Luís Miguel Cardoso Lourenço

Licenciado em Engenharia Informática

Linguagem Intermédia Tipificada para Concorrência em Memória Partilhada

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : João R. V. da Costa Seco, Prof. Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente:	Prof. Doutor Pedro Abílio Duarte de Medeiros
Arguente:	Prof. Doutor Luís Miguel Barros Lopes
Vogal:	Prof. Doutor João R. V. da Costa Seco

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2011

Linguagem Intermédia Tipificada para Concorrência em Memória Partilhada

Aos meus pais

Agradecimentos

Em primeiro lugar queria agradecer ao meu orientador, Professor João Costa Seco, pelo apoio dado durante a realização da dissertação, e também ao Professor Francisco Martins. Também agradeço a todos os meus colegas e professores que contribuíram para a minha formação académica ao longo da licenciatura e do mestrado.

Agradeço também aos meus pais, à minha irmã e aos restantes familiares pelo apoio total que me deram ao longo da minha vida.

Esta dissertação foi suportada por uma bolsa de investigação do projecto PTDC/EIA-CCO/104583/2008 da Fundação para a Ciência e Tecnologia do MCTES.

Resumo

O objectivo da dissertação consiste em implementar um compilador, e o sistema de suporte à execução, para uma linguagem de programação com mecanismos primitivos de controlo de concorrência em memória partilhada. A utilização de concorrência nos sistemas de *software* actuais é essencial, desde os servidores aplicacionais mais poderosos que disponibilizam serviços a múltiplos clientes simultaneamente, até aos "simples" interfaces gráficos de utilização comum. As linguagens ditas *general purpose*, as mais utilizadas para a implementação destes sistemas, como a linguagens Java e C#, suportam a utilização de vários fios de execução através de classes de biblioteca. O suporte específico da linguagem para controlar acessos concorrentes a zonas de memória partilhada restringe-se apenas à utilização dos objectos como monitores, o que torna difícil a implementação de mecanismos de verificação estática ao nível do sistema de tipos.

Nesta dissertação pretende-se implementar uma linguagem de programação que integra mecanismos de concorrência de forma nativa, tornando possível a construção de programas concorrentes de forma mais estruturada e modular. É proposta uma linguagem concreta implementada a partir de uma linguagem *core* desenvolvida no grupo de investigação onde se insere este trabalho. Também é desenvolvido um compilador, a respectiva máquina virtual de pilha e uma linguagem intermédia tipificada, com um modelo de objectos análogo ao da JVM/CLR, mas com suporte nativo para concorrência. A definição de uma linguagem intermédia tipificada, onde constarão instruções para a criação de múltiplos fios de execução e controlo de concorrência, visa suportar e antever o desenvolvimento de um sistema de tipos comportamental que permita detectar estaticamente as interferências entre os múltiplos fios de execução, ao nível de abstracção mais baixo, e que também espelhe as propriedades do sistema de tipos da linguagem fonte.

Palavras-chave: linguagem de programação, concorrência, máquina virtual, compilador, sistema de tipos, tipos comportamentais

Abstract

The goal of this thesis is to implement a compiler, and a runtime support system, for a programming language that includes native mechanisms of concurrency in shared memory model. The usage of concurrency in current software systems nowadays is crucial, from powerful application servers that provide services to multiple clients simultaneously, to the "simplest" of graphical user interfaces. General purpose languages, widely used to implement this systems, such as Java and C#, support the usage of multiple threads through library classes. The specific support of the language for controlling concurrent accesses to shared memory areas is restricted to use objects like monitors, that make harder the implementation of static analysis mechanisms with the aid of the type system.

This work aims at implementing a language that integrates concurrency in a native way, making it possible for the construction of concurrent programs in a more structured and modular way. We will introduce a concrete programming language for a core language implemented in the research group where this work is being developed. This work focuses on developing a compiler, a stack-based virtual machine and its typed assembly language, with an object model like JVM/CLR, but with native support for concurrency. The definition of the typed assembly language, where will be instructions to create multiple threads and to control their concurrent access, aims to support and foresee the development of a spatial-behavioral type system that is able to statically check the interferences between multiple threads, in a lower abstraction level, and also reflects the properties of the high-level language type system.

Keywords: programming language, concurrency, virtual machine, compiler, type system, spatial-behavioral types.

Conteúdo

Lista de Figuras

1

Introdução

A construção de programas concorrentes consiste na utilização de vários fios de execução (*threads*) para melhor aproveitar os recursos de uma máquina (*e.g.*, usando vários processadores para diminuir o tempo de execução de uma tarefa), ou para melhorar a reactividade de um sistema (*e.g.*, não bloquear uma computação ou um interface gráfico por causa de uma actividade demorada, ou atender vários pedidos simultaneamente num *webserver*). Os múltiplos fios de execução podem ser executados apenas por um processador, ou estarem mapeados em diferentes processadores físicos (ou *cores*), ocorrendo nesse caso paralelismo real. Os fios de execução de um programa concorrente podem, e em geral devem, interagir entre si. Normalmente divide-se o tipo de interacção que pode ocorrer em dois modelos base: troca de mensagens e memória partilhada. Embora tenhamos estudado alguns trabalhos sobre troca de mensagens, o foco deste trabalho é sobre programas concorrentes com memória partilhada, e foca os mecanismos de controlo de concorrência que lhes são associados.

Sempre que, num programa concorrente, há partilha de uma zona de memória, é necessário disciplinar os acessos a esses dados sob pena de ocorrerem interferências indesejadas (entre fios de execução). Definem-se as chamadas *regiões críticas*, zonas do código que acedem à zona de memória partilhada, e utilizam-se mecanismos dinâmicos para garantir a ausência dessas mesmas interferências. Os mecanismos utilizados vão desde os mais primitivos, implementados pelos sistemas operativos a um nível de abstracção muito baixo, e que são visíveis em algumas linguagens de programação através de bibliotecas, como no caso dos semáforos e dos *locks*. Os semáforos, propostos por Dijkstra [13], são mecanismos que permitem assegurar que só um fio de execução entra numa região crítica de cada vez, serializando a execução dos fios de execução nestas regiões. É este também o comportamento básico dos *locks* que encontramos mais regularmente [32], que podem ser implementados através de semáforos. A principal diferença consiste no facto de os semáforos serem mais flexíveis, permitindo, por exemplo, controlar um número

de fios de execução maior que um, mas fixo, numa região crítica. Outro mecanismo primitivo de controlo de concorrência, muito perto da sua implementação hardware, é a instrução *test and set lock*, que escreve um valor na memória e obtém o valor antigo de forma atómica [32], usado nos chamados algoritmos não bloqueantes.

No início dos anos setenta também foram propostos, por Hansen e Hoare, alguns mecanismos estruturados de sincronização, como os monitores [16] [18]. Ao contrário dos mecanismos de *locks* e semáforos, os monitores são abstracções ao nível das linguagens de programação. São entidades que protegem estruturas de dados partilhadas, com procedimentos próprios que permitem aceder de forma segura aos dados. Num monitor só um fio de execução pode executar (código) dentro do mesmo de cada vez. Ao monitor podem estar associadas várias variáveis de condição (filas de fios de execução em espera), sobre as quais se podem aplicar duas operações (*wait* e *signal*) que permitem uma sincronização e estruturação da execução concorrente mais fina que os *locks*. Os fios de execução dentro do monitor podem invocar a operação *wait* para ficarem à espera numa condição, deixando que outro fio de execução continue a execução no monitor, ou passar voluntariamente o controlo para um fio (em espera) com a operação *signal*.

Outro mecanismo de controlo de concorrência estruturado, bastante distinto e mais recente que os anteriores, chama-se *Software Transactional Memory* (STM) [35]. Ao contrário dos semáforos, *locks* e monitores, que se baseiam num modelo pessimista, as STMs baseiam-se no modelo optimista semelhante às transacções nas bases de dados, onde os fios de execução prosseguem sem se preocuparem com possíveis interferências com outros fios de execução, e o sistema de suporte à execução encarrega-se de ver no final se houve ou não interferências, podendo abortar a execução de alguns deles (que devem depois repetir a execução).

Vantagens do Controlo Estruturado Em geral, a vantagem de ter mecanismos estruturados nas linguagens, como os monitores ou as STMs, está relacionada com uma maior facilidade de programação, e também no evitar dos erros de programação mais comuns, em particular em programas concorrentes (*e.g.*, utilizar operação *lock* sem o respectivo *unlock*). O controlo estruturado de acessos concorrentes (com utilização de variáveis de condição) apesar de não dar garantias da ausência total de erros, previne erros de programação, tais como *deadlocks*, ou seja, situações de impasse entre vários fios de execução [32].

Outro erro comum que ocorre na programação concorrente são as chamadas *race conditions* [32]. Estas acontecem quando a correcção do programa ou o funcionamento correcto do próprio, de acordo com a sua especificação, depende da forma como os fios de execução são escalonados. Por exemplo, quando é tomada uma decisão num programa baseada numa determinada informação, e o processamento subsequente depende dessa informação sem que se garanta que se mantém actualizada. Semelhante erros de programação, mas não exactamente iguais às *race conditions*, são as *data races* [32]. Resultam de

acessos concorrentes de leitura e escrita sobre uma mesma zona de memória, sem quaisquer mecanismos de sincronização para a proteger de interferências (por exemplo, *locks*). Ocorrem, por exemplo, quando dois fios de execução incrementam uma variável inteira, onde é preciso primeiro ler o valor, calcular o incremento, e depois guardar o resultado na variável. A execução desta operação em dois fios concorrentes pode gerar no final resultados diferentes, dependendo do escalonamento dos mesmos. O exemplo que se segue, de uma actualização bancária, não apresenta *data races* mas apresenta uma *race conditon*, derivada do facto de o montante total poder ser visto num estado incoerente por um segundo fio de execução:

```
transfer(account1, account2, amount) {
    lock
    account1.balance += amount
    unlock
    lock
    account2.balance -= amount
    unlock
}
```

Linguagens de Programação Actuais As linguagens *general purpose*, as mais utilizadas actualmente para a implementação de sistemas concorrentes, tais como o C, C++, Java e C#, suportam o modelo de vários fios de execução e de memória partilhada através de bibliotecas. Nas duas últimas linguagens, o suporte incorporado/nativo na linguagem para concorrência restringe-se à utilização dos objectos como monitores (e com uma só condição). A linguagem Scala suporta concorrência baseada no modelo de actores [7] [31] através de bibliotecas, apesar de este suporte estar bastante bem integrado na linguagem. No entanto, esta linguagem não tem primitivas de controlo de concorrência entre múltiplos fios de execução, no modelo de memória partilhada. Existem outras linguagens cujo suporte para concorrência é nativo, tais como linguagens derivadas do Cálculo-π, nomeadamente a linguagem Pict [33], ConcurrentML [34], TyCO [40], entre outras. A linguagem Pict é uma linguagem construída de raiz a pensar no paradigma concorrente. A ideia geral consiste na composição paralela de vários fios de execução, que através de canais comunicam entre si trocando mensagens. Mas mais uma vez, tal como na linguagem Scala, estamos perante uma linguagem com suporte para concorrência apenas no modelo de troca de mensagens.

Tomando um nível de abstracção mais baixo, no domínio das linguagens intermédias e máquinas virtuais, a JVM e a CLR, para as quais é gerado código das linguagens Java e C#, respectivamente, também oferecem suporte para concorrência, em memória partilhada, através de bibliotecas. A principal desvantagem do suporte não ser oferecido de forma nativa, tanto no nível das linguagens de programação como nas linguagens intermédias, está relacionado com a impossibilidade, ou a maior dificuldade, em implementar mecanismos de verificação estática, com auxílio do sistema de tipos, em tempo

de compilação ou carregamento, que detectem interferências entre múltiplos fios de execução. Por exemplo, um sistema de tipos que consiga detectar *data races* sobre um objecto entre múltiplos fios de execução, ou apenas disciplinar os acessos sequênciais/concorrentes sobre o mesmo. Algumas linguagens intermédias, como por exemplo a linguagem MIL [41], oferecem suporte para concorrência de forma nativa, e no modelo de memória partilhada, garantindo, com auxílio de notações de tipo e do sistema de tipos, a ausência de interferências entre múltiplos fios de execução e de situações de impasse entre os mesmos. Porém, ao contrário da JVM e CLR, são destino de máquinas bastante primitivas, baseadas em registos, sem modelo de objectos nativo e sem pilhas distintas de chamada e avaliação, o que torna mais difícil a tradução de uma linguagem fonte, orientada aos objectos e com concorrência, preservando a informação de tipos entre os dois níveis de abstracção, durante o processo de compilação.

Motivação e Problema Neste trabalho temos como objectivo implementar uma linguagem que integre mecanismos de concorrência em memória partilhada, de forma nativa, numa linguagem de programação imperativa e com objectos, tornando possível a construção de programas concorrentes de forma mais estruturada e modular, através de primitivas específicas de concorrência e respectivo controlo. A linguagem será definida, de raiz, tendo como objectivo a construção de programas concorrentes. Será proposta uma linguagem concreta implementada a partir de uma linguagem *core* [11], desenvolvida no grupo de investigação onde se insere este trabalho. A linguagem *core* é orientada por objectos, é imperativa, considera as referências para os fios de execução como sendo valores primitivos, e está equipada com um sistema de tipos sofisticado para evitar erros derivados de interferências entre os múltiplos fios de execução.

De forma a garantir que código compilado preserva a informação de tipos, da linguagem fonte, definimos também uma máquina virtual e respectiva linguagem intermédia tipificada, para máquina de pilha, com objectos e com suporte nativo para concorrência. O sistema de tipos permitirá garantir, num passo apenas, com auxílio de notações de tipo no código, que o programa carregado na máquina virtual está bem tipificado, evitando os erros mais usuais (saltos para locais ilegais, utilização da pilha para além dos limites, operações sobre valores do tipo errado, etc). Este sistema de tipos é a base para uma futura extensão que permita disciplinar os acessos paralelos entre os múltiplos fios de execução, através de tipos comportamentais [10]. O mesmo se aplica para a linguagem de alto nível. A semelhança entre os dois modelos de objectos (nativos) torna mais fácil a implementação de um compilador que preserva a informação de tipos, permitindo uma abordagem que segue as linguagens intermédias tipificadas [25], que possa ser usada em cenários de mobilidade de código (*Proof carrying code* [29]).

A linguagem de alto nível e orientada a objectos, que vamos apresentar, inclui as expressões (fork e) e (wait e) para a manipulação estruturada de fios de execução, onde a expressão (wait (fork e)) denota o valor da expressão e. Também tem primitivas de controlo de concorrência (sync(e)$\{e\}$) e (shared(e)$\{e\}$), que permitem programar de forma nativa o

padrão bastante conhecido de um escritor e vários leitores [18]. A linguagem intermédia tipificada também é orientada a objectos, sendo um programa composto por classes, com campos e métodos, que por sua vez têm blocos de instruções. Por questões de simplicidade consideramos apenas os métodos de instância, isto é, que são relativos a um dado objecto. A máquina é composta por pilhas distintas de avaliação e de chamada. Como é de esperar, a pilha de chamada é constituída por registos de activação, que representam instâncias de métodos e contêm os argumentos, as variáveis locais, a pilha de avaliação e uma referência para o código a executar. Na pilha de avaliação são empilhados valores, realizadas operações sobre os mesmos e empilhado o resultado. As instruções da máquina mais relevantes para este trabalho são as de criação e manipulação de fios concorrentes (fork *method*(), wait) e para controlo de acessos (sync *method*(), shared *method*()). De notar que o processo de tradução destas primitivas, da linguagem de alto nível para a linguagem intermédia, como veremos nesta dissertação, não é directo devido ao aninhamento de expressões na linguagem de alto nível.

Contribuições Os maiores desafios que se apresentam no desenvolvimento desta dissertação estão na definição da máquina virtual e do compilador para código da linguagem intermédia tipificada, garantindo a ausência de interferências entre fios de execução quando são usadas explicitamente, pelo programador, as respectivas primitivas de controlo. Todo este desenvolvimento deve ser feito tendo em mente, como trabalho futuro, um outro desafio que é a extensão do sistema de tipos para tipos comportamentais, para depois permitir a análise, em tempo de compilação e carregamento, do controlo dos acessos concorrentes a recursos (objectos). A realização do trabalho toma como ponto de partida uma linguagem *core*, imperativa e orientada a objectos [11]. No decorrer da elaboração da dissertação foi publicado o artigo "Linguagem Intermédia Tipificada para Máquina de Pilha Concorrente com Objectos" [24] no INForum. Como resultados deste trabalho apresentar-se-ão:

1. Uma definição da semântica da linguagem fonte por tradução directa para a linguagem Java utilizando algumas classes de biblioteca de suporte.
2. Definição da semântica operacional e sistema de tipos da linguagem fonte.
3. A definição de uma linguagem intermédia tipificada com as primitivas de concorrência, para um modelo de objectos semelhante ao das máquinas JVM [22] e CLR [19].
4. Definição da semântica operacional e do sistema de tipos da linguagem intermédia.
5. Uma implementação da máquina de pilha para a definição proposta.
6. Uma tradução da linguagem fonte para a linguagem intermédia.
7. Implementação do compilador.

Organização do Documento No capítulo seguinte (capítulo 2) são apresentados e analisados de maneira breve alguns mecanismos conhecidos de controlo de concorrência (*locks*, semáforos, *test and set lock*, monitores e STMs). De seguida são mencionadas e analisadas algumas linguagens de programação com suporte para concorrência (3). Depois são descritas e analisadas algumas máquinas virtuais, e respectivas linguagens intermédias, com e sem suporte nativo para concorrência (capítulo 4).

No capítulo seguinte (5) definimos a linguagem de alto nível, a sua sintaxe concreta e introduzimos a sua semântica de forma informal, usando alguns exemplos. Apresentamos uma primeira proposta de implementação para a linguagem, que consiste na sua tradução para a linguagem Java. Posteriormente, ainda no capítulo 5, apresentamos a semântica operacional e o sistema de tipos da linguagem, que poderá permitir realizar provas sobre a preservação da semântica e de tipos relativamente à linguagem intermédia tipificada definida no capítulo 6. É neste capítulo que descrevemos ao pormenor a motivação para a definição e implementação da máquina virtual, assim como a sua sintaxe concreta, semântica operacional e sistema de tipos da linguagem intermédia. Segue-se depois a explicação do processo de compilação entre as duas linguagens (capítulo 7).

No último capítulo (8) apresentamos o balanço do trabalho realizado face aos objectivos da dissertação. Mencionamos alguns aspectos a melhorar como trabalho futuro, nomeadamente aquele que consiste na principal motivação da dissertação, que é a extensão do sistema de tipos da linguagem intermédia (e da linguagem de alto nível) de modo a suportar tipos comportamentais, que permitirá analisar estaticamente os acessos concorrentes entre os múltiplos fios de execução.

2

Mecanismos de Controlo de Concorrência em Memória Partilhada

A partilha de recursos, entre fios de execução concorrentes, é um tema importante tanto no nível dos sistemas de operação como no nível das linguagens de programação. A forma mais simples de gerir os acessos a esses recursos, tais como estruturas de dados partilhadas, é a puramente sequencial e sem quaisquer interrupções, considerando toda a estrutura como região crítica. No entanto, esta solução tem claras desvantagens no que diz respeito à eficiência, pois não permite maximizar o desempenho em máquinas com múltiplos processadores, ou até mesmo o caso de uma tarefa demorada impedir que outras (que demoraram menos tempo) sejam executadas. Existem várias soluções para resolver estes problemas, isto é, para aumentar o grau de concorrência, onde se destacam os *locks*, os semáforos, a instrução *test and set*, os monitores e, num modelo bastante distinto dos anteriores e do abordado neste documento, os STMs.

2.1 Locks e Semáforos

Os *locks* são mecanismos utilizados para controlar o acesso (de fios de execução concorrentes) a regiões críticas. Sempre que múltiplos fios de execução necessitam de aceder a um dado partilhado, e nem sempre é para leitura, executam a operação *lock*, e só depois de adquirir o direito de acesso prosseguem (um a um) a sua execução. Depois de terminadas as leituras/escritas na estrutura partilhada, o fio de execução deve executar a operação *unlock*, libertando o acesso, dando possibilidade a outros fios de execução de repetirem o mesmo procedimento.

Uma alternativa mais flexível foi proposta por Dijsktra, designada por semáforo [13].

Este é constituído por uma variável, do tipo inteiro, não negativa, que juntamente com as operações atómicas P e V permite controlar acessos a regiões críticas. Relativamente à semântica da operação P, quando um fio de execução invoca esta operação fica bloqueado no caso de o valor do semáforo ser igual a zero, ou decrementa o valor e prossegue se for positivo. A operação V permite incrementar o valor do semáforo. Se o valor inicial do semáforo for igual a 1 e se ambas as operações forem executadas por todos os fios de forma alternada (P ... V ... P ... V ...) então a semântica dos semáforos é igual à dos *locks*. Vejamos agora um pequeno exemplo, que permite controlar o acesso a um dado partilhado entre leitores e escritores:

```
Inicialização:
    sem = 1;
    semReaders = 1;
    readers = 0;

Leitor:
    P(semReaders);
    readers++;
    if(readers == 1) P(sem);
    V(semReaders);
    ... ler dados ...
    P(semReaders);
    readers--;
    if(readers == 0) V(sem);
    V(semReaders);
```

Sempre que um leitor deseja aceder a um recurso incrementa o número de leitores e, se for o primeiro, bloqueia se um escritor estiver activo. Como o contador é alterado pelos leitores, será necessário utilizar um segundo semáforo, de modo a que essa alteração seja executada em exclusão mútua. Depois de ler os dados, o leitor decrementa o número de leitores (novamente em exclusão mútua) e, caso seja o último leitor, liberta o direito de acesso, permitindo que escritores possam aceder ao recurso partilhado.

```
Escritor:
    P(sem);
    ... escrever dados ...
    V(sem);
```

Sempre que um escritor deseja aceder a um recurso partilhado, deverá tentar obter o direito de acesso, ficando eventualmente bloqueado enquanto existirem leitores activos ou um escritor activo. Depois de os dados serem alterados, o direito de acesso é libertado.

2.2 Test and Set

Ainda mais primitivo que os semáforos é a instrução *test and set* [32] que permite, de forma atómica, consultar o valor antigo de uma variável e alterá-lo. Vejamos um exemplo concreto da sua utilização:

```
int lock;
...
while( test_and_set(lock, 1) == 1 );
... aceder a dados partilhados ...
lock = 0;
```

Através da instrução *test and set* e de um ciclo, consegue-se definir uma espera activa e a obtenção de um direito de acesso (*lock*). Depois são acedidos dados partilhados e por último é libertado o *lock*. Esta é a base para algoritmos *lock-free*, onde o fio de execução não bloqueia, continuando repetitivamente a tentar aceder à região crítica [32].

2.3 Monitores

Os semáforos podem ser utilizados para controlar, de forma eficiente, o acesso a recursos partilhados. No entanto, são mecanismos muito primitivos e por vezes difíceis de usar (com garantias de certas propriedades desejadas). Ao contrário dos semáforos, os monitores são mecanismos de mais alto nível, permitindo uma maior abstracção. Cada monitor está associado a um ou mais recursos, permitindo que o acesso aos mesmos seja efectuado com controlo de concorrência entre os múltiplos fios de execução. Dentro do monitor só pode estar um fio de execução de cada vez, e estes têm a possibilidade de passar explicitamente o controlo a outros fios de execução. De seguida, apresentamos a definição do modelo de monitores de Hoare, e finalmente fazemos uma comparação entre os vários modelos/implementações.

Especificação de Hoare

Segundo a especificação de Hoare [18], os monitores são objectos que têm associados dados partilhados e determinados procedimentos. Cada procedimento serve apenas para controlar o acesso a dados partilhados e tem que ser executado em exclusão mútua. Nestes são usados duas primitivas: wait e signal. A primeira permite adormecer um processo. A segunda permite um processo acordar outro (previamente adormecido). No entanto, tendo em conta que podem existir várias condições para um processo adormecer, é possível ter, dentro do monitor, várias condições e aplicar as primitivas wait e signal directamente a uma determinada condição. Como exemplo ilustrativo, considere-se o seguinte:

```
Monitor {
    boolean busy = false;
    Condition nonBusy;

    proc acquire() {
        if(busy) nonBusy.wait();
        busy = true;
    }

    proc release() {
        busy = false;
        nonBusy.signal();
    }
}
```

O procedimento acquire é chamado antes de aceder a um recurso partilhado. O release é chamado depois de ser utilizado o recurso. O primeiro verifica se o recurso está ocupado, e em caso afirmativo adormece o processo. Em caso negativo, ou depois de ser acordado, assinala o recurso como ocupado e prossegue a sua execução. O segundo procedimento assinala o recurso como livre, e possivelmente acorda um processo previamente adormecido. De notar que a primitiva signal nada faz caso nenhum processo tenha feito wait ou todos os que fizeram wait já tenham acordado anteriormente. De salientar também que a primitiva wait liberta o acesso exclusivo. As duas primitivas, assim como a exclusão mútua dos procedimentos, podem ser implementadas através de semáforos. As implementações podem variar, pois podem ter em conta questões de justiça relativa aos tempos de espera entre os processos, entre outros aspectos.

Através de axiomática de Hoare [17] podem ser definidas invariantes no monitor, de modo a descrever certas propriedades desejáveis. As invariantes devem ser válidas antes e depois da chamada de procedimentos. Também podem ser definidas asserções, colocadas antes das operações signal e depois das operações wait, relativas à mesma condição. Por exemplo, num monitor que suportasse o modelo de um escritor e vários leitores, e que tivesse duas variáveis, uma para sinalizar um escritor activo (*busy*) e outra para contar o número de leitores (*counter*), podíamos definir a invariante ($busy \longrightarrow counter = 0$), indicando que se o recurso estiver ocupado (por um escritor) então não há leitores activos no monitor. Contudo, nada garante que não sejam cometidos erros e que não possam existir interferências entre os múltiplos fios do programa.

Diferenças entre as especificações/implementações

Existem várias especificações e implementações de monitores, desde as especificações de Hoare e de Hansen às implementações em Mesa e em Java, tanto no pacote base `java.lang`, como no pacote `java.util.concurrent` elaborado por Doug Lea. Na

Tabela 2.1: *Diferenças entre as especificações/implementações de monitores*

	Hoare	**Hansen**	**Java**	**Doug Lea**	**Mesa**
obtenção da exclusão mútua	na entrada do método ou quando é acordado	na entrada do método ou quando é acordado	na entrada do método ou bloco *synchronized*	através da operação *lock*	na entrada do método
libertação da exclusão mútua	na saída do método, quando é adormecido ou quando acorda outro	na saída do método, quando é adormecido ou quando acorda outro	no fim de cada método ou bloco *synchronized*, ou quando é adormecido	através do método *unlock* ou quando é adormecido	no fim de cada método ou quando é adormecido
semântica do wait	bloqueia o processo e quando for acordado ganha o *lock*	bloqueia o processo e quando for acordado ganha o *lock*	bloqueia o processo e quando acorda compete por um *lock*	bloqueia o processo e quando acorda compete por um *lock*	bloqueia o processo e quando acorda compete por um *lock*
semântica do signal	acorda um processo em espera, passa-lhe o *lock* e fica bloqueado	acorda um processo em espera, passa-lhe o *lock* e sai do monitor.	acorda um processo em espera	acorda um processo em espera	acorda um processo em espera
múltiplas filas	sim	não	não	sim	sim
várias fios de execução por fila	sim	não	sim	sim	sim

De notar que, na especificação de Hoare, a operação *signal* pode ter uma optimização. Se for sempre invocado no final do procedimento do monitor, então a processo que invocou não necessita de ficar bloqueado, passando o *lock* ao processo acordado e saindo do monitor.

tabela 2.1 são descritas e comparadas as várias especificações/implementações. Fundamentalmente, as especificações de Hoare e Hansen distinguem-se das implementações em Java e Mesa [4] no que diz respeito à semântica das operações wait e signal. Nas duas especificações a operação signal passa o *lock* para o fio de execução que for entretanto acordado. Nas implementações (Java e Mesa) o fio acordado apenas passa de um estado em que está adormecido, para um estado em que espera pela obtenção do *lock* [4]. Só depois de adquirido o *lock* é que prossegue a sua execução.

2.4 Memória transacional

Os *locks*, semáforos e monitores baseiam-se num modelo pessimista. Quando um fio acede, a um dado partilhado, são utilizados *locks*/semáforos para controlar o acesso, podendo ou não terem sido necessários, dependendo dos acessos e escalonamento dos outros fios de execução. No caso das bases de dados usa-se uma política bastante diferente

(optimista). As transacções (fios de execução) executam operações (leituras e escritas) livremente sobre os dados, que são registados num *log*. Só no fim de uma transacção é que são verificados possíveis conflitos com outras (transações). Se existirem, a transacção é abortada e todas as suas acções são desfeitas (*rollback*). Se não existirem, a transacção é completada com sucesso (*commited*).

Foi através da analogia entre os programas com múltiplos fios de execução e o modelo de transacções das bases de dados que surgiu o conceito de Software Transactional Memory [35]. As principais vantagens consistem na ausência de *deadlocks* e num maior grau de concorrência quando comparado com os modelos pessimistas, visto que não há necessidade de efectuar pontos de sincronização pois os conflictos são resolvidos no fim da transação. Porém, a validação/gestão de uma grande quantidade de conflictos no final pode prejudicar bastante o desempenho, nomeadamente quando é necessário "desfazer" as transações, restaurando o estado anterior (*rollback*).

3

Linguagens de Programação com Suporte para Concorrência

Analisamos, neste capítulo, algumas linguagens de programação com suporte para concorrência. Analisamos as linguagens Java e C# por serem as mais usadas actualmente e por seguirem o modelo de memória partilhada, onde a concorrência é suportada por classes biblioteca (`java.lang.Thread`) e também por um modelo de monitores análogo ao das linguagens Concurrent Pascal [16] e Modula-3 [12]. Também fazemos uma análise a algumas das extensões às linguagens Java e C# (Join Java e Polyphonic C#). Depois analisamos a linguagem Scala, mais concretamente o seu modelo de actores, assim como linguagens com suporte nativo para a concorrência, tais como Erlang, Pict e Go, tendo como base o modelo de troca de mensagens. Este estudo é necessariamente incompleto mas pretende apenas ilustrar os conceitos envolvidos na programação concorrente, servindo como motivação para a nossa linguagem com primitivas para concorrência em memória partilhada.

3.1 Concorrência em Java e C#

A linguagem Java oferece suporte para a programação concorrente, através da classe biblioteca `Thread` e da interface `Runnable`. Por convenção, o código a executar, num novo fio de execução, deverá estar inserido num método com a assinatura **void** run(). Vejamos o seguinte exemplo, em que um fio de execução calcula o somatório de 1 até n:

```
class Sum implements Runnable {
    int res = 0;
    public void run() { for(int i=0; i<n; i++) res += i; }
}
new Thread(new Sum()).start();
```

Para depois esperar pelo fio de execução lançado, invoca-se o método join da classe `Thread`. No que diz respeito ao controlo de concorrência, na linguagem Java cada objecto está associado (implicitamente) a um monitor. No caso de os métodos de uma classe serem declarados com a palavra **synchronized**, qualquer fio de execução que os invoque terá que obter direito de acesso (exclusão mútua). Também é possível aceder a um objecto, em exclusão mútua, através do bloco **synchronized**. Caso um fio de execução tenha já obtido o *lock*, e ainda não o tenha libertado, então um segundo fio de execução que tente obter o direito de acesso ficará à espera numa *lock queue* [22].

Existem dois procedimentos associados a qualquer objecto: wait e notify (e notifyAll). O primeiro adormece o fio de execução que o invoca, colocando-o numa *waiting queue*. O segundo permite a um fio de execução "acordar" outro que anteriormente tenha "adormecido". Esta operação não faz com que o fio de execução acordado prossiga imediatamente a sua execução (ao contrário da especificação de Monitores de Hoare). Apenas move o fio de execução (acordado) da *waiting queue* para a *lock queue*, competindo depois por um *lock* com outros fios de execução. Existem mais alguns métodos relacionados com fios de execução, nomeadamente o interrupt, da classe `Thread`, que na verdade não interrompe um fio, mas faz com que um determinado fio de execução saia do estado bloqueado (*e.g.*, invocou o método wait a um objecto), sendo necessário recorrer a variáveis partilhadas (*booleans*) e avaliar as mesmas periodicamente para terminar fios de execução.

Apesar da linguagem Java ter suporte para programação concorrente (classes biblioteca e objectos baseados em monitores) o suporte não é nativo. A própria JVM também não tem instruções base para criação de fios de execução, conforme mencionado no capítulo 4. A linguagem Java também tem como grande desvantagem o facto de não permitir associar, ao mesmo objecto, múltiplas filas de espera, uma para cada tipo de condição. A biblioteca `java.util.concurrent` resolve este problema, permitindo associar múltiplas fila de espera para um mesmo monitor (objecto).

À semelhança da linguagem Java, a linguagem C# também oferece suporte para concorrência. Para criar um novo fio de execução constrói-se um objecto da classe `Thread`, sendo passado no construtor um objecto `ThreadStart`, que por sua vez recebe uma função. Vejamos o mesmo exemplo anteriormente apresentado, mas agora em C#:

```
class A {
    int res = 0;
    void B() { for(int i=0; i<n; i++) res += i; }
}
new Thread(new ThreadStart(A.B)).Start()
```

```
class A
{
    int value;
    int count;

    void producer(int val) {
        lock(this) {
            while(count == 1) Monitor.Wait(this);
            val = value;
            count = 1;
            Monitor.Pulse(this);
        }
    }

    int consumer() {
        int res;
        lock(this) {
            while(count == 0) Monitor.Wait(this);
            res = value;
            count = 0;
            Monitor.Pulse(this);
        }
        return res;
    }
}
```

Figura 3.1: Produtor/consumidor em C#

Tal como no Java, oferece suporte para sincronização de fios concorrentes, através de um bloco (lock), que recebe o objecto sobre o qual serão executadas operações em exclusão mútua. Também oferece suporte para monitores. No Java é associado (implicitamente) a todos os objectos um monitor, sendo invocados, ao próprio objecto, os métodos wait, notify, entre outros, para sincronizar os fios concorrentes. No C# é necessário recorrer à classe Monitor e invocar os métodos wait e pulse, que recebem como argumento o próprio objecto. Como exemplo ilustrativo considere-se a figura 3.1. A classe, com os métodos producer e consumer, representa o conhecido problema do produtor-consumidor.

À semelhança do Java, o suporte para concorrência e sincronização tem algumas desvantagens, tais como não haver suporte para múltiplas filas associadas ao mesmo monitor. No capítulo 4 será descrito como é que este suporte está presente na máquina virtual.

3.2 Polyphonic C# e Join Java

A linguagem Polyphonic C# [9] é uma extensão à linguagem C#, baseada no *Join Calculus* [14], introduzindo novas primitivas relativas à programação concorrente. Seguindo o mesmo modelo, também existe a extensão Join Java [20] para linguagem Java. Por questões de simplicidade, focamos a nossa análise na primeira extensão.

```
class ProducerConsumer
{
    ProducerConsumer() { Empty(); }
    void Get() & private async Full() {}
    void Put() & private async Empty() {}
    void releaseGet() { Empty(); }
    void releasePut() { Full(); }
}
```

Figura 3.2: Produtor/consumidor em Polyphonic C#

Na maioria das linguagens existe uma relação única entre um método (assinatura) e respectivo corpo (a implementação). No entanto, nesta extensão um corpo pode estar associado a vários métodos (assinaturas). Esta forma de estruturação é designada por *chord*. Um mesmo método (assinatura) também pode aparecer em diferentes *chords*, e o código de um corpo apenas pode ser executado se todos os seus métodos forem chamados.

Os métodos de uma *chord* podem ser síncronos ou assíncronos. Como os próprios nomes indicam, os métodos síncronos bloqueiam os fios de execução que os invocam, até retornarem com um determinado resultado. Os assíncronos não bloqueiam o processo (nem retornam um resultado). Estes últimos métodos são assinalados pela palavra async. Através de métodos síncronos e assíncronos é possível definir protocolos de acesso a dados partilhados.

Vejamos primeiro um exemplo bastante simples (figura 3.2 [9]) que consiste numa classe que define o protocolo de uma zona de memória partilhada entre produtores e consumidores. Quando a classe `ProducerConsumer` é instanciada, é invocado o método Empty. Assim, quando um produtor invocar o método Put não vai necessitar de esperar, visto que já foi (anteriormente) invocado o método Emtpy. O único corpo associado ao método Put está vazio, servindo apenas para efeitos de controlo / sincronização. Depois do produtor colocar o valor, irá invocar o método releasePut, que por sua vez invoca o método assíncrono Full. Assim, se entretanto um consumidor já tiver invocado o método Get, não irá precisar de esperar. Tal como no caso anterior, o corpo associado ao método Get está vazio. Depois de o consumidor retirar o valor, será necessário invocar o método releaseGet, que por sua vez invoca o método assíncrono Empty, fazendo com que um dos possíveis produtores bloqueados prossiga a sua execução.

Vejamos agora um exemplo mais elaborado (figura 3.3 [9]) de uma classe para controlo de concorrência entre leitores e escritores. Um escritor quando vai escrever (numa estrutura de dados partilhada) tem que primeiro invocar o Exclusive e, depois de ter feito a escrita, tem que invocar o ReleaseExclusive. Já os leitores têm que invocar Shared no início e ReleaseShared no fim.

Um leitor, após invocar Shared, poderá prosseguir por uma de duas razões. Ou não está ninguém no monitor e é executado o corpo que tem associado o método Shared e Idle, ou um outro leitor está presente e invocou o método S(n), sendo executado o corpo que

```
class ReaderWriter
{
    ReaderWriter() { Idle(); }
    void Exclusive() & private async Idle() {}
    void ReleaseExclusive() { Idle(); }
    void Shared() & private async Idle() { S(1); }
    void Shared() & private async S(int n) { S(n+1); }
    void ReleaseShared() & private async S(int n) {
        if (n == 1) Idle(); else S(n−1);
    }
}
```

Figura 3.3: Leitor/escritor em Polyphonic C#

tem associado o método Shared e S(n). Se um escritor estiver activo, então o leitor terá que esperar. Quando um leitor termina de ler os dados e chama o método ReleaseShared, não terá que ficar bloqueado porque a última chord tem associado os métodos ReleaseShared e S(n), e entretanto já foi chamado o método S(n), garantidamente. Se for o último leitor, invoca o método Idle, permitindo que um escritor, entretanto bloqueado, prossiga a sua execução. Caso contrário, como existem outros leitores activos, invoca o método S(n−1).

Um escritor, após invocar o método Exclusive, poderá presseguir apenas se ninguém (nem leitores nem escritores) estiver no monitor, isto é, se alguém invocou Idle e mais ninguém entrou no monitor. O corpo do método está vazio, servindo apenas para efeitos de sincronização. Quando o escritor acaba de alterar os dados, invoca o método ReleaseExclusive, que por sua vez chama o método Idle, permitindo que outros escritores ou leitores entrem no monitor.

Esta extensão ao C# permite construir facilmente uma grande variedade de protocolos, num ambiente de memória partilhada. Por exemplo, permite abstrair ao programador a noção de filas de espera. Embora estas não estejam visíveis para o programador, elas de facto existem, visto que sempre que um processo invoca um método bloqueante, e se este apenas estiver associado a *chords* com métodos assíncronos ainda não chamados, o processo terá que ser adicionado uma fila (de espera). Só quando todos os métodos assíncronos forem invocados, é que esse processo (ou outro) será retirado da fila e poderá prosseguir a sua execução.

3.3 Modelo de Actores em Scala

A linguagem Scala tem construções que permitem programar de forma concorrente através do modelo de actores [7] [31], seguindo um modelo diferente das linguagens anteriormente analisadas. Os actores são entidades que representam processos que comunicam entre si através do envio/recepção de mensagens. Estas trocas tanto podem ser síncronas como assíncronas. As mensagens são guardadas numa caixa de correio associada ao actor, podendo este, posteriormente, retirar a mensagem utilizando filtros, através de

```
class Ping(count: int, pong: Actor) extends Actor
{
  def act() {
    var pingsLeft = count − 1
    pong ! Ping
    while (true) {
      receive {
        case Pong =>
          if (pingsLeft % 1000 == 0)
            Console.println("Ping: pong")
          if (pingsLeft > 0) {
            pong ! Ping
            pingsLeft −= 1
          }
          else {
            Console.println("Ping: stop")
            pong ! Stop
            exit()
} } } } }
```

Figura 3.4: Exemplo da utilização de actores na linguagem Scala

pattern-matching. O comportamento destas entidades é normalmente definido de forma reactiva, isto é, para cada tipo de mensagem a receber é definida uma acção apropriada.

Considere-se o exemplo de uma classe Ping da figura 3.4 [6]. O comportamento de um actor é definido no método abstracto act, numa classe que herda a classe Actor. A expressão pong ! Ping consiste no envio de uma mensagem Ping para o actor pong. Depois temos a operação receive, dentro de um ciclo, que bloqueia o fio de execução do respectivo Actor até que chegue uma nova mensagem, fazendo *pattern matching* e tomando a acção apropriada. Num programa em que o número de actores seja elevado, sendo prejudicial a criação de muitos fios de execução, deve-se substituir a operação receive (thread-based) pela operação react (event-based):

```
loop {
  react {
    case A => ...
    case B => ...
} }
```

Neste caso, se todos os Actors seguirem esta estrutura, em vez de termos um fio de execução (a nível do Sistema Operativo) por cada um deles, temos apenas um no total. De notar que, obviamente, esta solução é péssima (a nível de eficiência) para programas onde se tenta tirar proveito do paralelismo em máquinas com múltiplos processadores.

Este modelo de programação do Scala, baseado em actores, é um modelo bastante intuitivo e simples de usar. O modelo de comunicação da linguagem Scala, juntamente com as operações **!** e ?, permite abstrair ao programador certos pormenores, tais como

a gestão das caixas de correio associadas a cada actor. De salientar que este suporte de programação concorrente está incorporado, na linguagem Scala, apenas a nível de bibliotecas, apesar de estar bem integrado na linguagem. Além disso, este modelo de programação baseia-se na troca de mensagens, não havendo partilha directa de recursos (a não ser que essa partilha seja simulada através de um actor) simplificando bastante o problema dos acessos concorrentes.

3.4 Erlang

Na linguagem Erlang [8], tal como no Scala, é oferecido ao programador suporte para programação concorrente, através do modelo de actores. No entanto, neste caso o suporte é nativo, ao contrário da linguagem Scala, onde o modelo de actores é oferecido através de bibliotecas.

Para criar um processo (actor) usa-se a primitiva spawn(M, F, [A1,...An]) onde M representa um módulo, F representa a função do módulo a executar (pelo novo processo) seguido de uma lista de argumentos. Para enviar uma mensagem (entre dois processos) usa-se o operador !. A primitiva receive permite bloquear um processo até que receba uma mensagem, fazendo posteriormente *pattern-matching* e tomando a acção apropriada. Considere-se o seguinte módulo echo, com duas funções bastante simples:

```
start() ->
    spawn(echo, loop, []).

loop() ->
    receive
        {From, Body} ->
            From ! Body, loop()
    end.
```

A função start permite inicializar um processo, que irá executar a função loop, que por sua vez fica bloqueado até que receba uma mensagem com dois valores, neste caso a origem e o corpo. Quando tal se suceder, é enviado para a origem uma mensagem com o mesmo corpo, e é repetido o procedimento. Vejamos agora um exemplo de comunicação entre dois processos:

```
Pid = echo:start(),
Pid ! {self(), hello}
```

Depois de ser associado ao identificador Pid um novo processo, é enviado para este uma mensagem, através do operador !, que contém o identificador do processo "principal" e o corpo da mensagem (hello). Para além das primitivas de criação de processos e envio/recepção de mensagens, temos a primitva register(Atom, Pid) que permite registar um processo, dado um Atom (alias) e o identificador do processo.

O envio de mensagens entre processos é não bloqueante. Quando um processo envia uma mensagem prossegue de imediato a sua execução, não tendo garantias absolutas de que a mensagem foi ou irá ser entregue. No entanto, é garantido que as sequências de mensagens enviadas entre dois processos diferentes terão ordem de chegada igual à ordem de envio (se as mensagens chegaram aos destinatários). Se num dado programa for necessário introduzir sincronização de processos, ela terá que ser programada explicitamente.

À semelhança da linguagem Scala, as mensagens não são enviadas/recebidas directamente entre processos. Quando uma mensagem é enviada, de um processo A para um processo B, é colocada na caixa de correio do processo B, que posteriormente, através da primitiva receive, trata de fazer *pattern-matching* e tomar a acção apropriada.

Os valores que são transmitidos entre processos (no envio/recepção de mensagens) são passados por cópia (e não por referência). Assim sendo, este modelo baseia-se única e exclusivamente no modelo de troca de mensagens, não existindo partilha de memória, permitindo excluir alguns erros frequentes na programação concorrente (*data races*). No entanto, erros resultantes de situações de impasse entre processos (*deadlocks*) também estão presentes neste modelo, como por exemplo, dois processos estarem à espera de receber uma mensagem entre si (um processo A estar à espera de receber uma mensagem do processo B e vice-versa). O mesmo acontece com as *race conditions*, visto que a ordem de entrega de mensagens a um processo, enviadas por outros dois processos, pode influenciar a execução do programa de acordo com a sua especificação.

3.5 Pict

Pict [33] é uma linguagem de programação concorrente baseada no Cálculo-π, onde múltiplas entidades, designadas por "Processos", comunicam e sincronizam entre si, através do envio e recepção de mensagens em canais. Começamos por ilustrar a linguagem com um exemplo bastante simples:

```
run ( new ch: ^[Int] ( (ch ! 1+1) | (ch ? x = x+1) ) )
```

Foi criado um canal *ch* do tipo inteiro, ou seja, um canal por onde podem passar valores do tipo inteiro. De seguida são criados dois processos, em paralelo, que comunicam entre si através desse mesmo canal. O primeiro processo calcula o valor da expressão 1+1 e envia para o canal *ch*. O segundo processo fica bloqueado até que receba uma mensagem (um inteiro) do mesmo canal. De seguida soma 1 ao valor recebido e retorna-o. Ao contrário do que sintaxe possa sugerir, a expressão (ch ? x = x + 1) não soma os valores x e 1 e guarda o resultado na variável x. Em vez disso, o inteiro recebido no canal *ch* é associado ao identificador x. De seguida está especificado o comportamento a tomar depois da recepção da mensagem, que é somar os valores x e 1, sendo retornado o valor da soma.

Vejamos agora como está estruturada a linguagem e as principais construções nela presentes:

$$Program \quad ::= \quad \textbf{run}\ Proc$$

$$\begin{array}{rcl} Proc & ::= & Val\ !\ Val \\ & | & Val\ ?\ Abs \\ & | & (Proc \mid Proc) \\ & | & (Decl\ Proc) \\ & | & \textbf{if}\ Val\ \textbf{then}\ Proc\ \textbf{else}\ Proc \end{array}$$

Um programa é criado mediante o comando run, seguido de uma expressão relativa a processos. Pode ser um envio de uma mensagem, onde por exemplo, a expressão (a ! b) envia para o canal (a) um valor (b). O mesmo raciocínio para a recepção de um mensagem (a ? b). No entanto, b representa um *pattern-matching* face ao valor recebido e a acção a tomar depois de ter sido recebido. O seguinte operador (|) diz respeito à composição paralela entre dois processos. De seguida temos a declaração, que permite associar identificadores a novos canais, especificar definições recursivas e abreviar tipos. Por último temos a construção de controlo de fluxos.

De salientar que os canais têm tipos associados. Se um canal for declarado com o tipo "^ Int", então só podem passar pelo próprio valores do tipo inteiro. Se o canal for do tipo "^ ^ Int" então já podem passar por ele canais do tipo "^ Int", um pouco á semelhança com os apontadores na linguagem C.

Vejamos agora um exemplo mais elaborado, onde o output do programa consiste na string "Hello world":

```
run
(
  new ch: ^[String String]
  (
   ch ! ["Hello" "world"]
  | ch ? a@[b c] = ( print!b | print!c )
  )
)
```

Em primeiro lugar, é criado um novo canal *ch*. De seguida são lançados, em paralelo, dois processos, sendo que um deles envia para o canal *ch* um registo com dois campos, mais exactamente duas strings. O outro processo fica à espera (no canal *ch*) de uma mensagem, fazendo *pattern matching* de um registo, associando a este o identificador *a* e aos seus campos os identificadores *b* e *c*. De seguida, tendo em conta que já recebeu a mensagem, lança dois processos, em paralelo, que enviam para o canal padrão de I/O os valores dos identificadores *b* e *c*.

A linguagem está desenhada, de raiz, a pensar na programação concorrente. Mediante o modelo de trocas de mensagens entre processos, baseado em canais, permite-nos construir, de forma bastantes simples e intuitiva, uma grande variedade de protocolos, assim como mecanismos de sincronização entre os processos. Mais uma vez, distingue-se do modelo de concorrência abordado nesta dissertação.

3.6 Go

A linguagem Go [1] é uma linguagem de programação imperativa e com suporte para concorrência, desenvolvida pela Google. Tem construções específicas para criação de múltiplos fios de execução e também oferece mecanismos de sincronização entre os mesmos, através de canais. Por questões de simplicidade, focamos a nossa análise apenas no suporte para concorrência.

Para lançar um novo fio de execução utiliza-se o comando go, seguido de uma chamada a uma função. A criação de canais, que irá permitir a troca de valores entre fios de execução, é realizada mediante a primitiva make(chan T), onde T representa o tipo do canal, isto é, o tipo dos valores que podem passar pelo canal. Considere-se o seguinte exemplo simples:

```
func add(n1 int, n2 int, ch chan int) { ch <− n1+n2 }
func main()
{
    ch := make(chan int)
    go add(1, 1, ch)
    fmt.Println( <−ch + 1 )
}
```

A função add soma dois inteiros e envia o resultado para um canal, através do operador <−. A função principal (main) cria um canal e inicia um novo fio de execução, que executa a função add. De seguida fica à espera que o fio criado lhe envie um valor para o canal criado e posteriormente imprime a soma entre o inteiro 1 e o valor recebido.

Considere-se o exemplo da figura 3.5 que calcula os números primos. A função *generate* gera números inteiros infinitamente, a partir do inteiro 2, enviando-os para o canal que recebe como argumento (*ch*). A função *filter*, como o próprio nome indica, filtra os números provenientes do canal *in*, com base no número primo dado como argumento, e envia-os para o canal *out*. A função principal (*main*) cria um canal (*ch*) e lança um fio de execução que gera vários números (*generate*) para o canal anterior. Depois, iterativamente, recebe um dos números gerados (na primeira iteração o valor 2), imprime o número anterior (que é primo), cria um novo canal (*ch*1), lança um novo fio de execução que filtra os valores entre os dois canais (*ch* e *ch*1), com base no número primo anterior, e finalmente define como canal corrente o segundo canal, e repete todo este procedimento.

```
func generate(ch chan int) {
    for i := 2; ; i++ {
        ch <− i
    }
}
func filter(in, out chan int, prime int) {
    for {
        i := <−in
        if i % prime != 0 {
            out <− i
        }
    }
}
func main() {
    ch := make(chan int)
    go generate(ch)
    for i := 0; i < 100; i++ {
        prime := <−ch
        fmt.Println(prime)
        ch1 := make(chan int)
        go filter(ch, ch1, prime)
        ch = ch1
    }
}
```

Figura 3.5: Cálculo de números primos na linguagem Go

O suporte para a concorrência, nomeadamente o modelo de trocas de mensagens através de canais, permite a construção de programas que tirem proveito do paralelismo de modo a aumentar o desempenho de um programa.

4

Máquinas Virtuais e Linguagens Intermédias

Actualmente os compiladores de linguagens (*e.g.*, Java, C#) geram código para uma linguagem intermédia próxima mas diferente do código de máquina. Como se pode observar na figura 4.1, o código de algumas linguagens de programação é compilado para código intermédio, que por sua vez é interpretado por uma máquina virtual. No caso particular das linguagens Java e C#, a maior parte dos compiladores existentes, para ambas as linguagens, compilam o código fonte para código das máquinas virtuais JVM [22] e CLR [15], respectivamente. Ambas as máquinas têm verificadores que analisam o código estaticamente, permitindo rejeitar uma boa parte de programas que possam provocar erros que coloquem em perigo a execução da própria máquina e dos dados envolvidos, quer seja por manipulação directa do *bytecode* quer seja por um *bug* do compilador. Quando uma classe é carregada para memória todos os seus métodos são analisados, de modo a detectar erros, tais como: tipos errados nos argumentos de cada operação, utilização da pilha de avaliação para além dos limites, saltos para locais ilegais do código, uso de variáveis locais não inicializadas, entre outros. O mecanismo de verificação é descrito na documentação técnica [22] como um processo de análise estática de fluxo de informação, no grafo de controlo do programa. No entanto, as especificações existentes não são exaustivas, originando implementações diferentes dos verificadores. Existem algumas abordagens formais a este problema, a sua maioria inspiradas em [21, 30, 38].

Tomando outro ponto de vista, os sistemas de tipos são mecanismos eficientes para garantir, também através da verificação estática de código, que certos erros de execução não ocorrem. Contudo, são geralmente aplicados às linguagens ditas de alto nível,

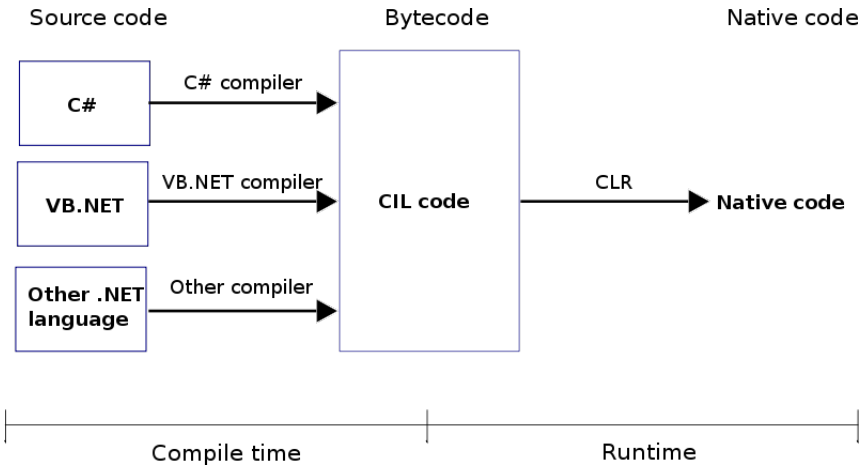

Figura 4.1: CLR Arquitectura

aproveitando o grau de abstracção das suas construções. No entanto, alguns sistemas de tipos tomam como ponto de partida linguagens mais primitivas [25–27], de modo a garantir que a tradução de código de uma linguagem de alto nível para código máquina preserva a informação de tipos de construções como *closures* e objectos. Ao preservar a informação de tipos, ao nível da linguagem intermédia, é possível demonstrar que o código produzido não contém os erros de execução detectados pelos sistemas de tipos das linguagens fonte. Deste modo, consegue-se garantir que um compilador é "correcto" em relação a uma especificação da linguagem de alto nível e que o código produzido pode ser carregado dinamicamente sem problemas.

As linguagens destino em [25–27] têm como destino máquinas de registos da família x86 em que, por exemplo, endereços do segmento de código são também valores, podem ser armazenados em registos e utilizados como argumentos para instruções de salto. Outra proposta semelhante às anteriores é a MIL [41], que se destaca por acrescentar instruções específicas para criação e controlo de concorrência, seguindo o modelo de memória partilhada, e um sistema de tipos capaz de rejeitar programas que apresentam potenciais situações de bloqueio, ou até mesmo *data races*, entre múltiplos fios de execução.

Por outro lado, as linguagens intermédias JVML [22] e CIL [15] são já linguagens tipificadas que, ao contrário das anteriores, utilizam análise de fluxos de dados. Também são, em certa medida, linguagens destino de compiladores que preservam informação de tipos a partir das linguagens Java ou C#. Em comparação com as linguagens [25–27] não são tão primitivas, apresentando um modelo de abstração superior, nomeadamente por terem presente um modelo de objectos implementado de raiz e com duas pilhas (uma avaliação de expressões e outra para chamadas de métodos). Outra proposta (orientada a objectos) é a linguagem intermédia iTalX [39], onde os tipos também são inferidos, em vez de serem explicitamente anotados no código. Tanto a JVML como a CIL não tratam da concorrência de forma nativa, ou seja, utilizam bibliotecas externas para suportar programas concorrentes, dificultando qualquer tentativa de implementar mecanismos de

verificação (*e.g.*, situações de bloqueio, *data races*) com auxílio do sistema de tipos.

Nas próximas secções apresentamos uma análise sobre linguagens intermédias e respectivas máquinas virtuais mais comuns (JVM e CLR), assim como sobre a TAL [25] e a MIL [41]. Também fazemos no final uma pequena referência à TyCO Virtual Machine, que é baseada num modelo de programação concorrente diferente das anteriores e do abordado neste documento (troca de mensagens).

4.1 CLR e JVM

A CLR e a JVM são máquinas virtuais bastante semelhantes, diferindo principalmente na sintaxe das respectivas linguagens intermédias, assim como alguns pormenores da semântica operacional e sistema de tipos. Por questões de simplicidade, apresentamos o modelo de execução e o sistema de tipos de uma só linguagem/máquina, mencionando apenas a sintaxe das instruções da CLR. Sempre que existam diferenças significativas serão mencionadas devidamente. Depois será apresentado um algoritmo de *garbage collection* utilizado numa das versões da CLR (Mono), o suporte para múltiplos fios de execução da CLR e JVM, e finalmente o processo de verificação de código incorporado nas mesmas.

Modelo de Execução

A CLR é uma máquina de pilha onde todas as instruções da respectiva linguagem intermédia (CIL) empilham/desempilham valores. A linguagem contém um modelo de objectos nativo, sendo os programas compostos por classes, que por sua vez têm campos e métodos. O modelo da máquina é composto por uma pilha de chamada e uma pilha de avaliação, que podem ter referências para objectos presentes na *heap*.

A pilha de chamada é composta por vários registos de activação, que guardam os argumentos e as variáveis locais de uma instância de um método. Sempre que é executada uma instrução relativa à chamada de um método (por exemplo, a instrução `call`) são desempilhados os argumentos, da pilha de avaliação do método instanciado anteriormente, e colocados no registo de activação da nova instância do método. Depois podem ser acedidos pelas instruções apropriadas (`ldarg.0`, `ldarg.1`, ...). De salientar que em algumas implementações da especificação da CLR (como por exemplo, a Mono implementada em C) não existe exactamente esta passagem/cópia de argumentos entre a pilha de avaliação e o registo de activação. É usada aritmética de apontadores para definir os argumentos do registo de activação corrente com base no apontador para o primeiro argumento presente na pilha de avaliação de uma instância anterior. No que diz respeito às variáveis locais, para cada uma pode ser associado um número lógico, para depois serem acedidas e alteradas mediante instruções próprias (`ldloc.0`, `ldloc.1`, `stloc.0`, ...).

A pilha de avaliação é uma estrutura associada a uma só instância de um método, que permite realizar cálculos e operações sobre valores previamente empilhados. Por

exemplo, a instrução ldc.i4 10 coloca o inteiro 10 no topo da pilha e a instrução add retira os dois primeiros elementos, calcula a soma e empilha o resultado. De salientar que, de acordo com a documentação [15], a pilha de avaliação é uma pilha lógica, podendo cada elemento ter um tamanho distinto. As operações também podem produzir efeitos laterais, isto é, efeitos para além da pilha de avaliação.

Sistema de Tipos

Os tipos da linguagem intermédia podem ser divididos em duas categorias principais: os tipos de valor e os tipos de referência. Dos tipos de valor destacam-se os escalares: inteiros, boleanos, caracteres, etc. Em relacção aos tipos de referência destacam-se os objectos e os apontadores.

Os tipos de objecto estão divídios em *arrays* e os derivados das classes. Os *arrays* são caracterizados pelo tipo dos seus elementos e pelo seu tamanho. A instrução newarr type permite criar um novo *array* com elementos do tipo *type*, deixando no topo da pilha uma referência para o mesmo. As instruções ldelem e stelem, a partir de uma referência para um *array* previamente empilhada, permitem empilhar o valor de uma posição do *array* e guardar um valor numa posição, respectivamente. Não existe suporte directo (na linguagem intermédia) para *arrays* com duas ou mais dimensões, mas podem ser programados através de *arrays* de *arrays*, e assim sucessivamente [15].

As classes são constituídas por campos, constructores e métodos. Para aceder ou alterar os valores dos campos, de um objecto, usam-se as intruções ldfld e stfld, respectivamente. A primeira instrução desempilha uma referência de um objecto e, dado o nome de um campo, coloca o seu valor no topo da pilha. A instrução stfld desempilha a referência do objecto e um valor e guarda-o no campo do objecto com o nome indicado. Para criar um novo objecto de uma determinada classe usa-se a instrução newobj (acompanhada do constructor a usar) que eventualmente retira valores do topo da pilha (argumentos) e que empilha uma referência para o objecto criado. Os métodos associados à classe podem ser estáticos ou não. Os métodos estáticos são invocados mediante a instrução call. Por exemplo, call **void class** [mscorlib]System.Console::Write(int32) retira o valor do topo da pilha (um inteiro) e executa o corpo do método estático Write. Neste caso, como o método é do tipo **void**, não é empilhado o valor de retorno. Por outro lado, métodos relativos aos objectos são chamados mediante a instrução callvirt. Para além de ser necessário empilhar os argumentos antes da instrução callvirt, é necessário, em primeiro lugar, empilhar a referência para o objecto. O próprio será o argumento com índice 0 do método instanciado. Dentro do método os argumentos são acedidos pela instrução ldarg., seguida do índice do argumento. As variáveis locais são definidas através da directiva .locals e acedidas ou alteradas pelas instruções ldloc e stloc. Ao contrário da CLR, onde os argumentos e variáveis locais são guardados em vectores separados [15], na JVM são guardados num único vector [22].

No caso particular da CIL, é possível encapsular valores primitivos em objectos através da instrução box, que retira do topo da pilha um valor e coloca uma referência para o valor encapsulado num objecto (guardado na *heap*). A instrução unbox desempilha uma referência de um objecto e coloca uma referência para o valor que antes estava encapsulado. Para empilhar depois o próprio valor utiliza-se a instrução ldobj, que retira do topo da pilha a referência e empilha o valor.

Para além dos objectos e respectivas referências, também existem outros tipos de referências, designadamente os apontadores, que se dividem em três categorias: *managed pointers*, *unmanaged pointer* e apontadores para funções. Os *managed pointers* são relativos a um campo de um objecto ou a uma posição de um *array*. Os *unmanaged pointers* são análogos aos apontadores da linguagem C, por exemplo. Neste caso, podem ser realizadas operações aritméticas, originando a possibilidade de erros de programação, se não forem cuidadosamente utilizados. Finalmente, os apontadores de funções permitem manipular referências para funções. As instruções ldvirtftn e ldftn permitem empilhar apontadores para funções relativas aos objectos ou estáticas, respectivamente. Por exemplo, ldvirtftn **void** f1(object) empilha um apontador para a função f1 (associada ao objecto do topo da pilha) com um objecto como argumento. Usa-se a instrução calli **void**(object) que verifica que a assinatura dada e a que está no topo da pilha são iguais e chama a função, desempilhado, para além do apontador da função e do objecto corrente (*this*), o (único) argumento dela. Como neste caso o método f1 não retorna nada (**void**) o resultado não é empilhado. De notar que, ao contrário da CIL, na JVML não há suporte para aritmética de apontadores e apontadores para funções.

Garbage Collection

O algoritmo de *Garbage Collection* varia consoante as implementações da CLR. No caso da implementação da CLR designada por Mono [5] é utilizado o *Boehm Garbage Collector*, que por sua vez utiliza o algoritmo *mark-sweep*. Este algoritmo considera que os objectos que podem ser acedidos directamente são aqueles que são referenciados através da pilha da instância do método (neste caso em particular, na pilha de avaliação e na pilha de chamada). A estas variáveis dá-se o nome de *roots*. Por outro lado, um objecto pode ser acessível, indirectamente, se for referenciado por um campo de um outro objecto (acessível). Portanto, o *roots* e todos os objectos acessíveis por estes, e assim sucessivamente, são ignorados pelo algoritmo. Já no caso dos objectos que não são directamente, nem indirectamente acessíveis (*trash*), o algoritmo será responsável por libertar o espaço reservado aos próprios.

Como mostra a figura 4.2, o algoritmo *mark-sweep* é decomposto em duas fases. Primeiro procura e marca todos os objectos acessíveis. Depois pesquisa pela *heap* e liberta memória dos objectos que não estão marcados. De notar que na fase de marcação o algoritmo ignora um objecto que já tenha sido marcado, visto que devido ao *aliasing* a pesquisa poderia originar ciclos. Assim sendo termina garantidamente e só termina a 1ª

```
for each root variable r
    mark(r);
sweep();

void mark(Object p) {
    if (!p.marked) {
        p.marked = true;
        for each Object q referenced by p
            mark (q);
    }
}

void sweep(){
    for each Object p in the heap
        if (p.marked)
            p.marked = false;
        else
            heap.release (p);
}
```

Figura 4.2: Algoritmo *mark-and-sweep*

fase quando todos os objectos acessíveis já tiverem sido marcados. Na 2ª fase, durante as pesquisas na *heap* por objectos não marcados, a libertação de espaço é realizada no momento.

Este algoritmo tem a principal vantagem de funcionar correctamente mesmo na presenças de ciclos de referências. A principal desvantagem consiste na execução do programa ter que ser suspendida quando o algoritmo é executado, principalmente em programas interactivos ou em sistemas de tempo real. Existem outros algoritmos que adoptam abordagens baseadas em dados empíricos, como é o caso do *Generational GC*, que assume que os objectos criados mais recentemente são aqueles que vão demorar mais tempo a estarem inacessíveis.

Suporte para Concorrência

A CIL não têm suporte nativo para concorrência, apesar das implementações da CLR suportarem múltiplos fios de execução recorrendo a bibliotecas externas. Por exemplo, nas implementações da CLR, no Sistema Operativo Windows, o mapeamento dos fios de execução, representado por a estrutura da figura 4.3, é realizado com auxílio da API de *Threads* do Windows. No caso particular da classe Thread do C#, o método Start (quando compilado para a linguagem intermédia) invoca um método externo que contém código nativo, que por sua vez trata do mapeamento do fio de execução entre o nível de utilizador e no nível de SO.

De forma semelhante temos o mesmo caso para a JVM. A linguagem intermédia JVML não apresenta suporte nativo para concorrência, sendo a mesma suportada pelas várias implementações da JVM, através de bibliotecas externas. O mapeamento dos fios

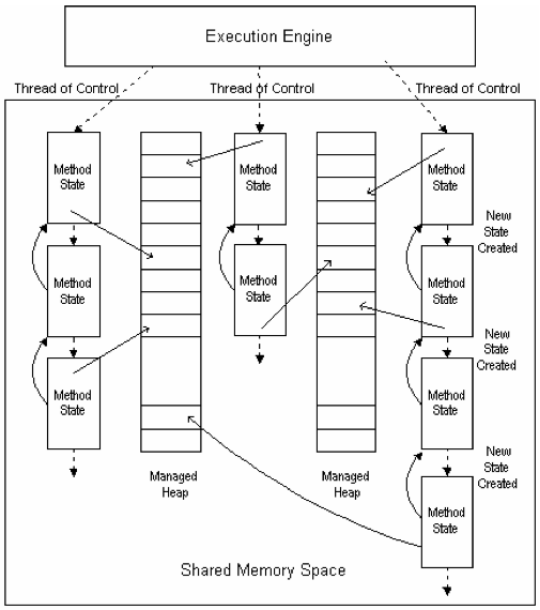

Figura 4.3: CLR Threads Arquitectura

de execução varia muito consoante as implementações. Nomeadamente existem duas formas bastante distintas de como os fios (*threads*) estão implementados: *green-threads* ou *native-threads*.

As *green-threads* não necessitam de suporte do sistema operativo. O próprio ambiente de execução da linguagem trata do escalonamento, da preempção, entre outros mecanismos. Têm como principal desvantagem o facto de os fios de execução não poderem ser distribuídos por múltiplos processadores (visto que são suportados pelo ambiente de execução). Têm como principal vantagem o facto de serem mais "leves"quando comparados com os fios de execução do nível do SO [36].

As *native-threads* estão disponíveis em implementações de JVM relativas a Sistemas Operativos com suporte para múltiplos fios de execução. São usadas bibliotecas do SO (*e.g.*, pthreads) para fazer o mapeamento entre os fios de execução de nível utilizador (*user level*) e os fios de baixo nível (*OS kernel level*). O Sistema Operativo depois trata de distribuir os fios de execução pelos vários processadores (ou núcleos), e também faz o escalonamento dos mesmos. Tem como principal vantagem a possibilidade de executar código em paralelo. Tem como desvantagem o facto de serem mais "pesados"que as *green-threads*, principalmente aquando da troca de contexto entre dois fios de execução [36].

A especificação da JVM contém duas instruções base para sincronização entre múltiplos fios de execução: monitorenter e monitorexit. A primeira retira do topo da pilha uma referência para um objecto e obtém o *lock* associado ao mesmo. A segunda também retira do topo da pilha uma referência para um objecto, mas liberta o *lock*. É com estas duas instruções que o bloco syncronized é compilado para bytecode. De relembrar que cada objecto está associado a um monitor. No que diz respeito às operações wait e notify (signal)

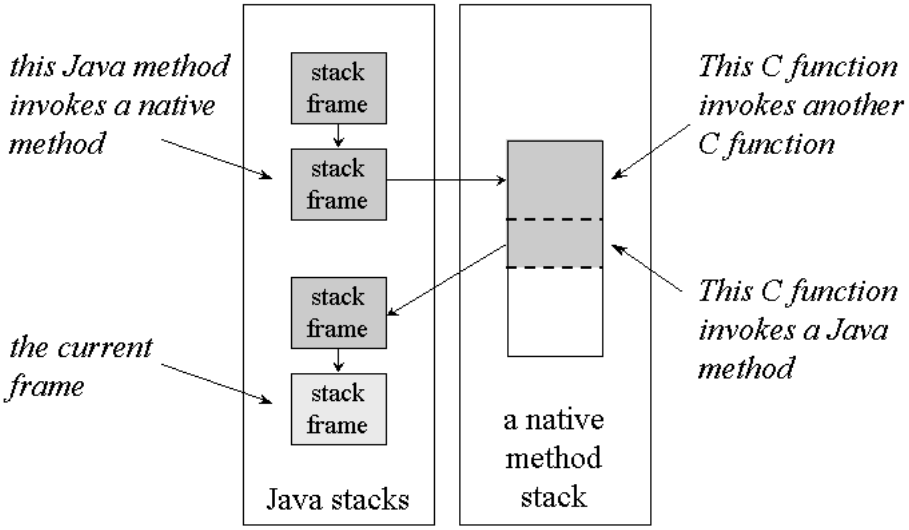

Figura 4.4: Chamada de métodos nativos na JVM [2]

dos objectos, não existem instruções base a nível do bytecode para as mesmas. Estas estão implementadas com chamadas a métodos nativos. O mesmo se aplica à criação de fios de execução. Quando um fio de execução é criado e iniciado, através do método start, a máquina virtual trata de criar uma nova pilha (Java stack) para o fio de execução [2]. Esta pilha é composta por registos de activação. Sempre que há a invocação de um método é empilhado um novo registo, que irá guardar o estado da chamada ao método. Ainda na criação do fio de execução, a chamada ao método start por sua vez chama o método nativo start0. Quando este método é chamado, por ser nativo, em vez de ser empilhado mais um registo na pilha (Java stack), é utilizada uma pilha nativa. Se a implementação da máquina estiver em C, então a pilha nativa será uma pilha em C (ver figura 4.4). É nesta fase que, em boa parte das implementações da JVM, é efectuado o mapeamento entre fios de execução em Java para fios nativos, com recurso à biblioteca `Pthread`, por exemplo. No entanto, para correr o código do fio de execução definido no método run, do código fonte, recorre-se novamente à pilha do Java reservada para o fio de execução, adicionando novos registos de activação a esta pilha à medida que novas chamadas são realizadas.

Actualmente, as mais recentes versões da HotSpot JVM, para Solaris, suportam concorrência com recurso aos fios de execução do próprio SO [3]. O mapeamento pode ser feito de duas maneiras: um-para-um ou muitos-para-muitos. No primeiro caso, um fio de execução de nível utilizador é mapeado directamente para um fio do núcleo do SO, com recurso às bibliotecas do próprio SO. Assim sendo, quando é criada um fio de execução, em Java, este acaba por ser gerido pelo escalonador do SO, competindo com outros fios de execução por ciclos do CPU. No caso do modelo muitos-para-muitos, vários fios de execução de nível utilizador são mapeadas em vários fios nativos. Se o número destes últimos for menor que o número de fios de nível utilizador, então o ambiente de execução

também terá que fazer gestão dos mesmos.

A principal desvantagem do suporte para concorrência presente na JVM e CLR, oferecido aos programadores ou aos compiladores de linguagens de alto nível através de classes biblioteca, consiste no facto de tornar mais difícil a tarefa de analisar estaticamente o código, com vista a detectar interferências entre os múltiplos fios de execução. Em secções seguintes iremos analisar casos em que o suporte para concorrência é nativo e quais as suas vantagens.

Verificador de Carregamento de Código

Nas máquina virtuais JVM [22] e CLR [15], sempre que uma classe é carregada para memória todos os seus métodos são analisados por um módulo, cuja função é detectar erros estaticamente. Para cada instrução é verificado se os valores retirados do topo da pilha são dos tipos esperados. Por exemplo, no caso da JVM, na verificação da instrução *iadd* é verificado se os dois elementos do topo da pilha são do tipo inteiro. Além disso, a próxima instrução a analisar irá encontrar no topo da pilha o tipo inteiro deixado pela instrução anterior (*iadd*), e assim sucessivamente, até se chegar ao fim do método. Através desta análise consegue-se detectar alguns erros, como por exemplo, somar um inteiro com uma referência de um objecto, utilização da pilha para além dos limites, saltos para locais ilegais, entre outros.

O problema surge ao analisar as instruções que envolvem saltos para outras instruções, porque neste caso uma instrução em vez de ter um antecessor passa a ter dois ou mais. Considere-se o seguinte exemplo (em pseudo-código) onde o 1º argumento é do tipo V, com um método m que não retorna valor, e o 2º do tipo A, com um método f que também não retorna valor. O tipo A é subtipo de V.

```
    ldarg 1
l0: call V.m()
    ldarg 2
    br l0
```

Quando se executa a instrução call pela 1ª vez, como anteriormente foi empilhado um objecto do tipo V, então a chamada do método m é válida. Depois é empilhado um objecto do tipo A e salta-se para a label $l0$. Anteriormente, quando se chegou a este ponto de execução a pilha tinha (apenas) o tipo V. Assim, como se chega (pela 2ª vez) ao mesmo local, e como o tipo do (único) elemento da pilha (A) é subtipo do tipo anterior (V) já não é necessário executar novamente a verificação. Vejamos agora um segundo exemplo:

```
    ldarg 2
l0: call A.f()
    ldarg 1
    br l0
```

Quando se executa a instrução call pela 1ª vez, como anteriormente foi empilhado um objecto do tipo A, então a chamada do método f é válida. De seguida é empilhado um objecto do tipo V e salta-se para a label $l0$. Anteriormente, quando se chegou a esta instrução a pilha tinha (apenas) o tipo A. No entanto, nesta 2ª vez que se chega ao mesmo ponto de execução, o tipo do (único) elemento da pilha (V) não é subtipo do tipo anterior (A). Portanto, é necessário efectuar uma segunda iteração, verificando-se (neste exemplo) que o código está mal tipificado.

Assim, o processo de verificação envolve *dataflow analysis*, onde cada ponto tem associado o tipo da pilha, ou seja, os tipos de cada elemento organizados em pilha. A função de união de dois antecessores consiste na união dos tipos das duas pilhas, isto é, consiste na união dos i-ésimos elementos (de ambas as pilhas). Desta união resulta o subtipo (mais próximo) entre ambos. No "pior caso", o resultado da união é o tipo *Object*. Existem algumas abordagens que eliminam a necessidade de fazer este tipo de análise, usando anotações no código [21] para qualquer instrução que seja alvo de um segundo antecessor (devido às instruções de salto). Neste caso, quando se analisa a instrução de salto, para uma determinada instrução etiquetada, basta comparar os tipos dos elementos da pilha "corrente"com aqueles que estão associados à etiqueta.

4.2 Linguagem Intermédia Tipificada

As máquina virtuais CLR e JVM são máquinas com alto nível de abstração, tendo as respectivas linguagens intermédias muitas instruções e algumas delas complexas. Têm também um modelo de objectos obrigatório para qualquer compilador que traduza uma linguagem de alto nível para estas linguagens intermédias. Por outro lado, linguagens intermédias e máquina virtuais com instruções muito básicas e em pequeno número, apesar de tornarem mais complexo o processo de compilação e a preservação de tipos (das linguagens fonte), permitem um suporte mais flexível para linguagens de alto nível e as respectivas optimizações (durante o processo de compilação).

A linguagem TAL [25] consiste numa linguagem intermédia, destinada a máquinas baseadas em registos, com instruções muito simples e com anotações no código que permitem verificar linearmente se o programa está bem tipificado, em tempo de carregamento, assegurando a ausência de boa parte dos erros que possam ser cometidos pelo programador, e também eliminado a necessidade de confiar no compilador da linguagem fonte.

Existem várias versões desta linguagem intermédia, pelo que analisamos apenas os princípios base da linguagem. O estado da máquina, à qual a linguagem se destina, é composto por uma *heap*, pelos registos e por uma sequência de instruções corrente. A *heap* é um mapa de etiquetas para sequências de instruções. O conjunto de registos é um mapa entre os registos e os respectivos valores, que podem ser inteiros ou etiquetas. As sequências de instruções são formadas por instruções de afectação de um valor a um registo, operações aritméticas, saltos condicionais e incondicionais. De salientar que

```
prod:
    r3 := 0;
    jump loop;

loop:
    if r1 jump done;
    r3 := r2 + r3;
    r1 := r1 + −1;
    jump loop

done:
    jump r4
```

Figura 4.5: Programa em TAL que calcula o produto entre dois registos

não se pode saltar para locais arbitrários do programa. Só se pode saltar para etiquetas presentes na *heap*. Também não é possível realizar operações aritméticas sobre etiquetas, o que irá permitir ao sistema de tipos assegurar certas propriedades básicas, como será explicado mais à frente.

Considere-se o exemplo da figura 4.5 onde é calculado o produto entre os registos r_1 e r_2. O primeiro bloco (*prod*) inicializa o resultado (r_3) a zero e salta para o segundo bloco (*loop*). Este bloco salta para o terceiro bloco (*done*) caso o registo r_1 tenha valor zero. Caso contrário soma o registo r_2 ao resultado, decrementa o registo r_1 e repete o procedimento. O último bloco (*done*) apenas salta para uma etiqueta, que se assume estar guardada no registo r_4.

O sistema de tipos, tendo em conta as instruções base da linguagem, assegura propriedades de segurança no controlo de fluxos, ou seja, assegura que o programa está bem tipificado mesmo na presença de saltos (condicionais ou incondicionais). Também não permite saltos para locais arbitrários do programa, permitindo apenas saltos para etiquetas presentes na *heap*. Para além dos tipos associados às etiquetas (Γ), também existe um tipo para os inteiros (*int*). Graças às etiquetas com os tipos associados, para se verificar que um bloco de instruções está bem tipificado basta assumir que o tipos dos registos para qualquer instrução de salto, de um segundo bloco, são iguais aos esperados. Depois, de forma composicional, tipifica-se cada instrução com base nos tipos dos registos anteriores, produzindo novos tipos de registos, e assim sucessivamente. O sistema de tipos também suporta polimorfismo, que permite tornar a linguagem mais flexível. É garantido formalmente que a máquina nunca atinge um estado incoerente (*does not get stuck*).

Vejamos novamente o exemplo da figura 4.5, considerando agora que os tipos dos 3 primeiros registos associados aos blocos *prod*, *loop* e *done* são do tipo inteiro. Facilmente verificamos que após a primeira instrução ($r_3 := 0$), do bloco *prod*, os tipos mantêm-se. Assim, quando se dá o salto para o bloco *loop* os tipos dos 3 primeiros registos são iguais aos esperados. Depois neste segundo bloco (*loop*), verificamos que o salto (condicional)

para o bloco *done* está bem tipificado, pois os tipos esperados tanto no segundo como no terceiro bloco são iguais (para além do registo r_1 ser do tipo inteiro). Continuando no segundo bloco, as próximas duas instruções são operações aritméticas sobre registos, pelo que não alteraram o tipo de cada um. Finalmente, quando se dá o salto (incondicional) para o bloco *done*, os tipos dos registos correntes são iguais aos esperados por esse bloco.

Continuando no exemplo anterior, falta definir qual o tipo do registo r_4 para os 3 blocos. Tem que ser obrigatoriamente um tipo de uma etiqueta, visto que é utilizado pela instrução de salto no bloco *done*. Essa etiqueta tem que ter os 3 primeiros registos com tipo inteiro, visto que, como vimos anteriormente, o bloco *done* tem associado o tipo inteiro aos 3 primeiros registos. O problema surge ao atribuir o tipo do r_4 (do tipo da etiqueta do registo r_4) e assim sucessivamente. Existem várias formas de resolver este problema, como por exemplo introduzindo os tipos polimórficos. Assim, para o código da figura 4.5 estar completo é necessário associar a cada etiqueta o tipo $\{r_1, r_2, r_3 : int,\ r_4 : \forall_\alpha code\{r1, r2, r3 : int,\ r_4 : \alpha\}\}$. Para além deste exemplo, o polimorfismo também é particularmente útil em programas com procedimentos (sequências de instruções) que não alteram determinados registos.

Até este ponto analisámos o núcleo da TAL. A linguagem possui mais instruções, nomeadamente para reservar espaço na *heap* (passando esta a ter, para além dos blocos de instruções, tuplos de valores), carregar/guardar valores entre os registos e a *heap*, entre outros. O sistema de tipos também consegue garantir propriedades de segurança na memória, nomeadamente verificar que nenhum programa lê ou altera valores (localizados na *heap*) excepto quando o respectivo acesso for autorizado numa determinada localização.

A TAL é, de facto, uma linguagem de máquina fortemente tipificada que, através das anotações de tipos no código, retira a necessidade de inferir os mesmos. Porém, não é fácil implementar compiladores que geram código para esta linguagem a partir de linguagens de alto nível, nomeadamente quando são orientadas a objectos. O mesmo se aplica na preservação de tipos entre as duas linguagens (fonte e alvo), em comparação com a CLR e a JVM. Também existe uma extensão designada por STAL (Stack-based TAL) [28] que acrescenta instruções para criar uma pilha, assim como transferir valores entre os registos e a pilha, entre outras. No entanto, não apresenta o mesmo nível de abstração da CLR e da JVM, as quais têm o modelo de objectos de origem e com pilhas de chamada e avaliação de expressões. Por outras palavras, apresentam níveis de abstracção bastante distintos, sendo o código para as máquinas JVM e CLR mais compacto.

4.3 Linguagem Intermédia Tipificada e Concorrente

A linguagem MIL [41] e a respectiva máquina virtual estão desenhadas, de raiz, em função do paradigma concorrente, mediante o modelo de memória partilhada. A máquina é baseada em registos e a arquitectura é composta por um conjunto de processadores e pela memória, que por sua vez é constituída por uma *heap* e por um conjunto de fios de

execução suspendidos, chamado *run pool*, para o caso de o número fios de execução ser maior que o número de processadores. Para cada fio suspenso é armazenado um par constituído por uma etiqueta associada ao código a executar, e um conjunto de valores relativos aos registos do processador (no momento em que foi criada). A *heap* é composta por tuplos de valores ou por segmentos de instruções. Os valores podem ser inteiros, *locks* (0 ou 1), registos ou referências para a *heap*.

As instruções presentes nesta linguagem intermédia estão organizadas em blocos e são as esperadas para uma máquina de registos, tais como: afectação de valores a registos, saltos condicionais e incondicionais, *loads/stores* entre os registos e a *heap*, entre outros. No entanto, também possui instruções específicas para criação e controlo de fios de execução, nomeadamente a instrução fork que cria um novo fio, que executa instruções associadas a uma dada etiqueta (de um bloco). Na verdade o fio de execução não é lançado imediatamente, mas sim colocado na *run pool*, onde permanece até que pelo menos um processador não esteja a realizar "trabalho útil", isto é, esteja livre e tome a iniciativa de executar as instruções nele contidas. Sempre que um processador selecciona um fio de execução são carregados os valores dos registos a ele associados. Um fio de execução depois só liberta o processador quando executa a instrução *yield*, não existindo mecanismos de preempção presentes na maior parte dos Sistemas Operativos [36].

Para além das instruções de criação e terminação de fios de execução, também estão disponíveis instruções para sincronização entre fios concorrentes, através de *locks*. Para criar um *lock* recorre-se à instrução newLock, acompanhada de um valor (0 ou 1). Com a instrução testSetLock, para um dado lock, conseguimos escrever um valor 1 e obter o valor antigo. Esta operação é realizada atomicamente. Assim, com auxílio desta instrução e sucessivas iterações, a aquisição de um *lock* pode ser realizada em modo de espera activa. Posteriormente, para libertar um dado *lock*, usa-se a instrução unlock. De salientar que nas aquisições de *locks* (quer através da criação ou mediante a instrução testSetLock) são adicionadas permissões ao processador "corrente", para depois serem garantidas propriedades no sistema de tipos. A figura 4.6 ilustra o modelo e a utilização dos *locks* nesta linguagem intermédia.

Os tuplos, presentes na *heap*, são vectores de valores protegidos por um lock α. Este *lock* não têm influência a nível de execução, funcionando apenas como uma asserção, conforme será explicado mais à frente. Por outro lado, blocos contidos na *heap* são compostos por uma assinatura e um corpo. A assinatura tem o formato Γ *requires* Λ, que representa os tipos de cada registo e os *locks* obtidos (e ainda não libertados) que são esperados antes de ser executado o corpo.

O exemplo da figura 4.7 representa uma solução, com recurso a múltiplos fios de execução, para o acesso exclusivo a uma região crítica. No bloco main é criado um *lock*, com valor inicial 0, isto é, disponível para ser requerido, e associado a α. De seguida é reservado espaço na memória para um inteiro protegido por α, e salta-se para o bloco *enterSleepRegion*. Neste bloco é realizada uma tentativa de adquirir o *lock* (com a instrução testSetLock). Em caso positivo salta-se para o bloco relativo à região crítica. Caso contrário

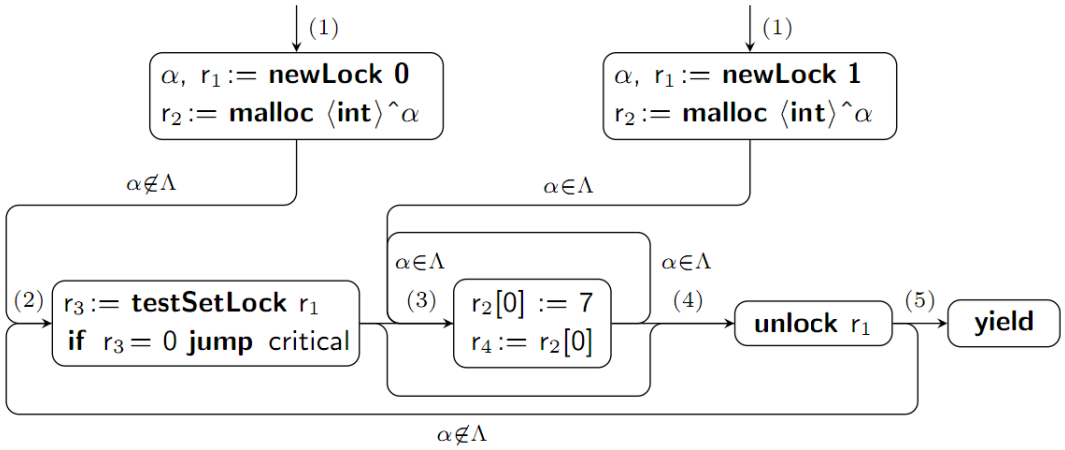

Figura 4.6: Modelo e utilização dos *locks* na MIL

```
enterSleepLockRegion(r1:Tuple, r2:lock(α)^α) {
    r3 := testSetLock r2
    if r3 = 0 jump criticalRegion
    fork enterSleepLockRegion
    yield
}
criticalRegion(r1:Tuple, r2:lock(α)^α) requires α {
    r1 := ...
    unlock r2
    jump continuation
}
main() {
    α, r2 := newLock 0
    r0 := malloc [int]^α
    jump enterSleepLockRegion
}
```

Figura 4.7: Exemplo de um programa na MIL

é lançado um novo fio de execução que repete o procedimento do fio "pai" (que liberta o processador mediante a instrução yield). O bloco relativo à região crítica tem associado a precondição α, ou seja, o processador deve ter adquirido o *lock* quando salta para este bloco, onde depois altera o primeiro valor do tuplo associado ao registo r1, liberta o *lock* e continua a execução.

O sistema de tipos da linguagem intermédia permite um bom funcionamento e controlo seguro entre os múltiplos fios em execução. Um fio de execução só pode libertar o processador (yield) se todos os *locks* adquiridos tiverem sido previamente libertados, evitando potenciais situações de bloqueio. Obviamente que também só pode libertar um *lock* previamente adquirido. Após um fio de execução lançar um segundo (mediante a instrução fork) o sistema de tipos obriga a que o conjunto de *locks* adquiridos seja repartido em dois conjuntos disjuntos, um para cada fio de execução. Os *locks* associados à

etiqueta do bloco, que a novo fio de execução vai executar, são associados ao próprio. Os restantes permanecem no fio de execução pai. De salientar que esta forma é mais correcta que outras abordagens, que consistem em o primeiro fio fazer *unlock* e o segundo *lock*, para passar o *lock* entre os dois fios, podendo um terceiro fio de execução entretanto adquirir o *lock*.

No momento de criação de um *lock* por um determinado fio de execução, com valor inicial igual a 1, é lhe associado esse mesmo *lock* ao seu conjunto de permissões adquiridas. Já os *locks* que são requeridos mediante a instrução testSetLock só são adicionados (à lista de permissões) após ser analisada a instrução de salto condicional para um dado bloco (que tem um conjunto de permissões associado).

As regras de tipos para as instruções load, store, entre outras, são as esperadas para uma máquina baseada em registos, excepto que neste caso têm em conta os *locks* anteriormente descritos. No caso da instrução que reserva espaço na *heap* (malloc), acompanhada de um *lock* α, é verificado se o *lock* α está no ambiente de tipificação. Tanto a regra para a instrução load, como para a instrução store, obrigam a que o fio de execução tenha adquirido o *lock* que está associado à zona de memória que está a ser lida ou escrita, respectivamente. Os *locks* desta linguagem estão implementados com *singleton types* [37].

Graças ao sistema de tipos descrito anteriormente, nomeadamente a utilização disciplinada dos *locks*, e a associação destes a cada tuplo na *heap*, consegue-se garantir ausência de situações de bloqueio e *data races*. Os autores desta linguagem intermédia [41] também asseguram outra propriedade importante. Um programa bem tipificado nunca origina um estado da máquina incoerente (*does not get stuck*).

Esta linguagem intermédia, face à TAL proposta por Greg Morisset, tem como principal vantagem o suporte nativo para criação e controlo de concorrência. Também consegue garantir várias propriedades interessantes (ausência de *data races* e situações de bloqueio) através do seu sistema de tipos. Tal como a TAL, não apresenta o mesmo nível de abstracção da JVML e da CIL, nomeadamente o modelo de objectos (de raiz) e as pilhas de chamada e avaliação.

4.4 TyCO VM

A linguagem TyCO (Typed Concurrent Objects [40]) é baseada no Cálculo-π, onde um número variado de processos comunica entre si por troca de mensagens, que são enviadas ou recebidas através de canais. Para além deste modelo de troca de mensagens, também tem incorporado um modelo de objectos. A sintaxe da linguagem é apresentada na figura 4.8. Um processo (P) pode estar no estado terminal (**inaction**), pode ser lançado em paralelo ($P \mid P$) ou pode criar novos canais (**new** $\overline{x}$ P). Um método com uma dada etiqueta (l_i) num determinado objecto (x ? $\{$ $\overline{l(\overline{x}) = P}$ $\}$) é invocado através do envio de uma mensagem (x ! $l_i[\overline{e}]$), dando origem a um processo (P_i), onde os identificadores ($\overline{x}_i$) são associados às expressões ($\overline{e}$). A este mecanismo dá-se o nome de "comunicação". De forma análoga temos o mecanismo de "instanciação" ($X_i[\overline{e}]$) onde é seleccionado um

$$
\begin{aligned}
P &::= \textbf{inaction} \\
&\mid P \mid P \\
&\mid \textbf{new}\ \overline{x}\ P \\
&\mid x\ !\ l[\vec{e}] \\
&\mid x\ ?\ M \\
&\mid X[\vec{e}] \\
&\mid \textbf{def}\ D\ \textbf{in}\ P \\
&\mid \textbf{if}\ e\ \textbf{then}\ P\ \textbf{else}\ P \\
&\mid (P) \\
D &::= \overline{X(\overline{x}) = P} \\
M &::= \{\ \overline{l(\overline{x}) = P}\ \} \\
e &::= e_1\ op\ e_2 \mid op\ e \mid x \mid c \mid (e)
\end{aligned}
$$

Figura 4.8: Sintaxe da linguagem TyCO

```
def Cell(self, u) =
self ? {
    read(r) = r![u] | Cell[self, u],
    write(v) = Cell[self, v]
}
in new x Cell[x, 9] | new y Cell[y, true]
```

Figura 4.9: Exemplo de um programa na linguagem TyCO

template (X_i) da declaração de *templates* ($\overline{X(\overline{x}) = P}$), dando origem a um processo (P_i). Também é possível construir definições recursivas (**def** D **in** P) e controlos de fluxos (**if** e **then** P **else** P). No exemplo da figura 4.9 está programada uma célula com as operações de leitura e escrita. O identificador *self* representa o canal para comunicação e o identificador u representa o valor da célula. Se no canal for recebido uma mensagem com a etiqueta *read* então é enviada uma resposta (mensagem) com o valor da célula, mantendo-se o próprio valor. No caso de a etiqueta ser *write* dá-se origem a uma nova instância com o novo valor recebido.

A máquina virtual [23] que suporta a linguagem (anteriormente definida) é baseada em registos e a sua arquitectura é composta pelo código do programa, *heap*, fila de fios de execução, pilha de canais e pilha de avaliação. O código do programa é composto pelas instruções em formato orientado ao *byte*. Na *heap* encontram-se armazenadas as estruturas faladas anteriormente, tais como mensagens, objectos, canais, entre outros. O bloco mais básico presente na *heap* é uma *word*. As *frames* por sua vez são constituídas por várias *words* e podem ser alocadas mediante instruções específicas, que serão mencionadas mais à frente. Quando se dá o mecanismo de comunicação ou instanciação (descritos anteriormente) um novo fio de execução é criado e colocado na fila dos fios de execução

(até ser seleccionado pelo escalonador). O fio de execução é composto por um apontador (para a primeira instrução a executar) e um conjunto de apontadores para canais. Quando se cria um novo canal é lhe associado uma *frame* alocada na *heap* e o respectivo apontador é guardado na pilha de canais. Apesar de os apontadores serem destruídos após o fio de execução terminar, o respectivo valor presente na *heap* é mantido pois podem existir outros apontadores exteriores ao ambiente local. Só depois de ser lançado o algoritmo de *Garbage Collection* (quando a memória disponível não for suficiente) o valor é eliminado da *heap*. Na pilha de avaliação, como seria de esperar, são colocados valores e realizadas operações sobre os mesmos. Para além destas estruturas de dados, a máquina contém instruções que manipulam diversos registos (*e.g.*, *Heap Pointer*, *Program Counter*, *Operand Stack*, *Channel Stack*, *Current Stack*, *Current Frame*).

Considere-se o exemplo da figura 4.10, onde é apresentado o código originado pelo compilador na tradução do exemplo da figura 4.9. Na definição da célula é criado um novo objecto com dois parâmetros (objf) sendo-lhe associado uma nova *frame* na *heap*. Posteriormente são copiados os dois parâmetros (o canal e o valor da célula) para a *frame* corrente (put). De seguida é definido o comportamento do objecto mediante as mensagens recebidas. No caso de uma mensagem com etiqueta *read* é reservada na *heap* uma *frame* para uma mensagem com um argumento (msgf), coloca-se o valor da célula (put), define-se o canal recebido (*r*) como o canal corrente e envia-se a mensagem (trmsg), e finalmente cria-se uma nova instância da célula (instof) com os mesmos argumentos (canal e valor da célula). De salientar que o envio da mensagem na verdade corresponde ao mecanismo de redução (comunicação) onde é lançado um novo fio de execução que irá executar o método com a etiqueta dada (na mensagem).

Caso a mensagem recebida tenha etiqueta *write* é criada uma nova instância da célula (instof), cujo primeiro argumento continua a ser igual ao canal da célula, e o segundo passa a ser o valor enviado na mensagem. Depois da definição da célula (descrita anteriormente) é apresentado o código responsável pelo início da execução do programa. É criado um novo canal (newc) e é instanciada uma célula (instof) com o 1º parâmetro igual ao canal anteriormente criado e o 2º com o inteiro 9. De forma análoga, de seguida cria-se um novo fio de execução para uma célula com um novo canal e o valor *true*.

A implementação desta máquina foi realizada na linguagem C [23]. No momento anterior à execução de um programa são associados os códigos das instruções às respectivas funções em C, sob a forma de endereço, de forma a aumentar a eficiência da máquina virtual. A estrutura principal do interpretador consiste num ciclo que, sucessivamente, vai processando as várias instruções, até eventualmente não existirem mais fios de execução na máquina. Em suma, trata-se de uma máquina virtual (e respectiva linguagem intermédia e de alto nível) seguindo um modelo distinto (troca de mensagens) tratado nesta dissertação (memória partilhada), sendo principalmente indicada para programas compilados de linguagens baseadas no Cálculo-π.

```
main = {
    def Cell = {
        objf 2
        put p0
        put p1
        trobj p0 = {
            {read, write}
            read = {
                msgf  1,0
                put f1
                trmsg p0
                instof 2, Cell
                put f0
                put f1
            }
            write = {
                instof 2, Cell
                put f0
                put p0
            }
        }
    }
    newc c0
    instof 2, Cell
    put c0
    put 9
    newc c1
    instof 2, Cell
    put c1
    put true
}
```

Figura 4.10: Resultado da compilação do exemplo da figura 4.9

5

Uma Linguagem de Programação Concorrente

Apresentamos neste capítulo uma linguagem baseada na linguagem *core* proposta por Caires e Seco [11]. Trata-se de uma linguagem imperativa, orientada a objectos, com expressões específicas para exprimir programas concorrentes e para exprimir controlo de concorrência. Sendo uma linguagem exclusivamente de expressões, destacam-se as expressões **fork** e que denota um valor especial que identifica um fio de execução, a expressão **wait** e que denota o resultado da computação feita pelo fio de execução denotado pela sub-expressão e, a expressão **shared**$(e_1)\{e_2\}$ que denota o valor da expressão e_2, quando executada em partilha com outros fios de execução que também executem um bloco **shared** sob o *lock* denotado pela expressão e_1, e a expressão **sync**$(e_1)\{e_2\}$ que corresponde à execução em exclusão mútua da expressão e_2 em relação ao *lock* denotado pela expressão e_1. Introduzimos agora a intuição destas construções por intermédio de exemplos simples. Considere-se a seguinte expressão:

val x = **fork**(1+1), y = **fork**(2+2) **in** (**wait** x) + (**wait** y)

onde a declaração de identificadores tem a sintaxe **val** x = e_1 **in** e_2. Neste caso, os identificadores x e y denotam dois fios de execução criados pela expressão **fork**. Ambos avaliam em concorrência, respectivamente, as expressões 1+1 e 2+2. O fio de execução, que inicialmente executa a expressão principal, espera o resultado dos dois fios de execução denotados por x e y para produzir o resultado final. Considere agora o exemplo da figura 5.1, no qual existe uma zona de memória partilhada. Verifica-se que para evitar conflitos entre fios de execução utiliza-se a expressão **sync**. A declaração de variáveis, neste exemplo, é semelhante à das constantes indicando sempre o âmbito da variável

```
var x, l in
    x := 5;
    l := newlock;
    val
        y = fork sync(l){x := x−1},
        z = fork sync(l){x := x+1}
    in
        (wait y) + (wait z)
```

Figura 5.1: Exemplo de utilização da primitiva sync

```
var x, l in
    x := 5;
    l := newlock;
    val
        w1 = fork sync(l){x := x−1},
        w2 = fork sync(l){x := x+1},
        r1 = fork shared(l){x−1},
        r2 = fork shared(l){x+1}
    in
        (wait r1) + (wait w1) +
        (wait r2) + (wait w2)
```

Figura 5.2: Exemplo de utilização das primitivas sync e shared

(**var** $\overline{x}$ **in** e). Os *locks* são valores especiais sobre os quais se podem aplicar as operações sync e shared. Neste caso, atribuído o valor 5 à variável x e inicializado o *lock* l, são lançados dois fios de execução que decrementam/incrementam o valor de x, respectivamente. Mais uma vez, o fio principal espera pelos fios de execução denotados por y e z, para calcular o resultado final. O exemplo da figura 5.2 usa a expressão **shared** para executar expressões em partilha com outros fios de execução. Neste caso existem dois fios de execução que apenas consultam o valor da variável x. Assim, podem-no fazer em partilha com outros leitores.

Um programa nesta linguagem, cuja sintaxe está definida na figura 5.3, é constituído por uma série de classes. Cada uma, associada a um identificador (c), é composta por variáveis de instância e por métodos. Apenas por questões de simplicidade, consideramos que todos os métodos são de instância. A assinatura do método é composta pelo identificador (m), pelo tipo de retorno (τ), e pelos nomes e tipos dos argumentos ($\overline{\tau\ x}$). O corpo dos métodos é baseado apenas em expressões, não existindo comandos. As expressões manipulam os valores inteiros, *locks*, *threads* e objectos. Os operadores para os valores inteiros são representados pelo símbolo *op*. A linguagem inclui a declaração de identificadores (**val** $\overline{\tau\ x = e_1}$ **in** e_2), onde os identificadores têm como âmbito a expressão e_2, de variáveis (**var** $\overline{\tau\ x}$ **in** e), a sequenciação de expressões ($e\ ;\ e$) e a atribuição ($x := e$). Também inclui as expressões habituais de controlo de fluxo (**while** $(e)\ \{e\}$) e (**if** e **then** e **else** e), assim como as que dizem respeito aos objectos, nomeadamente a criação (**new** c) e a invocação de um método ($e.m(\overline{e})$).

P	::=	$\overline{C}$	(Program)
C	::=	**class** c { $\overline{F}$ $\overline{M}$ }	(Class)
F	::=	τ f	(Field)
M	::=	τ $m(\overline{\tau\ x})$ { e }	(Method)

e	::=		(Expression)
	|	x	(Identifier)
	|	n	(Integers)
	|	$!e$	(Negation)
	|	op	(Binary Operations)
	|	e ; e	(Sequence)
	|	x := e	(Assign)
	|	**while** (e) $\{e\}$	(Loop)
	|	**if** e **then** e **else** e	(Decision)
	|	**var** $\overline{\tau\ x}$ **in** e	(Variables Declaration)
	|	**val** $\overline{\tau\ x\ =\ e}$ **in** e	(Constants Declaration)
	|	**new** c	(New Object)
	|	$e.m(\overline{e})$	(Method Call)
	|	**fork** e	(Fork)
	|	**wait** e	(Wait)
	|	**newlock**	(New Lock)
	|	**shared** (e) $\{e\}$	(Shared)
	|	**sync** (e) $\{e\}$	(Synchronized)

Figura 5.3: Sintaxe da linguagem concreta

A grande parte das expressões acima segue de perto o universalmente aceite para uma linguagem de expressões, imperativa e orientada a objectos. Descrevemos agora as restantes expressões cujo significado é bastante relevante para este trabalho. A expressão (**fork** e) avalia a expressão e num novo fio de execução e denota um valor do tipo *thread*. A expressão bloqueante (**wait** e) denota o valor da expressão avaliado pelo fio denotado pela expressão e. Intuitivamente, a semântica destas expressões é tal que **wait** $($**fork** $e)$ denota o mesmo valor que a expressão e. A expressão **newlock** cria um novo *lock*. As expressões (**sync**$(e_1)\{e_2\}$) e (**shared**$(e_1)\{e_2\}$) controlam o acesso dos fios de execução às zonas críticas protegidas pelo *lock* denotado pela expressão e_1, respectivamente em exclusividade ou em partilha com outros fios de execução também dentro de um bloco **shared**. No caso da expressão (**sync**$(e_1)\{e_2\}$), só um fio de execução pode executar um bloco **sync** para o mesmo *lock*, ficando todos os outros fios de execução bloqueados, à espera da sua vez de executar (**sync** ou **shared**). No caso da expressão (**shared**$(e_1)\{e_2\}$), podem executar um número indeterminado de fios de execução, também com expressões **shared** sobre o mesmo *lock*, ficando todos os fios de execução com as expressões **sync** bloqueados, à espera que todas as expressões **shared** sejam avaliadas na totalidade. O valor da expressão, em ambos os casos, é o valor da sub-expressão e_2. Estas duas expressões implementam o conhecido padrão concorrente de um escritor e múltiplos leitores [18].

```
class IntList {
    IntNode head;
    IntNode tail;
    int size;

    int getHead() { head.getInt() }
    int getTail() { tail.getInt() }
    int getSize() { size }

    int remove() {
        var int tmp in
            (tmp := head.getInt();
            head := head.getNext();
            tmp)
        ;
        size := size−1
    }

    int add(int i) {
        if size > 0 then
            var IntNode tmp in
                tmp := new IntNode;
                tmp.setInt(i);
                tail.setNext(tmp);
                tail := tmp
        else
            (head := new IntNode;
            head.setInt(i);
            tail := head)
        ;
        size := size+1;
    }

    int main() {
        var IntList list in
            list := new IntList;
            list.add(10);
            var lock l, th⟨int⟩ r1, r2, w1, w2 in
                l := newlock;
                r1 := fork shared(l){list.getHead().getValue()};
                r2 := fork shared(l){list.getHead().getValue())};
                w1 := fork sync(l){list.add(100)};
                w2 := fork sync(l){list.remove()};
                wait r1; wait r2; wait w1; wait w2;
                list.getHead().getValue()
    }
}
```

Figura 5.4: Exemplo de definição e utilização de uma lista ligada

5.1 Lista Ligada Concorrente (Exemplo)

Como exemplo mais elaborado da linguagem, considere-se o da figura 5.4, envolvendo uma lista ligada e os acessos concorrentes à própria. A lista é composta por um inteiro que indica o número de elementos, e por dois nós que representam a cauda e a cabeça, e que guardam um inteiro e o próximo nó (apontador). As operações são as usuais: consultar tamanho, primeiro valor, último valor, remover à cabeça e adicionar à cauda.

No método que inicia a execução (*main*), é adicionado o valor 10 à lista. De seguida, são declarados, e mais tarde inicializados, um *lock* (para a proteger a lista), dois leitores ($r1$ e $r2$) e dois escritores ($w1$ e $w2$), em paralelo. Os leitores apenas consultam elementos da lista, estando o acesso protegido numa expressão shared. Os escritores modificam a lista pelo que se torna necessário utilizar uma expressão sync, para não haver interferências. No final, o fio de execução "principal" espera que todos terminem e devolve o valor da cabeça, que será garantidamente o inteiro 100, independentemente da ordem de escalonamento dos múltiplos fios de execução.

De seguida, ilustramos a semântica por tradução para a linguagem Java, recorrendo a algumas classes de suporte. Posteriormente definimos a semântica formal da linguagem, assim como o seu sistema de tipos.

5.2 Tradução para Java

Podemos traduzir as construções (anteriormente definidas) para Java, tendo boa parte delas uma tradução directa. Porém, existem quatro expressões onde a tradução não é linear: fork, wait, shared e sync. As duas primeiras são traduzidas para expressões com fios de execução em Java (da classe `java.lang.Thread`), enquanto a implementação das primitivas shared e sync envolve a implementação de monitores com auxílio do pacote `java.util.concurrent`.

Os fios de execução, na linguagem que queremos implementar, denotam um valor. Definimos o interface ThreadValue com dois métodos: start e join. O primeiro cria e inicia um novo fio de execução, e retorna o próprio objecto que o representa. O segundo espera pela conclusão da expressão a executar e retorna o valor computado:

```
interface ThreadValue⟨T⟩ extends Runnable {
    public ThreadValue<T> start();
    public T join() throws InterruptedException;
}
```

Neste contexto, a tradução da primitiva fork é feita com base na criação de um objecto, que implementa a interface ThreadValue, em que é encapsulado, no seu método run, a tradução da expressão a avaliar. Depois de criado o objecto, é invocado o método start que inicia o fio de execução e retorna o próprio objecto que o representa:

```
[[fork E]] ≜ new ThreadValue<T>() {
                 private T val;
                 private Thread t;
                 public void run() { this.val = [[E]]; }
                 public ThreadValue<T> start() { t = new Thread(this); t.start(); return this; }
                 public T join() throws InterruptedException { t.join(); return this.val; }
             }.start();
```

No caso da expressão wait e é traduzida a sub-expressão e, que deverá corresponder a um fio de execução representado pela interface ThreadValue, sendo de seguida invocado o método (bloqueante) join, que retorna o resultado calculado pelo fio de execução:

$$[\![\textbf{wait } E]\!] \triangleq [\![E]\!].\text{join}()$$

As expressões shared e sync são implementadas com base no conceito de Monitor. O próprio Java já oferece suporte para monitores, tais como a primitiva **synchronized** e os métodos wait, notify e notifyAll da classe Object. No entanto, existem algumas limitações, como por exemplo o facto de não permitir estarem associadas, ao mesmo monitor, duas ou mais filas de espera. Assim, decidimos utilizar o pacote `java.util.concurrent` que permite associar ao mesmo monitor várias filas de espera. Na figura 5.5 apresentamos, parcialmente, a classe de suporte para a implementação dos *locks* na linguagem (ver anexo 9.1 para código completo). O objecto monitor contém um objecto da classe ReentrantLock, duas variáveis condição (uma para os leitores e outra para os escritores) associados a esse *lock*, e um valor inteiro que pode estar dentro de um de três conjuntos para representar o estado do monitor: -1 se está um escritor activo no monitor; 0 se não está nenhum leitor nem nenhum escritor activo no monitor; qualquer valor positivo, indicando o número de leitores activos. Todos os métodos do monitor são executados em exclusão mútua, visto que acedem e alteram variáveis partilhadas.

A expressão **shared**$(e_1)\{e_2\}$ é traduzida na sequência composta pela avaliação da expressão que denota o *lock*, a aquisição do direito de leitura (startShared), a tradução do corpo, seguida da libertação do mesmo *lock* (endShared):

$$[\![\textbf{shared}(e_1)\{e_2\}]\!] \triangleq \{Monitor\ m = [\![e_1]\!];\ m.startShared();\ [\![e_2]\!];\ m.endShared();\}$$

Assim, o método startShared bloqueia o fio de execução se um escritor tiver adquirido, e ainda não tiver libertado, o direito de escrita (contador < 0). O fio de execução pode avaliar a expressão e_2 quando garantidamente não houver escritores activos para esse monitor.

A expressão **sync**$(e_1)\{e_2\}$ é traduzida pela sequência composta pela avaliação da expressão que denota o *lock*, a aquisição do direito de escrita (startSync), a tradução do corpo, seguida da libertação do mesmo *lock* (endSync):

$$[\![\textbf{sync}(e_1)\{e_2\}]\!] \triangleq \{Monitor\ m = [\![e_1]\!];\ m.startSync();\ [\![e_2]\!];\ m.endSync();\}$$

```
class Monitor {
    private ReentrantLock mon;
    private Condition canRead;
    private Condition canWrite;
    private int count;

    public Monitor() {...}
    public void startShared() throws InterruptedException {...}
    public void endShared() {...}
    public void startSync() throws InterruptedException {...}
    public void endSync() {...}
}
```

Figura 5.5: Esquema da classe de suporte (Monitor)

O método startSync adormece o fio de execução se algum escritor, ou algum leitor, estiverem activos no monitor (contador != 0). O fio de execução só avalia a expressão e_2 quando garantidamente é o único activo no monitor.

De salientar que tanto no startShared como no startSync optámos por seguir uma abordagem em que, um fio de execução após ter sido acordado verifica novamente a condição, mesmo que o fio de execução que acorda (o outro) assegura essa mesma condição, isto porque não há garantias, tanto no Java, como no pacote `java.util.concurrent`, que o fio de execução não possa acordar, não apenas por ter recebido um sinal, mas também porque podem ocorrer *spurious wakeups*. Este fenómeno deve-se às implementações das operações bloqueantes dos fios de execução, a nível das chamadas ao sistema, de boa parte dos Sistemas de Operação. Podem existir casos em que fio de execução acorda sem que outro fio o tenha acordado. Além deste motivo, tanto no Java como no pacote utilizado, o método notify/signal não passa o *lock* do processo que o invocou para o processo que foi acordado. Apenas acorda o fio de execução, tendo este que voltar a adquirir o *lock*, ao contrário da especificação de Hoare [18]. Assim, o fio de execução que invocou o método signal pode ainda executar código dentro do monitor, podendo alterar o estado do monitor de tal forma que o fio de execução que foi acordado, quando voltar a adquirir o *lock*, encontre o monitor num estado incoerente. Além desta segunda razão, há uma terceira relacionada com o facto de que o fio acordado pode não ser o primeiro a entrar no monitor depois do fio de execução que invocou o método signal sair. É possível que um terceiro fio de execução consiga obter o *lock* antes do fio de execução acordado.

Declarações Como a linguagem Java não suporta declarações aninhadas, recorremos ao aninhamento de classes/objectos para obter o mesmo efeito. Damos como exemplo a tradução das declarações de variáveis (as declarações de constantes seguem lógica semelhante). Utilizamos a seguinte interface para representar uma expressão de declaração de variáveis:

```
public interface Decl { public Object get(); }
```

Sempre que uma declaração de variáveis (**var** $\overline{\tau\ x}$ **in** e) é traduzida, é criado um objecto que implementa a interface Decl, e é definido o método get a partir da tradução da expressão e. No final, invoca-se o próprio método get que avalia a expressão e num contexto em que existem as variáveis declaradas:

$$
\begin{aligned}
[\![\textbf{var } \tau_1 x_1, ..., \tau_n x_n \textbf{ in } e]\!] \triangleq\ & \textbf{new Decl() \{} \\
& \quad \textbf{private } \tau_1\ x_1; \\
& \quad ... \\
& \quad \textbf{private } \tau_n\ x_n; \\
& \quad \textbf{public Object get() \{ } [\![E]\!] \textbf{ \}} \\
& \textbf{\}.get()}
\end{aligned}
$$

Note-se que esta solução não é boa do ponto de vista da eficiência, visto que envolve a criação de um objecto auxiliar. Por exemplo, se dentro de um ciclo forem declaradas variáveis, então será instanciado um novo objecto repetitivamente, o que é uma operação com algum peso computacional. Este também é um dos motivos pelo qual a definição de uma máquina virtual dedicada, que veremos no próximo capítulo, é mais vantajosa.

5.3 Semântica Operacional

A semântica é definida numa relação de passo pequeno, sobre um conjunto de fios de execução (Σ) que avaliam, cada um, uma expressão (e), com a forma $t\langle e^{\mathtt{l}}_{\alpha}\rangle$, no contexto do objecto corrente ($\mathtt{l}$) e protegido pelo *lock* (α). Por questões de simplicidade, omitimos o objecto corrente e o *lock* sempre que sejam irrelevantes. A avaliação da expressão é efectuada numa dada memória partilhada ($\mathcal{M}$), que representa um mapa entre localizações e valores primitivos ou objectos. Como é de esperar, um objecto tem associado uma classe (c) e um mapa de campos/valores (F). Assim, definimos a relação de redução com a seguinte forma:

$$(\Sigma; \mathcal{M}) \longrightarrow (\Sigma'; \mathcal{M}')$$

onde a avaliação de uma expressão pode dar origem a novos fios de execução (Σ') e a alterações na memória ($\mathcal{M}'$). Os valores são os seguintes:

$$
\begin{array}{rcll}
v & ::= & & \text{(Valores)} \\
 & & n & \text{(Número inteiro)} \\
 & | & t & \text{(Identificador de fio de execução)} \\
 & | & \mathtt{l} & \text{(Referência para Objecto)} \\
 & | & \alpha & \text{(Referência para Lock)}
\end{array}
$$

Por convenção, definimos que a execução do programa começa com um único fio de execução que executa o método com identificador main.

$$
\begin{array}{rcll}
C & ::= & !C & \text{(Not)} \\
 & | & C \ + \ e & \text{(Add Left)} \\
 & | & n \ + \ C & \text{(Add Right)} \\
 & | & C \ ; \ e & \text{(Seq Left)} \\
 & | & v \ ; \ C & \text{(Seq Right)} \\
 & | & x := C & \text{(Assign)} \\
 & | & \textbf{if } C \textbf{ then } e \textbf{ else } e' & \text{(If)} \\
 & | & \textbf{val } \tau \ x \ = \ C \textbf{ in } e & \text{(Val)} \\
 & | & C.m(e') & \text{(Call-1)} \\
 & | & \mathtt{l}.m(C') & \text{(Call-2)} \\
 & | & \textbf{wait } C & \text{(Wait)} \\
 & | & \textbf{shared}(C)\{e\} & \text{(Shared)} \\
 & | & \textbf{sync}(C)\{e\} & \text{(Sync)} \\
 & | & C_\alpha & \text{(Protected)}
\end{array}
$$

Figura 5.6: Figura do contexto de avaliação de expressões da linguagem de alto nível

Para auxiliar a especificar a semântica recorremos ao contexto presente na figura 5.6. A utilização de um contexto, numa relação de redução de passo pequeno, auxilia a especificar a forma como as sub-expressões, de uma determinada expressão, são avaliadas. Assim, definimos a seguinte regra de avaliação de expressões num dado contexto:

$$\frac{(\Sigma, t\langle e\rangle; \mathcal{M}) \longrightarrow (\Sigma', t\langle e'\rangle; \mathcal{M}')}{(\Sigma, t\langle C[e]\rangle; \mathcal{M}) \longrightarrow (\Sigma', t\langle C[e']\rangle; \mathcal{M}')} \qquad \text{(R-CONTEXT)}$$

É com base na definição de contexto apresentada que definimos, de seguida, as regras para as expressões básicas (soma, negação, decisões, ciclos, etc), expressões sobre objectos (criação e chamada) e finalmente aquelas que têm maior relevância para este trabalho (criação e controlo de fios de execução).

Expressões Básicas

A avaliação de um identificador (x) denota o valor (v) que lhe está associada na memória ($\mathcal{M}$). No caso particular de o identificador x representar uma variável de instância, utilizamos a notação $t\langle x^{\mathtt{l}}\rangle$ onde $\mathtt{l}$ representa a referência do objecto corrente (*this*) do método onde a expressão está a ser avaliada.

$$\frac{\mathcal{M}(x) = v}{(\Sigma, t\langle x\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle v\rangle; \mathcal{M})} \qquad \text{(R-ID} - 1\text{)}$$

$$\frac{\mathcal{M}(\mathtt{l}) = \textbf{object}(\_, F) \qquad F(x) = v}{(\Sigma, t\langle x^{\mathtt{l}}\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle v\rangle; \mathcal{M})} \qquad \text{(R-ID} - 2\text{)}$$

Na negação de expressões, depois de avaliada a sub-expressão, dando origem a um inteiro (n), a negação denota o valor do inteiro negado.

$$\frac{n' = \textbf{not}\ n}{(\Sigma, t\langle !n\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle n'\rangle; \mathcal{M})} \qquad (\text{R-NOT})$$

Na soma de expressões, é avaliada a expressão da esquerda e posteriormente a da direita, dando origem a dois inteiros (n,n'), sendo o resultado igual à adição entre esses mesmos dois inteiros.

$$\frac{n'' = n + n'}{(\Sigma, t\langle n + n'\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle n''\rangle; \mathcal{M})} \qquad (\text{R-ADD})$$

Na sequência de expressões, depois de avaliada a expressão da esquerda, e de seguida a da direita, a sequência denota o valor originado pela avaliação da expressão da direita.

$$(\Sigma, t\langle v; v'\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle v'\rangle; \mathcal{M}) \qquad (\text{R-SEQ})$$

Na afectação, é associado na memória à variável local (x) o valor (v) correspondente à sub-expressão avaliada. No caso de o identificador representar uma variável de instância, é actualizado o respectivo valor no objecto corrente (`1`).

$$(\Sigma, t\langle x := v\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle v\rangle; \mathcal{M}[x \mapsto v]) \qquad (\text{R-ASSIGN} - 1)$$

$$(\Sigma, t\langle (x := v)^{\mathtt{l}}\rangle; \mathcal{M}[\mathtt{l} \mapsto \textbf{object}(\_, F)]) \longrightarrow (\Sigma, t\langle v\rangle; \mathcal{M}[\mathtt{l} \mapsto \textbf{object}(\_, F[x \mapsto v])] \qquad (\text{R-ASSIGN} - 2)$$

Como é de esperar, nas construções que envolvem decisões, a sub-expressão a avaliar depende do valor originado pela avaliação da condição.

$$(\Sigma, t\langle \textbf{if}\ 0\ \textbf{then}\ e\ \textbf{else}\ e'\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle e'\rangle; \mathcal{M}) \qquad (\text{R-IF} - 1)$$

$$(\Sigma, t\langle \textbf{if}\ 1\ \textbf{then}\ e\ \textbf{else}\ e'\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle e\rangle; \mathcal{M}) \qquad (\text{R-IF} - 2)$$

Definimos a semântica dos ciclos por equivalência à semântica das expressões de decisão (R-IF).

$$(\Sigma, t\langle \textbf{while}(e)\{e'\}\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle \textbf{if}\ e\ \textbf{then}\ e'; \textbf{while}(e)\{e'\}\ \textbf{else}\ 0\rangle; \mathcal{M}) \qquad (\text{R-WHILE})$$

A declaração de variáveis denota o valor (v) originado pela avaliação do corpo. Utilizamos a notação $e[x'/x]$ para especificar a substituição de todas as ocorrências de x por x'. A tradução da declaração de múltiplas variáveis (**var** $\overline{\tau\ x}$ **in** e) pode-se efectuar por *syntatic sugar* (**var** $\tau_1\ x_1$ **in var** $\tau_2\ x_2$ **in**...)

$$\frac{x' \text{ is fresh}}{(\Sigma, t\langle \textbf{var}\ \tau\ x\ \textbf{in}\ e\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle e[x'/x]\rangle; \mathcal{M})} \qquad \text{(R-VAR)}$$

A declaração de constantes denota o valor originado pela avaliação do corpo, com todas as ocorrências do identificador x substituídas pelo valor v. Tal como nas variáveis, a tradução da declaração de múltiplas constantes pode-se efectuar por *syntatic sugar*.

$$(\Sigma, t\langle \textbf{val}\ \tau\ x\ =\ v\ \textbf{in}\ e\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle e[v/x]\rangle; \mathcal{M}) \qquad \text{(R-VAL)}$$

Expressões sobre Objectos

A criação de um novo objecto denota uma referência para o próprio, alocado na memória, ainda sem as variáveis de instância inicializadas.

$$(\Sigma, t\langle \textbf{new}\ c\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle \mathtt{l}\rangle; \mathcal{M}[\mathtt{l} \mapsto \textbf{object}(c, \varepsilon)]) \qquad \text{(R-NEW)}$$

Depois de avaliada a sub-expressão que denota o objecto sobre o qual é invocado o método (m), seguida da avaliação do argumento, é avaliado o corpo (e), onde todas as ocorrências do argumento (x) são substituídas pelo valor (v). Por questões de simplicidade, utilizamos a notação $\mathcal{P}(c, m)$ para representar o corpo associado ao método m da classe c. Tal como nas declarações de variáveis e constantes, a avaliação de métodos com múltiplos argumentos pode-se efectuar com *syntatic sugar*.

$$\frac{\mathcal{M}(\mathtt{l}) = \textbf{object}(c, \_) \qquad \mathcal{P}(c, m) = \tau'\ m(\tau\ x)\{e\}}{(\Sigma, t\langle \mathtt{l}.m(v)^{\mathtt{l}'}\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle e[v/x]^{\mathtt{l}}\rangle; \mathcal{M})} \qquad \text{(R-CALL)}$$

Expressões de Suporte à Concorrência

A criação de um novo fio de execução (t') que avalia a sub-expressão (e), denota o identificador t'.

$$(\Sigma, t\langle \textbf{fork}\ e\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle t'\rangle, t'\langle e\rangle; \mathcal{M}) \qquad \text{(R-FORK)}$$

A primitiva bloqueante **wait**, sobre um dado fio de execução (t') resultante da avaliação da respectiva sub-expressão, denota o valor calculado pelo próprio.

$$(\Sigma, t\langle \textbf{wait}\ t'\rangle, t'\langle v\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle v\rangle; \mathcal{M}) \qquad \text{(R-WAIT)}$$

A criação de um novo *lock* denota uma referência para o próprio, alocado na memória, com valor inicial igual a 0.

$$(\Sigma, t\langle \textbf{newlock}\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle \alpha\rangle; \mathcal{M}[\alpha \mapsto 0]) \qquad (\text{R-NEWLOCK})$$

Tanto no acesso partilhado, como no acesso exclusivo, é avaliado em primeiro lugar a sub-expressão correspondente a um *lock*, é actualizado o respectivo valor e de seguida é avaliado o corpo. Quando a avaliação estiver concluída é actualizado o valor do *lock* (α). Em ambos os casos, a computação evolui apenas se as respectivas condições forem verdadeiras.

$$\frac{n \geq 0}{(\Sigma, t\langle \textbf{shared}(\alpha)\{e\}\rangle; \mathcal{M}[\alpha \mapsto n]) \longrightarrow (\Sigma, t\langle e_\alpha\rangle; \mathcal{M}[\alpha \mapsto n+1])} \qquad (\text{R-SHARED} - 1)$$

$$\frac{n > 0}{(\Sigma, t\langle v_\alpha\rangle; \mathcal{M}[\alpha \mapsto n]) \longrightarrow (\Sigma, t\langle v\rangle; \mathcal{M}[\alpha \mapsto n-1])} \qquad (\text{T-SHARED} - 2)$$

$$(\Sigma, t\langle \textbf{sync}(\alpha)\{e\}\rangle; \mathcal{M}[\alpha \mapsto 0]) \longrightarrow (\Sigma, t\langle e_\alpha\rangle; \mathcal{M}[\alpha \mapsto -1]) \qquad (\text{R-SYNC} - 1)$$

$$(\Sigma, t\langle v_\alpha\rangle; \mathcal{M}[\alpha \mapsto -1]) \longrightarrow (\Sigma, t\langle v\rangle; \mathcal{M}[\alpha \mapsto 0]) \qquad (\text{R-SYNC} - 2)$$

5.4 Sistema de Tipos

O sistema de tipos proposto permite-nos detectar os erros usuais, tais como operações aritméticas, de comparação, etc, sobre tipos de valores errados (não inteiros), afectação de valores a constantes, chamadas de métodos com argumentos errados, assim como operações de bloqueio (wait) sobre valores que não representam fios de execução. De seguida, definimos a tipificação de um programa através da verificação de cada uma das suas classes, onde são verificadas todas as expressões contidas nos respectivos métodos. Posteriormente definimos as regras de tipo para cada construção (expressão) presente na linguagem.

Programa

Os tipos dos valores podem ser inteiros, variáveis de um determinado tipo (τ), *locks*, fios de execução que calculam um valor de um determinado tipo (τ), ou podem ser objectos de uma determinada classe (c).

$$\tau ::= \textbf{int} \mid \textbf{var}\langle\tau\rangle \mid \textbf{lock} \mid \textbf{th}\langle\tau\rangle \mid c$$

Definimos o ambiente global (Ψ) para representar o conjunto de classes e respectivas interfaces, assim como o ambiente local (Φ) para representar as variáveis locais (e respectivos tipos) de um determinado método.

$$\Psi ::= \overline{c \ : \ (\overline{f{:}\tau} \ ; \ \overline{m{:}\tau(\overline{x{:}\tau})})} \qquad \text{(Ambiente Global)}$$

$$\Phi ::= \overline{x{:}\tau} \qquad \text{(Ambiente Local)}$$

A tipificação de um programa envolve a tipificação de cada uma das suas classes, no ambiente global (Ψ).

$$\frac{P = \overline{\textbf{class}\ c\ \{\ F\ M\ \}} \qquad \forall_i \qquad \overline{c \ : \ (\overline{f{:}\tau} \ ; \ \overline{m{:}\tau(\overline{x{:}\tau})})} \vdash \textbf{class}\ c_i\ \{\ F_i\ M_i\ \}}{\vdash P} \quad \text{(T-PROGRAM)}$$

A tipificação de uma dada classe envolve a tipificação de cada campo e de cada um dos métodos contidos na própria (classe), num ambiente global (Ψ) e no ambiente local (Φ) com o tipo do objecto corrente (*this*) igual ao tipo da classe (c).

$$\frac{\forall_{i,j} \qquad \Psi \vdash \sigma_j \qquad M_i = \tau_i\ m_i(\overline{\tau_i\ x_i})\{\ e_i\ \} \qquad \Psi, this{:}c \vdash \tau_i\ m_i(\overline{\tau_i\ x_i})\{\ e_i\ \} : \tau_i(\overline{\tau_i})}{\Psi \vdash \textbf{class}\ c\ \{\ \overline{\sigma\ f}\ M\ \}} \text{(T-CLASS)}$$

Ao verificarmos um determinado método é necessário averiguar se o corpo (expressão) está bem tipificado, e se o tipo do próprio coincide com o tipo de retorno da assinatura do método.

$$\frac{\Psi, \Phi \vdash_m e : \tau}{\Psi, \Phi \vdash \tau\ m(\overline{\tau\ x})\{\ e\ \} : \tau(\overline{\tau})} \qquad \text{(T-METHOD)}$$

Expressões Básicas

A tipificação de um identificador (x) varia consoante tenha sido declarado como variável local ou constante, ou caso seja um argumento ou uma variável de instância. Caso seja uma variável local ($\textbf{var}\langle\tau\rangle$), o identificador ($x$) denota o tipo encapsulado (τ). Se for uma constante, o identificador (x) denota o tipo (τ) que lhe está associado. O mesmo se segue no caso de ser um argumento ou um campo de uma classe.

$$\frac{\Phi(x) = \textbf{var}\langle\tau\rangle}{\Psi, \Phi \vdash x : \tau} \qquad \frac{\Phi(x) = \tau \qquad \tau \neq \textbf{var}\langle\_\rangle}{\Psi, \Phi \vdash x : \tau} \qquad (\text{T-ID} - 1, \text{T-ID} - 2)$$

$$\frac{\Phi(this) = c \qquad \Psi(c) = (\_\ ;\ \overline{M}, m{:}\_(..., x{:}\tau, ...), \overline{M'})}{\Psi, \Phi \vdash_m x : \tau} \qquad (\text{T-ID} - 3)$$

$$\frac{\Phi(this) = c \qquad \Psi(c) = (\overline{F}, x{:}\tau, \overline{F'}\ ;\ \_)}{\Psi, \Phi \vdash x : \tau} \qquad (\text{T-ID} - 4)$$

Os literais inteiros denotam o tipo **int**.

$$\Psi, \Phi \vdash n : \textbf{int} \qquad (\text{T-INT})$$

A negação é uma operação apenas permitida sobre uma expressão que denota um valor do tipo inteiro, e o valor resultante da própria negação também é do tipo inteiro.

$$\frac{\Psi, \Phi \vdash e : \textbf{int}}{\Psi, \Phi \vdash !e : \textbf{int}} \qquad (\text{T-NOT})$$

A soma, assim como outras operações aritméticas, de comparação, etc, requer duas expressões que denotam valores do tipo inteiro. A própria expressão, neste caso a soma, também denota um valor do tipo inteiro.

$$\frac{\Psi, \Phi \vdash e : \textbf{int} \quad \Psi, \Phi \vdash e' : \textbf{int}}{\Psi, \Phi \vdash e{+}e' : \textbf{int}} \qquad (\text{T-ADD})$$

Na expressão composta pela sequência de duas sub-expressões, estas duas podem denotar valores de diferentes tipos, sendo que o tipo da expressão (principal) é igual ao tipo da segunda sub-expressão.

$$\frac{\Psi, \Phi \vdash e : \tau \quad \Psi, \Phi \vdash e' : \tau'}{\Psi, \Phi \vdash e;e' : \tau'} \qquad (\text{T-SEQ})$$

Na afectação de uma valor a um identificador que representa uma variável do tipo τ, o próprio valor também tem que ser desse mesmo tipo τ. Na afectação de variáveis de instância, o tipo associado à própria também tem que ser do mesmo tipo do valor (da expressão) a afectar. Em ambos os casos, o tipo do valor denotado pela expressão principal ($x := e$) é do tipo do valor da sub-expressão e.

$$\frac{\Phi(x) = \textbf{var}\langle\tau\rangle \quad \Psi, \Phi \vdash e : \tau}{\Psi, \Phi \vdash x{:=}e : \tau} \qquad (\text{T-ASSIGN} - 1)$$

$$\frac{\Phi(this) = c \qquad \Psi(c) = (\overline{F}, x{:}\tau, \overline{F'}\ ;\ \_) \qquad \Psi, \Phi \vdash e : \tau}{\Psi, \Phi \vdash x{:=}e : \tau} \qquad (\text{T-ASSIGN} - 2)$$

Nos ciclos o tipo do valor denotado pela sub-expressão e tem que ser obrigatoriamente do tipo inteiro, enquanto que o da sub-expressão e' pode ser de qualquer tipo. O valor denotado pelo ciclo também é do tipo inteiro.

$$\frac{\Psi, \Phi \vdash e : \textbf{int} \quad \Psi, \Phi \vdash e' : \tau}{\Psi, \Phi \vdash \textbf{while}(e)\{e'\} : \textbf{int}} \qquad (\text{T-WHILE})$$

Restringimos os programas que utilizam expressões de decisão obrigando a que as duas sub-expressões (e', e'') sejam do mesmo tipo (τ). A condição (e), como seria de esperar, representa um valor do tipo inteiro, e a expressão principal denota um tipo igual ao das duas sub-expressões (e', e'').

$$\frac{\Psi, \Phi \vdash e : \textbf{int} \quad \Psi, \Phi \vdash e' : \tau \quad \Psi, \Phi \vdash e'' : \tau}{\Psi, \Phi \vdash \textbf{if}\ e\ \textbf{then}\ e'\ \textbf{else}\ e'' : \tau} \qquad (\text{T-IF})$$

Na declaração de variáveis, num dado ambiente Φ, tipificamos o corpo (e) num ambiente constituído por Φ mais as variáveis declaradas (e respectivos tipos). O tipo da expressão principal é o mesmo do tipo do corpo (e).

$$\frac{\overline{x'} \text{ are fresh} \qquad \Phi' = \Phi, \overline{x'{:}\textbf{var}\langle\tau\rangle} \qquad \Psi, \Phi' \vdash e[\overline{x'/x}] : \tau}{\Psi, \Phi \vdash \textbf{var}\ \overline{\tau\ x}\ \textbf{in}\ e : \tau} \qquad (\text{T-VAR})$$

Na declaração de constantes, num dado ambiente Φ, é averiguado se os tipos associados às mesmas são iguais aos tipos dos valores denotados pelas respectivas expressões. O corpo (e) é tipificado num ambiente Φ mais as constantes declaradas (e respectivos tipos). O tipo da expressão principal é o mesmo do tipo do corpo.

$$\frac{\overline{x'} \text{ are fresh} \qquad \forall_i \quad \Psi, \Phi \vdash e_i : \tau_i \qquad \Phi' = \Phi, \overline{x'{:}\tau} \qquad \Psi, \Phi' \vdash e[\overline{x'/x}] : \tau}{\Psi, \Phi \vdash \textbf{val}\ \overline{\tau\ x\ =\ e}\ \textbf{in}\ e : \tau} \qquad (\text{T-VAL})$$

Expressões sobre Objectos

A construção de um novo objecto de uma dada classe (c) denota um valor desse mesmo tipo (c).

$$\Psi \vdash \textbf{new}\ c : c \qquad (\text{T-NEW})$$

A chamada de um método (m) é obrigatoriamente efectuada sobre uma sub-expressão (e)

que denota um objecto de uma determinada classe (c) que contém esse mesmo método (m). Os tipos das sub-expressões passadas como argumento são iguais aos esperados na assinatura do método, e a expressão principal ($e.m(\overline{e})$) denota um valor do mesmo tipo daquele especificado como retorno na assinatura do método.

$$\frac{\Psi, \Phi \vdash e : c \qquad \forall_i \quad \Psi, \Phi \vdash e_i : \tau_i \qquad \Psi(c) = (\_\,;\, \overline{M}, m{:}\tau(\overline{x{:}\tau}), \overline{M'})}{\Psi, \Phi \vdash e.m(\overline{e}) : \tau} \qquad \text{(T-CALL)}$$

Expressões de Suporte à Concorrência

A expressão de criação de novos fios de execução denota um valor do tipo *thread* ($\mathbf{th}\langle\tau\rangle$) com um tipo paramétrico (τ) igual ao da sub-expressão (e).

$$\frac{\Psi, \Phi \vdash e : \tau}{\Psi, \Phi \vdash \mathbf{fork}\ e : \mathbf{th}\langle\tau\rangle} \qquad \text{(T-FORK)}$$

A operação bloqueante (**wait**) é utilizada sobre uma sub-expressão que denota um valor do tipo *thread* ($\mathbf{th}\langle\tau\rangle$). O tipo do valor desta operação corresponde ao valor paramétrico (τ).

$$\frac{\Psi, \Phi \vdash e : \mathbf{th}\langle\tau\rangle}{\Psi, \Phi \vdash \mathbf{wait}\ e : \tau} \qquad \text{(T-WAIT)}$$

A criação de um novo *lock* denota um valor desse mesmo tipo (**lock**).

$$\Psi, \Phi \vdash \mathbf{newlock} : \mathbf{lock} \qquad \text{(T-NEWLOCK)}$$

Tanto no acesso partilhado, como no acesso exclusivo, a sub-expressão e denota um valor do tipo **lock**. A expressão principal denota um valor do tipo (τ) igual ao que está associado ao corpo (e').

$$\frac{\Psi, \Phi \vdash e : \mathbf{lock} \quad \Psi, \Phi \vdash e' : \tau}{\Psi, \Phi \vdash \mathbf{shared}(e)\{e'\} : \tau} \qquad \text{(T-SHARED)}$$

$$\frac{\Psi, \Phi \vdash e : \mathbf{lock} \quad \Psi, \Phi \vdash e' : \tau}{\Psi, \Phi \vdash \mathbf{sync}(e)\{e'\} : \tau} \qquad \text{(T-SYNC)}$$

Resultados

Apesar de não apresentarmos provas formais (nomeadamente o teorema da preservação de tipos e do progresso) foram efectuados vários testes, que mesmo não dando garantias permitiram testar o sistema de tipos, invalidando programas com operações sobre tipos de operandos errados (somar um inteiro com uma referência de um objecto), afectação

de valores a constantes, chamadas de métodos inexestentes ou sobre valores que não são objectos (inteiros, por exemplo) e operações sobre concorrência (**wait**, **shared**/**sync**) sobre valores que não são do tipo *thread* e *lock*, respectivamente.

6

Linguagem Intermédia e Máquina Virtual

O suporte para concorrência implementado através bibliotecas (Java), no capítulo anterior, torna difícil qualquer tentativa de implementar procedimentos de verificação sobre programas com múltiplos fios de execução. Tanto na linguagem Java, como na linguagem C#, o suporte para concorrência, no modelo de memória partilhada, resume-se a bibliotecas (*e.g.*, classe *Thread*). Mesmo que ambas tivessem suporte nativo, e eventualmente um sistema de tipos que conseguisse detectar interferências entre fios de execução, ao ser compilado o código para as linguagens intermédias JVML e CIL, respectivamente, perdia-se informação sobre os fios de execução e sobre eventuais garantias dadas pelo sistema de tipos, pois tanto a JVML como a CIL também não têm suporte nativo para concorrência. Assim, decidimos desenvolver uma linguagem intermédia tipificada com suporte nativo para concorrência, em memória partilhada, com um modelo de objectos de raiz, antevendo um sistema de tipos comportamental sofisticado que garanta a ausência de interferências entre os múltiplos fios de execução.

Algumas linguagens intermédias, nomeadamente a linguagem intermédia alvo da linguagem TyCO [23] e a linguagem MIL [41], tratam da concorrência ao nível do código máquina. No primeiro caso estamos perante uma linguagem intermédia concorrente mais focalizada para suporte a linguagens derivadas do Cálculo-π. O segundo caso diz respeito a uma linguagem intermédia concorrente, segundo o modelo de memória partilhada que, através de anotações no código e do respectivo sistema de tipos, consegue garantir a ausência de situações de bloqueio ou *data races* entre múltiplos fios de execução. No entanto, é uma linguagem destino para máquinas muito primitivas, baseada

em registos e sem modelo de objectos implementado de raiz, o que torna difícil a geração de código de alto nível para código de máquina, nomeadamente tendo em conta a preservação de tipos entre as duas linguagens.

Por outro lado, as máquinas JVM e CLR apresentam um modelo de objectos nativo, e com pilhas distintas de chamada e de avaliação de expressões. Também apresentam um sistema de tipos que realiza análise de fluxo de dados de forma a evitar determinados erros, que eventualmente iriam ocorrer em tempo de execução se não fossem devidamente detectados, ao contrário da linguagem MIL que verifica o programa linearmente com auxílio de anotações no código. Decidimos seguir esta última abordagem, originalmente proposta pela *Typed Assembly Language* [25], mas agora associando às etiquetas os tipos dos elementos da pilha de avaliação, em vez dos tipos dos registos, e sem necessidade de incorporar tipos polimórficos, dado que a pilha de avaliação está associada apenas a uma instância de uma chamada.

Nas próximas secções vamos introduzir, em primeiro lugar, os conceitos básicos da nossa linguagem intermédia, com recurso a vários exemplos. Posteriormente definimos a sintaxe concreta da linguagem e a sua semântica operacional, através de uma relação de redução de passo pequeno. De seguida definimos o seu sistema de tipos base, que nos permite garantir, em apenas um passo, a ausência dos erros mais comuns de execução, nomeadamente tipos errados nos argumentos de cada operação, saltos para locais ilegais, utilização da pilha para além dos limites, entre outros. Por último, descrevemos a arquitectura da implementação da máquina virtual e alguns detalhes que consideramos ser relevantes. No próximo capítulo, apresentamos a tradução da linguagem de alto nível, do capítulo anterior, para a linguagem intermédia.

6.1 Linguagem Intermédia Tipificada

A linguagem intermédia aqui proposta [24] é uma linguagem de máquina de pilha semelhante às linguagens CIL [15] e JVML [22], sendo portanto baseada num modelo de objectos, com duas pilhas distintas de avaliação e de chamada. O código de cada programa consiste em um conjunto de declarações de classes, com declarações de variáveis de instância e de métodos, com assinatura e corpo. Apenas por questões de simplicidade, consideramos que todos os métodos são de instância. Adicionalmente, a sintaxe da linguagem define o corpo dos métodos como um conjunto de blocos de instruções. Os blocos começam com uma etiqueta associada a uma expressão de tipo, que caracteriza o conteúdo da pilha no início do bloco, seguido de várias instruções que operam sobre a pilha de avaliação e de chamada, e terminam com uma instrução de retorno (**ret**) ou com uma instrução de salto incondicional (**br**).

No extracto de código da Figura 6.1 encontra-se, no lado esquerdo, a definição do método `abs` que calcula o módulo de um inteiro, usando uma linguagem de alto nível (Java), e no lado direito usando a nossa linguagem intermédia. O código da linguagem intermédia é composto por dois blocos com as etiquetas `l0` e `l1`. A linguagem inclui as

```
class Abs {
    int abs(int x) {
        if(n > 0)
            return x;
        else
            return −1∗x;
    }
}
```

```
class Abs {
    method int abs(int x) {
    l0: empty
        ldarg x
        dup
        ldc 0
        cgt
        brfalse l1
        ret
    l1: {int}
        ldc −1
        mul
        ret
    }
}
```

Figura 6.1: Cálculo do módulo

```
class Factorial {
    int factorial(int x) {
        if(x == 0)
            return 1;
        else
            return x∗factorial(x−1);
    }
}
```

```
class Factorial {
    method int factorial(int x) {
    l0: empty
      ldarg x
      ldc 0
      ceq
      brfalse l1
      ldc x
      ret
    l1: empty
      ldarg x
      ldarg this
      ldarg x
      ldc 1
      sub
      call int factorial(int)
      mul
      ret
    }
}
```

Figura 6.2: Cálculo do factorial

instruções habituais de carregamento de valores na pilha de avaliação (**ldc**, **dup**), de manipulação de argumentos e variáveis locais (**ldarg**, **ldloc**, **stloc**), para além das instruções de salto (**br**, **brfalse**), operações aritméticas (**mul**), entre outras. As anotações de tipo que estão associadas às etiquetas (**empty** e {**int**}) descrevem o conteúdo esperado da pilha de avaliação à entrada de cada bloco. O tipo esperado para a etiqueta l0 é **empty** indicando que o bloco está bem tipificado num contexto com a pilha vazia. Cada instrução do bloco é tipificada em relação ao tipo da pilha deixado pela instrução anterior, produzindo um tipo de pilha modificado, que serve de entrada para a instrução seguinte. Neste caso, a

```
class Point {
    int x, y;
    int horizontal(Point p) {
        return this.x == p.x;
    }
    int main() {
        Point p = new Point;
        p.x = 5;
        Point p2 = new Point;
        p2.x = 10;
        return p.horizontal(p2);
    }
}
```

```
class Point {
    field int x, y;
    method int horizontal(Point p) {
            locals(int)
        l0: empty
            ldarg this
            ldarg p
            br l1
        l1: {Point,Point}
            ldfld x
            stloc 0
            ldfld x
            ldloc 0
            ceq
            ret
    }
    method int main() {
        l0: empty
            newobj Point
            dup
            ldc 5
            stfld x
            newobj Point
            dup
            ldc 10
            stfld x
            call int horizontal(Point)
            ret
    }
}
```

Figura 6.3: Exemplo com objectos

instrução **brfalse** l1 tem por tipo de entrada {**int**, **int**}, resultante do carregamento de um argumento da função e da sua comparação com 0. Este tipo permite verificar que depois de retirar um valor do topo da pilha como argumento da instrução de salto, o tipo {**int**} coincide com o tipo requerido pela etiqueta l1. A instrução **ret** está bem tipificada se o tipo à entrada for o tipo esperado na assinatura do método para o valor de retorno.

O exemplo da Figura 6.2 ilustra uma função recursiva que calcula o factorial de um inteiro não negativo. Tem dois blocos l0 e l1 que representam o caso base e o caso geral, respectivamente. No caso base é efectuada uma comparação para averiguar se o valor, recebido como argumento, é igual a 0. Em caso positivo, é retornado o (único) valor deixado na pilha de avaliação (o inteiro 0). Em caso negativo, dá-se um salto para o bloco que representa o caso geral. Neste caso, para um dado inteiro x, é calculado o factorial de $x-1$, que é multiplicado ao inteiro x e retornado o resultado final. Podemos verificar que a chamada recursiva (linha 16) é tipificada numa pilha com o tipo {**int**, `Factorial`, **int**}, que está de acordo com o tipo do argumento do método abs da classe `Factorial`, visto que tem no topo um valor do tipo inteiro. Após a invocação, a pilha fica associada ao

```
1  class Abs {
       int abs(int n) {...}
       int spawn() {
           th⟨int⟩ t1 = fork abs(−1);
5          th⟨int⟩ t2 = fork abs(−2);
6          return wait t1 + wait t2;
       }
   }
```

```
1  class Abs {
2      method int abs(int) {...}
       method int spawn() {
           locals(th⟨int⟩,th⟨int⟩)
5      l0: empty
6          ldarg this
7          dup
           br l1
       l1: {Abs,Abs}
10         ldc −1
           fork int abs(int)
12         stloc 0
           ldc −2
           fork int abs(int)
           stloc 1
           ldloc 0
17         wait
           ldloc 1
           wait
           add
           ret
22     }
   }
```

Figura 6.4: Exemplo com concorrência

tipo {**int**, **int**}, onde o topo representa o tipo do retorno do método (**int**).

Considere-se o exemplo da Figura 6.3, onde o método main cria dois objectos (**newobj**) do tipo `Point`, altera a variável de instância *x* (**stfld**) de ambos para 5 e 10, respectivamente, e de seguida invoca (**call**) o método horizontal. No primeiro bloco é empilhado o objecto corrente (*this*) e o recebido como argumento, saltando-se de seguida para um segundo bloco que espera um tipo de pilha igual a {`Point`,`Point`}, carrega a variável de instância *x* (**ldfld**) de cada um dos dois objectos, compara a mesmas de modo a averiguar se os dois pontos formam uma linha horizontal, e finalmente retorna o resultado (**ret**).

A linguagem intermédia, e a respectiva máquina virtual, têm suporte primitivo para programas concorrentes. A linguagem inclui duas instruções (**fork** e **wait**) para lançar um fio de execução novo com base num método existente e tendo como resultado o respectivo identificador, e também para esperar pela finalização de um fio de execução e recuperar o resultado correspondente. A linguagem inclui ainda outras três instruções de suporte à concorrência e sincronização de fios de execução: **newlock**, **sync** e **shared**. Os *locks* implementados são próximos do conceito de *locks* de "multiplos leitores, um escritor", que permitem o acesso a um recurso em exclusão mútua ou em partilha. A instrução **sync** permite chamar um método em exclusão mútua sobre um dado lock. Por outro lado, instrução **shared** permite chamar um método em simultâneo com outras chamadas do mesmo tipo, sobre o mesmo lock, mas em exclusão mútua em relação a métodos chamados em modo exclusivo (**sync**).

```
class Cell {
    int value;
    lock l;
    int spawn() {
        th⟨int⟩ t1 = fork sync(l){
            value = value − 1;
        };

        th⟨int⟩ t2 = fork sync(l){
            value = value + 1;
        };
        wait t1; wait t2;
        return value;
    }
}
```

```
class Cell {
    field int value;
    field lock l;
    method int spawn() {
    l0: empty
        ldarg this
        fork int f1()
        ldarg this
        fork int f2()
        br l1
    l1: {th⟨int⟩, th⟨int⟩}
        ...
    }
    method int f1() {
    l0: empty
        ldarg this
        ldfld l
        ldarg this
        sync int s1()
        ret
    }
    method int f2() {...}
    method int s1() {...}
    method int s2() {...}
}
```

Figura 6.5: Execução em modo exclusivo

No extracto de código na Figura 6.4 são empilhados dois objectos, do tipo `Abs`, e salta-se para o segundo bloco, que espera esses mesmos dois objectos no topo da pilha. São lançados dois novos fios de execução, em paralelo, que recorrem ao método abs para calcular o módulo dos números deixados no topo da pilha (-1 e -2, respectivamente). De seguida, o fio principal bloqueia a execução até que os restantes dois fios de execução terminem, e somam-se os valores calculados por ambos. De notar que a instrução **fork** nem sempre está associada ao paralelismo real, visto que podemos simular uma chamada convencional (**call** τ $m(\overline{\tau})$) através da criação de um fio de execução, seguida (imediatamente) da operação bloqueante (**fork** τ $m(\overline{\tau})$; **wait**).

O exemplo da Figura 6.5 ilustra a utilização da instrução **sync**, de modo a permitir que dois fios de execução alterem de forma segura o valor de uma célula. O método spawn lança dois novos fios de execução, que executam os métodos f1 e f2, respectivamente. Como se pode observar, o método f1 invoca, em modo exclusivo, o método s1 (que decrementa o valor da célula) sobre um *lock* l previamente empilhado. O mesmo raciocínio seguem os métodos executados pelo segundo fio de execução (f2 e s2). O fio de execução principal, que entretanto saltou para o bloco l1, associado a uma pilha com outros dois fios do tipo inteiro, trata de esperar por ambos, mediante a instrução **wait**, e no final retorna o novo valor da célula, que é igual ao anterior. Caso o acesso não fosse realizado no modo exclusivo, e dependendo do escalonamento dos fios de execução, teríamos

$$
\begin{array}{lcll}
P & ::= & \overline{C} & \text{(Programa)} \\
C & ::= & \textbf{class}\ c\ \{\ \overline{F}\ \overline{M}\ \} & \text{(Classe)} \\
F & ::= & \textbf{field}\ \tau\ f & \text{(Variável de instância)} \\
M & ::= & \textbf{method}\ \tau\ m(\overline{\tau\ x})\ \{\ \textbf{locals}(\overline{\tau\ x})\ \overline{B}\ \} & \text{(Método)} \\
B & ::= & l{:}\Gamma \mapsto A & \text{(Bloco)} \\
A & ::= & \textbf{ret} \mid \textbf{br}\ l \mid \iota \cdot A & \text{(Sequência de instruções)} \\
\iota & ::= & & \text{(Instruções)} \\
 & & op \mid \textbf{pop} \mid \textbf{dup} \mid \textbf{not} \mid \textbf{ldc}\ n \mid \textbf{brfalse}\ l \mid \textbf{ldarg}\ x \mid \textbf{ldloc}\ x \mid \textbf{stloc}\ x & \\
 & \mid & \textbf{call}\ \tau\ m(\overline{\tau}) \mid \textbf{newobj}\ c \mid \textbf{ldfld}\ f \mid \textbf{stfld}\ f \mid \textbf{fork}\ \tau\ m(\overline{\tau}) \mid \textbf{wait} \mid \textbf{newlock} & \\
 & \mid & \textbf{shared}\ \tau\ m(\overline{\tau}) \mid \textbf{sync}\ \tau\ m(\overline{\tau}) &
\end{array}
$$

Figura 6.6: Sintaxe

eventualmente um resultado diferente.

6.2 Sintaxe e Semântica

Descrevemos de seguida a sintaxe da linguagem e a sua semântica operacional. A sintaxe é dada pela gramática apresentada na Figura 6.6 onde se convencionam símbolos para identificadores de classes (c), para as variáveis de instância de objectos (f), para métodos (m), para as etiquetas do código (l), e para variáveis locais e argumentos (x). Como seria de esperar, os identificadores são obrigatoriamente únicos. Também se usa a letra n para representar literais inteiros. O símbolo op representa as operações binárias (*e.g.*, aritméticas, de comparação). Para representar os tipos (dos valores) na sintaxe da linguagem usa-se o símbolo τ, e para os tipos de pilha o símbolo Γ.

Um programa (P) é um conjunto de definições de classes, cada uma composta por um conjunto de definições de variáveis de instância e de métodos. Os métodos (M) são definidos por uma assinatura (nome, tipo de retorno e tipos dos parâmetros), um conjunto de tipos para variáveis locais e um conjunto de blocos de etiquetas/sequência de instruções. A cada etiqueta é associado um tipo de pilha (Γ), e o código de cada bloco é formado por uma ou várias instruções, e acaba obrigatoriamente numa instrução de retorno ou de salto incondicional.

Definimos a semântica operacional da linguagem através de uma relação de redução de passo pequeno, com base numa configuração da forma $\{\Sigma; H\}$, onde Σ representa o conjunto de fios de execução activos e H representa a memória contendo objectos e *locks* (Figura 6.7). A redução define-se em relação a um programa P, omitido nas regras para aumentar a legibilidade. Por convenção, definimos o início de execução de um programa com base num método com identificador *main*. A execução termina quando não existirem fios de execução (Σ) com instruções para processar.

Um fio de execução tem a forma (t, ρ) onde t é o seu identificador e ρ representa a pilha de chamadas, que contém os registos de activação activos nesse fio de execução.

Σ	::=	$\overline{(t, \rho)}$	(Fios de execução)
ρ	::=	$\overline{Fr}$	(Pilha de chamada)
Fr	::=	$\langle \overline{x \mapsto v}, \overline{x \mapsto v}, s, A \rangle_{\alpha}^{m}$	(Registo de activação)
s	::=	$\varepsilon \mid s \cdot v$	(Pilha de avaliação)
v	::=		(Valores)
		n	(Número inteiro)
	$\mid$	t	(Identificador de fio de execução)
	$\mid$	$\mathtt{l}$	(Referência para Objecto)
	$\mid$	α	(Referência para Lock)
obj	::=	$\textbf{object}(c, \overline{f \mapsto v})$	(Objecto)
H	::=	$\overline{\mathtt{l} \mapsto obj}, \ \overline{\alpha \mapsto n}$	(Heap)

Figura 6.7: Estado da máquina

Um registo de activação tem a forma $\langle a, lv, s, A \rangle_{\alpha}^{m}$, onde a e lv representam os argumentos e as variáveis locais da instância do método, respectivamente, s representa a pilha de avaliação, e A representa as instruções que faltam executar no bloco de código corrente. O registo de activação é ainda decorado com o identificador do método correspondente ao registo de activação (m) e, opcionalmente, com o identificador do *lock* adquirido na chamada do método (α). O *lock* α só é usado no caso de chamadas que adquirem locks (**sync** e **shared**). A memória (H) é um mapeamento de referências para objectos, e também de referências para *locks*. Os objectos são constituídos por um identificador de uma classe (c) e por um mapa de variáveis de instância para os seus valores ($\overline{f \mapsto v}$). A cada *lock* é associado um número inteiro que representa o seu estado. O valor 0 (zero) indica que o *lock* não foi adquirido. O valor -1 significa que um fio de execução obteve o *lock* em modo exclusivo. Por outro lado, um valor positivo indica o número de fios de execução que adquiriram o *lock* em modo partilhado. De forma a simplificar as regras relativas à semântica da linguagem, definimos as seguintes abreviaturas:

- $\mathcal{F}(c)$: o conjunto de variáveis de instância da classe c
- $\mathcal{M}(c, m)$: a assinatura do método m da classe c, incluindo os nomes dos argumentos.
- $\mathcal{P}(c, m)$: o código do primeiro bloco do método m, da classe c
- $\mathcal{P}(c, m, l)$: o código associado à etiqueta l do método m, da classe c
- $\mathcal{V}(c, m)$: as variáveis locais associadas ao método m da classe c

Apresentamos de seguida as regras de avaliação para as instruções básicas (**pop**, **brfalse**, etc). Depois apresentamos as instruções relativas a objectos (**newobj**, **call**, etc), seguidas das instruções de suporte à concorrência (**fork**, **wait**, etc), e finalmente ilustramos a semântica operacional com alguns exemplos.

Instruções básicas

Na regra (R-POP), partindo de um estado com pelo menos um fio de execução e um registo de activação, e com uma pilha de avaliação com pelo menos um valor, é descartado o valor presente no seu topo (v).

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, \textbf{pop} \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^m); H\} \qquad \text{(R-POP)}$$

De forma semelhante à regra anterior temos na duplicação de valores (R-DUP), mas neste caso em vez de ser descartado o valor do topo (v), é duplicado, ficando uma copia no topo da pilha.

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, \textbf{dup} \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v \cdot v, A\rangle^m); H\} \qquad \text{(R-DUP)}$$

No carregamento de constantes (R-LDC), é colocado no topo da pilha de avaliação um dado inteiro (n).

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{ldc}\ n \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n, A\rangle^m); H\} \qquad \text{(R-LDC)}$$

Na negação de valores (R-NOT), o inteiro (n) presente no topo de avaliação é desempilhado e é empilhada a sua negação (n'). Se o valor for igual 0 então o resultado da negação é igual a 1, caso contrário é igual a 0.

$$\frac{n' \ = \ \textbf{not}\ n}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n, \textbf{not} \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n', A\rangle^m); H\}} \qquad \text{(R-NOT)}$$

No caso das operações binárias (*e.g.*, R-ADD) são desempilhados 2 valores (n', n''), é calculado o resultado (n), sendo o mesmo colocado no topo da pilha.

$$\frac{n = n' + n''}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n' \cdot n'', \ \textbf{add} \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n, A\rangle^m); H\}} \qquad \text{(R-ADD)}$$

Nas instruções de salto incondicional (R-BR), o bloco de instruções corrente (A) do registo de activação do topo da pilha de chamadas, de um determinado fio de execução, é

trocado por outro (A'), com a etiqueta dada (l) e presente no método corrente (m), pertencente à classe (c) do objecto corrente (`l`).

$$\frac{a(this) = \texttt{l} \quad H(\texttt{l}) = \textbf{object}(c, \_) \quad A' = \mathcal{P}(c, m, l)}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{br}\ l \rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A' \rangle^m); H\}} \quad \text{(R-BR)}$$

Ao contrário da instrução anterior (R-BR), na instrução de salto condicional a troca do bloco corrente está directamente dependente do valor do inteiro, presente no topo da pilha. Caso seja igual a 0 (R-BRFALSE-2) dá-se a troca de blocos. Caso contrário (R-BRFALSE-1), as estruturas de dados mantêm-se inalteradas, com excepção do inteiro, presente no topo da pilha, que é sempre desempilhado.

$$\frac{n \neq 0}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n, \textbf{brfalse}\ l \cdot A \rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A \rangle^m); H\}} \ (\text{R-BRFALSE} - 1)$$

$$\frac{a(this) = \texttt{l} \quad H(\texttt{l}) = \textbf{object}(c, \_) \quad A' = \mathcal{P}(c, m, l)}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot 0, \textbf{brfalse}\ l \cdot A \rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A' \rangle^m); H\}} \ (\text{R-BRFALSE} - 2)$$

Finalmente, as regras (R-LDARG, R-LDLOC, R-STLOC) permitem carregar e guardar valores entre a pilha de avaliação e os argumentos e variáveis locais, respectivamente.

$$\frac{a(x) = v}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{ldarg}\ x \cdot A \rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A \rangle^m); H\}} \quad \text{(R-LDARG)}$$

$$\frac{lv(x) = v}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{ldloc}\ x \cdot A \rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A \rangle^m); H\}} \quad \text{(R-LDLOC)}$$

$$\{\Sigma, (t, \rho \cdot \langle a, lv[x \mapsto v], s \cdot v', \textbf{stloc}\ x \cdot A \rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv[x \mapsto v'], s, A \rangle^m); H\} \quad \text{(R-STLOC)}$$

Instruções relativas a objectos

Na criação de um novo objecto (R-NEWOBJ), de uma dada classe, é empilhada uma referência (`l`) para a sua localização na *heap*.

$$\frac{\texttt{l}\ is\ fresh \quad \mathcal{F}(c) = \overline{fv}}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{newobj}\ c \cdot A \rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \texttt{l}, A \rangle^m); H[\texttt{l} \mapsto \textbf{object}(c, \overline{fv})]\}} \quad \text{(R-NEWOBJ)}$$

As regras (R-LDFLD) e (R-STFLD) especificam como são carregadas ou alteradas as variáveis de instância de um objecto. No primeiro caso, para uma dado nome de uma variável de instância, é necessário ter a referência (`l`) do próprio objecto no topo da pilha. No caso da afectação de variáveis de instância, é necessário também ter, no topo da pilha, o valor a ser guardado (v).

$$\frac{H(\texttt{l}) = \textbf{object}(c, fv) \quad fv(f) = v}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \texttt{l}, \textbf{ldfld}\ f \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^m); H\}} \quad \text{(R-LDFLD)}$$

$$\begin{aligned}\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \texttt{l} \cdot v, \textbf{stfld}\ f \cdot A\rangle^m); H[\texttt{l} \mapsto \textbf{object}(c, fv)]\} \longrightarrow \\ \{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^m); H[\texttt{l} \mapsto \textbf{object}(c, fv[f \mapsto v])]\}\end{aligned} \quad \text{(R-STFLD)}$$

Na chamada de métodos (R-CALL) é criado um novo registo de activação para o método com a assinatura dada ($\tau\ m(\overline{\tau})$) de uma dada classe (c) do objecto (`l`) que estiver no topo da pilha. Os argumentos ($\overline{v}$) são retirados do topo da pilha de avaliação do registo de activação corrente e são colocados no novo registo de activação como argumentos. As restantes estruturas (variáveis locais e o primeiro bloco de instruções) são retirados do código do programa ($A' = \mathcal{P}(c, m)$ e $lv' = \mathcal{V}(c, m)$).

$$\begin{aligned}\frac{A' = \mathcal{P}(c, m) \quad \mathcal{M}(c, m) = \tau\ m(\overline{\tau\ x}) \quad lv' = \mathcal{V}(c, m) \quad H(\texttt{l}) = \textbf{object}(c, \overline{fv})}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \texttt{l} \cdot \overline{v}, \textbf{call}\ \tau\ m(\overline{\tau}) \cdot A\rangle^{m'}); H\} \longrightarrow} \\ \{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^{m'} \cdot \langle (this \mapsto \texttt{l}, \overline{x \mapsto v}), lv', \varepsilon, A'\rangle^m); H\}\end{aligned} \quad \text{(R-CALL)}$$

Quando estamos perante o retorno de um método (R-RET $-$ 1), o valor (v) deixado no topo da pilha de avaliação, do registo de activação corrente, é transferido para o registo de activação anterior (passando a ser este o registo de activação corrente).

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^m \cdot \langle a', lv', v, \textbf{ret}\rangle^{m'}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^m); H\} \quad \text{(R-RET} - 1\text{)}$$

Instruções de suporte à concorrência

A criação de um novo fio de execução (R-FORK) envolve também a criação de um novo registo de activação, relativo ao método da assinatura dada, da classe (c) do objecto (`l`), presente no topo da pilha (juntamente com os eventuais argumentos). Ao contrário da chamada de métodos (R-CALL), não se suspende o código que chama o método, ficando no topo da pilha de avaliação o identificador do novo fio de execução (t'), avançando-se

para a próxima instrução. De salientar que aqui, na semântica operacional da linguagem intermédia, não especificamos a forma como os fios de execução são escalonados, deixando este aspecto para a implementação concreta da máquina virtual.

$$\frac{A' = \mathcal{P}(c,m) \quad \mathcal{M}(c,m) = \tau\ m(\overline{\tau\ x}) \quad lv' = \mathcal{V}(c,m) \quad H(\mathtt{l}) = \textbf{object}(c,\overline{fv}) \quad t' \text{ is fresh}}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \mathtt{l} \cdot \overline{v}, \textbf{fork}\ \tau\ m(\overline{\tau}) \cdot A\rangle^{m'}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot t', A\rangle^{m'}), (t', \langle (this \mapsto \mathtt{l}, \overline{x \mapsto v}), lv', \varepsilon, A'\rangle^{m}); H\}} \text{(R-FORK)}$$

Para obter o valor calculado pelo novo fio de execução (R-WAIT), é retirado do topo da pilha de avaliação o respectivo identificador (t'), bloqueia-se a execução até que o mesmo termine, e finalmente transfere-se para o topo da pilha de avaliação o valor calculado (v). No final, o fio de execução é removido.

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot t', \textbf{wait} \cdot A\rangle^{m}), (t', \langle a', lv', v, \textbf{ret}\rangle^{m'}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^{m}); H\} \quad \text{(R-WAIT)}$$

A regra (R-NEWLOCK) especifica a criação de um novo *lock* na *heap*, assim como a colocação da respectiva referência (α) no topo da pilha.

$$\frac{\alpha \text{ is fresh}}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{newlock} \cdot A\rangle^{m}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \alpha, A\rangle^{m}); H[\alpha \mapsto 0]\}} \text{(R-NEWLOCK)}$$

As chamadas de métodos em modo partilhado ou exclusivo (R-SHARED e R-SYNC) têm comportamento mais próximo da chamada de métodos convencional (R-CALL). No entanto, acrescentam a aquisição de um *lock* (α) cuja referência deve estar presente no topo da pilha de avaliação (assim como os argumentos). Nestes dois casos, as chamadas bloqueiam à espera das respectivas condições para evoluir. No acesso em modo partilhado, só se avança se não existir nenhuma chamada sobre o mesmo *lock* ou, no caso de existir, se também foi efectuada em modo partilhado ($n \geq 0$). No acesso exclusivo, só se avança caso não exista nenhuma chamada, tanto em modo de partilha como em modo exclusivo, sobre o mesmo *lock* ($n = 0$).

$$\frac{A' = \mathcal{P}(c,m) \quad \mathcal{M}(c,m) = \tau\ m(\overline{\tau\ x}) \quad lv' = \mathcal{V}(c,m) \quad H(\mathtt{l}) = \textbf{object}(c,\overline{fv}) \quad n \geq 0}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \alpha \cdot \mathtt{l} \cdot \overline{v}, \textbf{shared}\ \tau\ m(\overline{\tau}) \cdot A\rangle^{m'}); H[\alpha \mapsto n]\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^{m'} \cdot \langle (this \mapsto \mathtt{l}, \overline{x \mapsto v}), lv', \varepsilon, A'\rangle^{m}_{\alpha}); H[\alpha \mapsto n{+}1]\}} \text{(R-SHARED)}$$

$$\frac{A' = \mathcal{P}(c,m) \quad \mathcal{M}(c,m) = \tau\ m(\overline{\tau\ x}) \quad lv' = \mathcal{V}(c,m) \quad H(\mathtt{l}) = \textbf{object}(c, \overline{fv})}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \alpha \cdot \mathtt{l} \cdot \overline{v},\ \textbf{sync}\ \tau\ m(\overline{\tau}) \cdot A\rangle^{m'}); H[\alpha \mapsto 0]\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^{m'} \cdot \langle (this \mapsto \mathtt{l}, \overline{x \mapsto v}), lv', \varepsilon, A'\rangle_{\alpha}^{m}); H[\alpha \mapsto -1]\}} \text{(R-SYNC)}$$

Ao contrário do retorno de métodos invocados da forma convencional (R-RET − 1), se os métodos tiverem sido invocados em modo partilhado ou exclusivo (R-RET − 2, R-RET − 3) é necessário actualizar o *lock*. De notar que aqui, na semântica operacional, não especificamos como são resolvidas as questões de justiça sobre o acesso, partilhado ou exclusivo, entre os múltiplos fios de execução, deixando este aspecto para a implementação concreta da máquina virtual.

$$\frac{n > 0}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^{m} \cdot \langle a', lv', v, \textbf{ret}\rangle_{\alpha}^{m'}); H[\alpha \mapsto n]\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^{m}); H[\alpha \mapsto n-1]\}} \quad (\text{R-RET} - 2)$$

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^{m} \cdot \langle a', lv', v, \textbf{ret}\rangle_{\alpha}^{m'}); H[\alpha \mapsto -1]\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^{m}); H[\alpha \mapsto 0]\} \quad (\text{R-RET} - 3)$$

Exemplo

Ilustramos agora a avaliação de algumas instruções anteriormente apresentadas, partindo de um bloco com a etiqueta $l0$ e um tipo de pilha vazio, retirado do exemplo da Figura 6.1, cuja função abs foi invocada com o argumento igual a 5:

```
l0: empty
    ldarg x
    dup
    ldc 0
    cgt
    brfalse l1
    ret
```

$$\{\Sigma, (t, \rho \cdot \langle a[x \mapsto 5], \varepsilon, \varepsilon, \textbf{ldarg}\ x \cdot A\rangle^{abs}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, \varepsilon, 5, A\rangle^{abs}); H\}$$

$$\{\Sigma, (t, \rho \cdot \langle a, \varepsilon, 5, \textbf{dup} \cdot A\rangle^{abs}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, \varepsilon, 5 \cdot 5, A\rangle^{abs}); H\}$$

$$\{\Sigma, (t, \rho \cdot \langle a, \varepsilon, 5 \cdot 5, \textbf{ldc}\ 0 \cdot A\rangle^{abs}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, \varepsilon, 5 \cdot 5 \cdot 0, A\rangle^{abs}); H\}$$

$$\{\Sigma, (t, \rho \cdot \langle a, \varepsilon, 5 \cdot 5 \cdot 0, \textbf{cgt} \cdot A\rangle^{abs}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, \varepsilon, 5 \cdot 1, A\rangle^{abs}); H\}$$

$$\{\Sigma, (t, \rho \cdot \langle a, \varepsilon, 5 \cdot 1, \textbf{brfalse}\ l1 \cdot A\rangle^{abs}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, \varepsilon, 5, A\rangle^{abs}); H\}$$

por último, a instrução **ret** transfere o valor do topo da pilha (5) para registo de activação anterior, destruindo ao registo de activação corrente. Vejamos agora um segundo exemplo, com criação de um fio de execução:

```
method int main()
{
l0: empty
    ldarg this
    ldc 5
    fork int abs(int)
    wait
    ret
}
```

$$\{\Sigma, (t, \rho \cdot \langle a[this \mapsto \mathtt{1}], \varepsilon, \varepsilon, \textbf{ldarg}\ this \cdot A\rangle^{main}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, \varepsilon, \mathtt{1}, A\rangle^{main}); H\}$$

$$\{\Sigma, (t, \rho \cdot \langle a, \varepsilon, \mathtt{1}, \textbf{ldc}\ 5 \cdot A\rangle^{main}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, \varepsilon, \mathtt{1} \cdot 5, A\rangle^{main}); H\}$$

$$\{\Sigma, (t, \rho \cdot \langle a, \varepsilon, \mathtt{1} \cdot 5, \textbf{fork int}\ abs(\textbf{int}) \cdot A\rangle^{main}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, \varepsilon, t', A\rangle^{main}), (t', \langle (this \mapsto \mathtt{1}, x \mapsto 5), \varepsilon, \varepsilon, A'\rangle^{abs}); H\}$$

$$\{\Sigma, (t, \rho \cdot \langle a, \varepsilon, t', \textbf{wait} \cdot A\rangle^{main}), (t', \langle a', \varepsilon, 5, \textbf{ret}\rangle^{abs}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, \varepsilon, 5, A\rangle^{main}); H\}$$

finalmente, a instrução **ret** transfere o valor do topo da pilha (5) para o registo de activação anterior, eliminando o registo de activação corrente.

$$
\begin{array}{rcll}
\tau & ::= & \textbf{int} \mid \textbf{lock} \mid \textbf{th}\langle\tau\rangle \mid c & \text{(Tipos)} \\
\Psi & ::= & \overline{c \ : \ (\ \overline{f{:}\tau}\ ,\ \overline{m{:}\tau(\overline{x{:}\tau})}\ )} & \text{(Ambiente Global)} \\
\Phi & ::= & \overline{l{:}\Gamma},\ \overline{x{:}\tau} & \text{(Ambiente Local)} \\
\Gamma & ::= & \varepsilon \mid \Gamma \cdot \tau & \text{(Pilha de Tipos)}
\end{array}
$$

Figura 6.8: Sintaxe dos tipos

6.3 Sistema de Tipos

O sistema de tipos proposto nesta secção para a linguagem intermédia atribui tipos a classes, a métodos, a variáveis de instância/locais e a etiquetas de blocos (de código). O tipo de uma classe define a sua interface, isto é, o tipo de cada método e variável de instância. O tipo de um método define os tipos dos parâmetros e o tipo do retorno. Por outro lado, o tipo de cada etiqueta caracteriza o conteúdo da pilha de avaliação no início da execução do seu bloco. Por exemplo, a atribuição de um tipo $\{\textbf{int}\}$ a uma etiqueta significa que qualquer bloco que salte (para essa etiqueta) manipulou a pilha de avaliação de tal forma que, no momento anterior ao salto, é composta por um único elemento do tipo inteiro. O sistema de tipos assegura de forma composicional que cada operação de um bloco bem tipificado é executada numa pilha de avaliação com o conteúdo adequado e que a manipula da forma esperada. Mais, garante que cada método consome todos os valores da pilha e que devolve um valor do tipo adequado. Assim, conseguimos garantir, com um passo apenas e sem recurso a análises de fluxos de dados, a ausência de erros derivados de tipos errados nos argumentos das instruções (presentes na pilha), utilização da pilha de avaliação para além dos limites, saltos para locais ilegais no código, ou saltos com um tipo de pilha corrente incoerente face ao tipo associado à etiqueta destino.

A sintaxe dos tipos apresentada na Figura 6.8 define 4 tipos: inteiro, *lock*, *thread* associado a um tipo, e o tipo de objecto. No ambiente global (Ψ) estão guardadas as interfaces de cada classe (c), enquanto que o ambiente local (Φ) é composto pelas etiquetas e variáveis locais, associadas a tipos de pilha (Γ) e a tipos (τ), respectivamente. No tipo de pilha (Γ) encontram-se vários tipos (τ) segundo o modelo LIFO (o último elemento a entrar é o primeiro a sair).

Tendo em conta as definições anteriormente apresentadas, elaborámos as regras que compõem o sistema de tipos. Um dado estado $\{P;\Sigma;H\}$ da máquina virtual, num ambiente Ψ, está bem tipificado (T-STATE) se o programa, o conjunto de fios e a *heap* estiverem bem tipificados.

$$
\frac{\Psi \vdash P \qquad \Psi \vdash \Sigma \qquad \Psi \vdash H}{\Psi \vdash \{P;\Sigma;H\}} \qquad \text{(T-STATE)}
$$

Um programa P está bem tipificado (T-PROGRAM) se qualquer das suas classes estiver

bem tipificada, num ambiente Ψ constituído pelas interfaces de todas as classes do programa.

$$\frac{P = \overline{\textbf{class } c \ \{ \ F \ M \ \}} \qquad \forall_i \quad \Psi, \ \overline{c : (\overline{f{:}\tau} \ , \ \overline{m{:}\tau(\overline{x{:}\tau})})} \vdash \textbf{class } c_i \ \{ \ F_i \ M_i \ \}}{\Psi \vdash P} \ (\text{T-PROGRAM})$$

A regra (T-CLASS) especifica que uma classe (c) está bem tipificada se todos os campos e métodos estiverem bem tipificados, no ambiente Ψ e com o tipo do objecto corrente (*this*) igual ao tipo da classe (c).

$$\frac{\begin{array}{c}\Psi \vdash c \ : \ (\ \overline{f{:}\tau} \ , \ \overline{m{:}\tau(\overline{x{:}\tau})} \ ) \qquad \forall_{i,j} \quad \Psi \vdash \sigma_j \\ M_i = \textbf{method } \tau_i \ m_i(\overline{\tau_i \ x_i}) \ \{\_ \ \_\} \quad \Psi, \mathit{this}{:}c \vdash \ M_i : \ \tau_i(\overline{\tau_i})\end{array}}{\Psi \vdash \textbf{class } c \ \{ \ \overline{\textbf{field } \sigma \ f} \quad \overline{M} \ \}} \qquad (\text{T-CLASS})$$

A regra (T-METHOD) permitir especificar que um método está bem tipificado se todos os seus blocos de instruções estiverem bem tipificados, no ambiente Ψ e num ambiente Φ constituído pelas etiquetas, associadas a tipos de pilha (Γ), e por variáveis locais, associadas a tipos (τ).

$$\frac{\Phi' = \Phi, \overline{l{:}\Gamma}, \overline{x'{:}\tau'} \qquad \forall_l \quad \Psi, \Phi' \vdash_\tau A_l : \Gamma_l \longrightarrow \Gamma'}{\Psi, \Phi \vdash \textbf{method } \tau \ m(\overline{\tau \ x}) \ \{ \ \textbf{locals}(\overline{\tau' \ x'}) \ \ \overline{l{:}\Gamma \mapsto A} \ \} \ : \ \tau(\overline{\tau})} \qquad (\text{T-METHOD})$$

Na verificação de uma sequência de instruções (T-INST), iniciando numa pilha com tipo Γ, é averiguado se a primeira instrução (ι) está bem tipificada partindo de Γ e originando um novo tipo de pilha Γ''. A partir deste último tipo é verificado se o resto da sequência de instruções está bem tipificada, repetindo-se o procedimento, até se chegar à última instrução, quer seja a instrução de retorno (**ret**) ou de salto incondicional (**br**). A tipificação de cada instrução é realizada sobre um tipo de retorno (τ) de um método, na qual a instrução está inserida, sendo omitido sempre que não é utilizado.

$$\frac{\Psi, \Phi \vdash_\tau \iota : \Gamma \longrightarrow \Gamma'' \qquad \Psi, \Phi \vdash_\tau A : \Gamma'' \longrightarrow \Gamma'}{\Psi, \Phi \vdash_\tau \iota \cdot A : \Gamma \longrightarrow \Gamma'} \qquad (\text{T-INST})$$

Finalmente, temos as regras para o tipo inteiro (T-INT), *lock* (T-LOCK), *thread* (T-THREAD) e objecto (T-CLASSV). Tendo em conta todas estas regras apresentadas até aqui, definimos as regras de tipo para cada instrução. Apresentamos de seguida as regras para as instruções básicas, seguidas das instruções relativas aos objectos, e das regras das instruções de suporte à concorrência. No final apresentamos um pequeno exemplo que ilustra o algoritmo de tipificação.

$$\Psi \vdash \textbf{int} \qquad \Psi \vdash \textbf{lock} \qquad (\text{T-INT}, \text{T-LOCK})$$

$$\frac{\Psi \vdash \tau}{\Psi \vdash \textbf{th}\langle\tau\rangle} \qquad \frac{c \in \text{dom}(\Psi)}{\Psi \vdash c} \qquad (\text{T-THREAD}, \text{T-CLASSV})$$

Instruções básicas

Na regra (T-POP), dada uma pilha $\Gamma \cdot \tau$, transita-se para uma segunda descartando-se o tipo τ presente no topo. O contrário se sucede na regra (T-LDC), onde o tipo de pilha resultante consiste no anterior (Γ) mais o tipo **int** no topo. No caso da regra (T-DUP), transita-se para um tipo de pilha igual ao anterior, mais uma cópia do tipo presente no topo.

$$\Psi, \Phi \vdash \textbf{pop}: \Gamma \cdot \tau \longrightarrow \Gamma \qquad \Psi, \Phi \vdash \textbf{ldc}\ n: \Gamma \longrightarrow \Gamma \cdot \textbf{int} \qquad (\text{T-POP}, \text{T-LDC})$$

$$\Psi, \Phi \vdash \textbf{dup}: \Gamma \cdot \tau \longrightarrow \Gamma \cdot \tau \cdot \tau \qquad (\text{T-DUP})$$

A tipificação da instrução **not** (T-NOT) requer um tipo de pilha não vazio, com um tipo inteiro no topo. O tipo de pilha de saída é igual ao anterior. No caso da regra (T-ADD) requerem-se dois tipos inteiros no topo da pilha, produzindo um tipo de pilha com um inteiro no topo.

$$\Psi, \Phi \vdash \textbf{not}: \Gamma \cdot \textbf{int} \longrightarrow \Gamma \cdot \textbf{int} \qquad \Psi, \Phi \vdash \textbf{add}: \Gamma \cdot \textbf{int} \cdot \textbf{int} \longrightarrow \Gamma \cdot \textbf{int} \qquad (\text{T-NOT}, \text{T-ADD})$$

No caso das regras para as instruções de salto (T-BR) e (T-BRFALSE) é necessário fazer a concordância entre o tipo da etiqueta destino (l) e o tipo corrente da pilha (Γ). De salientar que no caso de um salto condicional (T-BRFALSE) é retirado o inteiro do resultado do teste da pilha de entrada ($\Gamma \cdot \textbf{int} \longrightarrow \Gamma$).

$$\frac{\Phi(l) = \Gamma}{\Psi, \Phi \vdash \textbf{br}\ l: \Gamma \longrightarrow \Gamma} \qquad \frac{\Phi(l) = \Gamma}{\Psi, \Phi \vdash \textbf{brfalse}\ l: \Gamma \cdot \textbf{int} \longrightarrow \Gamma} \qquad (\text{T-BR}, \text{T-BRFALSE})$$

As regras (T-LDARG) e (T-LDLOC) indicam que é adicionado um tipo τ ao tipo da pilha ($\Gamma \longrightarrow \Gamma \cdot \tau$), de acordo com os tipos dos argumentos e variáveis locais, registados nos ambientes Ψ e Φ, respectivamente. No caso da regra (T-STLOC), verifica-se se o tipo do topo da pilha é o mesmo que o da variável local associada ao identificador x. As

anotações auxiliares (c,m), utilizadas apenas nestas regras, representam o método (m), da classe (c), onde a instrução está inserida.

$$\frac{\Psi(c) = (\_\,;\, \overline{M}, m{:}\_(\ldots, x{:}\tau, \ldots), \overline{M'})}{\Psi, \Phi \vdash^{c,m} \textbf{ldarg}\ x : \Gamma \longrightarrow \Gamma \cdot \tau} \qquad \text{(T-LDARG)}$$

$$\frac{\Phi(x) = \tau}{\Psi, \Phi \vdash \textbf{ldloc}\ x : \Gamma \longrightarrow \Gamma \cdot \tau} \qquad \frac{\Phi(x) = \tau}{\Psi, \Phi \vdash \textbf{stloc}\ x : \Gamma \cdot \tau \longrightarrow \Gamma} \qquad \text{(T-LDLOC, T-STLOC)}$$

Instruções relativas a objectos

No caso das instruções relacionadas com objectos, a regra (T-NEWOBJ) especifica que o topo da pilha passa a ter o tipo do objecto criado ($\Gamma \cdot c$).

$$\frac{\Psi \vdash c}{\Psi, \Phi \vdash \textbf{newobj}\ c : \Gamma \longrightarrow \Gamma \cdot c} \qquad \text{(T-NEWOBJ)}$$

As regras (T-LDFLD) e (T-STFLD) garantem a correcta manipulação das variáveis de instância de um objecto, de acordo com a informação de tipos da pilha corrente.

$$\frac{\Psi(c) = (F, \_) \quad F(f) = \tau}{\Psi, \Phi \vdash \textbf{ldfld}\ f : \Gamma \cdot c \longrightarrow \Gamma \cdot \tau} \qquad \frac{\Psi(c) = (F, \_) \quad F(f) = \tau}{\Psi, \Phi \vdash \textbf{stfld}\ f : \Gamma \cdot c \cdot \tau \longrightarrow \Gamma} \qquad \text{(T-LDFLD, T-STFLD)}$$

A regra (T-CALL) verifica a concordância dos tipos do topo da pilha (o tipo do objecto (c) e dos argumentos ($\overline{\tau}$)) com os da assinatura do método. Após a execução do método (m) espera-se que a pilha tenha no topo um valor do tipo do retorno do método (τ). Note-se que o tipo da classe e dos argumentos foram removidos da pilha e adicionado o tipo do retorno do método ($\Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau$). A regra (T-RET) verifica que a pilha contém um só elemento e que está de acordo com o tipo de retorno do método (τ).

$$\frac{\Psi(c) = (\_\,, M) \quad M(m) = \tau(\overline{\tau})}{\Psi, \Phi \vdash \textbf{call}\ \tau\ m(\overline{\tau}) : \Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau} \qquad \Psi, \Phi \vdash_\tau \textbf{ret} : \tau \longrightarrow \tau \qquad \text{(T-CALL, T-RET)}$$

Instruções de suporte à concorrência

A regra (T-FORK) verifica a chamada do método, do mesmo modo que a regra (T-CALL), mas o tipo resultante denota a inclusão de um identificador de um fio de execução que

irá devolver um elemento do mesmo tipo do método chamado ($\Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \textbf{th}\langle\tau\rangle$).

$$\frac{\Psi, \Phi \vdash \textbf{call}\ \tau\ m(\overline{\tau}) : \Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau}{\Psi, \Phi \vdash \textbf{fork}\ \tau\ m(\overline{\tau}) : \Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \textbf{th}\langle\tau\rangle} \qquad (\text{T-FORK})$$

No caso da regra (T-WAIT), o topo do tipo da pilha deve ser do tipo $\textbf{th}\langle\tau\rangle$, sendo que o tipo da pilha resultado contém o tipo τ no topo.

$$\Psi, \Phi \vdash \textbf{wait} : \Gamma \cdot \textbf{th}\langle\tau\rangle \longrightarrow \Gamma \cdot \tau \qquad (\text{T-WAIT})$$

A regra (T-NEWLOCK) apenas acrescenta ao tipo da pilha resultante um elemento de tipo **lock**.

$$\Psi, \Phi \vdash \textbf{newlock} : \Gamma \longrightarrow \Gamma \cdot \textbf{lock} \qquad (\text{T-NEWLOCK})$$

As regras (T-SHARED) e (T-SYNC) tipificam uma chamada de método (T-CALL) com a verificação adicional de que no topo da pilha (após retirar a informação da chamada do método) está um **lock**.

$$\frac{\Psi, \Phi \vdash \textbf{call}\ \tau\ m(\overline{\tau}) : \Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau}{\Psi, \Phi \vdash \textbf{shared}\ \tau\ m(\overline{\tau}) : \Gamma \cdot \textbf{lock} \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau} \qquad (\text{T-SHARED})$$

$$\frac{\Psi, \Phi \vdash \textbf{call}\ \tau\ m(\overline{\tau}) : \Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau}{\Psi, \Phi \vdash \textbf{sync}\ \tau\ m(\overline{\tau}) : \Gamma \cdot \textbf{lock} \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau} \qquad (\text{T-SYNC})$$

Exemplo

Partindo do seguinte exemplo, que lança um fio de execução associado ao método abs, da classe Abs, e que espera que o mesmo termine:

```
method int main() {
l0: empty
    ldarg this
    ldc 5
    fork int abs(int)
    wait
    ret
}
```

obtemos o seguinte processo de verificação, para o método main, também da classe Abs, de acordo com as regras anteriormente definidas:

$\Psi = Abs : (\varepsilon\ ;\ main{:}\textbf{int}()\ ,\ abs{:}\textbf{int}(\textbf{int})\ )$
$\Phi = this{:}Abs,\ l0{:}\textbf{emtpy}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Psi,\Phi \vdash \textbf{ldarg}\ this : \varepsilon \longrightarrow Abs}}{\Psi,\Phi \vdash \textbf{ldc}\ 5 : Abs \longrightarrow Abs \cdot \textbf{int}}}{\Psi,\Phi \vdash \textbf{fork}\ \ \textbf{int}\ abs(\textbf{int}) : Abs \cdot \textbf{int} \longrightarrow \textbf{th}\langle\textbf{int}\rangle}}{\Psi,\Phi \vdash \textbf{wait} : \textbf{th}\langle\textbf{int}\rangle \longrightarrow \textbf{int}}}{\Psi,\Phi \vdash \textbf{ret} : \textbf{int} \longrightarrow \textbf{int}}}{\Psi,\Phi \vdash \textbf{wait} \cdot \textbf{ret} : \textbf{th}\langle\textbf{int}\rangle \longrightarrow \textbf{int}}}{\Psi,\Phi \vdash \textbf{fork}\ \ \textbf{int}\ abs(\textbf{int}) \cdot \textbf{wait} \cdot \textbf{ret} : Abs \cdot \textbf{int} \longrightarrow \textbf{int}}}{\Psi,\Phi \vdash \textbf{ldc}\ 5 \cdot \textbf{fork}\ \ \textbf{int}\ abs(\textbf{int}) \cdot \textbf{wait} \cdot \textbf{ret} : Abs \longrightarrow \textbf{int}}}{\Psi,\Phi \vdash \textbf{ldarg}\ this \cdot \textbf{ldc}\ 5 \cdot \textbf{fork}\ \ \textbf{int}\ abs(\textbf{int}) \cdot \textbf{wait} \cdot \textbf{ret} : \varepsilon \longrightarrow \textbf{int}}$$

$$\Psi,\Phi \vdash \textbf{method int}\ main()\ \{\ l0{:}\textbf{empty} \mapsto \textbf{ldarg}\ this \cdot \textbf{ldc}\ 5 \cdot \textbf{fork int}\ abs(\textbf{int}) \cdot \textbf{wait} \cdot \textbf{ret}\ \}\ :\ \textbf{int}()$$

Resultados

Será de esperar que dadas as definições para a semântica e sistema de tipos apresentado que os teoremas habituais de preservação de tipos e de progresso sejam verdadeiros. Apesar de não termos elaborado provas formais realizamos vários testes que, mesmo não dando garantias, permitiram invalidar programas com chamadas a métodos não existentes, a utilização da pilha de avaliação para além dos seus limites, a não concordância de parâmetros, a utilização incorrecta das etiquetas para os saltos internos aos métodos, a não concordância dos tipos das variáveis locais e argumentos, etc. Alguns resultados mais elaborados, como serão devidamente mencionados no último capítulo, podem ser obtidos através dos tipos comportamentais e outros, como *singleton types* para os identificadores de fios de execução. Estas abordagens permitiriam resolver problemas derivados de operações bloqueantes (**wait**) sobre fios já destruídos, ou mesmo garantir o cumprimento dos protocolos dos objectos.

6.4 Implementação

Foi implementada na linguagem C uma máquina virtual para a linguagem intermédia definida nas secções anteriores. A arquitectura da máquina está dividida em cinco partes: os metadados, o *parser*, o interpretador, o verificador de código e algumas estruturas de dados. Os metadados, como o próprio nome sugere, servem para representar internamente as classes, os métodos, os blocos, entre outros.

O *parser* carrega as classes da linguagem intermédia para a máquina virtual. Em várias máquina virtuais, como por exemplo na JVM/CLR, uma classe em linguagem intermédia é serializada para código máquina e só depois é carregada na máquina virtual para ser executada. Por questões de simplicidade, na nossa máquina virtual o código de uma classe em linguagem intermédia é directamente carregado para a máquina virtual, ficando em memória as classes e os respectivos métodos com blocos de instruções em *bytes*. Também por questões de simplicidade, o carregamento não é dinâmico.

Relativamente ao interpretador, este módulo é responsável pelo processamento das várias instruções (no formato *byte*) que modificam as estruturas de dados, nomeadamente os registos de activação (variáveis locais, pilha de avaliação) e a *heap* (com um *Garbage Collector* baseado no algoritmo *mark-and-sweep*). Por outro lado, o verificador de código analisa as instruções de todos os blocos de cada método, de cada classe carregada na máquina, de forma a evitar determinados erros que poderiam comprometer o funcionamento correcto do interpretador. Tanto o interpretador como o verificador de código foram implementados de forma bastante modular, com vista a facilitar possíveis extensões/modificações. As instruções, em formato orientado ao *byte*, estão mapeadas em funções específicas para cada uma, tanto no interpretador como no verificador, através de um vector de apontadores de funções, cujo índice representa o código da instrução.

De forma a maximizar o desempenho de programas, em máquinas com múltiplos processadores, recorremos à API das `pthreads` para efectuar o mapeamento (um-para-um) entre os fios de execução da nossa linguagem e os do Sistema Operativo, ficando também o escalonamento dos mesmos entregue à biblioteca. No entanto, este mapeamento pode ser uma desvantagem em programas com um número elevado de fios de execução, isto é, bastante superior ao número de processadores da máquina real, visto que o escalonamento de fios de execução ao nível da aplicação seria mais leve que o escalonamento dos mesmos ao nível do Sistema de Operação [36]. Por outro lado, um mapeamento de muitos-para-um, apesar de tornar mais leve o escalonamento, não iria aproveitar os (eventuais) múltiplos processadores da máquina (real). O ideal, que poderá ser implementado no futuro, seria o modelo intermédio dos dois anteriores, designado por mapeamento de muitos-para-muitos, onde o programador não precisa de se preocupar com o número (eventualmente elevado) de fios de execução lançados no programa. O próprio sistema de suporte à execução iria aproveitar os vários recursos da máquina e, ao mesmo tempo, fazer a gestão entre os fios de execução do nível da linguagem e os do nível do SO.

7

Tradução para a Linguagem Intermédia

Apresentamos neste capítulo a função de tradução, que serve de base à implementação do compilador da linguagem de alto nível para linguagem intermédia, ambas apresentadas nos capítulos anteriores. No final descrevemos a arquitectura do compilador, assim como os detalhes mais importantes da implementação da função de tradução.

Como ambas as linguagens são baseadas num modelo de objectos com classes, definimos um mapeamento entre a estrutura de classes, campos e métodos, de uma linguagem para a outra. As principais diferenças, que requerem uma tradução mais elaborada, dizem respeito a algumas expressões, que formam o corpo dos métodos na linguagem de alto nível, conterem expressões de criação de novos fios de execução (**fork** e), ou acesso partilhado ou seguro sobre um dado *lock* (**shared**$(e)\{e'\}$, **sync**$(e)\{e'\}$). O aninhamento deste tipo de construções dá origem a vários métodos auxiliares, e a resolução estática de nomes é realizada através de passagem de referências como argumentos dos métodos. Tendo em conta este cenário, definimos a função de tradução de um programa (conjunto de definições de classes) da seguinte forma:

$$[\![\overline{C}]\!] \triangleq \overline{[\![C]\!]_\Psi}$$

que depende apenas da função de tradução de métodos com a forma:

$$[\![\tau\ m(\overline{\tau})\ \{\ e\ \}]\!]_\Psi \triangleq \textbf{method}\ \tau\ m(\overline{\tau})\ \{\ \textbf{locals}(\Phi)\ \overline{B}\ \}\ ,\ \overline{M}$$

onde o conjunto de blocos ($\overline{B}$) e o conjunto de métodos ($\overline{M}$) resultam da função de tradução de expressões com a forma:

$$[\![e]\!]^{\Gamma,l}_{\Psi,\Phi} \triangleq (A\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi'\ ;\ \tau)$$

A tradução de uma expressão (e), escrita da forma $[\![e]\!]^{\Gamma,l}_{\Psi,\Phi}$, é definida em relação a um tipo de pilha (Γ) que descreve o conteúdo da pilha antes da avaliação da expressão, em relação a uma etiqueta (l) que denota o ponto de saída da avaliação da expressão traduzida, e em relação aos ambientes (Ψ) e (Φ) que descrevem as interfaces das classes do programa e as constantes/variáveis locais associadas a tipos, respectivamente. O resultado da tradução é composto por uma sequência de instruções da linguagem intermédia (A), um conjunto de blocos auxiliares de instruções ($\overline{B}$), um conjunto de métodos auxiliares ($\overline{M}$), um conjunto de nomes e tipos de constantes/variáveis locais (Φ') declaradas na expressão (e), e um tipo τ que denota o tipo da expressão traduzida.

Note-se que o bloco de instruções A é o ponto de entrada para a avaliação da expressão, e que a etiqueta l denota o ponto de saída da avaliação da expressão. Entre o início e o fim da avaliação da expressão poderá haver um número variável de saltos entre os vários blocos gerados, garantindo-se que a execução termina no ponto anunciado (etiqueta l). De salientar também que o valor da expressão (traduzida) é deixado como resultado no topo da pilha de avaliação. Nas definições das próximas secções, por questões de simplicidade, omitimos os parâmetros (Γ, l, Ψ e Φ) nos casos em que não são usados.

Tendo em conta que, nos métodos auxiliares que são gerados a partir das expressões **fork** e, **shared**$(e)\{e'\}$ e **sync**$(e)\{e'\}$, as variáveis extravasam (eventualmente) o âmbito dos métodos, decidimos estender as classes da linguagem intermédia anteriormente apresentada introduzindo tipos paramétricos, de modo encapsular as variáveis em objectos do tipo $Cell\langle\tau\rangle$, com o tipo paramétrico τ igual ao tipo do valor a ser guardado na variável. O acesso a valores é realizado pela chamada ao método get (**call** τ $get()$), enquanto que a afectação dos mesmos é efectuada pela invocação do método set (**call** τ $set(\tau)$).

7.1 Tradução de um Programa

A tradução de um programa (conjunto de classes) da linguagem de alto nível para a linguagem intermédia envolve a tradução de cada classe, que é realizada no ambiente Ψ, que representa um mapa constituído pelos identificadores das classes e pelas respectivas interfaces (tipos dos campos e tipos dos métodos).

$$[\![\overline{C}]\!] \ \triangleq\ \overline{[\![C]\!]_\Psi}$$

$$\Psi = \overline{c : (\overline{f{:}\tau}\ ;\ \overline{m{:}\tau(\overline{\tau\ x})})}$$

Na tradução de uma classe, os respectivos campos são traduzidos (directamente) para

campos na linguagem intermédia, visto que apenas diferem na sintaxe, sendo omitido este pormenor. Por outro lado, a tradução de um método, na linguagem de alto nível, origina um ou mais métodos na linguagem intermédia.

$$[\![\textbf{class}\ c\ \{\ \overline{F}\ \overline{M}\ \}]\!]_\Psi \triangleq \textbf{class}\ c\ \{\ \overline{F}\ \overline{[\![M]\!]_\Psi}\ \}$$

Na tradução de um método a sua assinatura mantêm-se. Por outro lado, é necessário traduzir a expressão que compõe o corpo, que é efectuado num contexto em que a pilha actual e o conjunto de variáveis locais estão vazios (ε). Dado que estamos a traduzir uma expressão, a etiqueta final (l) está associada a uma pilha com um único tipo (τ). Os dois blocos do conjunto ($\overline{B'}$), separados por uma linha, contêm as primeiras instruções (A) da expressão traduzida (e) e a instrução de retorno (**ret**), respectivamente.

$$[\![\tau\ m(\overline{\tau\ x})\ \{\ e\ \}]\!]_\Psi \triangleq \textbf{method}\ \tau\ m(\overline{\tau\ x})\ \{\ \textbf{locals}(\Phi)\ \overline{B'},\overline{B}\ \}\ ,\ \overline{M}$$

where $[\![e]\!]^{\varepsilon,l}_{\Psi,\varepsilon} \triangleq (A\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi\ ;\ \tau)$
l, l' are fresh

$$\overline{B'} = \begin{array}{l} l'\text{:}\textbf{emtpy} \\ A \\ \\ l\text{:}\tau \\ \textbf{ret} \end{array}$$

7.2 Tradução de Expressões

Expressões Básicas

A tradução de um identificador x varia consoante seja uma constante, uma variável local, um argumento ou um campo definido numa classe. Se for uma constante carrega-se o valor para o topo da pilha (**ldloc** x). O mesmo raciocínio aplica-se se for uma variável, com a excepção que é necessário retirar o valor que está encapsulado na célula (**call** $\tau\ get()$). Se for uma argumento, empilha-se o respectivo valor (**ldarg** x). Caso contrário, o identificador só poderá ser um campo do objecto corrente, colocando-se o respectivo valor no topo da pilha (**ldfld** x). Utilizamos excepcionalmente os argumentos (c,m) da função de tradução para representar o método (m), da classe (c), onde a expressão está inserida.

$$[\![x]\!]^{\Gamma,l}_{\Psi,\Phi,c,m} \triangleq (A\ ;\ \varepsilon\ ;\ \varepsilon\ ;\ \varepsilon\ ;\ \tau)$$

where $\Phi(x) = \tau$ $(constants)$
$A =$ **ldloc** x
br l

where $\Phi(x) = Cell\langle\tau\rangle$ $(variables)$
$A =$ **ldloc** x
call $\tau\ get()$
br l

where $\Psi(c) = (\_\ ;\ \overline{M}, m{:}\_(...,x{:}\tau,...), \overline{M'})$ $(arguments)$
$A =$ **ldarg** x
br l

where $\Psi(c) = (\overline{F}, x{:}\tau, \overline{F'}\ ;\ \_)$ $(fields)$
$A =$ **ldarg** this
ldfld x
br l

A tradução de um literal inteiro (n) gera código máquina que coloca, no topo da pilha, a respectiva constante (n). O conjunto de blocos, de métodos e de variáveis locais são vazios, e a expressão é do tipo inteiro (**int**).

$$[\![n]\!]^{l} \triangleq (\langle \textbf{ldc}\ n\rangle, \langle \textbf{br}\ l\rangle\ ;\ \varepsilon\ ;\ \varepsilon\ ;\ \varepsilon\ ;\ \textbf{int})$$

A negação de uma expressão e envolve a tradução da própria expressão (cujo código resultante deverá colocar no topo da pilha um inteiro) juntamente com a instrução **not** que nega o valor do topo da pilha.

$$[\![!e]\!]^{\Gamma,l'} \triangleq (A\ ;\ \overline{B}, \overline{B'}\ ;\ \overline{M}\ ;\ \Phi\ ;\ \textbf{int})$$

where $[\![e]\!]^{\Gamma,l} = (A\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi\ ;\ \textbf{int})$
l is fresh

$\overline{B'} = l{:}\Gamma \cdot$ **int**
not
br l'

Na tradução de uma soma (ou qualquer outra operação matemática) são traduzidas as duas expressões, que deverão colocar no topo da pilha os respectivos valores, seguido da instrução **add**.

$$[\![e + e']\!]^{\Gamma,l''} \triangleq (A\ ;\ \overline{B},\overline{B'},\overline{B''}\ ;\ \overline{M},\overline{M'}\ ;\ \Phi,\Phi'\ ;\ \textbf{int})$$

$$\begin{array}{rrcl} \text{where} & [\![e]\!]^{\Gamma,l} & = & (A\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi\ ;\ \textbf{int}) \\ & [\![e']\!]^{\Gamma\cdot\textbf{int},l'} & = & (A'\ ;\ \overline{B'}\ ;\ \overline{M'}\ ;\ \Phi'\ ;\ \textbf{int}) \\ & & & l, l' \text{ are fresh} \end{array}$$

$$\begin{array}{rcl} \overline{B''} & = & l{:}\Gamma\cdot\textbf{int} \\ & & A' \\ & & \\ & & l'{:}\Gamma\cdot\textbf{int}\cdot\textbf{int} \\ & & \textbf{add} \\ & & \textbf{br}\ l'' \end{array}$$

Na tradução do operador de sequência traduz-se o código da expressão à esquerda, que deverá deixar no topo da pilha um valor, seguida da instrução **pop**, para desempilhar o valor, e finalmente seguida da tradução da expressão à direita do operador. O tipo da expressão, como seria de esperar, é igual ao tipo do operando à direita do operador.

$$[\![e\ ;\ e']\!]^{\Gamma,l'} \triangleq (A\ ;\ \overline{B},\overline{B'},\overline{B''}\ ;\ \overline{M},\overline{M'}\ ;\ \Phi,\Phi'\ ;\ \tau')$$

$$\begin{array}{rrcl} \text{where} & [\![e]\!]^{\Gamma,l} & = & (A\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi\ ;\ \tau) \\ & [\![e']\!]^{\Gamma,l'} & = & (A'\ ;\ \overline{B'}\ ;\ \overline{M'}\ ;\ \Phi'\ ;\ \tau') \\ & & & l \text{ is fresh} \end{array}$$

$$\begin{array}{rcl} \overline{B''} & = & l{:}\Gamma\cdot\tau \\ & & \textbf{pop} \\ & & A' \end{array}$$

A afectação de valores envolve o carregamento de uma célula associada ao identificador x, no caso de ser uma variável local, seguido do carregamento do valor e da chamada ao método set pela instrução apropriada (**call**). Caso seja um campo, é necessário carregar a referência para o objecto corrente (**ldarg** *this*), seguida do carregamento do valor e da instrução **stfld**. Em ambos os casos é deixado o próprio valor no topo da pilha, visto que a afectação de valores (na linguagem de alto nível) é uma expressão.

$$[\![x := e]\!]^{\Gamma,l}_{\Psi,\Phi,c} \triangleq (A\ ;\ B,\overline{B}\ ;\ \overline{M}\ ;\ \Phi'\ ;\ \tau)$$

where $[\![e]\!]^{\Gamma',l'}_{\Psi,\Phi,c} = (A'\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi'\ ;\ \tau)$
l' is fresh

and $\Phi(x) = Cell\langle\tau\rangle$ $(variables)$
$\Gamma' = \Gamma \cdot Cell\langle\tau\rangle$
$A =$ **ldloc** x
A'

$B = l'{:}\Gamma \cdot Cell\langle\tau\rangle \cdot \tau$
call τ $set(\tau)$
br l

and $\Psi(c) = (\overline{F},...x{:}\tau...,\overline{F'};\_)$ $(fields)$
$\Gamma' = \Gamma \cdot c$
$A =$ **ldarg** *this*
A'

$B = l'{:}\Gamma \cdot c \cdot \tau$
stfld x
ldarg *this*
ldfld x
br l

Como exemplo ilustrativo da tradução da afectação, considere-se a seguinte expressão ($x := 5 + 6$) de uma variável local (x) previamente declarada, com a pilha de avaliação actual vazia (ε):

$$
\begin{array}{rcl}
[\![x := 5 + 6]\!]^{\varepsilon,l}_{\Psi,\Phi} & \triangleq & (\langle \textbf{ldloc } x\rangle, \langle \textbf{ldc } 5\rangle, \langle \textbf{br } l'\rangle \ ; \ \overline{B} \ ; \ \varepsilon \ ; \ \varepsilon \ ; \ \textbf{int}) \\
 & & l' \text{ is fresh} \\
 & & \\
\overline{B} & = & l' : Cell\langle \textbf{int}\rangle \cdot \textbf{int} \\
 & & \textbf{ldc } 6 \\
 & & \textbf{br } l'' \\
 & & \\
 & & l'' : Cell\langle \textbf{int}\rangle \cdot \textbf{int} \cdot \textbf{int} \\
 & & \textbf{add} \\
 & & \textbf{br } l''' \\
 & & \\
 & & l''' : Cell\langle \textbf{int}\rangle \cdot \textbf{int} \\
 & & \textbf{call int } set(\textbf{int}) \\
 & & \textbf{br } l
\end{array}
$$

Após serem empilhados a célula e o inteiro 5, dá-se um salto para um bloco com uma etiqueta l', que espera que a pilha (de avaliação) tenha esses mesmos dois elementos. De seguida, é traduzida a sub-expressão da direita da soma (6), dá-se novamente um salto para outra etiqueta (com tipo igual à anterior mas com mais um tipo **int** no topo) que soma os dois valores do topo. Finalmente, dá-se um último salto para um bloco que efectua uma invocação para alterar o valor encapsulado na célula. A chamada deixa o próprio valor, neste caso o inteiro, no topo da pilha.

Na tradução de ciclos é necessário traduzir, em primeiro, a condição do ciclo (A), seguida de um salto condicional (**brfalse** l_3) para o final do código gerado. Posteriormente é traduzido o corpo do método (A'), seguido da operação que desempilha (**pop**), que por sua vez é seguida de um salto incondicional (**br** l_2) para início, isto é, para o código gerado pela tradução da condição. No final do código gerado é empilhado o valor 0 visto que o ciclo, sendo uma expressão, também denota um valor, neste caso do tipo inteiro (**int**), que é colocado no topo da pilha.

$$[\![\textbf{while}(e)\{e'\}]\!]^{\Gamma,l_4} \triangleq (\langle \textbf{br}\ l_2\rangle\ ;\ \overline{B},\overline{B'},\overline{B''}\ ;\ \overline{M},\overline{M'}\ ;\ \Phi,\Phi'\ ;\ \textbf{int})$$

where

$$\begin{array}{rcl} [\![e]\!]^{\Gamma,l_0} & = & (A\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi\ ;\ \textbf{int}) \\ [\![e']\!]^{\Gamma,l_1} & = & (A'\ ;\ \overline{B}'\ ;\ \overline{M'}\ ;\ \Phi'\ ;\ \tau) \\ & & l_0, l_1, l_2, l_3 \text{ are fresh} \end{array}$$

$$\begin{array}{rcl} \overline{B''} & = & l_2{:}\Gamma \\ & & A \\ & & \\ & & l_0{:}\Gamma \cdot \textbf{int} \\ & & \textbf{brfalse}\ l_3 \\ & & A' \\ & & \\ & & l_1{:}\Gamma \cdot \tau \\ & & \textbf{pop} \\ & & \textbf{br}\ l_2 \\ & & \\ & & l_3{:}\Gamma \\ & & \textbf{ldc}\ 0 \\ & & \textbf{br}\ l_4 \end{array}$$

A tradução decisões é efectuada, primeiro, traduzindo a condição, que deverá deixar um inteiro no topo da pilha, seguida da instrução de salto condicional (**brfalse** l'). Depois seguem-se as traduções das expressões e' (A′) e e'' (A″). Ambas as sequências de instruções saltam para o final do código gerado (l'').

$$[\![\textbf{if}\ e\ \textbf{then}\ e'\ \textbf{else}\ e'']\!]^{\Gamma,l''} \triangleq (A\ ;\ \overline{B},\overline{B'},\overline{B''},\overline{B'''}\ ;\ \overline{M},\overline{M'},\overline{M''}\ ;\ \Phi,\Phi',\Phi''\ ;\ \tau)$$

where

$$\begin{array}{rcl} [\![e]\!]^{\Gamma,l} & = & (A\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi\ ;\ \textbf{int}) \\ [\![e']\!]^{\Gamma,l''} & = & (A'\ ;\ \overline{B'}\ ;\ \overline{M}'\ ;\ \Phi'\ ;\ \tau) \\ [\![e'']\!]^{\Gamma,l''} & = & (A''\ ;\ \overline{B''}\ ;\ \overline{M}''\ ;\ \Phi''\ ;\ \tau) \\ & & l, l' \text{ are fresh} \end{array}$$

$$\begin{array}{rcl} \overline{B'''} & = & l{:}\Gamma \cdot \textbf{int} \\ & & \textbf{brfalse}\ l' \\ & & A' \\ & & \\ & & l'{:}\Gamma \\ & & A'' \end{array}$$

A tradução de uma declaração de uma variável dá origem à criação de uma nova célula (**newobj** $Cell\langle\tau\rangle$), seguida da instrução específica que permite guardar valores em variáveis locais na linguagem intermédia (**stloc**). É realizada uma substituição de identificador, na expressão (e) que compõe o corpo da declaração, e é avaliada essa mesma expressão (e') num ambiente Φ' constituído pelo novo identificador associado ao respectivo tipo. De forma análoga ao que foi explicado no capítulo 5, a tradução de declarações com múltiplas variáveis pode-se efectuar através de *syntatic sugar*.

$$[\![\textbf{var } \tau\ x \textbf{ in } e]\!]^{\Gamma,l}_{\Psi,\Phi} \triangleq (A'\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi',\Phi''\ ;\ \tau')$$

where

$$\begin{aligned} \Phi' &= \Phi, x' \mapsto \tau \\ e' &= e[x'/x] \\ [\![e']\!]^{\Gamma,l}_{\Psi,\Phi'} &= (A\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi''\ ;\ \tau') \\ & \quad x' \text{ is fresh} \end{aligned}$$

$$\begin{aligned} A' &= \textbf{newobj } Cell\langle\tau\rangle \\ & \quad \textbf{stloc } x' \\ & \quad A \end{aligned}$$

A tradução de uma declaração de uma constante dá origem à tradução da expressão (associada à constante), seguida da instrução específica que permite guardar valores em variáveis locais na linguagem intermédia (**stloc**). Tal como na declaração de variáveis, é realizada uma substituição de um identificador, na expressão (e) que compõe o corpo da declaração, e é avaliada essa mesma expressão (e') num ambiente Φ' constituído pelo novo identificador associado ao respectivo tipo.

$$[\![\textbf{val } \tau\ x = e'' \textbf{ in } e]\!]^{\Gamma,l_f}_{\Psi,\Phi} \triangleq (A'\ ;\ \overline{B},\overline{B'},\overline{B''}\ ;\ \overline{M},\overline{M'}\ ;\ ,\Phi',\Phi'',\Phi'''\ ;\ \tau')$$

where

$$\begin{aligned} [\![e'']\!]^{\Gamma,l}_{\Psi,\Phi} &= (A'\ ;\ \overline{B''}\ ;\ \overline{M'}\ ;\ \Phi'''\ ;\ \tau) \\ \Phi' &= \Phi, x' \mapsto \tau \\ e' &= e[x'/x] \\ [\![e']\!]^{\Gamma,l_f}_{\Psi,\Phi'} &= (A\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi''\ ;\ \tau') \\ & \quad l, x' \text{ is fresh} \end{aligned}$$

$$\begin{aligned} \overline{B'} &= l{:}\Gamma \cdot \tau \\ & \quad \textbf{stloc } x' \\ & \quad A \end{aligned}$$

Expressões sobre Objectos

A tradução da criação de um objecto gera código máquina que coloca, no topo da pilha, a referência para o próprio. O conjunto de blocos, de métodos e de variáveis locais são vazios, e a expressão é do tipo da classe (c).

$$[\![\textbf{new } c]\!]^{l} \triangleq (\langle \textbf{newobj } c\rangle,\langle \textbf{br } l\rangle \ ; \ \varepsilon \ ; \ \varepsilon \ ; \ \varepsilon \ ; \ c)$$

A tradução de uma chamada a uma função envolve a tradução da expressão, sobre a qual está a ser invocada, assim como a tradução do argumento, seguida da chamada ao método (**call**), que deixa no topo da pilha o tipo de retorno do método (τ).

$$[\![e.m(e')]\!]_{\Psi}^{\Gamma,l_f} \triangleq (A \ ; \ \overline{B},\overline{B'},\overline{B''} \ ; \ \overline{M},\overline{M'} \ ; \ \Phi,\Phi' \ ; \ \tau)$$

where

$$\begin{array}{rcl} [\![e]\!]_{\Psi}^{\Gamma,l} & = & (A \ ; \ \overline{B} \ ; \ \overline{M} \ ; \ \Phi \ ; \ c) \\ [\![e']\!]_{\Psi}^{\Gamma\cdot c,l'} & = & (A' \ ; \ \overline{B'} \ ; \ \overline{M'} \ ; \ \Phi' \ ; \ \tau') \\ \Psi(c) & = & (\_ \ ; \ \overline{M''},m{:}\tau(\tau'),\overline{M'''}) \\ & & l, l' \text{ are fresh} \end{array}$$

$$\begin{array}{rcl} \overline{B''} & = & l{:}\Gamma\cdot c \\ & & A' \\ & & \\ & & l'{:}\Gamma\cdot c\cdot\tau' \\ & & \textbf{call } \tau \ m(\tau') \\ & & \textbf{br } l_f \end{array}$$

Expressões de Suporte à Concorrência

A tradução da criação de um fio de execução, para uma dada expressão e, dá origem a chamada concorrente (**fork** $\tau \ m(\overline{\tau})$) e à geração de um novo método (m), cujos argumentos ($\overline{x}$) são as variáveis livres em e. O corpo (do novo método) é constituído pela tradução da expressão e.

$$[\![\textbf{fork}\ e]\!]_{\Phi}^{\Gamma,l_f} \triangleq (A'\ ;\ \varepsilon\ ;\ \overline{M},\overline{M'}\ ;\ \varepsilon\ ;\ \textbf{th}\langle\tau\rangle)$$

where
$$\begin{array}{rcl}
\overline{x} & = & \text{freeVars(e)} \\
\Phi(x_i) & = & \tau_i \\
[\![e]\!]^{\varepsilon,l} & = & (A\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi'\ ;\ \tau) \\
\overline{M'} & = & \textbf{method}\ \tau\ m(\overline{\tau\ x})\ \{\ \textbf{locals}(\overline{x \mapsto \tau}, \Phi')\ \overline{B'},\overline{B}\ \} \\
 & & m, l, l' \text{ are fresh}
\end{array}$$

$$\begin{array}{rcl}
\overline{B'} & = & l'\text{:}\textbf{empty} \\
 & & \textbf{ldarg}\ x_1 \\
 & & \textbf{stloc}\ x_1 \\
 & & \ldots \\
 & & \textbf{ldarg}\ x_n \\
 & & \textbf{stloc}\ x_n \\
 & & A \\
 & & \\
 & & l\text{:}\tau \\
 & & \textbf{ret}
\end{array}$$

$$\begin{array}{rcl}
A' & = & \textbf{ldarg}\ this \\
 & & \textbf{ldloc}\ x_1 \\
 & & \ldots \\
 & & \textbf{ldloc}\ x_n \\
 & & \textbf{fork}\ \tau\ m(\overline{\tau}) \\
 & & \textbf{br}\ l_f
\end{array}$$

Como exemplo bastante simples da tradução da criação de fios de execução, considere-se a seguinte expressão (**fork** 1), com a pilha de avaliação actual vazia (ε):

$$[\![\textbf{fork}\ 1]\!]_{\Phi}^{\varepsilon,l_f} \triangleq (\langle\textbf{ldarg}\ this\rangle,\langle\textbf{fork int}\ m()\rangle,\langle\textbf{br}\ l_f\rangle\ ;\ \varepsilon\ ;\ \textbf{method int}\ m()\ \{\ \overline{B}\ \}\ ;\ \varepsilon\ ;\ \textbf{th}\langle\textbf{int}\rangle)$$
m, l, l' are fresh

$$\begin{array}{rcl}
\overline{B} & = & l'\text{:}\textbf{empty} \\
 & & \textbf{ldc}\ 1 \\
 & & \textbf{br}\ l \\
 & & \\
 & & l\text{:}\textbf{int} \\
 & & \textbf{ret}
\end{array}$$

Como se pode observar, a tradução envolve dois passos. Primeiro a criação do novo fio de execução, que executa um método auxiliar (m). Depois, no próprio corpo do método auxiliar, a tradução da expressão executada pelo novo fio (1).

A geração de código, relativa à expressão **wait** e, consiste na tradução da expressão e, que deverá deixar no topo da pilha um identificador para um fio de execução, com o tipo $\mathbf{th}\langle\tau\rangle$, que será usado pela instrução bloqueante (**wait**), deixando no topo da pilha o tipo τ.

$$[\![\mathbf{wait}\ e]\!]^{\Gamma,l'} \triangleq (A\ ;\ \overline{B},\overline{B'}\ ;\ \overline{M}\ ;\ \Phi\ ;\ \tau)$$

$$\text{where} \quad [\![e]\!]^{\Gamma,l} = (A\ ;\ \overline{B}\ ;\ \overline{M}\ ;\ \Phi\ ;\ \mathbf{th}\langle\tau\rangle) \quad l \text{ is fresh}$$

$$\overline{B'} = l{:}\Gamma \cdot \mathbf{th}\langle\tau\rangle \quad \mathbf{wait} \quad \mathbf{br}\ l'$$

A tradução da criação de um *lock* gera código máquina que coloca, no topo da pilha, a referência para o próprio. O conjunto de blocos, de métodos e de variáveis locais são vazios, e a expressão é do tipo **lock**.

$$[\![\mathbf{newlock}]\!]^{l} \triangleq (\langle\mathbf{newlock}\rangle,\langle\mathbf{br}\ l\rangle\ ;\ \varepsilon\ ;\ \varepsilon\ ;\ \varepsilon\ ;\ \mathbf{lock})$$

A tradução da expressão de acesso partilhado é análoga à tradução de criação de fios de execução, excepto que neste caso é necessário traduzir a expressão e' que deixa um *lock* no topo da pilha que, juntamente com os argumentos $\overline{x}$ (variáveis livres em e), são utilizados pela chamada de acesso partilhado (**shared** $\tau\ m(\overline{\tau})$). Por questões de simplicidade, e visto que em comparação com a tradução de acesso exclusivo entre fios de execução (**sync**$(e')\{e\}$) só difere na chamada ao método auxiliar (**sync** $\tau\ m(\overline{\tau})$), omitimos essa mesma tradução nesta secção.

$$[\![\textbf{shared}(e')\{e\}]\!]_{\Phi}^{\Gamma,l_f} \triangleq (A'' \ ; \ \overline{B''},B''' \ ; \ \overline{M},\overline{M'},\overline{M''} \ ; \ \Phi'' \ ; \ \tau)$$

where

$$\begin{array}{rcl}
\overline{x} & = & \text{freeVars(e)} \\
\Phi(x_i) & = & \tau_i \\
[\![e]\!]^{\varepsilon,l} & = & (A \ ; \ \overline{B} \ ; \ \overline{M} \ ; \ \Phi' \ ; \ \tau) \\
\overline{M'} & = & \textbf{method} \ \tau \ m(\overline{\tau \ x}) \ \{ \ \textbf{locals}(\overline{x \mapsto \tau}, \Phi') \ \overline{B'},\overline{B} \ \} \\
[\![e']\!]^{\Gamma,l''} & = & (A'' \ ; \ \overline{B''} \ ; \ \overline{M''} \ ; \ \Phi'' \ ; \ \textbf{lock}) \\
 & & m, l, l', l'' \text{ are fresh}
\end{array}$$

$$\begin{array}{rcl}
\overline{B'} & = & l'\text{:}\textbf{empty} \\
 & & \textbf{ldarg} \ x_1 \\
 & & \textbf{stloc} \ x_1 \\
 & & ... \\
 & & \textbf{ldarg} \ x_n \\
 & & \textbf{stloc} \ x_n \\
 & & A \\
 & & \\
 & & l\text{:}\tau \\
 & & \textbf{ret}
\end{array}$$

$$\begin{array}{rcl}
B''' & = & l''\text{:}\Gamma \cdot \textbf{lock} \\
 & & \textbf{ldarg} \ this \\
 & & \textbf{ldloc} \ x_1 \\
 & & ... \\
 & & \textbf{ldloc} \ x_n \\
 & & \textbf{shared} \ \tau \ m(\overline{\tau}) \\
 & & \textbf{br} \ l_f
\end{array}$$

7.3 Compilador

Foi implementado um compilador, em Ocaml, que gera código da linguagem de alto nível para código equivalente na linguagem intermédia. A arquitectura é composta pelo *parser*, pelo tradutor e pelo *unparser*. O *parser* lê o código em texto, carrega as classes, carrega os respectivos métodos e forma as Árvores de Sintaxe Abstracta das expressões que compõem os corpos dos métodos.

O tradutor trata de mapear as classes (e respectivos métodos) da linguagem fonte para classes na linguagem alvo, e traduzir as árvores de sintaxe abstracta para blocos de instruções (e para eventualmente métodos auxiliares) fazendo *pattern-matching* dos nós da árvore e tomando a acção apropriada, seguindo as equivalências definidas nas secções

anteriormente apresentadas.

Finalmente o *unparser* é um módulo que dado um programa, na linguagem intermédia e carregado na memória, converte o próprio para formato de texto.

8

Conclusões e Trabalho Futuro

O objectivo desta dissertação consistiu na definição de uma linguagem intermédia tipificada, baseada numa máquina de pilha com objectos, e com suporte nativo para concorrência, no modelo de memória partilhada. Definimos a respectiva semântica operacional e um sistema de tipos que permite verificar o código num passo apenas, garantindo a ausência de determinados erros mais usuais. Paralelamente também definimos uma linguagem de alto nível orientada a objectos e concorrente, também sobre o modelo de memória partilhada. Especificámos a respectiva semântica formal e sistema de tipos, assim como o processo de tradução desta própria linguagem para a linguagem intermédia. No decorrer da realização da dissertação foi publicado o artigo "Linguagem Intermédia Tipificada para Máquina de Pilha Concorrente com Objectos" [24] no INForum.

Dada a elevada abrangência deste trabalho, ficaram vários aspectos em aberto para trabalho futuro. Seria interessante provar formalmente que o sistema de tipos dá as garantias desejadas, tanto na linguagem intermédia como na linguagem fonte. Também seria interessante provar a preservação da semântica e de tipos, no processo de tradução entre as duas linguagens. Conseguiria-se garantir que a informação de tipos não se perdia entre o programa na linguagem fonte e o programa na linguagem de máquina. Os erros detectados pelo sistema de tipos da linguagem fonte também seriam detectados num nível de abstracção mais baixo.

A principal motivação para esta linguagem intermédia tipificada, orientada a objectos, com instruções, valores e tipos específicos para a concorrência, e com um algoritmo de tipificação de um passo apenas, consiste numa futura extensão ao sistema de tipos que analise estaticamente os acessos (aos objectos) entre os múltiplos fios de execução para, por exemplo, detectar *data races*. Introduzimos de seguida uma proposta para incorporar os tipos comportamentais no sistema de tipos definido nesta dissertação, permitindo

restringir o uso de objectos entre os múltiplos fios de execução.

8.1 Tipos Comportamentais

Os tipos comportamentais permitem exprimir a maneira como o uso de determinados recursos deve ser disciplinado, usando operadores de tipos para indicar a dependência entre operações sobre os recursos. No caso de uma linguagem de objectos, certos operadores, como a composição sequencial e paralela de tipos, indicam se há dependência entre métodos ou se é permitida a sua invocação por fios de execução diferentes. Por exemplo, no caso de um ficheiro, o protocolo de utilização poderia ser expresso por (open;(read)$*$;close) indicando que o método open deve ser chamado em primeiro lugar, e só depois é que é permitido chamar (várias vezes) o método read, podendo terminar chamando o método close, e depois disso não é possível chamar mais métodos. Como exemplo mais abstracto, considere-se o protocolo de um objecto com os métodos m_1, m_2 e m_3, combinados no protocolo $\{m_1 \ ; \ (m_2 \mid m_3)\}$. Segundo este protocolo, o primeiro método invocado tem que ser obrigatoriamente o m_1. Depois temos uma composição paralela dos métodos m_2 e m_3, o que quer dizer que podem ser chamados por qualquer ordem ou até mesmo de forma paralela.

O suporte nativo para concorrência da linguagem intermédia, assim como a separação da pilha de avaliação da pilha de chamada, e também a utilização de um modelo de objectos primitivo, torna possível exprimir os protocolos dos objectos. Assim, através de um sistema de tipos com efeitos é possível exprimir a evolução dos tipos dos objectos e verificar o cumprimento dos protocolos. Considere-se o seguinte exemplo bastante simples:

$l0 : \{ \ (m_1; m_2) \ \}$	w	$\{w \mapsto (m_1; m_2)\}$
dup	$w \cdot w$	$\{w \mapsto (m_1; m_2)\}$
fork $m_1()$	$w \cdot t_1$	$\{w \mapsto (\mathbf{th}\langle t_1\rangle; m_2)\}$
wait	w	$\{w \mapsto (m_2)\}$
call $m_2()$	ε	$\{w \mapsto \varepsilon\}$
br l		

Do lado esquerdo temos um bloco de instruções, associado a uma pilha com apenas um elemento do tipo $\{m_1;m_2\}$. No centro temos o conteúdo da pilha de tipos. Dado que um objecto pode ter múltiplas cópias da respectiva referência na pilha de avaliação (*aliasing*), introduzimos o *singleton type* (w) [37], para depois conseguirmos garantir que o protocolo sobre o mesmo objecto é respeitado. Do lado direito temos o protocolo associado ao w. Como se pode verificar no exemplo apresentado, na instrução de duplicação (**dup**) é apenas duplicado o *singleton type* (w). Na chamada concorrente do método (m_1), é necessário indicar no protocolo corrente que o método está a ser avaliado por um segundo

fio (t_1). De seguida (**wait**) actualiza-se o protocolo, indicando que o fio de execução terminou a chamada ao método, seguida da chamada ao segundo método (m_2) e respectiva actualização do protocolo (ε). Como acabámos de observar, um protocolo estritamente sequencial não obriga a que os métodos não possam ser chamados por novos fios de execução. Se entre as duas chamadas (**fork** $m_1()$ e **call** $m_2()$) não existisse a instrução bloqueante (**wait**), então o programa estaria mal tipificado, pois o protocolo já não era respeitado.

Considere-se agora um segundo exemplo, assim como a respectiva verificação dos protocolos:

```
l0: { (m1;(m2 | m3)) }
    dup
    dup
    call void m1()
    fork void m2()
    stloc 0
    call void m3()
    ldloc 0
    wait
    ret
```

Na tipificação de cada uma das instruções, utilizando *singleton types*, mantendo na pilha apenas um símbolo (w) identificando o objecto, e num ambiente à parte o protocolo corrente, é possível descrever as várias evoluções do tipo com a asserção $\Psi \vdash A : \Gamma * \beta \longrightarrow \Gamma * \beta$ em que β é um mapeamento dos símbolos (w) e os respectivos protocolos:

$$\vdash \textbf{dup} : w * \{w \mapsto \{m_1; (m_2 \mid m_3)\}\} \longrightarrow w \cdot w * \{w \mapsto \{m_1; (m_2 \mid m_3)\}\ \}$$
$$\vdash \textbf{dup} : w \cdot w * \{w \mapsto \{m_1; (m_2 \mid m_3)\}\} \longrightarrow w \cdot w \cdot w * \{w \mapsto \{m_1; (m_2 \mid m_3)\}\ \}$$
$$\vdash \textbf{call}\ m_1() : w \cdot w \cdot w * \{w \mapsto \{m_1; (m_2 \mid m_3)\}\} \longrightarrow w \cdot w * \{w \mapsto \{m_2 \mid m_3\}\}$$
$$\vdash \textbf{fork}\ m_2() : w \cdot w * \{w \mapsto \{m_2 \mid m_3\}\} \longrightarrow w \cdot t_1 * \{w \mapsto \{\textbf{th}\langle t_1\rangle \mid m_3\}\}$$
$$\vdash \textbf{stloc}\ 0 : w \cdot t_1 * \{w \mapsto \{\textbf{th}\langle t_1\rangle \mid m_3\}\} \longrightarrow w * \{w \mapsto \{\textbf{th}\langle t_1\rangle \mid m_3\}\}$$
$$\vdash \textbf{call}\ m_3() : w * \{w \mapsto \{\textbf{th}\langle t_1\rangle \mid m_3\}\} \longrightarrow \varepsilon * \{w \mapsto \{\textbf{th}\langle t_1\rangle\}\}$$
$$\vdash \textbf{ldloc}\ 0 : \varepsilon * \{w \mapsto \{\textbf{th}\langle t_1\rangle\}\} \longrightarrow t_1 * \{w \mapsto \{\textbf{th}\langle t_1\rangle\}\}$$
$$\vdash \textbf{wait} : t_1 * \{w \mapsto \{\textbf{th}\langle t_1\rangle\}\} \longrightarrow \varepsilon * \{w \mapsto \varepsilon\}$$
$$\vdash \textbf{ret} : \varepsilon * \{w \mapsto \varepsilon\} \longrightarrow \varepsilon * \{w \mapsto \varepsilon\}$$

Mais uma vez, tal como o 1º exemplo, o que é mantido no topo da pilha é um identificador (w) que permite que haja um tipo único para as várias ocorrências de um objecto. Uma chamada de método só é bem tipificada se puder ser efectuada tendo em conta um dado protocolo corrente.

As anotações de tipos apresentadas ao longo deste documento terão que sofrer algumas alterações. O tipo associado a cada etiqueta, para além do tipo de pilha também tem que ter o mapa dos protocolos (β). Na instrução de salto verifica-se, para cada posição da pilha actual, se o tipo de objecto, com um protocolo corrente associado, é subtipo daquele que está anotado na etiqueta alvo (da instrução de salto). Por exemplo, com uma etiqueta associada a um tipo de pilha com um único elemento com o protocolo (get;get) admite-se um salto, para a própria, com um tipo de pilha actual com um elemento não exactamente igual, como por exemplo ($get|get$). O caso contrário já não é válido. O facto de termos o código dos métodos dividido em blocos com tipos de pilha associados faz com que, mesmo com instruções de salto e objectos com relação de subtipo, não seja necessário efectuar análise de fluxos de dados (ao contrário da JVM e CLR).

A solução apresentada apenas está em fase de desenvolvimento, visto que existem casos mais complexos por analisar, nomeadamente chamadas a métodos que recebem objectos como parâmetro ou que retornam um objecto. O controlo/análise de múltiplas referências para um mesmo objecto (com um determinado protocolo corrente) torna-se mais complexo nestes casos.

Data Races

Para além da verificação do cumprimento dos protocolos, os tipos comportamentais podem ser utilizados como auxílio para detecção de *data races*. Por exemplo, sem entrar em detalhes concretos e formais, dado um protocolo $\{m_1|m_2\}$, se ambos os métodos apenas lerem os campos do respectivo objecto, considera-se que o protocolo está correcto e que não há interferência entre fios de execução. Caso contrário, o algoritmo de tipificação poderá detectar a presença de *data races*, se existirem chamadas convencionais (**call**) aos dois métodos em fios paralelos, ou seja, a correcta utilização iria envolver a utilização das chamadas de acesso partilhado/exclusivo que definimos nesta dissertação (**shared**,**sync**).

Destruição de Fios de Execução

Como foi devidamente mencionado na semântica operacional, a instrução **wait** remove um fio de execução do conjunto de fios (Σ), dado um identificador do próprio. No entanto, se dois fios de execução tiverem, nas suas variáveis locais ou na pilha de avaliação, um mesmo identificador para um terceiro fio, existe a possibilidade de um fio de execução invocar a operação **wait** sobre outro fio de execução já destruído.

Existem várias soluções para este problema. Uma delas seria deixar para o interpretador a verificação adicional, sempre que se tenta remover um fio de execução. No entanto, pretendemos nesta dissertação um sistema de tipos que detecte, em tempo de carregamento, eventuais erros presentes no código. Uma segunda solução seria alterar a semântica da instrução **wait**, que deixava de remover o fio de execução, ficando essa

responsabilidade associada a um *garbage collector*. Porém, iríamos obter perdas de desempenho sempre que esse mesmo mecanismo era accionado (automaticamente). Outra opção seria ter tipos lineares [42] para os fios de execução. Se apenas existir uma única referência para qualquer fio de execução, nas estruturas de dados que compõem o estado da máquina, conseguimos garantir que a operação **wait** é apenas invocada uma só vez para qualquer fio de execução. No entanto esta solução seria demasiado restritiva.

Com a futura extensão do sistema de tipos base para tipos comportamentais, adicionando o *singleton type* (w), para associar ao mesmo objecto o mesmo protocolo corrente, conseguimos resolver este problema da mesma forma. Cada tipo de um fio de execução é representado por um *singleton type* (w), que no início está associado ao tipo **thread**. Depois de uma operação de bloqueio (**wait**) actualiza-se o respectivo w, para um tipo indefinido ($?\tau$), detectando outras operações de bloqueio para o mesmo fio de execução.

Bibliografia

[1] The go programming language: Documentation. http://golang.org/doc/docs.html.

[2] Java virtual machine architecture. http://www.artima.com/insidejvm/ed2/jvm2.html.

[3] Java virtual machine threads. http://www.oracle.com/technetwork/java/threads-140302.html.

[4] Monitor types. http://www.cs.mtu.edu/~shene/NSF-3/e-Book/MONITOR/monitor-types.html.

[5] Mono project: Documentation. http://www.mono-project.com.

[6] Scala actors: A short tutorial. http://www.scala-lang.org/node/242.

[7] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[8] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[9] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26:769–804, September 2004.

[10] Luís Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theoretical Computer Science*, 402:120–141, July 2008.

[11] Luis Caires and João Costa Seco. Dynamic control of interference. *DI-FCT-UNL*.

[12] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. *Modula-3 Reference Manual*. DEC Systems Research Center.

[13] Edsger W. Dijkstra. *Cooperating sequential processes*, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[14] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 372–385, New York, NY, USA, 1996. ACM.

[15] John John Gough and K. John Gough. *Compiling for the .Net Common Language Runtime*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[16] Per Brinch Hansen. The programming language concurrent pascal. In *Language Hierarchies and Interfaces, International Summer School*, pages 82–110, London, UK, 1976. Springer-Verlag.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[18] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17:549–557, October 1974.

[19] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. 4 edition, June 2006.

[20] G Stewart Itzstein and David Kearney. Applications of join java. *Aust. Comput. Sci. Commun.*, 24:37–46, January 2002.

[21] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001.

[22] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[23] Luís Lopes, Fernando Silva, and Vasco T. Vasconcelos. A virtual machine for the TyCO process calculus. In *PPDP'99*, volume 1702, pages 244–260, September 1999.

[24] Luis Miguel Lourenço, João Costa Seco, and Francisco Martins. Linguagem intermédia tipificada para máquina de pilha concorrente com objectos. In *INForum*, 2011.

[25] Greg Morrisett. Typed assembly language. In Benjamin Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 141–175. MIT Press, 2005.

[26] Greg Morrisett, David Tarditi, Perry Cheng, Chris Stone, P. Cheng, Peter Lee, C. Stone, Robert Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *Proceedings of the Workshop on Compiler Support for Systems Software*, 1996.

[27] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21:527–568, May 1999.

[28] Gregory J. Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Proc. Snd Intl. Workshop on Types in Compilation (TIC)*, pages 28–52, 1998.

[29] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[30] Robert O'Callahan. A simple, comprehensive type system for java bytecode subroutines. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 70–78, New York, NY, USA, 1999. ACM.

[31] Martin Odersky. *The Scala Language Specification*. Programming Methods Laboratory, 2.7 edition.

[32] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[33] Benjamim C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In *PROOF, LANGUAGE AND INTERACTION: ESSAYS IN HONOUR OF ROBIN MILNER*. MIT Press, 1997.

[34] John H. Reppy. Cml: A higher concurrent language. *SIGPLAN Not.*, 26:293–305, May 1991.

[35] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[36] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[37] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pages 366–381, London, UK, 2000. Springer-Verlag.

[38] Raymie Stata and Martín Abadi. A type system for java bytecode subroutines. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 149–160, New York, NY, USA, 1998. ACM.

[39] Ross Tate, Juan Chen, and Chris Hawblitzel. Inferable object-oriented typed assembly language. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 424–435, New York, NY, USA, 2010. ACM.

[40] Vasco T. Vasconcelos and Rui Bastos. Core-TyCO, the language definition, version 0.1. DI/FCUL TR 98–3, Department of Informatics, Faculty of Sciences, University of Lisbon, March 1998.

[41] Vasco T. Vasconcelos and Francisco Martins. A multithreaded typed assembly language. In *Proceedings of TV06 - Multithreading in Hardware and Software: Formal Approaches to Design and Verification*, pages 133–141, 2006.

[42] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.

9

Listagens

9.1 Monitor.java

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class Monitor
{
    private ReentrantLock mon;
    private Condition canRead;
    private Condition canWrite;
    private int count;

    public Monitor()
    {
        mon = new ReentrantLock();
        canRead = mon.newCondition();
        canWrite = mon.newCondition();
        count = 0;
    }

    public void startShared() throws InterruptedException
    {
        mon.lock();

        while(count < 0)
            canRead.await();
```

```
        count++;
        canRead.signal();
        mon.unlock();
    }


    public void endShared()
    {
        mon.lock();
        count−−;

        if(count == 0)
            canWrite.signal();

        mon.unlock();
    }

    public void startSync() throws InterruptedException
    {
        mon.lock();

        while(count != 0)
            canWrite.await();

        count = −1;
        mon.unlock();
    }


    public void endSync()
    {
        mon.lock();
        count = 0;
        canWrite.signal();
        canRead.signal();
        mon.unlock();
    }
}
```

9.2 Linguagem de Programação Concorrente - Sintaxe

P	::=	$\overline{C}$	(Program)
C	::=	**class** c { $\overline{F}$ $\overline{M}$ }	(Class)
F	::=	τ f	(Field)
M	::=	τ $m(\overline{\tau\ x})$ { e }	(Method)

e	::=		(Expression)
	|	x	(Identifier)
	|	n	(Integers)
	|	$!e$	(Negation)
	|	op	(Binary Operations)
	|	e ; e	(Sequence)
	|	x := e	(Assign)
	|	**while** (e) $\{e\}$	(Loop)
	|	**if** e **then** e **else** e	(Decision)
	|	**var** $\overline{\tau\ x}$ **in** e	(Variables Declaration)
	|	**val** $\overline{\tau\ x\ =\ e}$ **in** e	(Constants Declaration)
	|	**new** c	(New Object)
	|	$e.m(\overline{e})$	(Method Call)
	|	**fork** e	(Fork)
	|	**wait** e	(Wait)
	|	**newlock**	(New Lock)
	|	**shared** (e) $\{e\}$	(Shared)
	|	**sync** (e) $\{e\}$	(Synchronized)

9.3 Linguagem de Programação Concorrente - Semântica Operacional

$$(\Sigma; \mathcal{M}) \longrightarrow (\Sigma'; \mathcal{M}')$$

v	::=		(Valores)
		n	(Número inteiro)
	$\mid$	t	(Identificador de fio de execução)
	$\mid$	$\mathtt{l}$	(Referência para Objecto)
	$\mid$	α	(Referência para Lock)

C	::=	$!C$	(Not)
	$\mid$	$C \ + \ e$	(Add Left)
	$\mid$	$n \ + \ C$	(Add Right)
	$\mid$	$C \ ; \ e$	(Seq Left)
	$\mid$	$v \ ; \ C$	(Seq Right)
	$\mid$	$x := C$	(Assign)
	$\mid$	**if** C **then** e **else** e'	(If)
	$\mid$	**val** $\tau \ x \ = \ C$ **in** e	(Val)
	$\mid$	$C.m(e')$	(Call-1)
	$\mid$	$\mathtt{l}.m(C')$	(Call-2)
	$\mid$	**wait** C	(Wait)
	$\mid$	**shared**$(C)\{e\}$	(Shared)
	$\mid$	**sync**$(C)\{e\}$	(Sync)
	$\mid$	C_α	(Protected)

$$\frac{(\Sigma, t\langle e\rangle; \mathcal{M}) \longrightarrow (\Sigma', t\langle e'\rangle; \mathcal{M}')}{(\Sigma, t\langle C[e]\rangle; \mathcal{M}) \longrightarrow (\Sigma', t\langle C[e']\rangle; \mathcal{M}')} \quad \text{(R-CONTEXT)}$$

$$\frac{\mathcal{M}(x) = v}{(\Sigma, t\langle x\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle v\rangle; \mathcal{M})} \quad (\text{R-ID} - 1)$$

$$\frac{\mathcal{M}(\mathtt{l}) = \textbf{object}(\_, F) \qquad F(x) = v}{(\Sigma, t\langle x^{\mathtt{l}}\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle v\rangle; \mathcal{M})} \quad (\text{R-ID} - 2)$$

$$\frac{n' = \textbf{not}\ n}{(\Sigma, t\langle !n\rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle n'\rangle; \mathcal{M})} \quad \text{(R-NOT)}$$

$$\frac{n'' = n + n'}{(\Sigma, t\langle n + n' \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle n'' \rangle; \mathcal{M})} \quad \text{(R-ADD)}$$

$$(\Sigma, t\langle v; v' \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle v' \rangle; \mathcal{M}) \quad \text{(R-SEQ)}$$

$$(\Sigma, t\langle x := v \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle v \rangle; \mathcal{M}[x \mapsto v]) \quad \text{(R-ASSIGN} - 1\text{)}$$

$$(\Sigma, t\langle (x := v)^{\mathtt{l}} \rangle; \mathcal{M}[\mathtt{l} \mapsto \textbf{object}(\_, F)]) \longrightarrow (\Sigma, t\langle v \rangle; \mathcal{M}[\mathtt{l} \mapsto \textbf{object}(\_, F[x \mapsto v])] \quad \text{(R-ASSIGN} - 2\text{)}$$

$$(\Sigma, t\langle \textbf{if } 0 \textbf{ then } e \textbf{ else } e' \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle e' \rangle; \mathcal{M}) \quad \text{(R-IF} - 1\text{)}$$

$$(\Sigma, t\langle \textbf{if } 1 \textbf{ then } e \textbf{ else } e' \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle e \rangle; \mathcal{M}) \quad \text{(R-IF} - 2\text{)}$$

$$(\Sigma, t\langle \textbf{while}(e)\{e'\} \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle \textbf{if } e \textbf{ then } e'; \textbf{while}(e)\{e'\} \textbf{ else } 0 \rangle; \mathcal{M}) \quad \text{(R-WHILE)}$$

$$\frac{x' \text{ is fresh}}{(\Sigma, t\langle \textbf{var } \tau\ x \textbf{ in } e \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle e[x'/x] \rangle; \mathcal{M})} \quad \text{(R-VAR)}$$

$$(\Sigma, t\langle \textbf{val } \tau\ x\ =\ v \textbf{ in } e \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle e[v/x] \rangle; \mathcal{M}) \quad \text{(R-VAL)}$$

$$(\Sigma, t\langle \textbf{new } c \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle \mathtt{l} \rangle; \mathcal{M}[\mathtt{l} \mapsto \textbf{object}(c, \varepsilon)]) \quad \text{(R-NEW)}$$

$$\frac{\mathcal{M}(\mathtt{l}) = \textbf{object}(c, \_) \qquad \mathcal{P}(c, m) = \tau'\ m(\tau\ x)\{e\}}{(\Sigma, t\langle \mathtt{l}.m(v)^{\mathtt{l}'} \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle e[v/x]^{\mathtt{l}} \rangle; \mathcal{M})} \quad \text{(R-CALL)}$$

$$(\Sigma, t\langle \textbf{fork } e \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle t' \rangle, t'\langle e \rangle; \mathcal{M}) \quad \text{(R-FORK)}$$

$$(\Sigma, t\langle \textbf{wait } t' \rangle, t'\langle v \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle v \rangle; \mathcal{M}) \quad \text{(R-WAIT)}$$

$$(\Sigma, t\langle \textbf{newlock} \rangle; \mathcal{M}) \longrightarrow (\Sigma, t\langle \alpha \rangle; \mathcal{M}[\alpha \mapsto 0]) \quad \text{(R-NEWLOCK)}$$

$$\frac{n \geq 0}{(\Sigma, t\langle \textbf{shared}(\alpha)\{e\} \rangle; \mathcal{M}[\alpha \mapsto n]) \longrightarrow (\Sigma, t\langle e_\alpha \rangle; \mathcal{M}[\alpha \mapsto n + 1])} \quad \text{(R-SHARED} - 1\text{)}$$

$$\frac{n > 0}{(\Sigma, t\langle v_\alpha \rangle; \mathcal{M}[\alpha \mapsto n]) \longrightarrow (\Sigma, t\langle v \rangle; \mathcal{M}[\alpha \mapsto n-1])} \qquad (\text{T-SHARED} - 2)$$

$$(\Sigma, t\langle \textbf{sync}(\alpha)\{e\} \rangle; \mathcal{M}[\alpha \mapsto 0]) \longrightarrow (\Sigma, t\langle e_\alpha \rangle; \mathcal{M}[\alpha \mapsto -1]) \qquad (\text{R-SYNC} - 1)$$

$$(\Sigma, t\langle v_\alpha \rangle; \mathcal{M}[\alpha \mapsto -1]) \longrightarrow (\Sigma, t\langle v \rangle; \mathcal{M}[\alpha \mapsto 0]) \qquad (\text{R-SYNC} - 2)$$

9.4 Linguagem de Programação Concorrente - Sistemas de Tipos

$$\tau ::= \textbf{int} \mid \textbf{var}\langle\tau\rangle \mid \textbf{lock} \mid \textbf{th}\langle\tau\rangle \mid c$$

$$\Psi ::= \overline{c \ : \ (\overline{f{:}\tau} \ ; \ \overline{m{:}\tau(\overline{x{:}\tau})})} \qquad \text{(Ambiente Global)}$$

$$\Phi ::= \overline{x{:}\tau} \qquad \text{(Ambiente Local)}$$

$$\frac{P = \overline{\textbf{class}\ c\ \{\ F\ M\ \}} \qquad \forall_i \qquad \overline{c \ : \ (\overline{f{:}\tau} \ ; \ \overline{m{:}\tau(\overline{x{:}\tau})})} \vdash \textbf{class}\ c_i\ \{\ F_i\ M_i\ \}}{\vdash P} \quad \text{(T-PROGRAM)}$$

$$\frac{\forall_{i,j} \qquad \Psi \vdash \sigma_j \qquad M_i = \tau_i\ m_i(\overline{\tau_i\ x_i})\{\ e_i\ \} \qquad \Psi, this{:}c \vdash \tau_i\ m_i(\overline{\tau_i\ x_i})\{\ e_i\ \} : \tau_i(\overline{\tau_i})}{\Psi \vdash \textbf{class}\ c\ \{\ \overline{\sigma\ f}\ M\ \}} \text{(T-CLASS)}$$

$$\frac{\Psi, \Phi \vdash_m e : \tau}{\Psi, \Phi \vdash \tau\ m(\overline{\tau\ x})\{\ e\ \} : \tau(\overline{\tau})} \qquad \text{(T-METHOD)}$$

$$\frac{\Phi(x) = \textbf{var}\langle\tau\rangle}{\Psi, \Phi \vdash x : \tau} \qquad \frac{\Phi(x) = \tau \qquad \tau \neq \textbf{var}\langle\_\rangle}{\Psi, \Phi \vdash x : \tau} \qquad (\text{T-ID}-1, \text{T-ID}-2)$$

$$\frac{\Phi(this) = c \qquad \Psi(c) = (\_\ ; \ \overline{M}, m{:}\_(..., x{:}\tau, ...), \overline{M'})}{\Psi, \Phi \vdash_m x : \tau} \qquad (\text{T-ID}-3)$$

$$\frac{\Phi(this) = c \qquad \Psi(c) = (\overline{F}, x{:}\tau, \overline{F'}\ ; \ \_)}{\Psi, \Phi \vdash x : \tau} \qquad (\text{T-ID}-4)$$

$$\Psi, \Phi \vdash n : \textbf{int} \qquad \text{(T-INT)}$$

$$\frac{\Psi, \Phi \vdash e : \textbf{int}}{\Psi, \Phi \vdash !e : \textbf{int}} \qquad \text{(T-NOT)}$$

$$\frac{\Psi, \Phi \vdash e : \textbf{int} \quad \Psi, \Phi \vdash e' : \textbf{int}}{\Psi, \Phi \vdash e{+}e' : \textbf{int}} \qquad \text{(T-ADD)}$$

$$\frac{\Psi, \Phi \vdash e : \tau \quad \Psi, \Phi \vdash e' : \tau'}{\Psi, \Phi \vdash e{;}e' : \tau'} \qquad \text{(T-SEQ)}$$

$$\frac{\Phi(x) = \textbf{var}\langle\tau\rangle \quad \Psi, \Phi \vdash e : \tau}{\Psi, \Phi \vdash x{:=}e : \tau} \qquad (\text{T-ASSIGN} - 1)$$

$$\frac{\Phi(\mathit{this}) = c \qquad \Psi(c) = (\overline{F}{,}x{:}\tau{,}\overline{F'}\ ;\ \_) \qquad \Psi, \Phi \vdash e : \tau}{\Psi, \Phi \vdash x{:=}e : \tau} \qquad (\text{T-ASSIGN} - 2)$$

$$\frac{\Psi, \Phi \vdash e : \textbf{int} \quad \Psi, \Phi \vdash e' : \tau}{\Psi, \Phi \vdash \textbf{while}(e)\{e'\} : \textbf{int}} \qquad (\text{T-WHILE})$$

$$\frac{\Psi, \Phi \vdash e : \textbf{int} \quad \Psi, \Phi \vdash e' : \tau \quad \Psi, \Phi \vdash e'' : \tau}{\Psi, \Phi \vdash \textbf{if}\ e\ \textbf{then}\ e'\ \textbf{else}\ e'' : \tau} \qquad (\text{T-IF})$$

$$\frac{\overline{x'} \text{ are fresh} \qquad \Phi' = \Phi, \overline{x'{:}\textbf{var}\langle\tau\rangle} \qquad \Psi, \Phi' \vdash e[x'/x] : \tau}{\Psi, \Phi \vdash \textbf{var}\ \overline{\tau\ x}\ \textbf{in}\ e : \tau} \qquad (\text{T-VAR})$$

$$\frac{\overline{x'} \text{ are fresh} \qquad \forall_i \quad \Psi, \Phi \vdash e_i : \tau_i \qquad \Phi' = \Phi, \overline{x'{:}\tau} \qquad \Psi, \Phi' \vdash e[\overline{x'/x}] : \tau}{\Psi, \Phi \vdash \textbf{val}\ \overline{\tau\ x\ =\ e}\ \textbf{in}\ e : \tau} \qquad (\text{T-VAL})$$

$$\Psi \vdash \textbf{new}\ c : c \qquad (\text{T-NEW})$$

$$\frac{\Psi, \Phi \vdash e : c \qquad \forall_i \quad \Psi, \Phi \vdash e_i : \tau_i \qquad \Psi(c) = (\_\ ;\ \overline{M}{,}m{:}\tau(\overline{x{:}\tau}){,}\overline{M'})}{\Psi, \Phi \vdash e.m(\overline{e}) : \tau} \qquad (\text{T-CALL})$$

$$\frac{\Psi, \Phi \vdash e : \tau}{\Psi, \Phi \vdash \textbf{fork}\ e : \textbf{th}\langle\tau\rangle} \qquad (\text{T-FORK})$$

$$\frac{\Psi, \Phi \vdash e : \textbf{th}\langle\tau\rangle}{\Psi, \Phi \vdash \textbf{wait}\ e : \tau} \qquad (\text{T-WAIT})$$

$$\Psi, \Phi \vdash \textbf{newlock} : \textbf{lock} \qquad (\text{T-NEWLOCK})$$

$$\frac{\Psi, \Phi \vdash e : \textbf{lock} \quad \Psi, \Phi \vdash e' : \tau}{\Psi, \Phi \vdash \textbf{shared}(e)\{e'\} : \tau} \qquad (\text{T-SHARED})$$

$$\frac{\Psi, \Phi \vdash e : \textbf{lock} \quad \Psi, \Phi \vdash e' : \tau}{\Psi, \Phi \vdash \textbf{sync}(e)\{e'\} : \tau} \qquad (\text{T-SYNC})$$

9.5 Linguagem Intermédia Tipificada - Sintaxe

P	::=	$\overline{C}$	(Programa)
C	::=	**class** c { $\overline{F}$ $\overline{M}$ }	(Classe)
F	::=	**field** τ f	(Variável de instância)
M	::=	**method** τ $m(\overline{\tau\ x})$ { **locals**$(\overline{\tau\ x})$ $\overline{B}$ }	(Método)
B	::=	$l{:}\Gamma \mapsto A$	(Bloco)
A	::=	**ret** | **br** l | $\iota \cdot A$	(Sequência de instruções)
ι	::=		(Instruções)
	|	op	
	|	**pop**	
	|	**dup**	
	|	**not**	
	|	**ldc** n	
	|	**brfalse** l	
	|	**ldarg** x	
	|	**ldloc** x	
	|	**stloc** x	
	|	**call** τ $m(\overline{\tau})$	
	|	**newobj** c	
	|	**ldfld** f	
	|	**stfld** f	
	|	**fork** τ $m(\overline{\tau})$	
	|	**wait**	
	|	**newlock**	
	|	**shared** τ $m(\overline{\tau})$	
	|	**sync** τ $m(\overline{\tau})$	

9.6 Linguagem Intermédia Tipificada - Semântica Operacional

$$\{\Sigma; H\}$$

$$
\begin{array}{lcll}
\Sigma & ::= & \overline{(t, \rho)} & \text{(Fios de execução)} \\
\rho & ::= & \overline{Fr} & \text{(Pilha de chamada)} \\
Fr & ::= & \langle \overline{x \mapsto v}, \overline{x \mapsto v}, s, A \rangle^{m}_{\alpha} & \text{(Registo de activação)} \\
s & ::= & \varepsilon \mid s \cdot v & \text{(Pilha de avaliação)} \\
v & ::= & & \text{(Valores)} \\
 & & n & \text{(Número inteiro)} \\
 & \mid & t & \text{(Identificador de fio de execução)} \\
 & \mid & \mathtt{l} & \text{(Referência para Objecto)} \\
 & \mid & \alpha & \text{(Referência para Lock)} \\
obj & ::= & \textbf{object}(c, \overline{f \mapsto v}) & \text{(Objecto)} \\
H & ::= & \overline{\mathtt{l} \mapsto obj}, \ \overline{\alpha \mapsto n} & \text{(Heap)}
\end{array}
$$

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, \textbf{pop} \cdot A \rangle^{m}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A \rangle^{m}); H\} \qquad \text{(R-POP)}$$

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, \textbf{dup} \cdot A \rangle^{m}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v \cdot v, A \rangle^{m}); H\} \qquad \text{(R-DUP)}$$

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{ldc}\ n \cdot A \rangle^{m}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n, A \rangle^{m}); H\} \qquad \text{(R-LDC)}$$

$$\frac{n' \ = \ \textbf{not}\ n}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n, \textbf{not} \cdot A \rangle^{m}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n', A \rangle^{m}); H\}} \qquad \text{(R-NOT)}$$

$$\frac{n = n' + n''}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n' \cdot n'', \ \textbf{add} \cdot A \rangle^{m}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n, A \rangle^{m}); H\}} \qquad \text{(R-ADD)}$$

$$\frac{a(this) = \mathtt{l} \quad H(\mathtt{l}) = \textbf{object}(c, \_) \quad A' = \mathcal{P}(c, m, l)}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{br}\ l \rangle^{m}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A' \rangle^{m}); H\}} \qquad \text{(R-BR)}$$

$$\frac{n \neq 0}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot n, \textbf{brfalse}\ l \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^m); H\}}\ (\text{R-BRFALSE} - 1)$$

$$\frac{a(this) = \texttt{l} \quad H(\texttt{l}) = \textbf{object}(c, \_) \quad A' = \mathcal{P}(c, m, l)}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot 0, \textbf{brfalse}\ l \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A'\rangle^m); H\}}\ (\text{R-BRFALSE} - 2)$$

$$\frac{a(x) = v}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{ldarg}\ x \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^m); H\}}\ (\text{R-LDARG})$$

$$\frac{lv(x) = v}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{ldloc}\ x \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^m); H\}}\ (\text{R-LDLOC})$$

$$\{\Sigma, (t, \rho \cdot \langle a, lv[x \mapsto v], s \cdot v', \textbf{stloc}\ x \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv[x \mapsto v'], s, A\rangle^m); H\}\ (\text{R-STLOC})$$

$$\frac{\texttt{l}\ is\ fresh \quad \mathcal{F}(c) = \overline{fv}}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{newobj}\ c \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \texttt{l}, A\rangle^m); H[\texttt{l} \mapsto \textbf{object}(c, \overline{fv})]\}}\ (\text{R-NEWOBJ})$$

$$\frac{H(\texttt{l}) = \textbf{object}(c, fv) \quad fv(f) = v}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \texttt{l}, \textbf{ldfld}\ f \cdot A\rangle^m); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^m); H\}}\ (\text{R-LDFLD})$$

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \texttt{l} \cdot v, \textbf{stfld}\ f \cdot A\rangle^m); H[\texttt{l} \mapsto \textbf{object}(c, fv)]\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^m); H[\texttt{l} \mapsto \textbf{object}(c, fv[f \mapsto v])]\}\ (\text{R-STFLD})$$

$$\frac{A' = \mathcal{P}(c, m) \quad \mathcal{M}(c, m) = \tau\ m(\overline{\tau\ x}) \quad lv' = \mathcal{V}(c, m) \quad H(\texttt{l}) = \textbf{object}(c, \overline{fv})}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \texttt{l} \cdot \overline{v}, \textbf{call}\ \tau\ m(\overline{\tau}) \cdot A\rangle^{m'}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^{m'} \cdot \langle (this \mapsto \texttt{l}, \overline{x \mapsto v}), lv', \varepsilon, A'\rangle^m); H\}}\ (\text{R-CALL})$$

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^m \cdot \langle a', lv', v, \textbf{ret}\rangle^{m'}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^m); H\}\ (\text{R-RET} - 1)$$

$$\frac{A' = \mathcal{P}(c,m) \quad \mathcal{M}(c,m) = \tau\ m(\overline{\tau\ x}) \quad lv' = \mathcal{V}(c,m) \quad H(\mathtt{l}) = \textbf{object}(c,\overline{fv}) \quad t' \text{ is fresh}}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \mathtt{l} \cdot \overline{v}, \textbf{fork}\ \tau\ m(\overline{\tau}) \cdot A\rangle^{m'}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot t', A\rangle^{m'}), (t', \langle (this \mapsto \mathtt{l}, \overline{x \mapsto v}), lv', \varepsilon, A'\rangle^{m}); H\}} \text{(R-FORK)}$$

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot t', \textbf{wait} \cdot A\rangle^{m}), (t', \langle a', lv', v, \textbf{ret}\rangle^{m'}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^{m}); H\} \quad \text{(R-WAIT)}$$

$$\frac{\alpha \text{ is fresh}}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, \textbf{newlock} \cdot A\rangle^{m}); H\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \alpha, A\rangle^{m}); H[\alpha \mapsto 0]\}} \text{(R-NEWLOCK)}$$

$$\frac{A' = \mathcal{P}(c,m) \quad \mathcal{M}(c,m) = \tau\ m(\overline{\tau\ x}) \quad lv' = \mathcal{V}(c,m) \quad H(\mathtt{l}) = \textbf{object}(c,\overline{fv}) \quad n \geq 0}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \alpha \cdot \mathtt{l} \cdot \overline{v}, \textbf{shared}\ \tau\ m(\overline{\tau}) \cdot A\rangle^{m'}); H[\alpha \mapsto n]\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^{m'} \cdot \langle (this \mapsto \mathtt{l}, \overline{x \mapsto v}), lv', \varepsilon, A'\rangle^{m}_{\alpha}); H[\alpha \mapsto n{+}1]\}} \text{(R-SHARED)}$$

$$\frac{A' = \mathcal{P}(c,m) \quad \mathcal{M}(c,m) = \tau\ m(\overline{\tau\ x}) \quad lv' = \mathcal{V}(c,m) \quad H(\mathtt{l}) = \textbf{object}(c,\overline{fv})}{\{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot \alpha \cdot \mathtt{l} \cdot \overline{v}, \textbf{sync}\ \tau\ m(\overline{\tau}) \cdot A\rangle^{m'}); H[\alpha \mapsto 0]\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^{m'} \cdot \langle (this \mapsto \mathtt{l}, \overline{x \mapsto v}), lv', \varepsilon, A'\rangle^{m}_{\alpha}); H[\alpha \mapsto -1]\}} \text{(R-SYNC)}$$

$$\frac{n > 0}{\{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^{m} \cdot \langle a', lv', v, \textbf{ret}\rangle^{m'}_{\alpha}); H[\alpha \mapsto n]\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^{m}); H[\alpha \mapsto n{-}1]\}} \quad \text{(R-RET} - 2)$$

$$\{\Sigma, (t, \rho \cdot \langle a, lv, s, A\rangle^{m} \cdot \langle a', lv', v, \textbf{ret}\rangle^{m'}_{\alpha}); H[\alpha \mapsto -1]\} \longrightarrow \{\Sigma, (t, \rho \cdot \langle a, lv, s \cdot v, A\rangle^{m}); H[\alpha \mapsto 0]\} \quad \text{(R-RET} - 3)$$

9.7 Linguagem Intermédia Tipificada - Sistema de Tipos

$$
\begin{array}{rcll}
\tau & ::= & \textbf{int} \mid \textbf{lock} \mid \textbf{th}\langle\tau\rangle \mid c & \text{(Tipos)} \\
\Psi & ::= & \overline{c \ : \ (\ \overline{f{:}\tau}\ ,\ \overline{m{:}\tau(\overline{x{:}\tau})}\ )} & \text{(Ambiente Global)} \\
\Phi & ::= & \overline{l{:}\Gamma},\ \overline{x{:}\tau} & \text{(Ambiente Local)} \\
\Gamma & ::= & \varepsilon \ \mid \ \Gamma \cdot \tau & \text{(Pilha de Tipos)}
\end{array}
$$

$$
\frac{\Psi \vdash P \qquad \Psi \vdash \Sigma \qquad \Psi \vdash H}{\Psi \vdash \{P;\Sigma;H\}} \qquad \text{(T-STATE)}
$$

$$
\frac{P = \overline{\textbf{class}\ c\ \{\ F\ M\ \}} \qquad \forall_i \quad \Psi,\ \overline{c : (\overline{f{:}\tau}\ ,\ \overline{m{:}\tau(\overline{x{:}\tau})})} \vdash \textbf{class}\ c_i\ \{\ F_i\ M_i\ \}}{\Psi \vdash P} \qquad \text{(T-PROGRAM)}
$$

$$
\frac{\begin{array}{c}\Psi \vdash c \ : \ (\ \overline{f{:}\tau}\ ,\ \overline{m{:}\tau(\overline{x{:}\tau})}\ ) \qquad \forall_{i,j} \quad \Psi \vdash \sigma_j \\ M_i = \textbf{method}\ \tau_i\ m_i(\overline{\tau_i\ x_i})\ \{\_\ \_\} \quad \Psi, this{:}c \vdash\ M_i{:}\ \tau_i(\overline{\tau_i})\end{array}}{\Psi \vdash \textbf{class}\ c\ \{\ \overline{\textbf{field}\ \sigma\ f}\quad \overline{M}\ \}} \qquad \text{(T-CLASS)}
$$

$$
\frac{\Phi' = \Phi, \overline{l{:}\Gamma}, \overline{x'{:}\tau'} \qquad \forall_l \quad \Psi, \Phi' \vdash_\tau A_l : \Gamma_l \longrightarrow \Gamma'}{\Psi, \Phi \vdash \textbf{method}\ \tau\ m(\overline{\tau\ x})\ \{\ \textbf{locals}(\overline{\tau'\ x'})\ \ \overline{l{:}\Gamma \mapsto A}\ \}\ :\ \tau(\overline{\tau})} \qquad \text{(T-METHOD)}
$$

$$
\frac{\Psi, \Phi \vdash_\tau \iota : \Gamma \longrightarrow \Gamma'' \qquad \Psi, \Phi \vdash_\tau A : \Gamma'' \longrightarrow \Gamma'}{\Psi, \Phi \vdash_\tau \iota \cdot A : \Gamma \longrightarrow \Gamma'} \qquad \text{(T-INST)}
$$

$$
\Psi \vdash \textbf{int} \qquad \Psi \vdash \textbf{lock} \qquad \text{(T-INT, T-LOCK)}
$$

$$
\frac{\Psi \vdash \tau}{\Psi \vdash \textbf{th}\langle\tau\rangle} \qquad \frac{c \in \mathrm{dom}(\Psi)}{\Psi \vdash c} \qquad \text{(T-THREAD, T-CLASSV)}
$$

$$
\Psi, \Phi \vdash \textbf{pop}{:}\ \Gamma \cdot \tau \longrightarrow \Gamma \qquad \Psi, \Phi \vdash \textbf{ldc}\ n{:}\ \Gamma \longrightarrow \Gamma \cdot \textbf{int} \qquad \text{(T-POP, T-LDC)}
$$

$$
\Psi, \Phi \vdash \textbf{dup}{:}\ \Gamma \cdot \tau \longrightarrow \Gamma \cdot \tau \cdot \tau \qquad \text{(T-DUP)}
$$

$$
\Psi, \Phi \vdash \textbf{not}{:}\ \Gamma \cdot \textbf{int} \longrightarrow \Gamma \cdot \textbf{int} \qquad \Psi, \Phi \vdash \textbf{add}{:}\ \Gamma \cdot \textbf{int} \cdot \textbf{int} \longrightarrow \Gamma \cdot \textbf{int} \qquad \text{(T-NOT, T-ADD)}
$$

$$\frac{\Phi(l) = \Gamma}{\Psi, \Phi \vdash \textbf{br}\ l : \Gamma \longrightarrow \Gamma} \qquad \frac{\Phi(l) = \Gamma}{\Psi, \Phi \vdash \textbf{brfalse}\ l : \Gamma \cdot \textbf{int} \longrightarrow \Gamma} \qquad \text{(T-BR, T-BRFALSE)}$$

$$\frac{\Psi(c) = (\_\,;\, \overline{M}, m{:}\_(\ldots, x{:}\tau, \ldots), \overline{M'})}{\Psi, \Phi \vdash^{c,m} \textbf{ldarg}\ x : \Gamma \longrightarrow \Gamma \cdot \tau} \qquad \text{(T-LDARG)}$$

$$\frac{\Phi(x) = \tau}{\Psi, \Phi \vdash \textbf{ldloc}\ x : \Gamma \longrightarrow \Gamma \cdot \tau} \qquad \frac{\Phi(x) = \tau}{\Psi, \Phi \vdash \textbf{stloc}\ x : \Gamma \cdot \tau \longrightarrow \Gamma} \qquad \text{(T-LDLOC, T-STLOC)}$$

$$\frac{\Psi \vdash c}{\Psi, \Phi \vdash \textbf{newobj}\ c : \Gamma \longrightarrow \Gamma \cdot c} \qquad \text{(T-NEWOBJ)}$$

$$\frac{\Psi(c) = (F, \_) \quad F(f) = \tau}{\Psi, \Phi \vdash \textbf{ldfld}\ f : \Gamma \cdot c \longrightarrow \Gamma \cdot \tau} \qquad \frac{\Psi(c) = (F, \_) \quad F(f) = \tau}{\Psi, \Phi \vdash \textbf{stfld}\ f : \Gamma \cdot c \cdot \tau \longrightarrow \Gamma} \qquad \text{(T-LDFLD, T-STFLD)}$$

$$\frac{\Psi(c) = (\_, M) \quad M(m) = \tau(\overline{\tau})}{\Psi, \Phi \vdash \textbf{call}\ \tau\ m(\overline{\tau}) : \Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau} \qquad \Psi, \Phi \vdash_{\tau} \textbf{ret} : \tau \longrightarrow \tau \qquad \text{(T-CALL, T-RET)}$$

$$\frac{\Psi, \Phi \vdash \textbf{call}\ \tau\ m(\overline{\tau}) : \Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau}{\Psi, \Phi \vdash \textbf{fork}\ \tau\ m(\overline{\tau}) : \Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \textbf{th}\langle\tau\rangle} \qquad \text{(T-FORK)}$$

$$\Psi, \Phi \vdash \textbf{wait} : \Gamma \cdot \textbf{th}\langle\tau\rangle \longrightarrow \Gamma \cdot \tau \qquad \text{(T-WAIT)}$$

$$\Psi, \Phi \vdash \textbf{newlock} : \Gamma \longrightarrow \Gamma \cdot \textbf{lock} \qquad \text{(T-NEWLOCK)}$$

$$\frac{\Psi, \Phi \vdash \textbf{call}\ \tau\ m(\overline{\tau}) : \Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau}{\Psi, \Phi \vdash \textbf{shared}\ \tau\ m(\overline{\tau}) : \Gamma \cdot \textbf{lock} \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau} \qquad \text{(T-SHARED)}$$

$$\frac{\Psi, \Phi \vdash \textbf{call}\ \tau\ m(\overline{\tau}) : \Gamma \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau}{\Psi, \Phi \vdash \textbf{sync}\ \tau\ m(\overline{\tau}) : \Gamma \cdot \textbf{lock} \cdot c \cdot \overline{\tau} \longrightarrow \Gamma \cdot \tau} \qquad \text{(T-SYNC)}$$