por isso, como pré-condição da premissa da regra de inferência, $\{I \wedge E\}$. Esta regra condiciona o facto de este sistema permitir apenas provar a correcção parcial de um programa SPL. Tal, deve-se à não garantia de terminação de um programa SPL, porque um ciclo pode não terminar. Para provar a terminação de um ciclo é então necessário assegurar condições auxiliares de modo a garantir o seu progresso.

[iteração (while) - correcção total]

$$\frac{W(<, P) \quad \{I \wedge E \wedge V = n\}\ P\ \{I \wedge V < n\}}{\{I\}\ \textbf{while}\ E\ \textbf{do}\ P\ \{I \wedge \neg E\}}$$

Figura 2.4: Regra da iteração – Correcção Total

Na Figura 2.4, é apresentada a regra que garante a terminação de um ciclo. Para tal, é necessário introduzir a noção de variante de ciclo (denotado de V na regra de inferência), em contraste com a noção de invariante. Sendo assim, é necessário definir um mapeamento entre o espaço de estados e um conjunto bem fundado ($W(<, P)$) e garantir que, de cada vez que o corpo do ciclo é executado, o resultado desse mapeamento diminui estritamente. Como tal só pode acontecer um número finito de vezes, é garantida a terminação do ciclo.

Por último, a regra da dedução permite derivar novos teoremas, a partir de outros teoremas ou regras já provadas. Esta dita que se um programa termina num estado que satisfaz B, então também é verdade que termina num estado que satisfaz qualquer asserção logicamente implicada por B ($B \Rightarrow B'$). Analogamente, se um programa inicia num estado que satisfaz A, também é verdade que o estado inicial satisfaz qualquer asserção que implique A ($A' \Rightarrow A$), obtendo assim a conclusão da regra de inferência $\{A'\}\ P\ \{B'\}$.

$\{true\}$
if $x > y$ **then**
$\{true \wedge x > y\}$ (1)
$z := x$
$\{(z = x \vee z = y) \wedge z \geq x \wedge z \geq y\}$ (2)
else $//\ x <= y$
$\{true \wedge x \leq y\}$ (3)
$z := y$
$\{(z = x \vee z = y) \wedge z \geq x \wedge z \geq y\}$ (4)
$\{(z = x \vee z = y) \wedge z \geq x \wedge z \geq y\}$

Pela regra condicional obtêm-se (1), (2) e (3), (4)

Aplicação do axioma da afectação em $z := x$:
$\{(x = x \vee x = y) \wedge x \geq x \wedge x \geq y\} \Leftrightarrow \{x \geq y\}$ (5)
$z = x$
$\{(z = x \vee z = y) \wedge z \geq x \wedge z \geq y\}$ (2)
$(1) \Rightarrow (5),\ \therefore$ primeiro ramo válido

Aplicação do axioma da afectação em $z := y$:
$\{(y = x \vee y = y) \wedge y \geq x \wedge y \geq y\} \Leftrightarrow \{y \geq x\}$ (3)
$z = y$
$\{(z = x \vee z = y) \wedge z \geq x \wedge z \geq y\}$ (4),
$\therefore$ segundo ramo válido

$\therefore$ Programa válido

Figura 2.5: Verificação Manual de um Programa SPL

Como se pode observar, a Figura 2.5 demonstra a verificação manual do programa que calcula o máximo entre dois números (Figura 2.2), recorrendo às regras de inferência e axiomas propostos por Hoare. Após concluída, verifica-se que, de facto, o programa obedece à sua especificação.

2.2.2 Lógica de Separação

A lógica originalmente proposta por Hoare lida com uma linguagem imperativa composta por valores simples. No entanto, com o evoluir das linguagens de programação surgiu a noção de apontador, podendo múltiplas variáveis referenciar a mesma célula de memória (*aliasing*). Tal situação torna mais difícil raciocinar sobre a correcção de

programas porque, ao alterar uma célula estamos a alterar o valor de muitas expressões que sintacticamente não estão relacionadas. Esta situação não é detectada pela lógica de Hoare, o que leva a erros de correcção de um programa quando ocorrem tais situações de *alisiang*. A Figura 2.6 ilustra um exemplo deste fenómeno (x e y apontam para a mesma célula de memória) e como um programa que na realidade não está correcto de acordo com a sua especificação, não seria detectado ao utilizar o sistema de inferência proposto por Hoare.

$$\overline{\{x = 5 \wedge y = 5\}\ y = y + 1\ \{x = 5 \wedge y = 6\}}$$

Figura 2.6: Falha da Lógica de Hoare na Presença de *aliasing*

Como podemos observar na Figura 2.6, segundo o axioma da afectação o programa encontra-se correcto, uma vez que em SPL não ocorrem variáveis sintacticamente diferentes a referenciar a mesma célula de memória e a linguagem apenas lida com valores simples. No entanto, podemos constatar de forma evidente que quando x e y apontam para a mesma célula de memória, o programa não obedece à sua especificação, porque ao alterar o conteúdo da variável y, estamos também a alterar o da variável x, pois esta aponta para o mesmo.

A lógica de separação, proposta por Reynolds em [Rey02, OHRY01, PB05] é uma extensão à lógica de Hoare orientada a raciocinar sobre estruturas de dados partilhadas mutáveis. A componente chave é o facto de esta permitir um raciocínio local, onde as especificações e as provas se concentram numa porção da memória usada por um componente de um programa, e não no estado global do sistema. Assim sendo, a lógica proposta por Reynolds estende a lógica de Hoare ao adicionar conectivos de separação à linguagem de asserções, permitindo a separação entre duas partes do *heap* de um programa.

A linguagem de programação SPL é também estendida com novos comandos para a manipulação de estruturas de dados mutáveis, como ilustramos na Figura 2.7.

$$\begin{array}{rcll} P & ::= & \ldots & \text{(Programa)} \\ & | & x := \mathbf{cons}(\overline{E}) & \text{(Alocação)} \\ & | & x := [E] & \text{(Leitura)} \\ & | & [E] := E & \text{(Mutação)} \\ & | & \mathbf{dispose}\ E & \text{(Libertação)} \end{array}$$

Figura 2.7: Sintaxe MPL (*Mutable Programming Language*)

Foram assim adicionados comandos que permitem a alocação de células de memória contiguas ($x := \mathbf{cons}(\overline{E})$), inicializadas com as expressões em argumento, colocando em x o endereço da primeira célula desse conjunto. Temos também o comando que permite libertar a célula de memória com o endereço especificado em argumento (**dispose** E). Por último, temos as operações de leitura ($x := [E]$) e mutação do conteúdo de uma célula ($[E] := E$).

A linguagem é ainda estendida de forma a que os estados de um programa contenham duas componentes: uma pilha, que mapeia variáveis em inteiros, tal como na linguagem SPL, e um *heap* que mapeia endereços de memória em valores (representando as estruturas mutáveis). Esses endereços de memória são considerados um subconjunto dos inteiros. As expressões da linguagem dependem apenas da pilha do programa, não contendo notações como **cons** ou $[-]$ que se referem ao *heap*. Temos também, que os comandos que manipulam o *heap* não são instâncias do comando da afectação da linguagem SPL, apesar de serem escritos com o mesmo operador ($:=$) e como tal, não seguem a regra de Hoare da afectação, sendo introduzidas novas regras para tais casos.

Quanto à linguagem de asserções, o cálculo de predicados original é estendido com novas formas para descrever o *heap*:

- **emp** – representa o *heap* vazio;
- $E \mapsto E'$ – o *heap* contém apenas uma célula no endereço E, com o conteúdo E';
- $A_1 * A_2$ – o *heap* pode ser dividido em duas partes disjuntas, de tal forma que a asserção A_1 é garantida para uma parte e a asserção A_2 é garantida para a outra parte;
- $A_1 \mathrel{-\!\!*} A_2$ – se o *heap* corrente for estendido com uma parte disjunta na qual a asserção A_1 se verifica, então a asserção A_2 verifica-se para o *heap* estendido.

São introduzidas ainda abreviaturas para descrever, de uma forma mais compacta, expressões mais complexas nesta linguagem, tais como:

- $E \mapsto - \stackrel{\text{def}}{=} \exists x'.E \mapsto x'$, sendo x' não livre em E;
- $E \hookrightarrow E' \stackrel{\text{def}}{=} E \mapsto E' * \mathbf{true}$;
- $E \mapsto E_1, \ldots, E_n \stackrel{\text{def}}{=} E \mapsto E_1 * \cdots * E + n - 1 \mapsto E_n$
- $E \hookrightarrow E_1, \ldots, E_n \stackrel{\text{def}}{=} E \hookrightarrow E_1 * \cdots * E + n - 1 \hookrightarrow E_n$, se e só se $E \mapsto E_1, \ldots, E_n * \mathbf{true}$.

Na Figura 2.8 encontram-se ilustrados exemplos simples de asserções nesta linguagem, juntamente com uma representação esquemática das células de memória respeitantes a essas asserções. Podemos observar, por exemplo, para o terceiro caso, como seria expressa uma lista ligada com quatro elementos, onde as células contêm uma referência para outra célula (e.g. $x + 1$ contém referência para y).

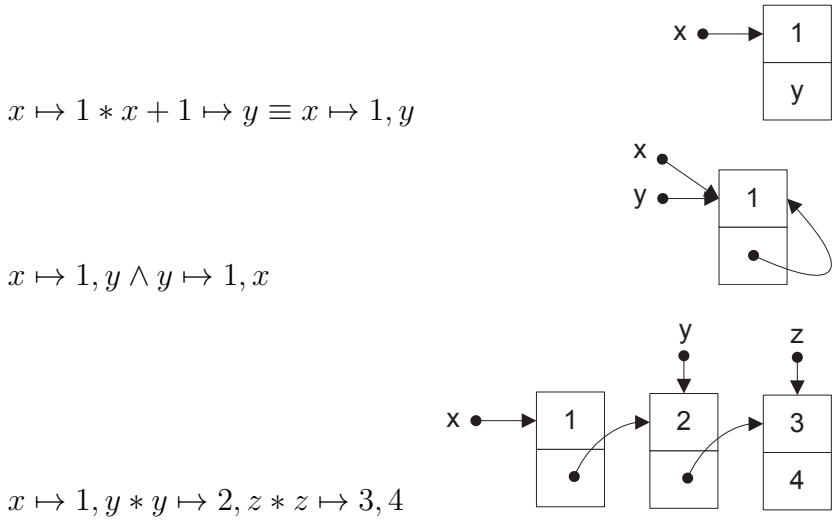

Figura 2.8: Exemplo de Asserções

No que diz respeito às regras de inferência deste sistema, as regras de Hoare permanecem coerentes, bem como as regras estruturais (e.g. regra da dedução). Uma excepção constitui a chamada regra de constância (*rule of constancy*) ilustrada abaixo, que é coerente na lógica de Hoare, mas não na lógica de separação.

[constância]

$$\frac{\{A\}\ P\ \{B\}}{\{A \wedge C\}\ P\ \{B \wedge C\}}$$

onde nenhuma variável que ocorra livre em C é modificada por P. Esta regra é bastante importante, porque permite estender uma especificação local sobre P que envolve apenas variáveis usadas por P com predicados arbitrários sobre variáveis que não são modificadas por este, sendo por isso preservados. Apesar de esta regra não ser coerente, foi introduzida uma nova regra, chamada de regra de *frame* que permite estender uma especificação local, através do uso da conjunção de separação, como se ilustra abaixo.

[frame]

$$\frac{\{A\}\ P\ \{B\}}{\{A * C\}\ P\ \{B * C\}}$$

onde nenhuma variável que ocorra livre em C é modificada por P. Podemos assim, com esta regra, adicionar predicados arbitrários sobre variáveis e partes do *heap* que não são modificadas por P, sendo esta essencial para raciocinar ao nível local acerca do *heap*. Sendo assim, com base na regra de *frame*, podemos passar de raciocínio local de regras de inferência dos comandos que manipulam o *heap* para versões equivalentes ao nível global.

[mutação - local]

$$\frac{}{\{E \mapsto -\}\ [E] := E'\ \{E \mapsto E'\}}$$

[mutação - global]

$$\frac{}{\{(E \mapsto -) * R\}\ [E] := E'\ \{(E \mapsto E') * R\}}$$

Figura 2.9: Regra de Hoare – Mutação

Na Figura 2.9 encontram-se ilustradas as regras de inferência para o comando que permite alterar o conteúdo de uma célula de memória do *heap*. Como podemos observar, esta é composta por duas regras, uma destinando-se ao raciocínio ao nível local e outra ao nível global, sendo ambas as regras bastante intuitivas.

Quanto à mutação local, esta pode ser explicada da seguinte forma: se na pré-condição da execução do comando tivermos um *heap* que contém apenas uma célula de memória com um determinado valor, o qual é irrelevante, pois irá ser alterado, após executar o comando que muda o conteúdo dessa célula de memória, concluímos num estado cuja pós-condição resulta num *heap* com a mesma célula de memória, mas cujo conteúdo foi alterado para o novo valor afectado.

No que diz respeito à versão global, está é uma generalização trivial desta regra, onde é efectuada uma extensão do *heap* com novos predicados sobre variáveis e partes do *heap* que não são modificadas pelo comando (através da regra de *frame*).

[libertação de célula - local]

$$\overline{\{E \mapsto -\}\ \textbf{dispose}\ E\ \{\textbf{emp}\}}$$

[libertação de célula - global]

$$\overline{\{(E \mapsto -) * R\}\ \textbf{dispose}\ E\ \{R\}}$$

Figura 2.10: Regra de Hoare – Libertação de Célula

Quanto ao comando que permite libertar células de memória (Figura 2.10), a sua versão local traduz o facto de que se iniciarmos num estado em que o *heap* é composto apenas por uma célula com um dado conteúdo, após efectuarmos o comando que liberta a referida célula, culminamos num estado em que o *heap* se encontra vazio (**emp**). Quanto à versão global, esta segue uma aproximação idêntica ao caso da mutação.

[alocação si - local] [alocação si - global]

$$\overline{\{\textbf{emp}\}\ v := \textbf{cons}(\overline{E})\ \{v \mapsto \overline{E}\}} \qquad \overline{\{R\}\ v := \textbf{cons}(\overline{E})\ \{(v \mapsto \overline{E}) * R\}}$$

Figura 2.11: Regra de Hoare – Alocação de Células sem Interferência

[alocação genérica - local]

$$\overline{\begin{array}{c}\{v = v' \wedge \textbf{emp}\}\ v := \textbf{cons}(\overline{E})\ \{v \mapsto \overline{E}'\} \\ \text{com } v' \text{ distinto de } v \\ \overline{E}' \stackrel{\text{abv}}{=} \overline{E}/v \to v'\end{array}}$$

[alocação genérica - global]

$$\overline{\begin{array}{c}\{R\}\ v := \textbf{cons}(\overline{E})\ \{\exists v'.(v \mapsto \overline{E}') * R'\} \\ \text{com } v' \text{ distinto de } v \text{ e não} \\ \text{pertence às variáveis livres de } \overline{E} \text{ ou } R \\ \overline{E}' \stackrel{\text{abv}}{=} \overline{E}/v \to v' \\ R' \stackrel{\text{abv}}{=} R/v \to v'\end{array}}$$

Figura 2.12: Regra de Hoare – Alocação de Células Genérica

Para o comando que efectua a alocação de células de memória, nas Figuras 2.11

e 2.12[2] são apresentadas quatro regras distintas, que se diferenciam em dois grupos. Um primeiro (Figura 2.11) respeitante aos casos de raciocínio local e global quando não ocorre interferência, isto é, quando são alocadas variáveis novas. O segundo (Figura 2.12) relaxa esta situação apresentando regras que contemplam a possibilidade de tais interferências, ou seja, poder ocorrer substituição do valor de variáveis existentes ao efectuar uma nova alocação para essa variável.

Quanto às regras no caso de não ocorrência de interferência (Figura 2.11), para a situação local, esta dita que se iniciarmos num estado em que o *heap* está vazio, se ao executar o comando alocarmos um conjunto de células com um determinado valor, então vamos terminar num estado em que o *heap* é composto por tais células com o conteúdo alocado pelo comando. Novamente a versão global usa a regra de *frame* para estender o *heap* com novos factos sobre variáveis ou zonas do *heap* não alteradas pelo comando.

Para as regras no caso de poder ocorrer interferência (Figura 2.12), para a situação global temos que se iniciarmos num estado em que se verifica R e uma vez que a variável v pode ocorrer em R e na expressão $\overline{E}$, ao efectuar uma nova alocação terminamos num estado em que o valor antigo da variável v, denotado por v' é substituído pela nova alocação. Para o caso local, este é um caso particular do caso global, onde o quantificador existencial é removido, passando v' a ser uma variável que não é modificada pela alocação, denotando as suas ocorrências na pós-condição o mesmo valor que na pré-condição.

[`leitura - local`]

$$\frac{}{\{v = v' \wedge (E \mapsto v'')\}\ v := [E]\ \{v = v'' \wedge (E' \mapsto v'')\}}$$
com v, v' e v'' distintos $\qquad E' \stackrel{\text{abv}}{=} E/v \to v'$

[`leitura - global`]

$$\frac{}{\{\exists v''.(E \mapsto v'') * (R/v' \to v)\}\ v := [E]\ \{\exists v'.(E' \mapsto v) * (R/v'' \to v)\}}$$
com v, v', v'' distintos
v', v'' não pertencem às variáveis livres de E
v não pertence às variáveis livres de R
$E' \stackrel{\text{abv}}{=} E/v \to v'$

Figura 2.13: Regra de Hoare – Leitura de Célula

[2] $\overline{E}/v \to v'$ tem como significado a substituição das ocorrências de v por v' para cada expressão em $\overline{E}$. Para $R/v \to v'$ o significado é idêntico

Por último, quanto ao comando de leitura de uma célula de memória resultando na afectação do seu conteúdo a uma variável (Figura 2.13), podemos observar para o caso local que esta é bastante simples. Ao executarmos este comando, se partirmos de um estado em que se verifica que o valor da variável v é v' e a célula de memória da qual estamos a efectuar a leitura (E) contém o valor v'', terminamos num estado em que o valor da variável v é alterado para v'', pois está a ser afectada ao conteúdo da célula E. O caso global resulta da sua generalização pela aplicação da regra de *frame*.

$$
\begin{gathered}
\{\mathbf{emp}\} \\
x := \mathbf{cons}(1,1) \\
\{x \mapsto 1, 1\} \\
y := \mathbf{cons}(2,2) \\
\{x \mapsto 1, 1 * y \mapsto 2, 2\} \\
z := \mathbf{cons}(3,4) \\
\{x \mapsto 1, 1 * y \mapsto 2, 2 * z \mapsto 3, 4\} \\
\{x \mapsto 1, - * y \mapsto 2, - * z \mapsto 3, 4\} \\
[x+1] := y \\
\{x \mapsto 1, y * y \mapsto 2, - * z \mapsto 3, 4\} \\
[y+1] := z \\
\{x \mapsto 1, y * y \mapsto 2, z * z \mapsto 3, 4\}
\end{gathered}
$$

Figura 2.14: Exemplo de Aplicação de Regras da Lógica de Separação

Na Figura 2.14 ilustramos um exemplo de aplicação das regras para o caso da lista ligada composta por quatro elementos, apresentada na Figura 2.8, verificando a sua correcção, ou seja, se iniciarmos com um *heap* vazio e executarmos os comandos terminamos num estado que satisfaz a pós-condição que modela a lista.

2.2.3 Cálculo WP

Para verificar a validade de um triplo de Hoare, é comum usar a técnica de cálculo de pré-condições mais fracas que foi proposta por Dijkstra em [Dij75] e, posteriormente, explicada em mais detalhe no seu livro [Dij76]. Dijkstra estendeu a lógica de Hoare, criando um método para definir a semântica de um programa numa linguagem imperativa ao atribuir a cada comando, na linguagem, um transformador de predicados (*predicate transformer*).

Assumindo que se pretende verificar um programa com uma pré-condição desconhecida, mas sobre o qual sabemos a sua pós-condição, temos então, o seguinte triplo de Hoare:

$$\{?\}\ P\ \{R\}$$

Para atingir um estado que satisfaz a pós-condição R ao executar P, geralmente, existem várias pré-condições possíveis que verificam esta situação. No entanto, existe apenas uma pré-condição (C) que descreve o maior conjunto possível de estados iniciais, de tal forma que a execução de P culmina num estado que satisfaz R, ou seja:

$$\exists C.\forall C' \ \{C'\} \ P \ \{R\} : C' \Rightarrow C$$

A pré-condição C, é chamada de pré-condição mais fraca.

Definição 2.2 (Pré-condição mais Fraca)**.** Sendo o programa denotado por P e a pós-condição por R, então, a pré-condição mais fraca correspondente é representada na forma:

$$wp(P, R)$$

Se o estado inicial satisfaz $wp(P, R)$, então, ao executar P, este atinge um estado que satisfaz R. Se este estado não for satisfeito, então, esta garantia não pode ser dada, ou seja, ou o programa P termina num estado que não satisfaz R, ou nem sequer termina. $wp(P, R)$, é assim visto como uma função, que transforma um predicado R (pós-condição) num outro predicado $wp(P, R)$ (pré-condição), sendo por isso, um transformador de predicados.

Normalmente, quando se pretende verificar um programa, tanto as pré-condições como as pós-condições são conhecidas, como tal, não estamos directamente interessados se o programa satisfaz a pré-condição mais fraca, mas sim, uma pré-condição provavelmente mais forte que esta, que representa um seu subconjunto. Para verificar que essa pré-condição (Q) é de facto um subconjunto da pré-condição mais fraca, tem de se provar que, em qualquer estado, se a pré-condição Q é verdade, então $wp(P, R)$ também é verdade, ou seja:

$$Q \Rightarrow wp(P, R), \text{ para todos os estados}$$

Podem ser derivadas algumas propriedades do predicado wp, que são fundamentais para a sua compreensão, e estão na base da definição das regras wp, para as construções de uma linguagem de programação.

De entre estas propriedades destacam-se as seguintes [Dij76]:

Lei do Milagre Excluído:

$$wp(P, F) = F,$$

onde F é um predicado que representa os estados de um programa que satisfazem a proposição falso.

Distributividade da Conjunção:

$$wp(P, Q) \wedge wp(P, R) = wp(P, Q \wedge R)$$

Lei da Monotonia:

$$\text{Se } Q \Rightarrow R \text{ então } wp(P, Q) \Rightarrow wp(P, R)$$

Distributividade da Disjunção:

$$wp(P, Q) \vee wp(P, R) \Rightarrow wp(P, Q \vee R)$$

Distributividade da Disjunção (P determinista):

$$wp(P, Q) \vee wp(P, R) = wp(P, Q \vee R)$$

De notar a diferença no caso da propriedade distributiva da disjunção, que é apresentada em duas propriedades que se distinguem pelo facto de P ser determinista ou não. Um programa diz-se não determinista, se a sua execução não é sempre a mesma quando este inicia sempre no mesmo estado, ou seja, produz resultados distintos, ou segue um traço de execução diferente.

[skip]

$$wp\left(\textbf{skip}, R\right) = R$$

[afectação]

$$wp\left(x := E, R\right) = R[x/E]$$

[iteração rec - while]

$$\frac{H_0 = \neg E \wedge R \qquad H_k = H_0 \vee wp\left(P, H_{k-1}\right), k > 0}{wp\left(\textbf{while } E \textbf{ do } P, R\right) = \exists k \geq 0 : H_k}$$

[condicional - if else]

$$wp\left(\textbf{if } E \textbf{ then } P_1 \textbf{ else } P_2, R\right) = (E \Rightarrow wp\left(P_1, R\right)) \wedge (\neg E \Rightarrow wp\left(P_2, R\right))$$

[condicional]

$$wp\left(\textbf{if } E \textbf{ then } P, R\right) = (E \Rightarrow wp\left(P, R\right)) \wedge (\neg E \Rightarrow R)$$

[composição]

$$wp\left(P_1 \ ; \ P_2, R\right) = wp\left(P_1, wp\left(P_2, R\right)\right)$$

Figura 2.15: Cálculo WP para SPL

Na Figura 2.15, é apresentado o cálculo wp desenvolvido por Dijkstra [Dij75, Dij76], com as devidas adaptações para a linguagem SPL. De referir que a regra respeitante ao comando condicional só com um ramo (`condicional`) é equivalente ao cálculo da pré-condição mais fraca para o comando condicional com dois ramos (`if else`), com $P_2 = \textbf{skip}$, como é a seguir demonstrado.

$$wp\,(\textbf{if } E \textbf{ then } P, R) \equiv wp\,(\textbf{if } E \textbf{ then } P \textbf{ else skip}, R)$$

$$= (E \Rightarrow wp\,(P,\ R)) \wedge (\neg E \Rightarrow wp\,(\textbf{skip}, R))$$

$$= (E \Rightarrow wp\,(P,\ R)) \wedge (\neg E \Rightarrow R)$$

Como se pode observar, a definição formal do predicado transformador para a regra da iteração é recursiva, sendo difícil a sua utilização. Sendo assim, é comum seguir outra abordagem que recorre à noção de invariante de ciclo explicado na secção anterior para provar que um ciclo é correcto, dado que este termina. Temos então:

[`iteração - while`]

$$\frac{(I \wedge \neg E) \Rightarrow R \quad (I \wedge E) \Rightarrow wp\,(P, I)}{wp\,(I \textbf{ while } E \textbf{ do } P, R) = I}$$

onde I é o invariante do ciclo.

[`alocação`]

$$wp\left(v := \textbf{cons}(\overline{E}), R\right) = \forall v'.(v' \mapsto \overline{E}) \mathrel{-\!\!*} R'$$

v' não pertence às variáveis livres de $\overline{E}$ e R

v' distinto de v

$$R' \stackrel{\text{abv}}{=} R/v \to v'$$

[`libertação de célula`]

$$wp\,(\textbf{dispose } E, R) = (E \mapsto -) * R$$

[`mutação`]

$$wp\,([E] := E', R) = (E \mapsto -) * ((E \mapsto E') \mathrel{-\!\!*} R)$$

[`leitura`]

$$wp\,(v := [E], R) = \exists v'.(E \hookrightarrow v') \wedge R'$$

v' não pertence às variáveis livres de $\overline{E}$ e R a menos que seja igual a v

$$R' \stackrel{\text{abv}}{=} R/v \to v'$$

Figura 2.16: Cálculo WP Estendido para MPL

Este cálculo de pré-condições mais fracas foi também estendido para a lógica de separação [Rey02], sendo adicionados predicados transformadores para os novos comandos da linguagem MPL, como ilustramos na Figura 2.16.

Foram ainda desenvolvidas outras técnicas para a verificação de programas, mais concretamente dirigidas a linguagens orientadas a objectos como o Java, por exemplo, técnicas de verificação baseadas em lógica dinâmica [Bec01, BK07] e lógicas de ordem superior (Higher-order Logic) [HJ00, JP01].

2.3 Linguagens com Suporte à Verificação de Programas

Até à data, foram desenvolvidas várias ferramentas que fornecem suporte à verificação de programas de uma dada linguagem de programação face à sua especificação (e.g. ESC/Java2 [FLL+02], KRAKATOA [MMU04, FM07]), e ainda linguagens que foram desenhadas com suporte para tal, como por exemplo, Gypsy [AGB+77], Euclid [LHL+77], Eiffel [Mey00], Spec# [BLS05], SPARK [Bar03], D [Ale10]. Algumas destas linguagens, como o Eiffel e D, transformam as especificações em código executável e realizam as verificações em tempo de execução. Outras suportam tanto esta técnica, como a verificação formal sem executar o programa (e.g. Spec#).

Apesar da sua existência, é necessário um grande esforço e experiência para usar estes mecanismos, não só devido à sua linguagem de especificação que, apesar de bastante expressiva, é muito complexa, mas também devido ao peso requerido para efectuar essas verificações que recorrem, por exemplo, a demonstradores de teoremas muito complexos que, muitas vezes, precisam da intervenção do utilizador, tornando difícil a sua utilização em ambientes de desenvolvimento de larga escala.

Como tal, e constituindo a principal motivação desta tese, uma forma de tornar o uso de especificações nos programas prática comum, no âmbito de desenvolvimento de software, passa por criar ferramentas menos complexas, integrando-as nas linguagens de programação, ainda que apresentem um grau de expressividade mais reduzido, mas que, no entanto, permitam raciocinar sobre propriedades interessantes e fáceis de especificar. A análise estática já efectuada em muitos casos, como em linguagens orientadas a objectos tipificadas (e.g. Java), pelos sistemas de tipos, é assim estendida com novas especificações formais e precisas sobre o comportamento de um programa, verificando-as em tempo de compilação.

São então apresentadas a seguir algumas das ferramentas e a linguagem de programação Spec# que foram desenvolvidas neste âmbito, explicando de uma forma resumida o seu funcionamento, e destacando as técnicas usadas por cada uma para verificar a correcção de um programa face à sua especificação.

Uma vez que nesta dissertação é realizada uma extensão do Java, uma linguagem orientada a objectos, as ferramentas escolhidas aplicam-se a esta linguagem ou a um seu subconjunto, Java Card, que exclui algumas das funcionalidades do Java, como threads, clonagem de objectos, carregamento dinâmico de classes, e apresenta uma API menor.

2.3.1 Spec#

O Spec# [BLS05, BDF+08, BCD+06] é um exemplo da integração de especificações no desenho de uma linguagem. Spec# estende a linguagem de programação orientada a objectos C#, com suporte para tipos não nulos, permitindo a distinção de referências não nulas de possíveis referências nulas, especificações (e.g. pré-condições, pós-condições, invariantes) e um melhor tratamento do mecanismo de excepções. A sua linguagem de especificação é bastante expressiva, permitindo por exemplo, o uso de quantificadores universais e existenciais, pós-condições excepcionais, referência numa pós-condição ao resultado do método e ao valor de uma expressão na sua pré-condição. Os invariantes de ciclo são inferidos automaticamente [BCD+06] usando interpretação abstracta.

Este apresenta uma noção distinta de quando são mantidos os invariantes da classe. Na aproximação mais comum os invariantes são mantidos no final dos construtores e entre as chamadas dos métodos, ou seja, um invariante tem de se manter antes da chamada do método e após esta, podendo portanto ser quebrado durante a sua execução, desde que no final seja restabelecido. No Spec# não é seguida essa aproximação no que diz respeito aos métodos, isto é, só é permitida a quebra temporária de invariantes no corpo de um método através do uso de um comando específico na linguagem.

Tal como outra linguagem de programação tipificada (e.g. Java), apresenta verificação de tipos, durante o processo de compilação. As verificações das especificações podem ser efectuadas de uma forma dinâmica, ou seja, em tempo de execução, de forma estática, em tempo de compilação, ou combinando os dois métodos.

No que diz respeito à verificação dinâmica, as pré-condições e pós-condições são transformadas em código executável, sendo embutidas no corpo dos procedimentos. Quanto aos invariantes, são criados métodos que os declaram. Após efectuada a compilação, as especificações são preservadas como metadados, juntamente com o código gerado na compilação, permitindo a reutilização das especificações por outras ferramentas.

Quanto à verificação estática, o código compilado é submetido ao verificador de programas do Spec#, Boogie que o traduz numa linguagem intermédia, BoogiePL. O

BoogiePL é uma linguagem imperativa com procedimentos. Todo o código a ser verificado, e inclusive a axiomatização do sistema de tipos, são traduzidos para BoogiePL. Posteriormente, o Boogie gera condições de verificação usando cálculo wp, e submete-as ao demonstrador de teoremas automático, Simplify. São gerados contra-exemplos pelo demonstrador, que são transformados em erros sobre o código e apresentados ao utilizador.

2.3.2 *Extended Static Checker for Java 2 (ESC/Java2)*

O ESC/Java2 [FLL+02] é uma ferramenta que foi construída com o principal objectivo de conseguir detectar erros de programação na linguagem Java 1.4, sem ter de executar o programa (análise estática). Esta foi desenvolvida por David R. Cok e Joseph R. Kiniry e sucede a sua primeira versão ESC/Java, criada no centro de pesquisa da Compaq (SRC).

O tipo de erros detectados pelo ESC/Java2 pode ser dividido em três grupos distintos. Num primeiro grupo, encontram-se as verificações de condições de erro que apenas são efectuadas pela linguagem Java em tempo de execução, como por exemplo, desreferenciações nulas, índices de arrays fora dos seus limites, erros de casting e divisão por zero. Do segundo grupo fazem parte os erros de sincronização, como por exemplo, corridas (*race conditions*) e situações de impasse (*deadlocks*). Por último, num terceiro grupo, estão presentes os que dizem respeito à verificação das especificações do programa, isto é, erros que ocorrem quando um programa não obedece à sua especificação.

As especificações de um programa são escritas sobre a forma de anotações em JML, sendo os principais elementos de uma especificação as pré-condições (anotadas com $@requires\ P$), pós-condições ($@ensures\ Q$) e invariantes das classes ($@invariant\ I$). São ainda suportadas muitas outras anotações, como por exemplo, $@signals$ que especifica uma pós-condição excepcional. As expressões usadas nas pré/pós-condições e invariantes (P, Q e I) têm de ser booleanas e são uma extensão à linguagem de expressões do Java, suportando quantificação universal ($\backslash forall$) e existencial ($\backslash exists$), implicação (==>) e equivalência (<==>) lógica, referência numa pós-condição ao estado de uma expressão no momento anterior à chamada do procedimento alvo da especificação ($\backslash old$) e referência numa pós-condição ao resultado do método ($\backslash result$).

Esta ferramenta suporta herança, ou seja, as especificações são herdadas pelos métodos que são redefinidos nas subclasses de uma dada classe. As pós-condições de um método podem ainda ser fortalecidas, através da adição de novas pós-condições.

ESC/Java2 efectua verificação modular (*modular checking*), para se tornar escalável, verificando um método ou construtor de cada vez, não necessitando de ter o código

fonte de todo o programa para correr. Esta ferramenta não é coerente (*sound*), uma vez que podem ocorrer situações onde o programa tem erros e esta não os detecta, nem completa (*complete*), pois podem ocorrer erros que não o são (falsos positivos).

A verificação de um programa por esta ferramenta é automática, não necessita da intervenção do utilizador, sendo realizada da seguinte forma:

1. Efectua-se a análise sintáctica e verificação de tipos, das anotações e do código Java produzindo árvores de sintaxe abstracta (ASTs - *abstract syntax trees*) e um predicado específico, para cada classe cujas rotinas vão ser verificadas. Este predicado é uma fórmula em lógica de primeira ordem que contém informação dos tipos e das variáveis que as rotinas dessa classe usam.

2. Traduz-se o corpo de cada rotina a ser verificada numa linguagem intermédia baseada na linguagem de comandos guardados (GCs - *guarded commands*), proposta por Dijkstra [Dij75].

3. Geram-se condições de verificação (VCs) para cada GC. Uma condição de verificação para um dado comando guardado é um predicado numa lógica de primeira ordem, que é verdade para os estados do programa onde nenhuma execução desse comando pode errar.

4. As VCs de cada rotina e o predicado específico da classe onde a rotina está definida são submetidos, juntamente com um predicado universal, que contém factos acerca da semântica do Java, a um demonstrador de teoremas automático, Simplify.

5. É processado o output do demonstrador de teoremas, produzindo mensagens informativas que contêm normalmente contra-exemplos, no caso de não conseguir provar as condições de verificação.

2.3.3 JACK

Esta ferramenta [BBC+08], inicialmente desenvolvida no laboratório de pesquisa em Gemplus e posteriormente, pela equipa Everest em INRIA Sophia Antipolis, tal como a anterior, permite a verificação de programas Java anotados em JML. Contudo, esta implementa o cálculo de pré-condições mais fracas, gerando condições de verificação que podem ser consumidas tanto por demonstradores de teoremas interactivos como automáticos, fornecendo diferentes tipos de suporte para os criadores de aplicações. O demonstrador automático usado é o mesmo do ESC/Java2 (Simplify), já o interactivo é o Coq.

Uma grande vantagem desta ferramenta, foi o desenvolvimento de um plugin para o IDE Eclipse, que permite a sua total integração neste, incluindo a possibilidade do uso do demonstrador Coq dentro deste ambiente de desenvolvimento.

O cálculo de pré-condições mais fracas implementado é directo, isto é, trabalha directamente sobre uma AST que representa a aplicação, não efectuando uma tradução, tal como no ESC/Java2, para comandos guardados, sendo usada uma variante do cálculo proposto por Dijkstra adaptado ao Java.

A ferramenta JACK destaca-se ainda pelo facto de permitir, não só a verificação ao nível do código fonte, como também ao nível do código gerado (bytecode). Tal é bastante importante para suportar código certificado (*proof carrying code*), onde as aplicações em bytecode são expedidas em conjunto com a sua especificação e uma prova da sua correcção. O uso de verificação a este nível é também bastante importante, no caso de aplicações críticas que são desenvolvidas ao nível de bytecode, por forma a não dependerem da correcção do compilador.

Para suportar a verificação ao nível de código gerado, foi desenvolvida uma linguagem de especificação BML (Bytecode Modeling Language) e um gerador de condições de verificação para aplicações anotadas em BML, implementando o cálculo wp para o código gerado.

2.3.4 KRAKATOA

Esta ferramenta [MMU04, FM07], desenvolvida por Claude Marché, Christine Paulin e Xavier Urbain (INRIA Futurs and Université Paris-Sud), tal como a anterior, tem como objectivo final a verificação estática de um programa em Java/Java Card, anotado em JML.

A verificação ocorre ao nível do código fonte, apenas são considerados programas sequenciais sem carregamento dinâmico de classes, e só é suportada uma parte das anotações em JML. As anotações consideradas são as que dizem respeito a: invariantes de classe; pré-condições; pós-condições; invariantes de ciclo ($@loop\_invariant$); pós-condições excepcionais; variantes de ciclo ($@decreases$); e a cláusula $@assignable$, que permite especificar as variáveis que podem ser modificadas por um método. Nas asserções, são suportadas, como no ESC/Java2, construções específicas, tais como: quantificação universal e existencial ($\backslash forall$ e $\backslash exists$), $\backslash old$, $\backslash result$.

O KRAKATOA depende da ferramenta Why e do demonstrador de teoremas Coq, no seu processo de verificação. A ferramenta Why tem como objectivo a produção de provas (*proof obligations*) para a certificação de programas, e assenta num método baseado em interpretação funcional e cálculo wp, gerando, tal como no ESC/Java2, condições de verificação. Esta apresenta suporte para vários formatos de output consoante

o demonstrador de teoremas a usar, neste caso é usado o output para o demonstrador Coq. Estas ferramentas têm de ser usadas em separado pelo utilizador, e é necessária a sua intervenção ao longo do processo de verificação.

O processo de verificação passa então pela ferramenta KRAKATOA traduzir um programa anotado em JML para:

- Um programa Why equivalente;
- Um conjunto de ficheiros de especificação em Why;
- Um conjunto de ficheiros Coq que contêm definições de predicados e funções que correspondem aos métodos puros e às invariantes das classes;
- A instanciação de um modelo genérico Coq que contém informação sobre os tipos de valores da linguagem Java (primitivos e referências) e o tipo da memória *heap*.

Posteriormente, os ficheiros Why são submetidos à ferramenta Why, cujo output é submetido, juntamente com os outros ficheiros Coq, ao demonstrador de teoremas Coq de forma a este provar se o programa é valido de acordo com a sua especificação.

2.3.5 LOOP

A ferramenta LOOP (Logic Object-Oriented Programs) [JP04], desenvolvida na Universidade de Nijmegen, tal como a anterior, traduz o código Java anotado em JML num conjunto de ficheiros que são, posteriormente, submetidos a um demonstrador de teoremas, neste caso concreto, o PVS. Esta suporta grande parte da linguagem Java, excepto threads e classes internas.

Inicialmente, o projecto LOOP começou com o propósito de explorar a semântica das linguagens orientadas a objectos no seu âmbito geral, e da linguagem Java em particular. Como ponto de partida, foi definida uma semântica formal denotacional para a linguagem Java numa lógica de ordem superior (Higher-order logic - HOL), embutindo-a no demonstrador de teoremas PVS (*shallow embedding*). Foi também desenvolvido um compilador, implementado em Ocaml, denominado de LOOP, que traduz um programa sequencial escrito em Java, num conjunto de teorias PVS, descrevendo a sua semântica. Posteriormente, a ferramenta evoluiu para dar suporte à especificação de programas Java anotados com JML, tendo então sido definida uma semântica formal para JML em PVS.

A ferramenta LOOP tem como input o código Java especificado através de anotações em JML e produz, como output, ficheiros que descrevem o significado do programa Java e da sua especificação. Estes ficheiros contendo obrigações de prova são,

posteriormente, submetidos ao demonstrador de teoremas PVS, juntamente com arquivos que definem os blocos de construção básicos para a semântica do Java e JML, e um conjunto de teorias PVS e lemas para suportar a verificação do programa. As obrigações de prova são expressas numa extensão à notação de Hoare e são provadas usando lógica de Hoare e cálculo wp para Java e JML, ambas formalizadas em PVS e provadas coerentes.

2.3.6 jStar

jStar [DM08] é uma ferramenta que permite também a verificação de programas escritos em Java. Esta ainda se encontra em desenvolvimento por Dino Distefano e Matthew Parkinson, e constitui apenas um protótipo implementado em Ocaml.

O jStar apresenta um método para a verificação de programas inovador, combinando a ideia de família abstracta de predicados e a ideia de execução simbólica e abstracção, usando lógica de separação [Rey02].

A ferramenta é constituída por um demonstrador de teoremas e um módulo de execução simbólica. O demonstrador de teoremas é usado pelo módulo de execução simbólica durante o processo de verificação, por forma a decidir implicações ou realizar inferência de *frames*. A componente de execução simbólica é responsável pelo cálculo de invariantes usando um algoritmo de ponto fixo. Os estados simbólicos respeitantes à execução simbólica, são expressos em termos de fórmulas em lógica de separação.

À semelhança de outras ferramentas, esta não usa o código Java directamente, ou seja, a ferramenta jStar tem como input programas escritos em Jimple, que é uma representação intermédia do Java, que faz parte da ferramenta Soot. Inicialmente, um programa Java tem de ser submetido à ferramenta Soot que efectua a análise sintáctica de Java traduzindo-o em Jimple, podendo então ser utilizado pelo jStar.

Para além do código Jimple, a ferramenta tem como input ficheiros com a especificação dos métodos, um conjunto de regras lógicas e um conjunto de regras de abstracção, expressas em lógica de separação.

As especificações são descritas numa linguagem de especificação própria e para cada método existem dois tipos de especificação: estática e dinâmica. A estática é utilizada para especificar, de uma forma precisa, o comportamento do código. Já a dinâmica é usada para especificar, de uma forma abstracta, o comportamento do método, permitindo que as subclasses satisfaçam essa especificação, mas possam alterar o comportamento concreto do método. Esta é usada no polimorfismo dos métodos, tendo todas as subclasses de definir essa especificação.

No que diz respeito às regras lógicas, estas são usadas pelo demonstrador de teoremas para decidir implicações. Quanto às regras de abstracção, são uma extensão das regras lógicas e são utilizadas para garantir a convergência do algoritmo de ponto fixo na computação de invariantes de ciclos, inferidos automaticamente.

Esta ferramenta foi desenhada com o intuito de ser bastante geral e flexível, introduzindo um mecanismo para definir novas regras de abstracção que são entendidas pelo demonstrador de teoremas, resultando na possibilidade de utilização de novas técnicas e ideias na verificação de programas. Contudo, os utilizadores necessitam de ter algum conhecimento sobre demonstração de teoremas para desenharem as regras lógicas correctamente, o que constitui um enorme inconveniente, pois torna a ferramenta muito complexa para ser usada por programadores.

2.3.7 KeY

Esta ferramenta [ABB+04] destaca-se das anteriores porque não usa a linguagem JML no processo de especificação, mas sim OCL. Permite aos seus utilizadores realizar especificação e verificação formal como parte do desenvolvimento de software baseado em UML. Esta é construída estendendo a ferramenta Together Control Center, que permite a modelação em UML. A linguagem à qual se destina a ferramenta KeY é a linguagem Java Card.

São então modelados diagramas de classe em UML, aos quais são associadas as especificações formais em OCL, e a sua implementação em Java. Posteriormente, um componente trata da tradução do modelo UML, da especificação e da implementação para obrigações de prova em Java Card Dynamic Logic que são, em seguida, submetidas ao demonstrador KeY Prover, para a sua verificação, automática ou interactiva.

Contrariamente às outras ferramentas, é usada lógica dinâmica que pode ser vista como uma extensão à lógica de Hoare. O mecanismo de dedução em lógica dinâmica é baseado em execução simbólica do programa e transformações deste. Em lógica dinâmica, o conjunto de fórmulas pode conter programas na sua descrição ao invés da lógica de Hoare, onde estas são fórmulas de primeira ordem puras.

2.3.8 Forge

A ferramenta Forge [Den09, DCJ06] efectua uma abordagem à verificação da especificação de um programa escrito em Java, usando uma aproximação diferente das outras ferramentas, através de uma técnica nomeada de verificação limitada (*bounded verification*), que utiliza execução simbólica e reduz o problema a um de satisfação de variáveis booleanas (SAT).

A referida técnica diz-se limitada porque a análise efectuada é controlada pelo utilizador, no sentido de este fornecer certos parâmetros – número de vezes que é desenrolado um ciclo, intervalo dos números inteiros, e o escopo de cada tipo, isto é, um limite do número de instâncias desse tipo que podem existir ao longo da execução – que irão determinar a sua amplitude.

A verificação efectuada é modular, e tal como nas técnicas de *model checking*, a análise é exaustiva, mas apenas nos limites descritos pelo utilizador. Aceita como input procedimentos, a sua especificação e os limites da análise. Os procedimentos e a especificação são expressos numa linguagem intermédia chamada de FIR, que é uma linguagem de programação relacional e também uma linguagem de especificação.

O Forge possui então um front-end para Java, que traduz Java e especificações em JFSL (Java Forge Specification Langue) para FIR. As especificações em JFSL são escritas sobre a forma de anotações Java. Recentemente, foi também desenvolvido um front-end que traduz anotações JML para FIR.

Após a tradução para FIR, são efectuadas transformações nos procedimentos a ser analisados (e.g. desenrolar de ciclos) e são construídas fórmulas em lógica relacional, usando a técnica de execução simbólica, que são verdade quando há um caminho na execução do procedimento em questão, que não satisfaz a especificação. Posteriormente, estas fórmulas e os limites impostos pelo utilizador são submetidos à ferramenta Kodkod, um *model finder* que os traduz num problema de satisfação de variáveis booleanas e invoca um SAT-solver para tentar encontrar a sua solução. Como resultado final, a ferramenta Kodkod traduz esta solução, numa em lógica relacional que é, em seguida, transformada num contra-exemplo permitindo identificar se o programa não está de acordo com a sua especificação.

A análise realizada não é coerente, uma vez que esta é limitada pelo utilizador, não efectuando uma pesquisa exaustiva sobre todas as possibilidades. Portanto, quando não são encontrados contra-exemplos a identificar que o programa não está correcto devido a tal situação, não é garantida a sua correcção.

Capítulo 3

Linguagem de Especificação Leve para Java

Neste capítulo, começando com uma descrição ampla e abrangente da motivação e do trabalho efectuado, apresentamos a linguagem de especificação desenvolvida e a lógica dual (Secção 3.1) que se encontra na base desta linguagem. Na Secção 3.2, abordamos a linguagem de especificação, explicando como esta é constituída e a sua integração num subconjunto da linguagem Java, recorrendo a exemplos ilustrativos da sua utilização. Por último, na Secção 3.3, é identificado em pormenor a extensão do cálculo de pré-condições mais fracas, no qual se baseia a verificação de um programa SpecJava, e o algoritmo de verificação.

Como foi referido no Capítulo 1, o objectivo desta dissertação consiste no desenvolvimento de uma linguagem de especificação leve, integrada na linguagem Java. O intuito final passa por provar a especificação de um programa em tempo de compilação, estendendo os mecanismos de verificação efectuados pelo Java (e.g. verificação de tipos, semântica). Com recurso a especificações leves e a uma lógica proposicional podemos, então, aproximar as verificações realizadas pelo sistema de tipos de verificações lógicas mais informativas. Como resultado, foi implementado um compilador verificador que garante a correcção de um programa, em tempo de compilação, sem ter de o executar.

Um primeiro ponto nesta aproximação, passou por formalizar a linguagem de especificação e extensão da linguagem Java, desenvolvendo a sua sintaxe abstracta. Esta foca-se numa parte da linguagem Java, simplificando alguns dos seus aspectos, como ponto de partida para uma posterior cobertura de outros aspectos desta linguagem que ainda não são suportados (Secção 5.1).

3.1 Lógica Dual Hoare-Separação

Nesta secção, é introduzida a lógica subjacente à linguagem de especificação desenvolvida, nomeada de lógica dual, explicando a sua sintaxe e apresentando definições que podem ser derivadas da sua divisão dual. A definição de uma nova lógica provém da necessidade de ter algum mecanismo de controlo de *aliasing* sobre a nossa linguagem de especificação, uma vez que estamos a estender uma linguagem orientada a objectos podendo ocorrer situações de *aliasing*. A lógica originalmente proposta por Hoare (Secção 2.2.1), foi desenvolvida para uma linguagem de programação imperativa composta apenas por valores simples, onde, portanto, não ocorrem problemas de *aliasing*. Com a nossa abordagem inovadora, tiramos partido das características inerentes das linguagens orientadas a objectos como é o caso do Java, onde podemos diferenciar os objectos em dois grupos, isto é, objectos que alteram o seu estado ao longo da execução do programa (mutáveis) e objectos sem estado ou imutáveis (puros). Como resultado e inspirados na lógica dual intuicionista linear (Dual Intuitionistic Linear Logic) [BP96] onde é efectuada uma divisão de contexto em duas partes (uma linear e outra intuicionista), efectuamos uma formulação dual separando as propriedades de objectos puros e lineares (mutáveis). Sobre os objectos mutáveis, é necessário controlar a ocorrência de *aliasing*, porque este fenómeno pode levar a erros na verificação de um programa. Para os objectos imutáveis, pode ocorrer *aliasing* livremente pois estes nunca mudam de estado, mesmo quando referenciados por múltiplas variáveis. Assim sendo, podemos observar que esta divisão é bastante intuitiva e tem como principal motivação tornar o processo de especificação mais simples para o programador.

Na Figura 3.1 apresentamos a sintaxe da lógica dual. Como o nome sugere, uma fórmula dual é composta por duas partes: a parte esquerda (ϕ), denominada de fórmula pura, é composta por uma fórmula proposicional em lógica clássica e refere propriedades sobre objectos imutáveis, ou elementos sobre os quais não pode ocorrer *aliasing* (e.g. tipos primitivos da linguagem Java); a direita (φ), denominada de fórmula linear, é constituída por um conjunto de fórmulas proposicionais em lógica clássica, e modela a parte linear do *heap* (inspirada pela lógica de separação – Secção 2.2.2). Na nossa abordagem um objecto é considerado puro, se todos os métodos da classe do objecto são puros, ou seja, se não alteram o estado do objecto e só efectuam invocações puras. Um objecto puro tem também os construtores puros, contudo estes podem afectar as variáveis de instância do método. As fórmulas na parte linear são disjuntas e cada fórmula apenas refere propriedades sobre um objecto linear. Nesta aproximação, um objecto linear não tem construtores puros, no entanto pode ter métodos puros. De

notar a existência de predicados e funções, que permitem, respectivamente, expressar propriedades (tais como, relações entre dois termos como e.g. $>(2,3)$) e compor constantes e variáveis com operadores para fazer determinado cálculo (e.g. $-(2,3)$).

ψ	$::=$	$\phi + \varphi$	(Fórmula Dual)
φ	$::=$	$\emptyset \mid \phi \mid \phi * \varphi$	(Fórmula Linear)
ϕ	$::=$		(Fórmula Clássica)
		$\bot$	(Absurdo)
	$\mid$	$\phi\ lc\ \phi$	(Fórmula Binária)
	$\mid$	$\neg\phi$	(Negação)
	$\mid$	(ϕ)	(Fórmula entre Parênteses)
	$\mid$	$P(t_1, t_2, \cdots, t_n)$	(Símbolos de Predicado)
lc	$::=$	$\vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow$	(Conectivos Lógicos)
t	$::=$		(Termos)
		c	(Constantes)
	$\mid$	x	(Variáveis)
	$\mid$	$f(t_1, t_2, \cdots, t_n)$	(Símbolos de Função)

Figura 3.1: Sintaxe da Lógica Dual

Após apresentada a sintaxe da lógica dual e uma vez que dividimos uma fórmula numa parte pura e noutra linear, podemos derivar certos conceitos, como por exemplo, se uma variável presente numa fórmula dual é pura ou não.

Definição 3.1 (Variável Pura)**.** Seja x uma variável de um programa, x é considerada pura se o seu tipo é um tipo primitivo da linguagem ou uma referência para um objecto imutável. Em ambos os casos os seus tipos são consideradas puros e pertencem ao conjunto dos tipos puros $\mathcal{P}$.

Como complementar da definição de variável pura, temos a de variável linear.

Definição 3.2 (Variável Linear)**.** Seja x uma variável de um programa, x é considerada linear se é uma referência para um objecto cuja classe não é pura, sendo por isso considerada como sendo de um tipo linear. O seu tipo pertence, assim, ao conjunto dos tipos lineares $\mathcal{L}$.

Uma vez que uma fórmula dual pode referir múltiplas variáveis, podemos então definir como são compostos o conjunto das variáveis puras e o das lineares de uma fórmula. Esta definição é indutiva na estrutura de uma fórmula dual, ou seja, começamos por definir esses conjuntos para termos.

Definição 3.3 (Conjunto das Variáveis Puras de um Termo)**.** Seja t um termo, $\mathcal{C}, \mathcal{L}$ e $\mathcal{P}$, respectivamente, os conjuntos das constantes, dos tipos lineares e dos tipos puros. O conjunto de variáveis puras do termo t, denotado por $V_{pure}(t)$, é definido indutivamente na sua estrutura pelas seguintes regras:

$$V_{pure}(t) = \begin{cases} \emptyset & t \equiv c, c \in \mathcal{C} \\ \emptyset & t \equiv x, x{:}T \in \mathcal{L} \\ x & t \equiv x, x{:}T \in \mathcal{P} \\ \bigcup_{i=1}^{n} V_{pure}(t_i) & t \equiv f(t_1, \ldots, t_n), n > 0 \end{cases}$$

onde $x{:}T$ denota o tipo da variável x.

O conjunto de variáveis lineares de um termo é semelhante ao das variáveis puras como passamos a definir.

Definição 3.4 (Conjunto das Variáveis Lineares de um Termo)**.** Seja t um termo, $\mathcal{C}, \mathcal{L}$ e $\mathcal{P}$, respectivamente, os conjuntos das constantes, dos tipos lineares e dos tipos puros. O conjunto de variáveis lineares do termo t, denotado por $V_{lin}(t)$, é definido indutivamente na sua estrutura pelas seguintes regras:

$$V_{lin}(t) = \begin{cases} \emptyset & t \equiv c, c \in \mathcal{C} \\ x & t \equiv x, x{:}T \in \mathcal{L} \\ \emptyset & t \equiv x, x{:}T \in \mathcal{P} \\ \bigcup_{i=1}^{n} V_{lin}(t_i) & t \equiv f(t_1, \ldots, t_n), n > 0 \end{cases}$$

onde $x{:}T$ denota o tipo da variável x.

Após definidos os conjuntos das variáveis puras e o das lineares de um termo, podemos então introduzir a definição para as fórmulas.

Definição 3.5 (Conjunto das Variáveis Puras de uma Fórmula)**.** Seja ϕ uma fórmula clássica em lógica proposicional, $\odot \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, φ uma fórmula linear e $\psi = \phi + \varphi$ a fórmula dual composta por ϕ e φ. O conjunto de variáveis puras da fórmula ϕ é definido indutivamente na sua estrutura pelas seguintes regras:

$$V_{pure}(\phi) = \begin{cases} \emptyset & \phi \equiv \bot \\ \bigcup_{i=1}^{n} V_{pure}(t_i) & \phi \equiv P(t_1, \ldots, t_n), n > 0 \\ V_{pure}(\phi_1) & \phi \equiv \neg\phi_1 \\ V_{pure}(\phi_1) \cup V_{pure}(\phi_2) & \phi \equiv \phi_1 \odot \phi_2 \end{cases}$$

O conjunto de variáveis puras da fórmula linear φ é definido como:

$$V_{pure}(\varphi) = \bigcup_{i=1}^{n} V_{pure}(\varphi_i) \quad \varphi \equiv \varphi_1 * \cdots * \varphi_n, n > 0$$

O conjunto de variáveis puras da fórmula dual $\psi = \phi + \varphi$ é definido como:

$$V_{pure}(\psi) = V_{pure}(\phi) \cup V_{pure}(\varphi)$$

Quanto ao conjunto das variáveis lineares de uma fórmula, este é idêntico ao das variáveis puras como definimos a seguir.

Definição 3.6 (Conjunto das Variáveis Lineares de uma Fórmula)**.** Seja ϕ uma fórmula clássica em lógica proposicional, $\odot \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, φ uma fórmula linear e $\psi = \phi + \varphi$ a fórmula dual composta por ϕ e φ. O conjunto de variáveis lineares da fórmula ϕ é definido indutivamente na sua estrutura pelas seguintes regras:

$$V_{lin}(\phi) = \begin{cases} \emptyset & \phi \equiv \bot \\ \bigcup_{i=1}^{n} V_{lin}(t_i) & \phi \equiv P(t_1, \ldots, t_n), n > 0 \\ V_{lin}(\phi_1) & \phi \equiv \neg\phi_1 \\ V_{lin}(\phi_1) \cup V_{lin}(\phi_2) & \phi \equiv \phi_1 \odot \phi_2 \end{cases}$$

O conjunto de variáveis lineares da fórmula linear φ é definido como:

$$V_{lin}(\varphi) = \bigcup_{i=1}^{n} V_{lin}(\varphi_i) \quad \varphi \equiv \varphi_1 * \cdots * \varphi_n, n > 0$$

O conjunto de variáveis lineares da fórmula dual $\psi = \phi + \varphi$ é definido como:

$$V_{lin}(\psi) = V_{lin}(\varphi)$$

De notar o facto de a definição do conjunto de variáveis lineares de uma fórmula dual se traduzir nas variáveis lineares da parte linear da formula dual. A omissão da parte pura resulta da característica inerente de fórmula pura, onde não podem ser referidas propriedades sobre variáveis lineares, ou seja, o seu conjunto é vazio, resultando apenas nas variáveis lineares da fórmula linear.

Temos ainda definições de disjunção, conjunção e implicação de fórmulas duais. Uma vez que uma fórmula dual é composta por uma fórmula pura e uma linear, primeiro definimos a disjunção, conjunção e implicação de uma fórmula pura com uma linear e entre duas fórmulas lineares[1].

Definição 3.7 (Disjunção Linear)**.** Sejam ε uma fórmula pura, φ, ω, γ e δ fórmulas lineares. A disjunção de uma fórmula pura ε e uma linear φ resulta na disjunção da fórmula pura a todas as fórmulas que compõem a fórmula linear, ou seja:

$$\varepsilon \vee \varphi = \varphi \vee \varepsilon = \varepsilon \vee \varphi_1 * \cdots * \varepsilon \vee \varphi_n, n > 0$$

[1] As definições de negação dual e de equivalência dual podem ser obtidas através das definições de implicação e conjunção dual, pois $\neg\psi \stackrel{\text{abv}}{=} \psi \Rightarrow \bot$ e $\psi_1 \Leftrightarrow \psi_2 \stackrel{\text{abv}}{=} \psi_1 \Rightarrow \psi_2 \wedge \psi_2 \Rightarrow \psi_1$

A utilização do símbolo de disjunção entre duas fórmulas lineares φ e ω abrevia a seguinte fórmula linear:

$$\varphi \vee \omega \stackrel{\text{abv}}{=} \gamma * \delta * \alpha$$

onde

$$\gamma \subseteq \varphi : \forall_{v \in \varphi} \left((V_{lin}(v) \cap V_{lin}(\omega) = \emptyset) \Leftrightarrow (v \notin \gamma) \right)$$

$$\delta \subseteq \omega : \forall_{v \in \omega} \left((V_{lin}(v) \cap V_{lin}(\varphi) = \emptyset) \Leftrightarrow (v \notin \delta) \right)$$

$$\forall_{v \in \varphi, \theta \in \omega} \left((V_{lin}(v) \cap V_{lin}(\theta) \neq \emptyset) \Leftrightarrow ((v \vee \theta) \in \alpha) \right)$$

Definição 3.8 (Conjunção Linear)**.** Sejam ε uma fórmula pura, φ, ω, γ e δ fórmulas lineares. A conjunção de uma fórmula pura ε e uma linear φ resulta na conjunção da fórmula pura a todas as fórmulas que compõem a fórmula linear, ou seja:

$$\varepsilon \wedge \varphi = \varphi \wedge \varepsilon = \varepsilon \wedge \varphi_1 * \cdots * \varepsilon \wedge \varphi_n, n > 0$$

A utilização do símbolo de conjunção entre duas fórmulas lineares φ e ω abrevia a seguinte fórmula linear:

$$\varphi \wedge \omega \stackrel{\text{abv}}{=} \gamma * \delta * \alpha$$

onde

$$\gamma \subseteq \varphi : \forall_{v \in \varphi} \left((V_{lin}(v) \cap V_{lin}(\omega) = \emptyset) \Leftrightarrow (v \in \gamma) \right)$$

$$\delta \subseteq \omega : \forall_{v \in \omega} \left((V_{lin}(v) \cap V_{lin}(\varphi) = \emptyset) \Leftrightarrow (v \in \delta) \right)$$

$$\forall_{v \in \varphi, \theta \in \omega} \left((V_{lin}(v) \cap V_{lin}(\theta) \neq \emptyset) \Leftrightarrow ((v \wedge \theta) \in \alpha) \right)$$

Definição 3.9 (Implicação Linear)**.** Sejam ε uma fórmula pura, φ, ω, γ e δ fórmulas lineares. A implicação de uma fórmula pura ε a uma linear φ resulta na implicação da fórmula pura a todas as fórmulas que compõem a fórmula linear. A implicação de uma fórmula linear φ a uma fórmula pura resulta na implicação de todas as fórmulas que fazem parte da fórmula linear à fórmula pura, ou seja:

$$\varepsilon \Rightarrow \varphi = \varepsilon \Rightarrow \varphi_1 * \cdots * \varepsilon \Rightarrow \varphi_n, n > 0$$
$$\varphi \Rightarrow \varepsilon = \varphi_1 \Rightarrow \varepsilon * \cdots * \varphi_n \Rightarrow \varepsilon, n > 0$$

A utilização do símbolo de implicação entre duas fórmulas lineares φ e ω abrevia a seguinte fórmula linear:

$$\varphi \Rightarrow \omega \stackrel{\text{abv}}{=} \gamma * \delta * \alpha$$

onde

$$\gamma \subseteq \varphi : \forall_{v \in \varphi} \left((V_{lin}(v) \cap V_{lin}(\omega) = \emptyset) \Leftrightarrow (v \notin \gamma) \right)$$

$$\delta \subseteq \omega : \forall_{v \in \omega} \left((V_{lin}(v) \cap V_{lin}(\varphi) = \emptyset) \Leftrightarrow (v \in \delta) \right)$$

$$\forall_{\upsilon \in \varphi, \theta \in \omega} \left((V_{lin}(\upsilon) \cap V_{lin}(\theta) \neq \emptyset) \Leftrightarrow ((\upsilon \Rightarrow \theta) \in \alpha) \right)$$

Após definida a disjunção, conjunção e implicação de fórmulas lineares, introduzimos as definições de disjunção conjunção e implicação, entre uma fórmula pura e uma fórmula dual.

Definição 3.10 (Disjunção Pura)**.** Sejam ε uma fórmula pura, $\psi = \phi + \varphi$ uma fórmula dual. A disjunção de uma fórmula pura com uma dual tem como resultado uma fórmula dual, cuja parte pura se obtém pela disjunção da fórmula pura ε com a parte pura da fórmula dual ϕ. A parte linear resulta da disjunção da fórmula pura ε com a parte linear da fórmula dual φ.

$$\varepsilon \vee \psi = \varepsilon \vee \phi + \varepsilon \vee \varphi$$

Definição 3.11 (Conjunção Pura)**.** Sejam ε uma fórmula pura, $\psi = \phi + \varphi$ uma fórmula dual. A conjunção de uma fórmula pura com uma dual tem como resultado uma fórmula dual, cuja parte pura se obtém pela conjunção da fórmula pura ε com a parte pura da fórmula dual ϕ. A parte linear resulta da conjunção da fórmula pura ε com a parte linear da fórmula dual φ.

$$\varepsilon \wedge \psi = \varepsilon \wedge \phi + \varepsilon \wedge \varphi$$

Definição 3.12 (Implicação Pura)**.** Sejam ε uma fórmula pura, $\psi = \phi + \varphi$ uma fórmula dual. A implicação de uma fórmula pura com uma dual tem como resultado uma fórmula dual, cuja parte pura se obtém pela implicação da fórmula pura ε com a parte pura da fórmula dual ϕ. A parte linear resulta da implicação da fórmula pura ε com a parte linear da fórmula dual φ.

$$\varepsilon \Rightarrow \psi = \varepsilon \Rightarrow \phi + \varepsilon \Rightarrow \varphi$$

Podemos, ainda, derivar noções de implicação, equivalência e conjunção de fórmulas duais, com base nas definições anteriores, como definimos a seguir.

Definição 3.13 (Conjunção Dual)**.** Sejam $\psi_1 = \phi_1 + \varphi_1$ e $\psi_2 = \phi_2 + \varphi_2$ fórmulas duais tem-se que a conjunção dual de ψ_1 e ψ_2 resulta numa fórmula dual cuja parte pura corresponde à conjunção das fórmulas puras ϕ_1 e ϕ_2 e a parte linear à conjunção linear de φ_1 e φ_2, isto é:

$$\psi_1 \wedge \psi_2 = (\phi_1 \wedge \phi_2) + (\varphi_1 \wedge \varphi_2)$$

Definição 3.14 (Disjunção Dual)**.** Sejam $\psi_1 = \phi_1 + \varphi_1$ e $\psi_2 = \phi_2 + \varphi_2$ fórmulas duais tem-se que a disjunção dual de ψ_1 e ψ_2 resulta numa fórmula dual cuja parte pura

corresponde à disjunção das fórmulas puras ϕ_1 e ϕ_2 e a parte linear à disjunção linear de φ_1 e φ_2, isto é:

$$\psi_1 \vee \psi_2 = (\phi_1 \vee \phi_2) + (\varphi_1 \vee \varphi_2)$$

Definição 3.15 (Implicação Dual)**.** Sejam $\psi_1 = \phi_1 + \varphi_1$ e $\psi_2 = \phi_2 + \varphi_2$ fórmulas duais tem-se que a implicação dual de ψ_1 e ψ_2 resulta numa fórmula dual cuja parte pura corresponde à implicação das fórmulas puras ϕ_1 e ϕ_2 e a parte linear à implicação linear de φ_1 e φ_2, isto é:

$$\psi_1 \Rightarrow \psi_2 = (\phi_1 \Rightarrow \phi_2) + (\varphi_1 \Rightarrow \varphi_2)$$

Adicionalmente, podemos referir zonas do *heap* específicas sobre as quais pretendemos mencionar propriedades através de restrição do *heap*, como se define a seguir.

Definição 3.16 (Restrição do *Heap*)**.** Seja φ a parte direita de uma fórmula dual, isto é, a fórmula linear, e $x_1, x_2, \ldots, x_n$ variáveis, $\varphi\downarrow\{x_1, x_2, \ldots, x_n\}$ é denominada de fórmula linear restrita a $x_1, x_2, \ldots, x_n$, e é composta pelo subconjunto de fórmulas contidas em φ, que apenas contém informação sobre propriedades referentes às variáveis $x_1, x_2, \ldots, x_n$.

Como complementar desta definição, podemos excluir zonas do *heap* sobre as quais não pretendemos referir propriedades.

Definição 3.17 (Exclusão do *Heap*)**.** Seja φ a parte direita de uma fórmula dual, isto é, a fórmula linear, e $x_1, x_2, \ldots, x_n$ variáveis, $\varphi - \{x_1, x_2, \ldots, x_n\}$ é denominada de fórmula linear excluindo $x_1, x_2, \ldots, x_n$, e é composta pelo subconjunto de fórmulas contidas em φ, que não contém informação sobre propriedades referentes às variáveis $x_1, x_2, \ldots, x_n$.

Como exemplo, na Figura 3.2, definimos uma fórmula linear (φ) e aplicamos as duas definições anteriores.

$$\begin{gathered}\varphi \equiv P_1(x) * P_2(y) * P_3(z)\\ \varphi\downarrow\{x, z\} = P_1(x) * P_3(z)\\ \varphi - \{x, z\} = P_2(y)\end{gathered}$$

Figura 3.2: Exemplo de Aplicação da Restrição e Exclusão do *Heap*

3.2 Linguagem de Especificação

A linguagem de especificação desenvolvida assemelha-se à do JML e Spec#, mas é leve e tem como base uma lógica monádica dual. É mais simples, não apresentando,

por exemplo, quantificadores, nem referência ao valor de uma expressão na sua pré-condição. Tal como no Spec# e JML, usamos as palavras reservadas **requires** e **ensures** para descrever, respectivamente, pré-condições e pós-condições de um procedimento. Quanto às invariantes de classe e de ciclo, usamos também a palavra reservada **invariant**.

3.2.1 Asserções

Nesta secção, apresentamos a sintaxe abstracta respeitante à parte das asserções da linguagem. Como podemos observar na Figura 3.3, esta permite descrever o estado em que se encontra determinado objecto ou tipo primitivo da linguagem SpecJava, mais concretamente, referir-nos a estados que dizem respeito às variáveis de instância de uma classe (fn), aos parâmetros (pn) e retorno (**return**) de um método, ou ao próprio estado da classe (**this**).

D, I	::=	*CF* + *SLF*	(Fórmula Dual)
SLF	::=	*CF* | *CF* $\star$ *SLF*	(Fórmula Linear)
CF	::=	**true** | **false** | *CF bop CF* | ! *CF* | $b : S$	(Fórmula Clássica)
bop	::=	`&&` | `||` | => | <=>	(Conectivos Lógicos)
b	::=	*fn* | **this** | **return** | *pn*	(Propriedades/Estados – Alvo)
S	::=	**true** | **false** | **pos** | **neg** | **zero** | **null** | *sn*	(Propriedades/Estados)

$pn \in$ nomes dos parâmetros
$fn \in$ nomes das variáveis de classe
$sn \in$ nomes dos estados

Figura 3.3: Sintaxe Abstracta – Asserções

Os estados são compostos por um conjunto de estados base, que se aplicam aos tipos primitivos, ou seja, no caso de variáveis primitivas booleanas estão associados os estados **true** e **false**, no caso de variáveis primitivas numéricas, **pos**, **neg** ou **zero**. No que diz respeito a referências de objectos, estas podem ser referências nulas (**null**) ou, então, referir estados definidos na classe do tipo do objecto (sn).

3.2.2 Classes

Nesta secção é apresentada a sintaxe abstracta para classes. Como podemos observar na Figura 3.4, a novidade é a especificação de classe. É suportada a noção de invariante, declarados como **invariant** D, para exprimir propriedades que todas as instâncias de uma classe devem satisfazer.

classDecl	::=	**class** cn { *classMember** }	(Declaração de Classe)
classMember	::=	...	(Membros de Classe)
	|	*field*	(Decl. de Variáveis de Instância)
	|	*method*	(Declaração de Métodos)
	|	*constructor*	(Declaração de Construtores)
	|	*classSpec*	(Especificação de Classe)
classSpec	::=		(Especificação de Classe)
		define sn**;**	(Definição Abstracta)
	|	**define** $sn = D$**;**	(Definição Concreta)
	|	**invariant** D**;**	(Invariante de Classe)
field	::=	$T\ fn\ [= E]?$ **;**	(Decl. de Variáveis de Instância)

$cn \in$ nomes das classes
$sn \in$ nomes dos estados
$fn \in$ nomes das variáveis de classe

Figura 3.4: Sintaxe Abstracta – Classes

Uma classe preserva os seus invariantes se todos os métodos públicos dessa classe preservam esses invariantes. Contudo, destacando-se do Spec#, onde apenas podemos quebrar invariantes através de um comando explícito, na nossa aproximação, estes podem ser quebrados durante a execução do método, desde que sejam restabelecidos no fim da sua execução. Os construtores têm de garantir, para além das suas pós-condições, os invariantes da classe.

Para além dos invariantes, ao nível da classe, as especificações são compostas por mais duas construções, que dizem respeito à definição de estados/propriedades associadas à classe. Estas definições podem ser concretas ou abstractas, sendo declaradas, respectivamente, como **define** $sn = D$ e **define** sn. No que diz respeito às definições concretas, estas são construídas com base em estados/propriedades observadas nas variáveis de instância da classe e/ou noutras definições ao nível da classe (abstractas ou concretas). Quanto às definições abstractas, estas representam um estado/propriedade da classe, de uma forma abstracta, sem recorrer a outras definições e/ou estados observados nas variáveis de instância da classe.

```
1  public class Buffer {
2      define empty = count:zero;
3      define full;
4
5      private int[] buffer;
6      invariant + !buffer:null;
7
8
9      private int tail;
10     private int head;
11     invariant !tail:neg && !head:neg;
12
13     private int count;
14     invariant !count:neg;
15 }
```

Figura 3.5: Buffer – Especificação ao Nível da Classe

A Figura 3.5 é um exemplo de especificação ao nível da classe. Esta mostra uma classe que representa um buffer de inteiros, implementado em array circular. Como se pode observar, são definidos dois estados na classe Buffer, um abstracto, `full`, identificando que o buffer se encontra "cheio" não podendo albergar mais elementos, e outro concreto, `empty`, que identifica o estado do buffer quando este não contém nenhum elemento, ou seja, a contagem do número de elementos é igual a zero (`count:zero`), e um conjunto de invariantes sobre os membros de instância da respectiva classe.

3.2.3 Procedimentos

Nesta secção, apresentamos a sintaxe abstracta para os procedimentos. A especificação dos métodos e construtores é composta por duas fórmulas que dizem respeito às pré-condições e pós-condições, declarados, respectivamente, como **requires** D, **ensures** D, como é mostrado na Figura 3.6.

As pré-condições de um método especificam condições que devem ser verdade ao início da sua execução. Nestas condições, apenas podem ser envolvidos os estados da classe, das variáveis de instância da classe, ou os parâmetros do método. No que diz respeito às pós-condições, estas referem-se ao estado do objecto após executar a operação e pode envolver nas suas condições, para além do referido para as pré-condições, o estado do retorno de um método.

Para além destas especificações, são ainda suportados os comandos **assume** e **sassert**, com o sentido usual de assumir ou verificar uma condição em dado ponto

method	::=	*modifier* T $mn(\overline{arg})$ $\overline{spec}$ { ST }	(Declaração de Métodos)
constructor	::=	*modifier* $cn(\overline{arg})$ $\overline{spec}$ { ST }	(Declaração de Construtores)
modifier	::=	**public** | ... | **pure**	(Modificadores)
$spec$	::=		(Especificação de Procedimentos)
		requires D	(Pré-condição)
	|	**ensures** D	(Pós-condição)
ST	::=	...	(Comandos)
	|	**assume** D	(Assume)
	|	**sassert** D	(Assert Estático)

$mn \in$ nomes dos métodos
$cn \in$ nomes das classes
$sn \in$ nomes dos estados

Figura 3.6: Sintaxe Abstracta – Procedimentos

do programa e são declarados, respectivamente, como **assume** D and **sassert** D. Por último, para especificar que um procedimento é puro, os modificadores do Java são estendidos com a palavra reservada **pure**.

```
 1 public Buffer(int size)
 2     requires size:pos
 3     ensures tail:zero && head:zero
 4     ensures + empty
 5     ensures + !full
 6 {
 7     buffer = new int[size + 1];
 8     // size + 1, pois é desperdiçada
 9     // um posição para controlar se
10     // o buffer está vazio ou cheio
11     tail = 0;
12     head = 0;
13     count = 0;
14     assume + !full;
15 }
16 public pure int dataSize()
17     ensures !return:neg
18 { return count; }
```

Figura 3.7: Buffer – Especificação de Método e Construtor

A Figura 3.7 demonstra a especificação ao nível dos métodos e construtores da classe `Buffer`. Por exemplo, a pré-condição **requires** `size:pos` indica que o argumento `size` tem de ser positivo (`size:pos`), ou seja, o buffer tem de ter capacidade

para um ou mais elementos.

O método `dataSize`, que retorna o número de elementos no buffer, demonstra uma asserção associada ao retorno de um método, citando que este não é negativo e ainda a utilização do modificador **pure** que ilustra que o método é puro, não alterando o estado do buffer.

A Figura 3.8 ilustra um exemplo de especificações numa situação em que é necessário seguir um protocolo, como é o caso de um ficheiro. Primeiro, deve-se abrir o ficheiro, posteriormente, podem ser efectuadas leituras e escritas, e por último, este é fechado.

```
1  public class File {
2      define open;
3      public void open()
4          requires + !open
5          ensures + open;
6      public void write(int b)
7          requires + open
8          ensures + open;
9      public int read()
10         requires + open
11         ensures + open;
12     public void close()
13         requires + open
14         ensures + !open;
15 }
```

Figura 3.8: Especificação do Protocolo de um Ficheiro

Na Figura 3.9, encontra-se ilustrado parte do restante da sintaxe do SpecJava, que diz respeito à composição de um programa e à sintaxe abstracta dos comandos e expressões da linguagem.

Uma vez sendo uma extensão do Java, a sintaxe total corresponde à desta linguagem. No entanto, decidimos colocar aqui apenas a parte mais relevante, pois os aspectos abordados pelo cálculo desenvolvido incidem nesta parte. Apesar de ainda não serem suportadas algumas das características do Java, decidimos não efectuar alterações à sua sintaxe, de forma a apenas permitir os comandos e expressões aqui descritos, pois é nossa intenção, como trabalho futuro, tratar de outros aspectos que foram deixados de parte no desenvolvimento desta dissertação (Secção 5.1).

P	::=	$classDecl^*$	(Programa)
ST	::=		(Comandos)
		skip	(Skip)
	|	**while** (ε) **invariant** I ST	(While)
	|	**for** (*ForInits*; ε; *ForUpdates*) **invariant** I ST	(For)
	|	**do** ST **while** (ε) **invariant** I	(Do While)
	|	**if** (ε) ST **else** ST	(If Else)
	|	**if** (ε) ST	(If)
	|	**synchronized** (E) ST	(Sincronização)
	|	ST ; ST	(Composição)
	|	{ ST }	(Bloco)
	|	**return** E	(Retorno)
	|	**this**$(\overline{E})$	(Invocação de Construtor)
	|	*local*	(Decl. de Variáveis Locais)
	|	*STE*	(Comando-Expressão)
local	::=	$T\ ln\ [= E]?$	(Decl. de Variáveis Locais)
E, ε	::=		(Expressões)
		$E\ bop\ E$	(Expressão Binária)
	|	$uop\ E$	(Expressão Unária)
	|	(E)	(Expressão entre Parênteses)
	|	*STE*	(Comando-Expressão)
	|	x	(Identificador)
	|	n	(Número)
	|	s	(String)
ForUpdates	::=	$\overline{STE}$	(Actualizações – For)
ForInits	::=	$\overline{STE}$ | *local*	(Inicializações – For)
STE	::=	$x = E$	(Afectação)
	|	$E.mn(\overline{E})$	(Invocação de Método)
	|	**new** $cn(\overline{E})$	(Instanciação de Objecto)
bop	::=		(Operadores Binários)
		+ | - | * | / | %	(Aritméticos)
	|	== | !=	(Igualdade)
	|	> | < | >= | <=	(Relacionais)
	|	&& | ||	(Condicionais)
uop	::=	- | !	(Operadores Unários)

$cn/mn/ln \in$ nomes dos construtores/métodos/variáveis locais

Figura 3.9: Sintaxe Abstracta – Comandos e Expressões

Após definida a linguagem de especificação e sua integração na linguagem Java, o passo seguinte consiste em determinar como se verifica um programa de acordo com a sua especificação.

3.3 Regras de Verificação

A abordagem seguida para a verificação de um programa na linguagem SpecJava tem como base o cálculo wp. Sendo assim, uma vez que a aproximação desenvolvida por Dijkstra tem como alvo uma linguagem de programação imperativa, foi necessário estender o cálculo proposto por Dijkstra para uma linguagem orientada a objectos, neste caso concreto o Java.

3.3.1 Cálculo WP

Tendo como base as propriedades referidas na Secção 2.2.3, é necessário, para provar que um programa na linguagem da Figura 3.9 está correcto de acordo com a sua especificação, associar a cada comando da linguagem o respectivo predicado transformador. Esta extensão é bastante intuitiva e apesar de não ter sido efectuada uma prova formal para as diversas regras, que constitui um dos pontos mais importantes como trabalho futuro (Secção 5.1), podemos constatar que estas permanecem coerentes e correctas. Uma vez que o cálculo desenvolvido distingue propriedades puras de lineares, as regras para o cálculo de pré-condições mais fracas dividem-se em dois grupos, atendendo ao facto de os comandos serem independentes ou não dessa separação.

3.3.1.1 Comandos Independentes

Nesta secção, apresentamos o cálculo de pré-condições mais fracas para os comandos que não dependem da separação entre propriedades puras e lineares, respectivamente, para os ciclos e para os comandos composicional, **skip**, **assume**, **sassert** e condicional, que passamos a explicar.

Composição. No caso da composição sequencial de comandos, podemos observar que esta regra é idêntica à proposta por Dijkstra no seu cálculo. Temos então, que a pré-condição mais fraca de uma sequência de comandos resulta da pré-condição mais fraca de ST_1 ($wp\,(ST_1,\ R)$), respeitante à pós-condição que se obtém do cálculo da pré-condição mais fraca de ST_2 ($R = wp\,(ST_2,\ C + S)$).

[composição]

$$wp\,(ST_1;\ ST_2, C + S) = wp\,(ST_1, wp\,(ST_2, C + S))$$

Skip. Quanto ao comando vazio (**skip**), esta regra também é idêntica à proposta por Dijkstra. A sua pré-condição mais fraca corresponde à pós-condição (A), pois o comando vazio não altera o estado de um programa.

[skip]

$$wp\,(\textbf{skip}, A) = A$$

Assume. Para o comando **assume**, como estamos a assumir uma condição num dado ponto do programa, temos de assegurar que nesse ponto a condição implica a pós-condição actual (cf. Definição 3.15), sendo essa a sua pré-condição mais fraca.

[assume]

$$wp\,(\textbf{assume}\ A, R) = A \Rightarrow R$$

Assert Estático. No que diz respeito ao comando **sassert**, uma vez que este comando verifica se uma condição é verdade em dado ponto do programa, temos de garantir, adicionalmente, essa condição nesse ponto. Assim sendo, a pré-condição mais fraca resulta da conjunção dual (cf. Definição 3.13) da pós-condição com a condição a ser verificada ($A \wedge R$).

[assert estático]

$$wp\,(\textbf{sassert}\ A, R) = A \wedge R$$

Ciclo While. No caso dos ciclos "while", a expressão da condição (ε) tem de ser pura, composta por constantes ou variáveis de tipos primitivos. Podemos observar que a regra respeitante a este comando é bastante simples e corresponde ao seu invariante, visto que o invariante tem de ser mantido em cada iteração e tem também de ser válido ao início do ciclo, conforme se encontra nas premissas da regra.

[ciclo while]

$$\frac{(I \wedge \neg\varepsilon) \Rightarrow R \quad (I \wedge \varepsilon) \Rightarrow wp\,(ST, I)}{wp\left(\begin{array}{l}\textbf{while}\ (\varepsilon) \\ \quad \textbf{invariant}\ I\ , R \\ ST\end{array}\right) = I}$$

Ciclo For. No caso dos ciclos "for", a expressão da condição (ε) tem de ser pura, tal como referido para os ciclos "while". A pré-condição mais fraca do ciclo "for" pode ser obtida através da pré-condição mais fraca da composição sequencial e do ciclo "while",

uma vez que um ciclo "for" pode ser traduzido num ciclo "while"[2], como passamos a demonstrar.

$$wp\left(\begin{array}{l}\textbf{for}\ (\mathit{ForInits};\ \varepsilon;\ \mathit{ForUpdates})\\ \quad\textbf{invariant}\ I\\ ST\end{array}, R\right) \equiv$$

$$\equiv wp\left(\mathit{ForInits};\begin{array}{l}\textbf{while}\ (\varepsilon)\\ \quad\textbf{invariant}\ I\\ \{ST;\ \mathit{ForUpdates}\}\end{array}, R\right) =$$

$$= wp\left(\mathit{ForInits}, wp\left(\begin{array}{l}\textbf{while}\ (\varepsilon)\\ \quad\textbf{invariant}\ I\\ \{ST;\ \mathit{ForUpdates}\}\end{array}, R\right)\right) = wp\,(\mathit{ForInits}, I)$$

As premissas da regra também podem ser obtidas de forma idêntica. De seguida ilustramos a regra completa com as premissas.

$$\texttt{[ciclo for]}$$

$$\frac{(I \wedge \neg\varepsilon) \Rightarrow R \quad (I \wedge \varepsilon) \Rightarrow wp\,(ST; \mathit{ForUpdates}, I)}{wp\left(\begin{array}{l}\textbf{for}\ (\mathit{ForInits};\ \varepsilon;\ \mathit{ForUpdates})\\ \quad\textbf{invariant}\ I\\ ST\end{array}, R\right) = wp\,(\mathit{ForInits}, I)}$$

Ciclo Do While. Quanto aos ciclos "do while", a aproximação ao cálculo da sua pré-condição mais fraca é semelhante à seguida para os ciclos "for", ou seja, efectuamos a tradução de um ciclo "do while" para um ciclo "while"[2], calculando a pré-condição mais fraca do resultado dessa transformação. Identicamente aos outros tipos de ciclo, a expressão da condição tem de ser pura.

$$wp\left(\begin{array}{l}\textbf{do}\\ \quad ST\\ \textbf{while}\ (\varepsilon)\\ \quad\textbf{invariant}\ I\end{array}, R\right) \equiv wp\left(\begin{array}{l}\quad\textbf{while}\ (\varepsilon)\\ ST;\quad\textbf{invariant}\ I\\ \quad ST\end{array}, R\right) =$$

$$= wp\left(ST, wp\left(\begin{array}{l}\textbf{while}\ (\varepsilon)\\ \quad\textbf{invariant}\ I\\ ST\end{array}, R\right)\right) = wp\,(ST, I)$$

As premissas da regra podem ser obtidas de forma idêntica ao caso dos ciclos "for".

[2]Esta transformação é efectuada tendo em conta que não ocorrem os comandos **break**, **continue** e **return** no corpo do ciclo "for" ou "do while" a traduzir, uma vez que estes alteram o fluxo de execução normal de uma iteração de um ciclo, não sendo a tradução idêntica à apresentada (Secção 5.1)

$$\frac{(I \wedge \neg\varepsilon) \Rightarrow R \quad (I \wedge \varepsilon) \Rightarrow wp\,(ST, I)}{wp\left(\begin{array}{l}\textbf{do}\\ \quad ST\\ \textbf{while}\ (\varepsilon)\\ \quad \textbf{invariant}\ I\end{array}, R\right) = wp\,(ST, I)}\ [\texttt{ciclo do while}]$$

Condicional. Quanto ao comando condicional, a expressão da condição (ε) tem de ser pura, à semelhança dos ciclos, e duas situações podem ocorrer durante a sua execução, ou a expressão na condição é verdadeira entrando no primeiro ramo do comando condicional, ou a expressão é falsa entrando no segundo. Tomando como base estas situações, podemos concluir que, se entrarmos no primeiro ramo então a expressão da condição (ε) tem de implicar a pré-condição mais fraca deste ramo, caso contrário, se for falsa, a negação da condição ($\neg\varepsilon$) tem de implicar a pré-condição mais fraca do segundo ramo. Como ambos os casos podem ocorrer, conclui-se que a pré-condição mais fraca do comando condicional é a conjunção destas duas alternativas.

$$wp\left(\begin{array}{l}\textbf{if}\ (\varepsilon)\ ST_1\\ \textbf{else}\ ST_2\end{array}, C + S\right) = \varepsilon \Rightarrow wp\,(ST_1, C + S) \wedge \neg\varepsilon \Rightarrow wp\,(ST_2, C + S)\ [\texttt{condicional}]$$

3.3.1.2 Comandos Puros/Lineares

Nesta secção, apresentamos o cálculo de pré-condições mais fracas para os comandos da linguagem, que dependem da separação entre propriedades puras e lineares. Como referimos anteriormente, na nossa aproximação, um objecto é considerado puro se todos os métodos e construtores da classe do objecto forem puros. Temos ainda que os métodos de um objecto puro não podem retornar objectos lineares. Uma vez que podemos efectuar *aliasing* de objectos puros, se permitíssemos que estes retornassem objectos lineares, poderiam ocorrer situações de *aliasing* dos objectos lineares, indirectamente. As regras apresentadas nesta secção assumem que os procedimentos são públicos e não estáticos. Para as outras situações possíveis, estes são casos particulares destas regras.

Afectação Pura. Quanto à regra para o cálculo da pré-condição mais fraca de uma afectação pura, esta é bastante semelhante à regra de Dijkstra sendo o seu resultado a substituição de x por ε[3] na fórmula pura ($C[x/\varepsilon]$) e na fórmula linear ($S[x/\varepsilon]$), uma

[3] ε pode ser uma referência nula – `null`

vez que numa fórmula linear podem ser referidas propriedades sobre variáveis puras. Apesar desta substituição ser efectuada, os factos sobre variáveis lineares não são alterados no *heap*.

[afectação pura]

$$wp\,(x = \varepsilon, C + S) = C[x/\varepsilon] + S[x/\varepsilon]$$

Afectação Linear. A regra do cálculo da pré-condição mais fraca de uma afectação linear é composta por dois casos. Ambas as regras têm como resultado a substituição de x por y ou `null` na fórmula linear, mantendo-se a parte pura igual à pós-condição, pois sendo uma afectação linear, apenas os factos da parte linear da fórmula são alterados. Quando a afectação é não nula ($y \neq$ `null`) e se as duas variáveis presentes na afectação são diferentes ($x \neq y$), temos ainda de garantir nas premissas que não temos informação sobre a variável y no lado linear, uma vez que o nosso cálculo é linear e o estado de y é transferido para x após a afectação, removendo todos os factos de y do *heap*.

[afectação linear]

$$\frac{y \neq \texttt{null} \quad x \neq y \rightarrow S \downarrow \{y\} = \emptyset}{wp\,(x = y, C + S) = C + S[x/y]}$$

[afectação linear nula]

$$wp\,(x = \texttt{null}, C + S) = C + S[x/\texttt{null}]$$

Retorno Puro. No que diz respeito à pré-condição mais fraca do comando de retorno de uma expressão pura, esta é semelhante à da afectação pura. Contudo, esta regra depende da pós-condição do método respectivo a que o retorno pertence, ou seja, apenas pode ser aplicada a essa pós-condição. Como resultado temos a substituição de **return** por ε[4] na fórmula pura da pós-condição do método ($R_{mn_A}[\textbf{return}/\varepsilon]$). Quanto à parte linear, esta resulta da mesma substituição efectuada para a parte pura na fórmula linear da pós-condição do método ($R_{mn_B}[\textbf{return}/\varepsilon]$), à semelhança da afectação pura.

[retorno puro]

$$wp\,(\textbf{return}\ \varepsilon, R_{mn_A} + R_{mn_B}) = R_{mn_A}[\textbf{return}/\varepsilon] + R_{mn_B}[\textbf{return}/\varepsilon]$$

Retorno Linear. A regra do cálculo da pré-condição mais fraca do comando de retorno de uma variável linear é semelhante à da afectação linear. No entanto, tal como no caso da pré-condição mais fraca para o retorno puro, depende da pós-condição do método respectivo a que o retorno pertence. Como resultado, temos a substituição de **return** por y ou `null` na fórmula linear, mantendo-se a parte pura igual à pós-condição do método, pois sendo uma afectação linear, apenas os factos da parte linear

[4] ε pode ser uma referência nula – `null`

da fórmula são alterados. Quando o retorno é não nulo ($y \neq \texttt{null}$), temos ainda de garantir nas premissas que não temos informação sobre a variável y no lado linear, à semelhança da afectação linear.

[retorno linear]

$$\frac{y \neq \texttt{null} \quad R_{mn_B} \downarrow \{y\} = \emptyset}{wp\,(\textbf{return}\ y, R_{mn_A} + R_{mn_B}) = R_{mn_A} + R_{mn_B}[\textbf{return}/y]}$$

[retorno linear nulo]

$$wp\,(\textbf{return}\ \texttt{null}, R_{mn_A} + R_{mn_B}) = R_{mn_A} + R_{mn_B}[\textbf{return}/\texttt{null}]$$

Instanciação Pura. Quanto à criação de objectos puros, é necessário ter como pré-condição mais fraca na fórmula pura resultante, relativamente a uma pós-condição, $C + S$, a parte pura da pré-condição do construtor da classe do objecto a instanciar (Q_{cn_A}), e ainda que se culmine num estado cuja parte pura da pós-condição do construtor (R_{cn_A}) implique C. Para a parte linear, uma vez que a criação de objectos pode receber como parâmetros objectos lineares ($\overline{z}$), tem de se garantir, como pré-condição mais fraca, a parte linear da pré-condição do construtor (Q_{cn_B}), e ainda que se termine num estado em que a parte linear da pós-condição (R_{cn_B}) implique a parte linear do *heap* que contém propriedades sobre o objecto e sobre os seus parâmetros (através de restrição do *heap* cf. Definição 3.16), permanecendo todos os factos no *heap* sobre elementos que não são afectados pela chamada do procedimento, ou seja, é efectuada a exclusão do *heap* (cf. Definição 3.17) do objecto instanciado e dos seus parâmetros lineares. Podemos observar também, que a variável x é substituída por um nome fresco f, uma vez que estamos a afectar à variável um novo valor, que corresponde ao novo objecto puro criado, objecto esse que não existe na pré-condição, daí essa substituição.

[instanciação pura]

$$wp\,(x = \textbf{new}\ cn(\overline{y}, \overline{z}), C + S) = \begin{array}{c} \left(\begin{array}{c} Q_{cn_A}[\overline{p_1}/\overline{y}] \\ \wedge \\ \left(\left(\begin{array}{c} f \neq \texttt{null} \\ \wedge \\ R_{cn_A}[this/f, \overline{p_1}/\overline{y}] \end{array} \right) \Rightarrow C[x/f] \right) \end{array} \right) \\ + \\ \left(\begin{array}{c} \left(\begin{array}{c} Q_{cn_B}[\overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \\ \wedge \\ (R_{cn_B}[this/f, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \Rightarrow S \downarrow \{x, \overline{z}\}[x/f]) \end{array} \right) \\ * \\ S - \{x, \overline{z}\} \end{array} \right) \end{array}$$

Instanciação Linear. No que diz respeito à criação de objectos lineares, é necessário ter como pré-condição mais fraca, na fórmula pura resultante, a parte pura da pré-condição do construtor da classe do objecto a instanciar (Q_{cn_A}), e ainda que se culmine num estado cuja parte pura da pós-condição do construtor (R_{cn_A}) implique a pós-condição à qual está a ser aplicado o cálculo (C), de forma semelhante à instanciação pura. Quanto à parte linear, tem de se garantir como pré-condição mais fraca, a parte linear da pré-condição do construtor (Q_{cn_B}), permanecendo todos os factos no *heap* sobre elementos que não são afectados pela chamada do procedimento, ou seja, é efectuada a exclusão do *heap* (cf. Definição 3.17) da variável x e dos parâmetros lineares do construtor ($\overline{z}$). Adicionalmente, nas premissas, temos de garantir que após instanciar o objecto, se atinja um estado em que a parte linear da pós-condição do construtor (R_{cn_B}) implique a zona linear do *heap* alterada pelo construtor, sendo efectuada a restrição do *heap* (cf. Definição 3.16). Como premissa da regra, temos de garantir também que em S não ocorra informação sobre os parâmetros lineares, assegurando assim a linearidade do cálculo desenvolvido e a detecção de situações de *aliasing*. Caso tal aconteça, encontramo-nos na presença de uma situação de *aliasing*, pois estamos a referir propriedades sobre objectos previamente consumidos pelo procedimento, sobre os quais nada podemos garantir, porque estes podem ter sido copiados. Para a instanciação pura, não é necessário efectuar esta verificação porque um objecto puro não altera o estado dos objectos lineares, pois todos os seus procedimentos são puros. Contudo, é excluída desta verificação a variável x que corresponde ao objecto a instanciar, pois este também é linear e podemos estar a atribuir um novo valor a essa variável, que é diferente do valor passado em argumento. Temos ainda, que a variável x é substituída por um nome fresco f, uma vez que estamos a afectar à variável um novo valor x, que corresponde ao novo objecto criado, objecto esse que não existe na pré-condição. Esta substituição também ocorre na parte pura da pós-condição C, uma vez que um objecto linear pode conter variáveis puras sobre a quais pode garantir propriedades.

$$\frac{S \downarrow \{\overline{z} \backslash x\} = \emptyset \qquad \texttt{true} + \left((f \neq \texttt{null} \wedge R_{cn_B}[this/f, \overline{p_1}/\overline{y}]) \Rightarrow S \downarrow \{x, \overline{z}\}[x/f]\right)}{wp\,(x = \textbf{new}\ cn(\overline{y}, \overline{z}), C + S) = \begin{pmatrix} Q_{cn_A}[\overline{p_1}/\overline{y}] \\ \wedge \\ (R_{cn_A}[this/f, \overline{p_1}/\overline{y}] \Rightarrow C[x/f]) \end{pmatrix} + (Q_{cn_B}[\overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] * S - \{x, \overline{z}\})} \quad \text{[instanciação linear]}$$

Invocação Especial Pura. No caso da invocação especial pura do construtor da classe, podemos observar que é semelhante à regra da instanciação pura. Contudo, como permanecemos no mesmo escopo (ou seja, na mesma classe), pois estamos a instanciar

o objecto com base noutro construtor, não é efectuada a substituição pelo nome fresco.

$$[\texttt{invocação especial pura}]$$

$$wp\,(\textbf{this}(\overline{y},\overline{z}), C + S) = \left(\begin{array}{c} \Big(Q_{cn_A}[\overline{p_1}/\overline{y}] \wedge (R_{cn_A}[\overline{p_1}/\overline{y}] \Rightarrow C)\Big) \\ + \\ \left(\begin{array}{c} Q_{cn_B}[\overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \\ \wedge \\ (R_{cn_B}[\overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \Rightarrow S \downarrow \{this, \overline{z}\}) \end{array} \right) \\ * \\ S - \{this, \overline{z}\} \end{array} \right)$$

Invocação Especial Linear. Quanto à invocação especial linear do construtor da classe, podemos observar que é semelhante à regra da instanciação linear, retirando os factos sobre a variável x, uma vez que não está a ocorrer nenhuma afectação, e estamos a instanciar um objecto com base noutro construtor, permanecendo assim no escopo da mesma classe.

$$[\texttt{invocação especial linear}]$$

$$\frac{S \downarrow \{\overline{z}\} = \emptyset \qquad \texttt{true} + (R_{cn_B}[\overline{p_1}/\overline{y}] \Rightarrow S \downarrow \{this, \overline{z}\})}{wp\,(\textbf{this}(\overline{y},\overline{z}), C + S) = \begin{array}{c} (Q_{cn_A}[\overline{p_1}/\overline{y}] \wedge (R_{cn_A}[\overline{p_1}/\overline{y}] \Rightarrow C)) \\ + \\ (Q_{cn_B}[\overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] * S - \{this, \overline{z}\}) \end{array}}$$

Invocação de Método Puro com Retorno Puro em Objecto Puro. Para as chamadas de métodos puros a um objecto puro, com tipo de retorno puro, a pré-condição mais fraca é semelhante à criação de objectos, sendo necessário garantir adicionalmente, que não se está a efectuar uma chamada de método a uma referência nula ($k \neq \texttt{null}$). Temos ainda, de forma semelhante, que o retorno do método é substituído por um nome fresco f, uma vez que estamos a afectar à variável um novo valor, que corresponde ao resultado do método, que não existe na pré-condição. Como os métodos têm de preservar os invariantes da classe à qual pertencem, temos de assumir esses invariantes (I_c) como pré-condição da chamada do método. Esta assumpção é válida, pois quando o objecto é criado, é verificado que este garante os invariantes da classe e ainda porque todos os métodos os preservam[5].

[5]No nome das regras de invocações de métodos, o título é abreviado sobre a forma `X/Y/Z`. `X` refere-se ao tipo de invocação, ou seja invocação de método puro (`mp`) ou método linear (`ml`). `Y` diz respeito ao retorno, ou seja, se este é puro (`rp`) ou linear (`rl`). Por último, `Z` corresponde ao facto de o objecto ser puro (`op`) ou linear (`ol`). Para os métodos sem retorno, `Y` é omitido.

[invocação pura - mp/rp/op]

$$wp\,(x = k.mn(\overline{y},\overline{z}), C + S) = I_c[this/k] \Rightarrow \left(\begin{array}{c} k \neq \texttt{null} \wedge Q_{mn_A}[this/k, \overline{p_1}/\overline{y}] \\ \wedge \\ \left(\left(\begin{array}{c} k \neq \texttt{null} \\ \wedge \\ R_{mn_A}[this/k, \overline{p_1}/\overline{y}, return/f] \end{array}\right) \Rightarrow C[x/f]\right) \end{array}\right) + \left(\begin{array}{c} Q_{mn_B}[this/k, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \\ \wedge \\ (R_{mn_B}[this/k, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}, return/f] \Rightarrow S \downarrow \{k, x, \overline{z}\}[x/f]) \end{array}\right) * \left(S - \{k, x, \overline{z}\}\right)$$

Invocação de Método Puro com Retorno Puro em Objecto Linear. Para a invocação de métodos puros a um objecto linear, com retorno puro, a sua pré-condição mais fraca é semelhante à anterior. Contudo, uma vez que o objecto ao qual está a ser efectuada a invocação é linear, temos de garantir na fórmula linear resultante, que não se está a realizar uma invocação a uma referência nula ($k \neq \texttt{null}$), contrariamente ao caso anterior onde esta situação é verificada na fórmula pura.

[invocação pura - mp/rp/ol]

$$wp\,(x = k.mn(\overline{y},\overline{z}), C + S) = I_c[this/k] \Rightarrow \left(\begin{array}{c} Q_{mn_A}[this/k, \overline{p_1}/\overline{y}] \\ \wedge \\ (R_{mn_A}[this/k, \overline{p_1}/\overline{y}, return/f] \Rightarrow C[x/f]) \end{array}\right) + \left(\begin{array}{c} k \neq \texttt{null} \wedge Q_{mn_B}[this/k, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \\ \wedge \\ \left(\left(\begin{array}{c} k \neq \texttt{null} \\ \wedge \\ R_{mn_B}[this/k, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}, return/f] \end{array}\right) \Rightarrow S \downarrow \{k, x, \overline{z}\}[x/f]\right) \end{array}\right) * \left(S - \{k, x, \overline{z}\}\right)$$

Invocação de Método void Puro em Objecto Puro. Quanto às chamadas de métodos puros sem retorno a um objecto puro, a pré-condição mais fraca é um caso particular de chamada de métodos puros a objectos puros com retorno puro (porque o tipo `void` é um tipo primitivo da linguagem, logo é puro), onde simplesmente não mencionamos o seu retorno e não ocorre nenhuma afectação a uma variável, sendo estes elementos eliminados dessa regra.

[invocação pura sem retorno - mp/op]

$$wp\,(k.mn(\overline{y},\overline{z}),C+S) = I_c[this/k] \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} k \neq \texttt{null} \wedge Q_{mn_A}[this/k,\overline{p_1}/\overline{y}] \\ \wedge \\ \left(\left(\begin{array}{c} k \neq \texttt{null} \\ \wedge \\ R_{mn_A}[this/k,\overline{p_1}/\overline{y}] \end{array}\right) \Rightarrow C\right) \end{array}\right) \\ + \\ \left(\left(\begin{array}{c} Q_{mn_B}[this/k,\overline{p_1}/\overline{y},\overline{p_2}/\overline{z}] \\ \wedge \\ (R_{mn_B}[this/k,\overline{p_1}/\overline{y},\overline{p_2}/\overline{z}] \Rightarrow S \downarrow \{k,\overline{z}\}) \end{array}\right)\right) \\ * \\ S - \{k,\overline{z}\} \end{array}\right)$$

Invocação de Método void Puro em Objecto Linear. Para as chamadas de métodos puros sem retorno a um objecto linear, a sua pré-condição mais fraca é idêntica à da chamada de métodos puros com retorno puro em objectos lineares, não mencionando o seu retorno, sendo eliminado da regra, à semelhança da regra anterior.

[invocação pura sem retorno - mp/ol]

$$wp\,(k.mn(\overline{y},\overline{z}),C+S) = I_c[this/k] \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} Q_{mn_A}[this/k,\overline{p_1}/\overline{y}] \\ \wedge \\ (R_{mn_A}[this/k,\overline{p_1}/\overline{y}] \Rightarrow C) \end{array}\right) \\ + \\ \left(\left(\begin{array}{c} k \neq \texttt{null} \wedge Q_{mn_B}[this/k,\overline{p_1}/\overline{y},\overline{p_2}/\overline{z}] \\ \wedge \\ \left(\left(\left(\begin{array}{c} k \neq \texttt{null} \\ \wedge \\ R_{mn_B}[this/k,\overline{p_1}/\overline{y},\overline{p_2}/\overline{z}] \end{array}\right) \Rightarrow S \downarrow \{k,\overline{z}\}\right)\right) \end{array}\right)\right) \\ * \\ S - \{k,\overline{z}\} \end{array}\right)$$

Invocação de Método com Retorno Linear em Objecto Linear. Quanto às chamadas de métodos a um objecto linear, com tipo de retorno linear, a pré-condição mais fraca é semelhante à criação de objectos lineares, sendo necessário garantir adicionalmente, que não se está a efectuar uma chamada de método a uma referência nula ($k \neq \texttt{null}$). Temos também, de forma semelhante, que o retorno do método é substituído por um nome fresco f, uma vez que estamos a afectar à variável um novo valor, que corresponde ao resultado do método, que não existe na pré-condição, à semelhança da regra anterior. Tal como nas outras regras sobre invocações de métodos, são assumidos os invariantes da classe na pré-condição mais fraca.

$$\frac{S \downarrow \{\overline{z}\backslash x\} = \emptyset \qquad \texttt{true} + (k \neq \texttt{null} \wedge R_{mn_B}[this/k, \overline{p_1}/\overline{y}, return/f]) \Rightarrow S \downarrow \{k, x, \overline{z}\}[x/f]}{wp\,(x = k.mn(\overline{y}, \overline{z}), C + S) = I_c[this/k] \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} Q_{mn_A}[this/k, \overline{p_1}/\overline{y}] \\ \wedge \\ (R_{mn_A}[this/k, \overline{p_1}/\overline{y}] \Rightarrow C[x/f]) \end{array} \right) \\ + \\ \left(\begin{array}{c} \left(\begin{array}{c} k \neq \texttt{null} \\ \wedge \\ Q_{mn_B}[this/k, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \end{array} \right) \\ * \\ S - \{k, x, \overline{z}\} \end{array} \right) \end{array} \right)} \texttt{[invocação método - rl/ol]}$$

Invocação de Método Linear com Retorno Puro em Objecto Linear. Para as chamadas de métodos lineares a um objecto linear com retorno puro, a pré-condição mais fraca é semelhante à da invocação de métodos com retorno linear em objecto linear.

Como podemos observar, esta regra é bastante intuitiva. Suponhamos que, ao executar um método, estamos num estado que satisfaz uma dada condição, digamos, $D + T$ e que terminamos num estado com condição $C + S$. Para executar o método, em primeiro lugar não podemos estar a efectuar uma invocação a uma referência nula ($k \neq \texttt{null}$), senão ocorreria uma excepção. Temos também de satisfazer as pré-condições do método, tanto na parte pura ($Q_{mn_A}[this/k, \overline{p_1}/\overline{y}]$), como na parte linear ($Q_{mn_B}[this/k, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}]$), ou alguma condição que implique essas pré-condições, pois este só pode executar, se estas se verificarem. Assim sendo, a condição $D + T$ tem de implicar as pré-condições do método (regra da dedução da lógica de Hoare).

Ao ser chamado o método, apenas pode ser alterado o estado do objecto e dos parâmetros que foram passados em argumento. Como tal, todos os factos sobre zonas do *heap* que não são afectadas por esta chamada permanecem na mesma ($S - \{k, x, \overline{z}\}$).

Adicionalmente, após efectuarmos a chamada do método, este garante um conjunto de pós-condições, possivelmente sobre o retorno do método e ainda sobre o estado do objecto. Contudo, nada pode garantir sobre os seus parâmetros porque estes foram consumidos pelo método ($S \downarrow \{\overline{z}\} = \emptyset$). Esta pós-condição tem então de implicar a pós-condição $C + S$ (regra da dedução da lógica de Hoare). Portanto, a parte pura da pós-condição do método tem de implicar C ($R_{mn_A}[this/k, \overline{p_1}/\overline{y}, return/f] \Rightarrow C[x/f]$), verificando-se o mesmo para a parte linear, envolvendo novamente apenas as partes do *heap* que estão sujeitas a alteração de estado pela chamada do método, como ilustra a premissa da regra ($(k \neq \texttt{null} \wedge R_{mn_B}[this/k, \overline{p_1}/\overline{y}, return/f]) \Rightarrow S \downarrow \{k, x, \overline{z}\}[x/f]$).

Uma vez que o retorno é puro, podem ocorrer factos sobre o retorno em ambos os lados da pós-condição dual do método, ou seja, tanto na fórmula pura, como na linear,

sendo por isso efectuada a sua substituição por um nome fresco, em ambos os lados da fórmula dual da pós-condição do método. Temos ainda nas premissas, também devido ao facto de o retorno ser puro, que a variável x não é excluída do conjunto dos parâmetros lineares ($\overline{z}$), aquando da verificação de situações de *aliasing* destes parâmetros. Quanto aos invariantes da classe do objecto ao qual está a ser invocado o método ($I_c[this/k]$), como quando o objecto foi instanciado os seus invariantes foram garantidos, e como qualquer chamada de método ao referido objecto os preserva, podemos assumir que estes são verdade ao início da execução do método, como se encontra na regra.

Sendo assim, podemos constatar que esta regra permanece coerente e correcta, muito embora não tenha sido realizada uma prova formal (Secção 5.1).

$$\frac{S \downarrow \{\overline{z}\} = \emptyset \qquad \texttt{true} + \left((k \neq \texttt{null} \wedge R_{mn_B}[this/k, \overline{p_1}/\overline{y}, return/f]) \Rightarrow S \downarrow \{k, x, \overline{z}\}[x/f]\right)}{wp\,(x = k.mn(\overline{y}, \overline{z}), C + S) = I_c[this/k] \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} Q_{mn_A}[this/k, \overline{p_1}/\overline{y}] \\ \wedge \\ (R_{mn_A}[this/k, \overline{p_1}/\overline{y}, return/f] \Rightarrow C[x/f]) \end{array}\right) \\ + \\ \left(\begin{array}{c} \left(\begin{array}{c} k \neq \texttt{null} \\ \wedge \\ Q_{mn_B}[this/k, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \end{array}\right) \\ * \\ S - \{k, x, \overline{z}\} \end{array}\right) \end{array}\right)}$$

[invocação linear - ml/rp/ol]

Invocação de Método void Linear. Quanto às chamadas de métodos lineares sem retorno, a pré-condição mais fraca é um caso particular de chamada de método linear com retorno puro a um objecto linear, onde simplesmente não mencionamos o seu retorno e não ocorre nenhuma afectação a uma variável, sendo estes elementos eliminados dessa regra.

$$\frac{S \downarrow \{\overline{z}\} = \emptyset \qquad \texttt{true} + \left((k \neq \texttt{null} \wedge R_{mn_B}[this/k, \overline{p_1}/\overline{y}]) \Rightarrow S \downarrow \{k, \overline{z}\}\right)}{wp\,(k.mn(\overline{y}, \overline{z}), C + S) = I_c[this/k] \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} Q_{mn_A}[this/k, \overline{p_1}/\overline{y}] \\ \wedge \\ (R_{mn_A}[this/k, \overline{p_1}/\overline{y}] \Rightarrow C) \end{array}\right) \\ + \\ \left(\begin{array}{c} \left(\begin{array}{c} k \neq \texttt{null} \\ \wedge \\ Q_{mn_B}[this/k, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \end{array}\right) \\ * \\ S - \{k, \overline{z}\} \end{array}\right) \end{array}\right)}$$

[invocação linear sem retorno - ml/ol]

Sincronização Pura. No que diz respeito ao cálculo da pré-condição mais fraca para o comando **synchronized**, quando o objecto ao qual se está a adquirir o *lock* é puro, temos de garantir, para além da pré-condição mais fraca do seu corpo, que não se está a adquirir um *lock* a uma referência nula, sendo este facto adicionado à parte pura da fórmula dual.

[sincronização pura]

$$wp\,(\textbf{synchronized}(x)\ ST, C+S) = \big(x \neq \texttt{null} \wedge wp\,(ST,\ C+S)_A\big)\ +\ wp\,(ST,\ C+S)_B$$

Sincronização Linear. Quanto ao cálculo da pré-condição mais fraca para o comando **synchronized**, quando o objecto ao qual se está a adquirir o *lock* é linear, temos de garantir na parte linear, de forma semelhante ao caso puro, que não se está a adquirir um *lock* a uma referência nula.

[sincronização linear]

$$wp\,(\textbf{synchronized}(x)\ ST, C+S) = wp\,(ST,\ C+S)_A + (x \neq \texttt{null} \wedge wp\,(ST,\ C+S)_B)$$

Para verificar que um programa SpecJava está de acordo com a sua especificação, os triplos de Hoare seguintes têm de ser válidos:

$$\forall_{mn} : \{Q_{mn} \wedge I_c\}\ ST\ \{R_{mn} \wedge I_c\}$$

$$\forall_{cn} : \{Q_{cn}\}\ ST\ \{R_{cn} \wedge I_c\} \quad ,$$

onde ST é o corpo do procedimento, constituído por um bloco de comandos e I_c corresponde aos invariantes da classe. Sendo assim, os métodos têm de preservar os invariantes da classe. Já os construtores têm de garantir, para além das suas pós-condições, os invariantes da classe.

Suponhamos que temos um programa SpecJava, queremos então verificar se este é válido de acordo com a sua especificação usando o cálculo de pré-condições mais fracas referido acima. O processo de verificação é modular, isto é, apenas é verificado um procedimento de cada vez. Considerando o corpo de um procedimento desta linguagem como a sequência de comandos $s_1, s_2, \ldots, s_n$ com a pré-condição $\{Q\}$ e pós-condição $\{R\}$, ao aplicar a regra do cálculo wp que diz respeito à composição, e para cada comando aplicar as respectivas regras de cálculo, como resultado final obtém-se uma pré-condição mais geral ($\{Q_0\}$), como foi descrito na Secção 2.2.3. Após este processo, para o programa ser válido de acordo com a sua especificação, a pré-condição do procedimento tem de implicar a mais geral ($Q \Rightarrow Q_0$).

As fórmulas obtidas no processo de geração de condições de verificação, são fórmulas numa lógica proposicional, contrariamente a outras situações onde são geradas

fórmulas numa lógica de primeira ordem devido à existência de quantificadores, que não estão presentes na linguagem de especificação leve desenvolvida nesta dissertação, sendo por isso reduzidas a um cálculo proposicional. Uma fórmula em lógica proposicional é dita satisfazível se podem ser atribuídos valores lógicos às suas variáveis de forma a que esta seja verdade. Este problema de satisfação de variáveis booleanas é NP-completo, no entanto, é decidível e pode ser resolvido recorrendo a um SAT-Solver. Contudo, as fórmulas obtidas no nosso cálculo contêm predicados e funções não interpretadas, que requerem o suporte de um conjunto de teorias para serem resolvidas. Para obter soluções para estes problemas, é comum recorrer a SMT-Solvers [DMB09].

Apesar de ser um problema NP-completo, existem algoritmos finitos que permitem obter a solução para este problema. Porém, o tempo necessário à execução desses algoritmos pode ser demasiado elevado devido ao tamanho das fórmulas a ser verificadas. Contudo, pensamos que esta situação não constitui um problema na solução desenvolvida, uma vez que esta é modular, sendo verificado procedimento a procedimento, e o tamanho das fórmulas não é muito elevado.

```
1  public class Math {
2      public static pure int abs(int x)
3          ensures !return:neg
4      {   if (x > 0) return x;
5          else return -x;
6      }
7  }
```

$$wp\left(\begin{array}{l}\textbf{if } (x>0) \textbf{ return } x\\ \textbf{else return } -x\end{array}, \neg Neg(\textbf{return}) + \emptyset\right) =$$

$$= \begin{array}{l}\Big(>(x,0) \Rightarrow wp\big(\textbf{return } x, \neg Neg(\textbf{return}) + \emptyset\big)\Big) \wedge \\ \Big(\neg >(x,0) \Rightarrow wp\big(\textbf{return } -x, \neg Neg(\textbf{return}) + \emptyset\big)\Big)\end{array} =$$

$$= \Big(\Big(>(x,0) \Rightarrow \neg Neg(x)\Big) \wedge \Big(\neg >(x,0) \Rightarrow \neg Neg(-x)\Big)\Big) + \emptyset =$$

$$= \Big(\Big(>(x,0) \Rightarrow \neg <(x,0)\Big) \wedge \Big(\neg >(x,0) \Rightarrow \neg <(-x,0)\Big)\Big) + \emptyset \equiv (1)$$

$$VC = \left\{\Big(\top + \emptyset\Big) \Rightarrow \Big(\Big(\Big(>(x,0) \Rightarrow \neg <(x,0)\Big) \wedge \Big(\neg >(x,0) \Rightarrow \neg <(-x,0)\Big)\Big) + \emptyset\Big)\right\}$$

$$(1) \equiv \Big(\Big(>(x,0) \Rightarrow \geq(x,0)\Big) \wedge \Big(\leq(x,0) \Rightarrow \neg >(x,0)\Big)\Big) + \emptyset =$$

$$= \Big(\Big(>(x,0) \Rightarrow \geq(x,0)\Big) \wedge \Big(\leq(x,0) \Rightarrow \leq(x,0)\Big)\Big) + \emptyset = \top + \emptyset$$

$$\Vdash \big(\top + \emptyset\big) \Rightarrow \big(\top + \emptyset\big) \qquad \Vdash \big(\top \Rightarrow \top\big) + \emptyset$$

Figura 3.10: Exemplo de geração de VC

A Figura 3.10 exemplifica a aplicação do cálculo wp proposto, a um método muito simples, que calcula o módulo de um número inteiro, obtendo como resultado final um conjunto de condições de verificação, que neste caso, é composto apenas por uma condição. Esta condição de verificação diz respeito ao facto de a pré-condição de um método ter de implicar a pré-condição mais fraca. Como podemos observar, esta é válida pois ambas as partes da fórmula são válidas. Assim sendo, o procedimento está correcto, face à sua especificação.

As condições de verificação (VCs) obtidas resultam de um processo de reescrita das expressões da linguagem Java em predicados na lógica proposicional, como podemos observar. As propriedades da linguagem de especificação também são reescritas seguindo a mesma aproximação. Esta reescrita é efectuada seguindo um conjunto de regras, como ilustramos na Figura 3.11.

$$
\begin{array}{l}
x > 0 \Rrightarrow >(x,0) \\
x < 0 \Rrightarrow <(x,0) \\
x >= 0 \Rrightarrow \geq(x,0) \\
x <= 0 \Rrightarrow \leq(x,0) \\
x == 0 \Rrightarrow =(x,0) \\
x == true \Rrightarrow x \Leftrightarrow \top \\
x == false \Rrightarrow x \Leftrightarrow \bot \\
x == null \Rrightarrow Null(x) \\
x \Rrightarrow True(x) \\
!x \Rrightarrow \neg True(x)
\end{array}
\qquad
\begin{array}{l}
x : pos \Rrightarrow Pos(x) \\
x : neg \Rrightarrow Neg(x) \\
x : zero \Rrightarrow Zero(x) \\
x : true \Rrightarrow True(x) \\
x : false \Rrightarrow False(x) \\
x : null \Rrightarrow Null(x) \\
x : sn \Rrightarrow Sn(x)
\end{array}
$$

$$
\begin{array}{l}
Pos(x) \equiv >(x,0) \\
Neg(x) \equiv <(x,0) \\
Zero(x) \equiv =(x,0) \\
sn \in \text{nome de estado}
\end{array}
$$

Figura 3.11: Exemplo de Reescrita em Predicados Unários da Lógica Proposicional

As VCs, são posteriormente submetidas a um SMT-solver que possui um conjunto de teorias (e.g. teoria dos números reais, números inteiros, vectores de bits) que contêm propriedades sobre cada teoria, como as ilustradas na Figura 3.12. Uma fórmula dual é considerada válida se tanto a parte pura como a linear são válidas. A parte linear é válida se todas as fórmulas que a compõem são válidas. Se as VCs não forem todas válidas, então o programa não está de acordo com a sua especificação, não sendo compilado.

$$\nVdash <(x,x) \quad \nVdash >(x,x) \quad \Vdash \leq(x,x) \quad \Vdash \geq(x,x)$$

$$\leq(x,y) \equiv \geq(y,x) \qquad <(x,y) \equiv >(y,x)$$

$$\leq(x,y) \equiv <(x,y) \vee =(x,y)$$

$$\geq(x,y) \equiv >(x,y) \vee =(x,y)$$

$$+(x,y) \equiv +(y,x)$$

$$>(x,y) \equiv \neg =(x,y) \wedge \neg <(x,y)$$

$$=(x,y) \equiv \neg >(x,y) \wedge \neg <(x,y)$$

Figura 3.12: Exemplo de Lemas sobre Teoria de Números Inteiros e Reais

3.3.2 Desafios

De forma a realizar a extensão do cálculo de pré-condições mais fracas para uma linguagem orientada a objectos, tiveram de ser resolvidos alguns desafios, tais como: forma de tratamento dos ciclos, lidar com situações de *aliasing*.

Como explicamos na Secção 2.2.3, para provar a correcção de um programa que contenha ciclos, decidimos usar a noção de invariante de ciclo. Como tal, surgiram duas opções a seguir, ou o utilizador anotava o programa com o invariante do ciclo, ou este seria calculado automaticamente, não necessitando da anotação [RCK05].

Um problema da primeira abordagem passa pela obrigatoriedade da anotação, tornando a especificação mais complexa para o utilizador. Na segunda, o cálculo automático de invariantes pode ser realizado seguindo duas aproximações: estática ou dinâmica.

Na aproximação dinâmica [ECGN02], o programa é executado com vários inputs e infere invariantes pelos traços de execução do programa. Esta não se adequava ao nosso problema, uma vez que pretendemos analisar o programa estaticamente, sem o executar.

Segundo a aproximação estática, estas operam no código do programa e existem diversas abordagens, sendo as mais conhecidas as baseadas em interpretação abstracta, que utilizam um algoritmo iterativo de ponto fixo e usa heurísticas para a sua convergência [SSM04]. Apesar da sua conveniência pois evitam as anotações, estas técnicas podem demorar um certo tempo a convergir para inferir os invariantes e realizam um cálculo aproximado, o que pode resultar num invariante que, apesar de estar correcto, pode não ser o mais apropriado.

Com base nestes casos e uma vez que a complexidade da especificação não aumenta significativamente, decidimos optar pelo uso de anotações de invariante nos ciclos. Na

realidade, resulta apenas de uma anotação extra para cada ciclo, face ao que teria de ser especificado sem recorrer a tal anotação. Contudo, seria interessante estudar a abordagem de inferência automática de invariantes.

Quanto à situação de *aliasing*, esta pode ocorrer nas linguagens de programação como o Java, onde podemos ter múltiplos nomes a referir uma mesma célula de memória, o que constitui um problema porque ao alterar o valor de um nome podemos estar a alterar todos os outros que apontem para o mesmo objecto. Este problema torna o processo de verificação de um programa mais complicado, porque tal situação pode levar, por exemplo, a quebra de invariantes, como ilustra a Figura 3.13.

```
 1  public class Exposer {
 2      private Buffer f;
 3      invariant + !f:null;
 4      invariant + !f:full;
 5      ...
 6      public Buffer expose()
 7          + !return:null
 8      {
 9          return f;
10      }
11  }
```

```
 1  public class Main {
 2      public static void main(String[] args) {
 3          Exposer a = new Exposer();
 4          Buffer x = a.expose();
 5          // alias - x e f apontam para o mesmo
 6          x.write(3);
 7          // pode quebrar o invariante devido ao aliasing
 8          // pois a classe Exposer não tem controlo sobre
 9          // as escritas que dizem respeito à variável x
10      }
11  }
```

Figura 3.13: Exemplo de Situação de *aliasing*

Existem diversas aproximações para lidar com o problema de *aliasing*, como por exemplo: proibir totalmente *aliasing*; usar lógica de separação [OHRY01, PB05]; *ownership model* e variantes [SD03, CD02]; separar objectos com estado dos objectos imutáveis.

A primeira aproximação é muito restritiva, não constituindo uma boa solução para o nosso problema.

Na nossa abordagem, inspirados na lógica de separação, separamos as propriedades dos objectos lineares dos puros (imutáveis). Assim sendo, na parte pura podem ocorrer situações de *aliasing*, uma vez que o seu estado não altera ou são efectuadas passagens por cópia, como é o caso dos tipos primitivos onde não ocorrem múltiplas referências para um tipo primitivo. Na parte linear da nossa linguagem, é efectuada a detecção de situações de *aliasing* pelas regras do cálculo wp, assegurando que o cálculo é usado de forma linear e que as zonas do *heap* são disjuntas, levando a emissão de erros quando tal não ocorre.

Tomando como exemplo o caso da Figura 3.13, podemos observar que a situação de *aliasing* irá ser detectada aquando da verificação da especificação do método `expose`, pela regra do cálculo wp que diz respeito ao retorno linear. Como temos invariantes sobre `f`, as quais o método `expose` tem de garantir como sua pós-condição, ao efectuar o cálculo da pré-condição mais fraca do comando de retorno, as suas premissas não vão ser satisfeitas. Tal deve-se ao facto de a pós-condição ter informação sobre uma variável que foi exposta ao exterior, ou seja, a classe `Exposer` perdeu o controlo que tinha sobre esse objecto linear, portanto, nada pode garantir sobre este objecto, pois poderá ocorrer *aliasing* e serem efectuadas operações sobre este, externamente (`x.write(3)`).

3.3.3 Consistência das Regras do Cálculo WP

Com base nas regras apresentadas, podemos observar que esta extensão ao cálculo wp, originalmente proposto por Dijkstra, é bastante intuitiva. Apesar de não ter sido efectuada uma prova formal, podemos constatar que as regras permanecem coerentes e correctas. Os casos mais simples dizem respeito aos comandos independentes da separação, bem como a afectação pura, que são derivações directas do cálculo original. Os não triviais dizem respeito à instanciação e chamadas de métodos. Contudo, a pré-condição mais fraca para estes casos corresponde às suas pré-condições, pois estas são as condições necessárias à sua execução. É necessário ainda garantir que não estão a ser efectuadas invocações nulas, e que a pós-condição do método implica a pós-condição a ser verificada. Para efectuar a prova formal destas regras, é necessário realizar um estudo mais aprofundado sobre a semântica da lógica dual, definindo-a formalmente. Este estudo constitui um dos pontos mais importantes a desenvolver como trabalho futuro (Secção 5.1).

Capítulo 4

SpecJava: Uma Extensão ao Compilador do Java com Verificação de Especificações Leves

Neste capítulo apresentamos os detalhes de implementação do protótipo desenvolvido nesta dissertação, que permite a verificação de um programa SpecJava face à sua especificação. A extensão da linguagem Java foi implementada recorrendo à ferramenta Polyglot, que é um compilador extensível para esta linguagem, o qual descrevemos em traços gerais na Secção 4.1. Na Secção 4.2, explicamos alguns pormenores da implementação. Por último apresentamos exemplos de programas validados pelo protótipo desenvolvido (Secção 4.4) e concluímos com uma análise comparativa entre as ferramentas e linguagens de programação descritas na Secção 2.3 versus o nosso protótipo (Secção 4.5).

4.1 Polyglot

O Polyglot [NCM03, NQ08] é uma ferramenta que implementa um compilador extensível, para a linguagem Java 1.4. Esta ferramenta é também implementada em Java, e na sua forma mais simples apenas efectua verificação semântica do Java, podendo ser estendida de forma a definir alterações no processo de compilação, incluindo mudanças na árvore de sintaxe abstracta (AST) e na análise semântica efectuada.

Esta ferramenta tem vindo a ser usada em vários projectos e revelou-se bastante útil no desenvolvimento de compiladores para novas linguagens, semelhantes ao Java.

Uma extensão em Polyglot é um compilador de código para código, ou seja, aceita um programa escrito na nova linguagem, traduzindo-o para código Java que, posteriormente, pode ser convertido para bytecode com um compilador Java.

O primeiro passo no processo de compilação constitui a análise sintáctica do código fonte, utilizando a gramática definida para a nova linguagem como extensão à gramática base do Java, e produz uma árvore de sintaxe abstracta do programa. A gramática é definida usando a ferramenta PPG, um preprocessador do gerador CUP, que permite a alteração da gramática base do Java através da introdução, modificação ou remoção de produções e símbolos.

Como passo seguinte neste processo, é efectuado um conjunto de passagens sobre a AST, produzindo uma nova, de forma totalmente funcional, não modificando de um modo destrutivo a AST anterior, sendo durante estas passagens realizada a análise semântica e tradução para Java. Algumas passagens, como por exemplo, a que diz respeito à verificação de tipos, podem interromper o processo de compilação emitindo erros em vez de criarem uma nova AST. Uma extensão pode então adicionar, remover ou alterar passagens para lidar com as novas características da sua linguagem.

4.2 Implementação

Nesta secção enfatizamos os pontos mais importantes no desenvolvimento do protótipo. Após estudada a arquitectura do Polyglot e a sua implementação, concluímos que seria necessário efectuar como primeiro passo, a extensão à gramática do Java com a sintaxe apresentada na Secção 3.2, recorrendo à ferramenta PPG, descrita na secção anterior.

Posteriormente, foi necessário criar as classes Java correspondentes aos novos nós introduzidos na AST. No que diz respeito à lógica dual desenvolvida foram implementadas classes, para além das que representam os nós da AST de uma fórmula dual, que têm como objecto representar genericamente informação sobre uma fórmula em lógica proposicional clássica. Sendo assim, foram criadas classes para representar os diversos tipos de termos (constantes, funções, variáveis), os predicados e as operações de conjunção, disjunção, implicação, equivalência e negação. Foi também criada uma arquitectura de *visitors* que permite efectuar alterações nas fórmulas, como por exemplo convertê-la para forma normal conjuntiva e efectuar as substituições necessárias aquando do cálculo de pré-condições mais fracas.

Após este passo de modo a integrar o cálculo wp desenvolvido na Secção 3.3.1, foi então criada uma passagem adicional que visita a AST e tem como finalidade calcular as pré-condições mais fracas de cada procedimento. Esta passagem adicional é efectuada após a passagem sobre a AST que realiza a verificação de tipos, que foi estendida de modo a verificar a conformidade das especificações, como por exemplo, não poder

referir um estado/propriedade de um objecto que não esteja definida na classe do objecto, garantir que os tipos primitivos só podem estar associados às propriedades base, garantir que não ocorrem predicados sobre variáveis lineares na parte pura de uma fórmula dual, etc.

Como passo seguinte as condições de verificação geradas pelo cálculo de pré-condições mais fracas são submetidas a um SMT-Solver, neste caso o SMT CVC3 [BT07]. Foi efectuada uma pesquisa sobre vários SMT Solvers existentes destacando-se os SMT Solvers Boolector[1], MathSAT[2], Beaver[3], STP[4], Z3[5] e CVC3. Contudo o que se apresentou mais promissor e adequado foi o CVC3, uma vez que este suporta teorias sobre números inteiros e reais, necessária para provar os programas em SpecJava. Boolector, Beaver e STP apenas suportam a teoria de vectores de bits que não é adequada para raciocinar sobre programas que contenham números reais, como é o caso. O SMT MathSAT, apesar de suportar as teorias de números inteiros e reais, não suporta o operador de divisão, o que seria algo limitativo na análise dos programas. Quanto ao Z3 apesar de suportar as mesmas teorias que o CVC3, revelou-se à partida mais lento que este ao provar as condições de verificação, tomando assim a decisão de usar o SMT CVC3. A submissão das condições de verificação ao SMT-Solver é efectuada numa nova passagem sobre a AST, traduzindo a representação genérica das fórmulas para o formato específico do Solver que verifica se estas são válidas. Caso não sejam válidas o programa não é compilado com sucesso, uma vez que não está de acordo com a sua especificação, sendo emitida a causa do erro associada à localização do código fonte, juntamente com um contra-exemplo facilitando a correcção do programa/especificação pelo utilizador.

A representação genérica das fórmulas permite assim, para além de uma maior flexibilidade caso seja necessário estender o sistema, tornar possível a integração de qualquer outro SMT-Solver de uma forma relativamente simples, bastando criar uma nova classe para o SMT-Solver que efectua a tradução para o seu formato interno e a verificação de validade das fórmulas. Esta independência é bastante importante pois permite ser realizada uma análise mais detalhada sobre como se comporta o sistema desenvolvido na presença de diferentes SMT-Solver, e até a combinação de múltiplos Solvers juntando as suas melhores características. Outro ponto passa pelo facto de recentemente estar a ser adoptado um formato genérico de input para os SMT-Solvers[6],

[1] `http://fmv.jku.at/boolector/`
[2] `http://mathsat4.disi.unitn.it/`
[3] `http://uclid.eecs.berkeley.edu/newwiki/beaver/start`
[4] `http://sites.google.com/site/stpfastprover/`
[5] `http://research.microsoft.com/en-us/um/redmond/projects/z3/`
[6] `http://smtlib.org/`

tornando assim mais fácil a sua comparação e utilização, o que seria facilmente adicionado ao nosso sistema através do mesmo método de tradução.

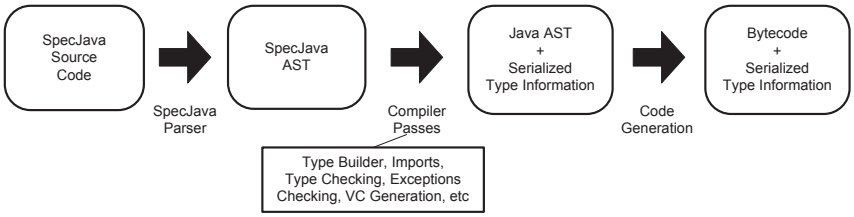

Figura 4.1: Arquitectura do Compilador

A Figura 4.1 esquematiza a arquitectura do compilador para a linguagem SpecJava desenvolvida, que concretiza a implementação de uma extensão do Polyglot e ilustra as várias etapas do processo descrito acima.

4.3 Listagem de Ficheiros

Nesta secção é apresentada uma listagem dos ficheiros Java que constituem o protótipo, juntamente com o número de linhas de código de cada ficheiro. No seu total compõem aproximadamente 11500 linhas de código[7].

```
./src:
specjava

./src/specjava:
ast
extension
ExtensionInfo.java (141)
frontend
lex
logic
Main.java (21)
package.html (5)
parse
Topics.java (14)
types
UniqueID.java (26)
util
Version.java (13)
visit

./src/specjava/ast:
extension
factory
package.html (5)
specification
```

[7] http://ctp.di.fct.unl.pt/~tsantos/tools/specjava.zip

```
./src/specjava/ast/extension:
SpecJavaConstructorDecl_c.java (330)
SpecJavaDo_c.java (56)
SpecJavaFor_c.java (62)
SpecJavaLoop.java (20)
SpecJavaMethodDecl_c.java (306)
SpecJavaProcedureDecl.java (52)
SpecJavaSynchronizedDel_c.java (26)
SpecJavaWhile_c.java (56)

./src/specjava/ast/factory:
SpecJavaDelFactory_c.java (15)
SpecJavaExtFactory_c.java (90)
SpecJavaExtFactory.java (14)
SpecJavaNodeFactory_c.java (258)
SpecJavaNodeFactory.java (95)

./src/specjava/ast/specification:
clazz
formula
procedure
SpecificationNode_c.java (16)
SpecificationNode.java (22)

./src/specjava/ast/specification/clazz:
ClassDefineNode_c.java (157)
ClassDefineNode.java (39)
ClassInvariantNode_c.java (99)
ClassInvariantNode.java (14)
ClassSpecificationNode_c.java (15)
ClassSpecificationNode.java (13)

./src/specjava/ast/specification/formula:
AmbiguousFormulaNode_c.java (54)
AmbiguousFormulaNode.java (10)
atomic
BinaryFormulaNode_c.java (114)
BinaryFormulaNode.java (42)
DualNode_c.java (161)
DualNode.java (35)
FormulaNode_c.java (90)
FormulaNode.java (39)
UnaryFormulaNode_c.java (76)
UnaryFormulaNode.java (23)

./src/specjava/ast/specification/formula/atomic:
FieldPath.java (42)
NormalPropertyNode_c.java (100)
NormalPropertyNode.java (17)
SinglePropertyNode_c.java (44)
SinglePropertyNode.java (16)
SpecialPropertyNode_c.java (84)
SpecialPropertyNode.java (24)
TruthConstantNode_c.java (38)
TruthConstantNode.java (15)
```

```
./src/specjava/ast/specification/procedure:
Assume_c.java (62)
Assume.java (18)
LoopInvariantNode_c.java (52)
LoopInvariantNode.java (16)
ProcedureAssertionNode_c.java (66)
ProcedureAssertionNode.java (26)
StaticAssert_c.java (62)
StaticAssert.java (18)

./src/specjava/extension:
package.html (5)
SpecJavaProcedureDeclExt_c.java (42)
SpecJavaProcedureDeclExt.java (12)
statement
WPCalculusException.java (34)
WPCalculus.java (3175)

./src/specjava/extension/statement:
AssumeExt_c.java (20)
SpecJavaBlockExt_c.java (20)
SpecJavaConstructorCallExt_c.java (33)
SpecJavaEmptyExt_c.java (22)
SpecJavaEvalExt_c.java (21)
SpecJavaIfExt_c.java (20)
SpecJavaLocalDeclExt_c.java (20)
SpecJavaLoopExt_c.java (20)
SpecJavaReturnExt_c.java (20)
SpecJavaStmtExt_c.java (36)
SpecJavaStmtExt.java (22)
SpecJavaSynchronizedExt_c.java (19)
StaticAssertExt_c.java (20)

./src/specjava/frontend:
goals

./src/specjava/frontend/goals:
SpecificationChecked.java (35)
WeakestPreconditionBuilt.java (43)

./src/specjava/lex:
NumberLiteral.java (29)

./src/specjava/logic:
CVCProver.java (541)
DualLogic.java (384)
formula
ProverException.java (24)
Prover.java (14)
Result.java (25)
Utils.java (110)
visit
VisitorException.java (22)


./src/specjava/logic/formula:
```

```
AbstractFormula.java (328)
binary
DualImpl.java (68)
Dual.java (23)
False.java (52)
Formula.java (30)
predicate
term
True.java (52)
unary
UnknownFormula.java (39)

./src/specjava/logic/formula/binary:
And.java (70)
Equivalence.java (70)
Implication.java (70)
Or.java (70)

./src/specjava/logic/formula/predicate:
AbstractPredicate.java (94)
Eq.java (18)
Gt.java (18)
Lt.java (19)
Predicate.java (9)
StatePredicate.java (29)

./src/specjava/logic/formula/term:
function
SpecialTerm.java (69)
Term.java (16)
VariableTerm.java (221)

./src/specjava/logic/formula/term/function:
AbstractFunction.java (89)
Constant.java (70)
Div.java (25)
Function.java (10)
Minus.java (22)
Mod.java (18)
Mul.java (19)
Plus.java (22)

./src/specjava/logic/formula/unary:
Not.java (51)

./src/specjava/logic/visit:
AbstractPLVisitor.java (132)
DistributeOrOverAnd.java (34)
ImplicationsOut.java (26)
NegationsIn.java (47)
PLVisitor.java (65)
Simplifier.java (125)
Substitutor.java (106)
VisitorException.java (24)

./src/specjava/parse:
```

```
Grm.java (10093)
Lexer_c.java (1264)
package.html (5)
specjava.flex (498)
specjava.ppg (527)
specjava_ppg.cup (1672)
sym.java (131)

./src/specjava/types:
BooleanProperty_c.java (56)
BooleanProperty.java (5)
NamedProperty_c.java (77)
NamedProperty.java (3)
NullProperty_c.java (57)
NullProperty.java (5)
NumberProperty_c.java (59)
NumberProperty.java (18)
package.html (5)
Property.java (23)
PropertyNotFoundException.java (22)
SpecJavaClassType.java (87)
SpecJavaConstructorInstance_c.java (74)
SpecJavaContext_c.java (84)
SpecJavaContext.java (15)
SpecJavaFlags.java (9)
SpecJavaMethodInstance_c.java (72)
SpecJavaParsedClassType_c.java (119)
SpecJavaParsedClassType.java (14)
SpecJavaPrimitiveType_c.java (55)
SpecJavaPrimitiveType.java (11)
SpecJavaProcedureInstance.java (23)
SpecJavaReferenceType.java (23)
SpecJavaTypeSystem_c.java (96)
SpecJavaTypeSystem.java (16)

./src/specjava/util:
CollectionUtil.java (48)

./src/specjava/visit:
DeepCopier.java (11)
LocalAssignVisitor.java (34)
package.html (5)
PureVisitor.java (61)
SpecificationChecker.java (25)
VariableVisitor.java (27)
WeakestPreconditionBuilder.java (52)
```

4.4 Exemplos

Nesta secção são apresentados alguns dos exemplos que foram validados por esta ferramenta, o primeiro diz respeito a uma classe matemática (`Math`) que contém métodos para calcular o módulo, o quadrado e o factorial de um número. São apresentadas

duas versões para o cálculo do factorial, sendo uma recursiva (`fact`) e a outra iterativa (`factI`) que utiliza a noção de invariante. No caso do módulo e do quadrado de um número garantimos como pós-condição que o resultado não é negativo. Para o factorial, temos como pré-condição que o número não seja negativo, pois o factorial apenas se encontra definido para números maiores ou iguais a zero, e garantimos como pós-condição que o resultado do factorial é um número positivo, conforme esperado.

O segundo exemplo ilustra uma pilha (`Stack`) de elementos não negativos, sem limite de capacidade, implementada em lista ligada. Neste exemplo é garantido um invariante estrutural implicitamente, uma vez que ao adicionar cada elemento na pilha temos como pré-condição que este seja positivo.

Por último, o terceiro ilustra o exemplo completo da especificação de um ficheiro, onde é necessário seguir um determinado protocolo, ou seja, primeiro é necessário abrir o ficheiro, posteriormente, podem ser efectuadas leituras e escritas, sendo no fim fechado.

4.4.1 Math

```
 1 public class Math {
 2
 3   private pure Math() { }
 4
 5   public static pure int abs(int x)
 6     ensures !return:neg
 7   {
 8     if (x > 0) return x;
 9     else return -x;
10   }
11
12   public static pure int sqr(int x)
13     ensures !return:neg
14   {
15     return x * x;
16   }
17
18   public static pure int fact(int x)
19     requires !x:neg
20     ensures return:pos
21   {
22     if (x == 0) return 1;
23     else {
24       int z = x - 1;
25       int y = fact(z);
```

```
26        return x * y;
27      }
28    }
29
30    public static pure int factI(int x)
31      requires !x:neg
32      ensures return:pos
33    {
34      int i = 1;
35      while (x > 1)
36        invariant i:pos
37      {
38        i = i * x;
39        x = x - 1;
40      }
41      return i;
42    }
43  }
```

4.4.2 Stack

```
1  public class Stack {
2
3    private Entry head;
4
5    class Entry {
6      public Entry(int x)
7        requires x:pos
8      {
9        this(x, null);
10     }
11
12     public Entry(int x, Entry n)
13       requires x:pos
14     {
15       this.x = x;
16       this.next = n;
17     }
18
19     private int x;
20     invariant x:pos;
21     private Entry next;
22
23     public Entry getNext() {
24       return next;
```

```
25      }
26
27      public int getElement()
28        ensures return:pos
29      {
30        return x;
31      }
32    }
33
34    public Stack()
35        ensures + head:null
36    {
37      head = null;
38    }
39
40    public void push(int i)
41      requires i:pos
42      ensures + !head:null
43    {
44      head = new Entry(i, head);
45    }
46
47    public int pop()
48      requires + !head:null
49      ensures return:pos
50    {
51      int res = head.getElement();
52      head = head.getNext();
53      return res;
54    }
55  }
```

4.4.3 File

```
1  public class File {
2    define open;
3
4    public File()
5      ensures + !open
6    {
7      assume + !open;
8    }
9
10   public void open()
11     requires + !open
```

```
12      ensures + open
13    {
14      assume + open;
15    }
16
17    public void write(int b)
18      requires + open
19      ensures + open
20    { }
21
22    public int read()
23      requires + open
24      ensures + open
25    { return 0; }
26
27    public void close()
28      requires + open
29      ensures + !open
30    {
31      assume + !open;
32    }
33
34    public pure void nothing() {}
35
36    public static void main(String[] args) {
37      File f = new File();
38      f.open();
39      f.nothing();
40      f.nothing();
41      f.close();
42      f.nothing();
43      f.open();
44      f = f;
45      //f.open(); // Falha como esperado
46      f.close();
47      //f.close(); // Falha como esperado
48    }
49  }
```

4.5 Análise Comparativa

As ferramentas descritas na Secção 2.3 distinguem-se, na sua generalidade, no que diz respeito a quatro vertentes: linguagem de especificação usada; abrangência à linguagem de programação inerente à verificação; técnicas de verificação usadas; e modo de

verificação.

No que diz respeito à linguagem de especificação usada, ESC/Java2, KRAKATOA, LOOP, JACK e Forge utilizam JML. Na ferramenta KeY, as especificações são descritas em OCL. A ferramenta Forge pode utilizar ainda a sua própria linguagem de especificação, denominada de JFSL. Quanto à linguagem de programação Spec# e à ferramenta jStar, estas possuem a sua própria linguagem de especificação. A linguagem de especificação do Spec# apresenta certas semelhanças ao JML, e a do jStar é a que menos se assemelha às restantes. Na nossa abordagem, a linguagem de especificação é semelhante à do Spec# e JML, mas é mais simples que estas e apresenta uma técnica inovadora onde se separa propriedades puras de lineares, modelando o *heap* na parte linear da fórmula dual por forma a detectar problemas de *aliasing*, o que não pode ser expresso usando estas linguagens. Contrariamente, no Spec# é usada a técnica de *ownership model* para lidar com *aliasing* e são especificadas explicitamente condições de *frame* através da cláusula **modifies** que denota as zonas de um programa que um método pode modificar.

Quanto à abrangência à linguagem de programação, de entre as ferramentas descritas, as que apresentam uma maior cobertura são o ESC/Java2 e Spec#. As ferramentas KRAKATOA, KeY, LOOP e Forge apresentam algumas limitações, não suportando alguns dos aspectos da linguagem Java, como por exemplo, carregamento dinâmica de classes e multithreading. No que diz respeito à nossa abordagem, certas características do Java ainda não são suportadas, como por exemplo, herança, excepções, os comandos **break** e **continue**, que alteram o fluxo normal de execução de um programa.

No que diz respeito às técnicas de verificação usadas, podemos destacar várias abordagens. As ferramentas ESC/Java2, KRAKATOA, LOOP, JACK e Spec# utilizam o cálculo de pré-condições mais fracas, proposto por Dijsktra ou variantes, na geração de condições de verificação. Já a ferramenta KeY utiliza lógica dinâmica, onde o mecanismo de dedução é baseado em execução simbólica do programa. A ferramenta Forge usa a técnica de verificação limitada, que utiliza execução simbólica e reduz o problema a um de satisfação de variáveis booleanas. Quanto à ferramenta jStar, esta combina a ideia de família abstracta de predicados e a ideia de execução simbólica e abstracção, usando lógica de separação. LOOP define uma semântica denotacional no PVS, em contraste com as abordagens seguidas por ESC/Java2, JACK e Spec# que dependem directamente de uma semântica axiomática. A nossa aproximação é baseada numa variante do cálculo de pré-condições mais fracas para linguagens orientadas a objectos, assemelhando-se ao ESC/Java2, JACK e Spec# que também dependem de uma semântica axiomática. Esse cálculo é efectuado directamente a partir da árvore de sintaxe abstracta (AST).

Por último, quanto aos modos de verificação, nas ferramentas ESC/Java2, JACK, KeY, Forge, jStar e Spec# o processo de verificação é automático, assim como no nosso protótipo (SpecJava), onde as condições de verificação geradas são provadas com recurso a um SMT-Solver, em tempo de compilação. Quanto às ferramentas KRAKATOA e LOOP, o processo de verificação é interactivo necessitando da intervenção do utilizador. As ferramentas KeY e JACK suportam ambos os modos.

Estas ferramentas permitem a verificação de programas de uma forma estática e têm um poder expressivo muito elevado, podendo verificar especificações bastante complexas. No entanto, esta é uma das principais razões para a sua rejeição por parte dos programadores, que na sua generalidade, não têm conhecimentos elevados nas áreas de lógica, nem pretendem lidar com todos os mecanismos complexos associados à maior parte destas ferramentas. Outro ponto passa pelo facto de muitas não estarem integradas nas linguagens de programação, o que obriga à utilização das ferramentas em separado do processo de desenvolvimento de um programa. Por outro lado, e constituindo a principal motivação desta dissertação, a nossa aproximação recorre a especificações leves e, apesar de ter um poder expressivo menor, permite ainda assim descrever propriedades interessantes. Além disso, a sua simplicidade é apelativa ao uso por parte dos programadores, e encontra-se integrada na linguagem de programação.

Capítulo 5

Considerações Finais

O objectivo desta dissertação consiste no desenvolvimento de uma extensão ao compilador do Java com verificação de especificações leves. A linguagem de especificação desenvolvida é baseada numa lógica proposicional, e é bastante simples, permitindo aos programadores especificar os seus programas de uma forma relativamente fácil, e verificá-los automaticamente em tempo de compilação.

Neste documento, é descrito de uma forma detalhada o desenvolvimento deste trabalho. Numa fase inicial, foi realizado um estudo do tema da especificação e verificação, salientando os trabalhos mais relevantes que contribuíram para o seu desenvolvimento (Capítulo 2). Foi ainda descrito em mais detalhe aspectos que constituem os alicerces desta dissertação, e apresentado um conjunto de ferramentas e linguagens de programação existentes que têm, igualmente, o objectivo de verificar a correcção de programas de acordo com a sua especificação. O Capítulo 3, após devidamente motivados, introduziu a linguagem de especificação desenvolvida, a lógica subjacente e o mecanismo inerente à verificação de um programa em SpecJava, que estende o cálculo wp proposto por Dijstra, para a linguagem Java, ilustrando exemplos da sua utilização. Finalmente, concluímos no Capítulo 4, destacando os pontos mais importantes no desenvolvimento do protótipo, que estende a ferramenta Polyglot aplicando os conceitos desenvolvidos no capítulo anterior, e onde é efectuada uma análise comparativa entre as várias ferramentas descritas no Capítulo 2 e a abordagem desenvolvida.

5.1 Trabalho Futuro

Nesta secção apresentamos algumas melhorias e desafios para trabalho futuro com base nesta dissertação, destacando-os por ordem de importância.

Estudo Teórico sobre a Lógica Dual: Apesar de termos apresentado a lógica dual, é necessário efectuar um estudo mais detalhado sobre a sua semântica, de forma a poder provar a sua coerência formalmente, bem como a das regras para o cálculo de pré-condições mais fracas. Como tal, este estudo constitui um dos pontos mais importantes para trabalho futuro.

Extensão do Cálculo WP: Apesar do cálculo desenvolvido conseguir demonstrar a potencialidade da nossa solução, é necessário efectuar algumas extensões a este cálculo de modo a permitir construções que foram deixadas de parte nesta dissertação, como é o caso de os mecanismos de herança, comandos **break** e **continue** que alteram o fluxo normal de execução de um programa e lidar com excepções. Destas extensões, o maior desafio recai sobre os comandos **break**, **continue** e mecanismos de excepções. No entanto, julgamos que basta tomar em atenção novos pontos de saída do CFG (*Control Flow Graph*) de um programa, com uma aproximação semelhante ao comando **return**, onde a sua pré-condição mais fraca é independente da pós-condição actual, isto é, o comando diz sempre respeito à pós-condição do método, pois é um ponto de saída do método, e como tal, tem de se verificar a sua pós-condição a seguir a este comando.

Extensão do Suporte de Especificações: Outro ponto de interesse seria suportar especificações ao nível da interface e em ficheiros separados, de modo a permitir especificar as classes da API do Java. Ao nível da interface e juntamente com o suporte de herança, poder-se-ia, por exemplo, verificar se todas as implementações de uma dada interface obedecem a uma especificação.

```
1  specification interface X {
2      ...
3      define sn1 = ...;
4      define sn2;
5
6      public void m1()
7          requires ...
8          ensures ...;
9
10     public Y m2(Z arg)
11         requires ...
12         ensures ...;
13     ...
14 }
```

Figura 5.1: Esboço de Especificação ao Nível da Interface

A Figura 5.1 ilustra uma possível representação de especificação de interface, onde

adicionamos uma palavra reservada **specification**, denotando que está a ser efectuada uma nova especificação da interface `X`. Para as classes core do Java, a abordagem seria semelhante, substituindo a palavra **interface** por **class**. Em ambos os casos são apenas descritos os esqueletos, pois no caso das classes não teríamos acesso ao código fonte da classe, podendo apenas referir as assinaturas dos métodos.

Sintaxe: A solução desenvolvida apresenta algumas restrições quanto à forma como a sintaxe do Java pode ser utilizada. As expressões e comandos têm de estar todas expandidas, não podendo ser efectuadas, por exemplo chamadas de métodos encadeadas ou instanciar parâmetros durante a chamada do procedimento. Para resolver este problema, podem ser efectuadas transformações na árvore sintáctica do programa de forma a expandi-la, mantendo o mesmo significado semântico. Estas transformações podem ser realizadas ao mesmo tempo que o cálculo wp ou numa passagem prévia.

```
1  class A {
2      ...
3      A(B b) { ... }
4      ...
5  }
6  class B { ... }
7
8  class Main {
9      void main(String[] args) {
10         int b = 1;
11         int c = b + (b += 2) + b;
12         A x = new A(new B());
13     }
14 }
```

```
8  class Main {
9      void main(String[] args) {
10         int b = 1;
11         int f$2 = b + 2;
12         int c = b + f$2 + f$2;
13         B f$1 = new B();
14         A x = new A(f$1);
15     }
16 }
```

Figura 5.2: Exemplo de alterações na AST

A Figura 5.2 ilustra a aplicação de tais alterações, onde é substituída a instanciação de `B` por uma declaração local prévia (`f$1`), sendo efectuada a mesma aproximação no caso da declaração da variável `c`.

Para além do referido, existem também limitações quanto à tradução dos ciclos "for" e "do while" para ciclos "while" que terão de ser resolvidas quando for efectuada a extensão do cálculo de pré-condições mais fracas para suportar os comandos **break** e **continue**. A tradução realizada nesta versão assume que estes comandos não ocorrem no corpo do ciclo, pois estes ainda não são suportados pelo cálculo de pré-condições mais fracas desenvolvido. Também assumimos que o comando **return** não

ocorre no corpo dos ciclos "for" e "do while", garantindo assim que a tradução efectuada é correcta. Por forma a contemplar a possibilidade de ocorrência destes comandos, a tradução realizada terá de ser diferente. Por exemplo, no caso dos ciclos "for", as últimas instruções correspondem aos comandos que efectuam actualizações de variáveis para a próxima iteração do ciclo, inclusive quando ocorre o comando **continue**, assim sendo, as instruções de actualização terão de ser adicionadas no momento da tradução, antes de cada **continue** do ciclo "for" em questão. Para os comandos **break** e **return**, tem de se ter em atenção a possibilidade de ocorrência de código que nunca irá ser executado (*unreachable code*) aquando do processo de tradução. A Figura 5.3 ilustra ambas as situações, bem como a tradução correcta para estes casos. No que diz respeito aos ciclos "do while", também terá de ser efectuada uma abordagem semelhante.

```
1 for (i = 0; i < 10; i = i + 1) {
2     break;
3 }
```

```
1 // tradução actual
2 i = 0;
3 while(i < 10) {
4     break;
5     // código que nunca irá
6     // ser executado!
7     i = i + 1;
8 }
```

```
1 // tradução correcta
2 i = 0;
3 while(i < 10) {
4     break;
5 }
```

```
1 for (i = 0; i < 10; i = i + 1) {
2     if (i < 2) {
3         continue;
4     }
5 }
```

```
1 // tradução actual
2 i = 0;
3 while(i < 10) {
4     if (i < 2) {
5         continue;
6     }
7     i = i + 1;
8 }
```

```
1 // tradução correcta
2 i = 0;
3 while(i < 10) {
4     if (i < 2) {
5         i = i + 1;
6         continue;
7     }
8     i = i + 1;
9 }
```

Figura 5.3: Tradução de ciclo For em ciclo While

Concorrência: A nossa solução foi desenhada para programação sequencial, não se focando sobre aspectos de concorrência. No entanto, recentemente a programação concorrente tem vindo a ser alvo de grande atenção, tendo vindo a crescer o número de

aplicações que a utiliza. Este desafio talvez seja o mais difícil de concretizar, podendo necessitar de extensão da linguagem e outras mudanças no cálculo de pré-condições mais fracas, contudo, não deixa de ser uma opção interessante a investigar.

Bibliografia

[ABB+04] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, et al. The KeY Tool. *Software and Systems Modeling*, páginas 32–54, Abril 2004.

[AGB+77] Allen L. Ambler, Donald I. Good, James C. Browne, et al. Gypsy: A Language for Specification and Implementation of Verifiable Programs. *SIGOPS Oper. Syst. Rev.*, 11(2):1–10, 1977.

[AL99] Sten Agerholm e Peter G. Larsen. A Lightweight Approach to Formal Methods. In *Applied Formal Methods — FM-Trends 98*, volume 1641 de *Lecture Notes in Computer Science*, páginas 168–183. Springer Berlin / Heidelberg, 1999.

[Ale10] Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley, primeira edição, 2010.

[AP98] V.S. Alagar e K. Periyasamy. *Specification of Software Systems*. Springer Verlag, 1998.

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, primeira edição, Março 2003.

[BBC+08] G. Barthe, L. Burdy, J. Charles, et al. JACK – A Tool for Validation of Security and Behaviour of Java Applications. *Lecture Notes in Computer Science*, 4709:152, 2008.

[BCD+06] Mike Barnett, Bor-Yuh Chang, Robert DeLine, et al. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, Lecture Notes in Computer Science, capítulo 17, páginas 364–387. 2006.

[BDF+08] M. Barnett, R. DeLine, M. Fahndrich, et al. The Spec# Programming System: Challenges and Directions. *Lecture Notes In Computer Science*, 4171:144–152, 2008.

[Bec01] B. Beckert. A Dynamic Logic for the Formal Verification of Java Card Programs. *Java on Smart Cards: Programming and Security*, páginas 6–24, 2001.

[BK07] Bernhard Beckert e Vladimir Klebanov. A Dynamic Logic for Deductive Verification of Concurrent Java Programs With Condition Variables. In *Proceedings, 1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP), Satellite Workshop CONCUR 2007, Lisbon, Portugal*. 2007.

[BLS05] Mike Barnett, Leino, e Wolfram Schulte. *The Spec# Programming System: An Overview*, volume 3362/2005 de *Lecture Notes in Computer Science*, páginas 49–69. Springer, Berlin / Heidelberg, Janeiro 2005.

[BP96] A. Barber e G. Plotkin. Dual Intuitionistic Linear Logic. *LFCS Report Series-Laboratory for Foundations of Computer Science ECS LFCS*, 1996.

[BT07] Clark Barrett e Cesare Tinelli. CVC3. In *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 de *Lecture Notes in Computer Science*, páginas 298–302. Springer-Verlag, Julho 2007.

[CD02] Dave Clarke e Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, páginas 292–310. ACM, New York, NY, USA, 2002.

[CWA+96] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, et al. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[DCJ06] Greg Dennis, Felix Sheng-Ho Chang, e Daniel Jackson. Modular Verification of Code with SAT. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, páginas 109–120. ACM, New York, NY, USA, 2006.

[Den09] G.D. Dennis. *A Relational Framework for Bounded Program Verification*. Tese de Doutoramento, Massachusetts Institute of Technology, 2009.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, Agosto 1975.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., Outubro 1976.

[DM08] Dino Distefano e Matthew. jStar: Towards Practical Verification for Java. *SIGPLAN Not.*, 43(10):213–226, 2008.

[DMB09] L. De Moura e N. Bjørner. Satisfiability Modulo Theories: An Appetizer. *Formal Methods: Foundations and Applications*, páginas 23–36, 2009.

[ECGN02] M. D. Ernst, J. Cockrell, W. G. Griswold, et al. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, Agosto 2002.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, et al. Extended Static Checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, páginas 234–245. ACM, New York, NY, USA, 2002.

[Flo67] Robert W. Floyd. Assigning Meanings to Programs. In *Proc. Sympos. Appl. Math., Vol. XIX*, páginas 19–32. Amer. Math. Soc., Providence, R.I., 1967.

[FM07] Jean-Christophe Filliâtre e Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Computer Aided Verification*, Lecture Notes in Computer Science, páginas 173–177. Springer Berlin / Heidelberg, 2007.

[HJ00] M. Huisman e B. Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. *Fundamental Approaches to Software Engineering*, páginas 284–303, 2000.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, Outubro 1969.

[Hoa03] Tony Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *J. ACM*, 50(1):63–69, Janeiro 2003.

[Hoa09] C. A. R. Hoare. Retrospective: An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 52(10):30–32, 2009.

[Jac02] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, Abril 2002.

[JJW96] C.B. Jones, D. Jackson, e J. Wing. Formal Methods Light. *ACM Computing Surveys*, 28(4es):121, 1996.

[Jon03] Cliff B. Jones. The Early Search for Tractable Ways of Reasoning about Programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.

[JP01] B. Jacobs e E. Poll. A Logic for the Java Modeling Language JML. *Fundamental Approaches to Software Engineering*, páginas 284–299, 2001.

[JP04] B. Jacobs e E. Poll. Java Program Verification at Nijmegen: Developments and Perspective. *Lecture Notes in Computer Science*, páginas 134–153, 2004.

[Lam80] Leslie Lamport. The 'Hoare Logic' of Concurrent Programs. *Acta Informatica*, 14(1):21–37, Junho 1980.

[LBR99] Gary T. Leavens, Albert L. Baker, e Clyde Ruby. JML: A Notation for Detailed Design, 1999.

[LHL+77] B. W. Lampson, J. J. Horning, R. L. London, et al. Report on the Programming Language Euclid. *SIGPLAN Not.*, 12(2):1–79, 1977.

[LS07] K.R.M. Leino e W. Schulte. A Verifying Compiler for a Multi-Threaded Object-Oriented Language. *Software System Reliability and Security*, página 351, 2007.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, segunda edição, Março 2000.

[MMU04] C. Marché, Paulin C. Mohring, e X. Urbain. The Krakatoa Tool for Certification of Java/JavaCard Programs Annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[MS95] S. P. Miller e M. Srivas. Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *WIFT '95: Proceedings of the 1st Workshop on Industrial-Strength Formal Specification Techniques*, volume 95, páginas 2–16. IEEE Computer Society, Washington, DC, USA, 1995.

[Nau66] P. Naur. Proof of Algorithms by General Snapshots. *BIT Numerical Mathematics*, 6(4):310–316, 1966.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, e Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *12th International Conference on Compiler Construction*, páginas 138–152. Springer-Verlag, 2003.

[NQ08] Nathaniel Nystrom e Xin Qi. Polyglot Extensible Compiler Framework. `http://www.cs.cornell.edu/projects/polyglot/`, 2008.

[OHRY01] P. O Hearn, J. Reynolds, e H. Yang. Local Reasoning about Programs that Alter Data Structures. *Lecture Notes in Computer Science*, páginas 1–19, 2001.

[Oui08] M. Ouimet. Formal Software Verification: Model Checking and Theorem Proving, 2008.

[PB05] Matthew Parkinson e Gavin Bierman. Separation Logic and Abstraction. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, páginas 247–258. ACM, New York, NY, USA, 2005.

[RCK05] E. Rodriguez-Carbonell e D. Kapur. Program Verification Using Automatic Generation of Invariants. In *Theoretical aspects of computing: ICTAC 2004: first international colloquium, Guiyang, China, September 20-24, 2004: revised selected papers*, página 325. Springer Verlag, 2005.

[Rey02] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. *Logic in Computer Science, Symposium on*, 0:55–74, 2002.

[SD03] Matthew Smith e Sophia Drossopoulou. Cheaper Reasoning With Ownership Types. In *IWACO 2003 - Workshop affilicated to ECOOP 2003*. Junho 2003.

[SSM04] Sriram Sankaranarayanan, Henny B. Sipma, e Zohar Manna. Non-linear Loop Invariant Generation Using Gröbner Bases. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, páginas 318–329. ACM Press, New York, NY, USA, 2004.

[Tur49] Alan M. Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, páginas 67–69. University Mathematical Laboratory, Cambridge, Junho 1949.

[Win90] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *Computer*, 23(9):8–24, 1990.