Paralelização de Código Puro numa Linguagem Imperativa

Diogo G. Sousa, João C. Martins e João Costa Seco

CITI — Departamento de Informática,
Universidade Nova de Lisboa, Portugal
{dm.sousa,jmc.martins}@campus.fct.unl.pt jrcs@fct.unl.pt

Resumo A programação concorrente é cada vez mais importante no aproveitamento dos recursos oferecidos pelos sistemas multiprocessador, que são agora ubíquos. No desenvolvimento de *software*, apesar das linguagens imperativas serem as mais utilizadas, não fornecem suporte real à programação concorrente. O programador necessita de prever as dependências entre dados e prever os efeitos no estado global para os proteger de maneira coerente de interferências indesejadas entre *threads*. É exigido ao programador o controlo explícito dos vários *threads* no acesso a zonas de memória partilhada no sentido de evitar erros de execução como *race conditions* e *deadlocks*. É sabido que as linguagens funcionais puras facilitam a escrita de código concorrente pois a pureza funcional implica a ausência de estado partilhado e de efeitos laterais. Procuramos neste artigo aplicar a noção de pureza funcional a uma linguagem imperativa concorrente de modo a garantir, por construção, a ausência de interferências entre *threads*.

Keywords: Concorrência, Pureza Funcional, Linguagem Imperativa, Verificação Estática, Sistema de Tipos

1 Introdução

Actualmente, os sistemas multiprocessador são ubíquos, o que estimula o desenvolvimento de programas que executam concorrentemente e assim aumentar a sua eficiência. Contudo, a programação concorrente com linguagens imperativas é propensa a erros. O programador tem de ter em conta os vários entrelaçamentos possíveis e a combinatória de interações possíveis entre os vários *threads*. Essa imprevisibilidade pode levar aos problemas conhecidos, como *race conditions* e *deadlocks*, que são não só difíceis de identificar como de reproduzir e corrigir. Estes problemas comprometem frequentemente a escalabilidade de sistemas de média e grande dimensão que tiram partido de sistemas multiprocessador. Os problemas acentuam-se quando se desenha um sistema segundo um paradigma sequencial e posteriormente se adapta para um paradigma concorrente.

A noção de pureza funcional é comum em linguagens funcionais. A linguagem *Haskell* tira partido disto para permitir a fácil paralelização de código [5]. No

entanto esta noção é raramente empregue explicitamente em linguagem imperativas, e não é aplicada à paralelização de código.

Os problemas na utilização de linguagens imperativas, normalmente advêm do controlo no acesso a zonas de memória partilhada ou de sincronização entre *threads*. Algumas abordagens baseadas em linguagens de programação, desde as abordagens seminais como [10], até outras mais recentes baseadas em tipos [1,3], disciplinam o *aliasing* de nomes para evitar as interferências. É um facto aceite que a programação paralela na ausência de partilha de estado evita este tipo de problemas por construção [7]. A conciliação entre a utilização de linguagens imperativas e ausência de partilha de estado é o objectivo deste artigo.

Apresentamos a linguagem imperativa *Impure*, onde o conceito de funções puras é central, e cujo sistema de tipos garante estaticamente que a execução paralela de código não coloca em causa a semântica esperada do programa. A ausência de problemas relacionados com a concorrência baseia-se na separação forçada pelo sistema de tipos entre código puro e impuro, sendo que só código puro pode ser executado em contexto paralelo.

Começamos por definir em maior detalhe a noção de código puro na secção 2. De seguida, apresentamos, na secção 3, um exemplos de programas que beneficiam da paralelização de algumas operações. A linguagem é apresentada através da sua sintaxe formal, na Secção 4, da semântica operacional, na Secção 5, e respectivo sistema de tipos, na Secção 6. Finalmente, apresentamos algum trabalho relacionado e delineamos as possíveis evoluções deste trabalho.

2 Pureza Funcional

A definição de pureza funcional não é consensual na comunidade científica [11]. No contexto deste artigo uma função é considerada pura se não provocar nenhum efeito lateral e se for determinista, i.e. se para um determinado conjunto de argumentos a função denota sempre o mesmo valor. Entende-se por efeito lateral qualquer mudança no estado não-local do programa, como por exemplo escrita em variáveis globais, *input/output*, etc. Decorre desta definição que funções puras não causam interferências com a execução de outro código, sendo a sua execução segura num contexto concorrente.

A utilização de funções puras apresenta ainda outras vantagens no desenvolvimento de *software* como a sua reutilização de forma independente do estado global.

Na linguagem *Impure*, as computações que provocam efeitos laterais ou que efectuam operações de *input/output*, são isoladas numa construção própria, colocada semanticamente ao nível dos valores da linguagem, a que chamamos *blocos impuros*. Estes blocos encerram e suspendem a computação impura que só poderá ser executada num contexto em que se esperam efeitos laterais. A execução destes blocos é sempre explicita. À imagem da linguagem *Haskell* [5], na linguagem *Impure*, todas as funções são puras, sendo que uma função pode conter código impuro se este for encerrado num bloco impuro, podendo ser devolvido como resultado. Blocos impuros podem ser compostos para representar compu-

```
Int fib(Int n) {
    Int z = 0;

    if n == 0 then z = 0;
    else if n == 1 then z = 1;
    else {
        Int a = 0;
        Int b = 0;
        a, b = fib(n − 1), fib(n − 2);
        z = a + b;
    }

    return z;
}
```

Listagem 1: Função Fibonacci na linguagem *Impure*.

tações maiores. O programa principal é representado por um único bloco impuro que encerra toda a computação do programa. Esta separação explícita entre código puro e impuro, bem como a composição do mesmo e a sua utilização em contexto de paralelização e sequenciação, é garantida pelo sistema de tipos.

Em favor da nossa abordagem, pode-se pensar que restrigir execuções concorrentes a contextos puros pode ser limitativa, no entanto, a maioria dos procedimentos que são candidatos a ser paralelizados são puros.

3 Exemplos

Consideremos então o pequeno exemplo da Listagem 1 na linguagem *Impure*, que lê um inteiro n, calcula o n-ésimo número de Fibonacci e imprime-o na consola de sistema. É de notar neste exemplo a utilização do operador de afectação paralela $x_1, \cdots, x_n = e_1, \cdots, e_n$, que lança n *threads* para calcular as n expressões $e_1, \cdots, e_n$, e obter os seus resultados nas variáveis correspondentes $x_1, \cdots, x_n$. Neste caso, as chamadas recursivas na função de Fibonacci são feitas em paralelo através do comando de afectação paralela.

A Listagem 2 apresenta um programa completo que lê dez inteiros, ordena-os usando uma implementação paralela do algoritmo *quicksort* e imprime-os ordenados.

A função quicksort$(\cdot)$ recebe uma lista e retorna uma cópia dessa lista ordenada com o algoritmo *quicksort*. As chamadas recursivas são executadas em paralelo. A função readInts$(\cdot)$, que seria impura numa linguagem imperativa convencional pois faz leitura do *input*, apenas cria e devolve um valor que contem uma computação impura. Da mesma maneira a função main() denota toda a computação impura que constitui o programa. Dentro dessa computação impura vemos que a expressão **run**$(\cdot)$ é usada para executar a computação que foi

```
[Int] quicksort([Int] list) {
    [Int] sorted = [];
    if length(list) > 0 then {
        [Int] front = [];
        [Int] rear = [];
        Int pivot = 0;
        Int i = 1;
        pivot = list[0];
        while i < length(list) do {
            if list[i] < pivot then
                front = [list[i]] ++ front;
            else
                rear = [list[i]] ++ rear;
            i = i + 1;
        }
        front, rear = quicksort(front),
                      quicksort(rear);
        sorted = front ++ [pivot] ++ rear;
    }
    return sorted;
}
```

```
(Impure [Int]) readInts(Int n) {
    return impure {
        [Int] list = [];
        Int i = 0;
        while i < n do {
            list = list ++ [read()];
            i = i + 1;
        }
        return list;
    };
}

(Impure Void) main() {
    return impure {
        [Int] list = run(readInts(10));
        [Int] sorted = [];
        Int i = 0;
        sorted=quicksort(list);
        while i < 10 do {
            print sorted[i];
            i = i + 1;
        }
        return nil;
    };
}
```

Listagem 2: Programa com implementação paralela do algoritmo *quicksort*.

suspensa no bloco impuro devolvido pela função readInts(·), e assim compô-la com a computação impura da função main(), i.e., a chamada a readInts(·) não é impura mas a chamada de um **run**(·) é. Podemos considerar que a execução de um programa corresponde à avaliação da expressão **run**(main()).

4 Sintaxe

A sintaxe da linguagem *Impure* é dada pela gramática apresentada na Figura 1. Convencionamos que os nomes x, y, z representam variáveis, e f, g, h representam nomes de funções. Convencionamos ainda que $\overline{\tau\ x}$ representa uma sequência $\tau_1\ x_1,\ \ \tau_2\ x_2,\ \ \cdots\ \ \tau_n\ x_n$ e que *op* representa as operações binárias usuais.

Um programa é uma sequência de definições de funções. Note-se que sendo uma linguagem pura não é possível declarar variáveis no escopo global. Esta linguagem oferece como valores primitivos inteiros, booleanos, listas, funções, computações impuras e *nil*. As listas são valores imutáveis e podem ser acedidas por índice (e.g. $l[i]$) ou concatenadas com o operador ++. Também é possível consultar o tamanho da lista com a função length(·). O bloco impuro, **impure** ·

$$
\begin{array}{lll}
\tau & ::= \textsf{Void} \mid \textsf{Int} \mid \textsf{Bool} \mid [\tau] \mid \textsf{Impure}\ \tau \mid \tau \to \tau & \text{(Tipos)} \\
P & ::= F \mid F\ P & \text{(Programas)} \\
F & ::= \tau\ f(\tau\ x)\ E & \text{(Declaração de Funções)} \\
S & ::= \textsf{skip} & \text{(Comando vazio)} \\
 & \mid S\,;\,S & \text{(Sequências)} \\
 & \mid \textsf{while}\ E\ \textsf{do}\ S & \text{(Ciclos)} \\
 & \mid \textsf{if}\ E\ \textsf{then}\ S\ \textsf{else}\ S & \text{(Condicionais)} \\
 & \mid x_1, \cdots, x_n = E, \cdots, E & \text{(Afectações Paralelas)} \\
 & \mid \textsf{print}\ E & (\textit{Output}) \\
E & ::= num \mid bool \mid \textsf{nil} & \text{(Literais)} \\
 & \mid x & \text{(Variáveis)} \\
 & \mid \lambda(\tau\ x)\ E & \text{(Abstrações)} \\
 & \mid E(E) & \text{(Aplicações)} \\
 & \mid E\ op\ E & \text{(Operadores Binários)} \\
 & \mid [E, \cdots, E] \mid E +\!\!+ E \mid E[E] \mid \textsf{length}(E) & \text{(Operadores de Listas)} \\
 & \mid \textsf{read}() & (\textit{Input}) \\
 & \mid \textsf{impure}\ E & \text{(Blocos Impuros)} \\
 & \mid \textsf{run}\ E & \text{(Execuções Impuras)} \\
 & \mid S\ \textsf{return}\ E & \text{(Comandos)} \\
 & \mid \{\overline{\tau\ x = E}\ E\} & \text{(Declarações)}
\end{array}
$$

Figura 1: Gramática da linguagem *Impure*.

representa um valor que suspende uma computação impura que pode ser executada com a função **run**$(\cdot)$ num contexto impuro.

5 Semântica Operacional

Definimos nesta secção a semântica operacional para a linguagem *Impure* através de duas relações de redução, uma para expressões e outra para os comandos da linguagem. A relação representada por $\langle E,\, s\rangle \longrightarrow \langle E', s'\rangle$ reduz a expressão E para a expressão E' em relação ao estado s e com o efeito no estado s'. A relação representada por $\langle S,\, s\rangle \longrightarrow \langle S', s'\rangle$ representa o efeito da redução do comando S para o comando S' em relação ao estado s e com o efeito no estado s'.

Considera-se ainda a divisão sintática usual para valores na linguagem.

$$v ::= x \mid num \mid bool \mid \textsf{nil} \mid \lambda_s(\tau\ x)\ E \mid [v_1, \cdots, v_n] \mid \textsf{impure}_s\ E$$

Definimos o estado s como uma pilha de mapeamentos entre identificadores e valores da linguagem, a *stream* de entrada **in**, a *stream* de saída **out** são atributos do estado. Definimos a notação $s(x)$ para representar o valor da primeira ocorrência da variável x no estado s, $s.\textsf{in}$ e $s.\textsf{out}$ para representar as *streams* de entrada e saída. Estas *streams* são listas de inteiros. Assume-se que a lista **in** tem comprimento arbitrariamente grande, i.e., tem quantos inteiros forem precisos para a execução do programa. Definimos as operações $s \cdot s'$ e $s[x \leftarrow E]$.

$$\frac{\langle E,\, s\rangle \longrightarrow \langle E',\, s'\rangle}{\langle \mathcal{E}[E],\, s\rangle \longrightarrow \langle \mathcal{E}[E'],\, s'\rangle} \quad \text{(R-Expressão)}$$

$$\langle x,\, s\rangle \longrightarrow \langle s(x),\, s\rangle \quad \text{(R-Identificador)}$$

$$\langle \lambda(\tau\ x)\ E,\, s\rangle \longrightarrow \langle \lambda_s(\tau\ x)\ E,\, s\rangle \quad \text{(R-Abstração)}$$

$$\langle (\lambda_s(\tau\ x)\ E)(v),\, s'\rangle \longrightarrow \langle \{E\{^{v}/_{x}\}\},\, s'\cdot s\rangle \quad \text{(R-Aplicação)}$$

$$\langle v\ op\ v',\, s\rangle \longrightarrow \langle \mathcal{A}[\![v\ op\ v']\!],\, s\rangle \quad \text{(R-Operação)}$$

$$\langle [v_1,\cdots,v_n] +\!\!+ [v'_1,\cdots,v'_n],\, s\rangle \longrightarrow \langle [v_1,\cdots,v_n,v'_1,\cdots,v'_n],\, s\rangle \quad \text{(R-Concatenação)}$$

$$\langle [v_1,\cdots,v_n][m],\, s\rangle \longrightarrow \langle v_m,\, s\rangle \quad \text{(R-Elemento)}$$

$$\langle \textsf{length}([v_1,\cdots,v_n]),\, s\rangle \longrightarrow \langle n,\, s\rangle \quad \text{(R-Comprimento)}$$

$$\frac{s.\textsf{in} = [v] +\!\!+ s'}{\langle \textsf{read}(),\, s\rangle \longrightarrow \langle v,\, s[s.\textsf{in} \leftarrow s']\rangle} \quad \text{(R-Leitura)}$$

$$\langle \textsf{impure}\ E,\, s\rangle \longrightarrow \langle \textsf{impure}_s\ E,\, s\rangle \quad \text{(R-Impure)}$$

$$\langle \textsf{run}\ (\textsf{impure}_s\ E),\, s'\rangle \longrightarrow \langle \{E\},\, s'\cdot s[s.\textsf{in} \leftarrow s'.\textsf{in},\, s.\textsf{out} \leftarrow s'.\textsf{out}]\rangle \quad \text{(R-Execução)}$$

$$\langle \{\overline{\tau\ y\ =v}\ E\},\, s\rangle \longrightarrow \langle \{E\},\, s\cdot[\overline{y \leftarrow v}]\rangle \quad \text{(R-Bloco-1)}$$

$$\langle \{v\},\, s\cdot s'\rangle \longrightarrow \langle v,\, s\rangle \quad \text{(R-Bloco-2)}$$

Figura 2: Semântica para expressões

A primeira combina os dois estados s e s', i.e., $(s \cdot s')(x)$ procura por x em s' e, caso não encontre, procura em s. A segunda a operação afecta x com o valor da avaliação da expressão E. A semântica operacional é definida com base em contextos sintáticos de avaliação, e na extensão da sintaxe dos programas com uma expressão de execução $\textsf{while}_E$ utilizada na avaliação de ciclos.

$$\begin{aligned}\mathcal{C} ::= {} & \square \mid \mathcal{C}\,;\, S \mid \textsf{while}_E\ \mathcal{E}\ \textsf{do}\ S \mid \textsf{if}\ \mathcal{E}\ \textsf{then}\ S\ \textsf{else}\ S \mid x_1,\cdots,x_n = \mathcal{E}_1,\cdots,\mathcal{E}_n \\ & \mid \textsf{print}\ \mathcal{E}\end{aligned}$$

$$\begin{aligned}\mathcal{E} ::= {} & \mathcal{E}(E_1,\cdots,E_n) \mid v(\mathcal{E}_1,\cdots,\mathcal{E}_m,E_{m+1},\cdots,E_n) \mid \mathcal{E}\ op\ E \mid v\ op\ \mathcal{E} \\ & \mid [\mathcal{E}_1,\cdots,\mathcal{E}_m,E_{m+1},\cdots,E_n] \mid \mathcal{E} +\!\!+ E \mid v +\!\!+ \mathcal{E} \mid \mathcal{E}[E] \mid v[\mathcal{E}] \mid \textsf{length}(\mathcal{E}) \\ & \mid \{\mathcal{E}\} \mid \mathcal{C}\ \textsf{return}\ E\end{aligned}$$

A semântica operacional é descrita pelas regras de inferência da Figura 2. A regra da semântica de expressões (R-Expressão) permite que quando uma expressão é reduzível a outra (podendo modificar o estado), também é possível reduzi-la quando num contexto sintático (alterando o estado da mesma maneira), generalizando a semântica de várias expressões. A notação $\mathcal{A}[\![v\ op\ v']\!]$ (R-Operação) representa a avaliação de expressões aritméticas, booleanas e comparações com operadores binários. A regra (R-Leitura) remove um valor do início da *stream* de entrada e devolve esse valor.

A avaliação da expressão da função lambda (R-Abstração) deve ser feita no ambiente original onde ela foi definida, para garantir escopo estático. Deste

$$\frac{\langle S,\ s\rangle \longrightarrow \langle S',s'\rangle}{\langle \mathcal{C}[S],\ s\rangle \longrightarrow \langle \mathcal{C}[S'],s'\rangle} \quad \text{(R-Comando)}$$

$$\langle \textsf{skip}\ ;\ S,\ s\rangle \longrightarrow \langle S,s\rangle \quad \text{(R-Skip)}$$

$$\langle \textsf{while}\ E\ \textsf{do}\ S,\ s\rangle \longrightarrow \langle \textsf{while}_E\ E\ \textsf{do}\ S,s\rangle \quad \text{(R-Ciclo)}$$

$$\langle \textsf{while}_E\ \textsf{true do}\ S,\ s\rangle \longrightarrow \langle S\ ;\ \textsf{while}\ E\ \textsf{do}\ S,s\rangle \quad \text{(R-Ciclo-T)}$$

$$\langle \textsf{while}_E\ \textsf{false do}\ S,\ s\rangle \longrightarrow \langle \textsf{skip},s\rangle \quad \text{(R-Ciclo-F)}$$

$$\langle \textsf{if true then}\ S\ \textsf{else}\ S',\ s\rangle \longrightarrow \langle S,s\rangle \quad \text{(R-Condicional-T)}$$

$$\langle \textsf{if false then}\ S\ \textsf{else}\ S',\ s\rangle \longrightarrow \langle S',s\rangle \quad \text{(R-Condicional-F)}$$

$$\langle x_1,\cdots,x_n = v_1,\cdots,v_n,\ s\rangle \longrightarrow \langle \textsf{skip}, s[\overline{x \leftarrow v}]\rangle \quad \text{(R-Afectação)}$$

$$\langle \textsf{print}\ v,\ s\rangle \longrightarrow \langle \textsf{skip}, s[s.\textsf{out} \leftarrow s.\textsf{out}{+\!+}[v]]\rangle \quad \text{(R-Escrita)}$$

$$\langle \textsf{skip return}\ E,\ s\rangle \longrightarrow \langle E,s\rangle \quad \text{(R-Retorno)}$$

Figura 3: Semântica para comandos

modo, a expressão lambda é reduzida a uma outra construção λ_s (R-Aplicação), que captura o estado s do programa no instante da declaração. Quando é efectuada a chamada, é utilizado este estado que corresponde ao estado da declaração. O estado capturado pela função lambda contêm também as *streams* de input e output. Isto não é problema pois o sistema de tipos garante que as funções lambda são puras e portanto não é feito qualquer I/O dentro da função.

À semelhança da semântica da abstração, a regra (R-Impure) tem de capturar o estado de onde é definida a expressão, para na sua avaliação (R-Execução) obter a semântica esperada, i.e., escopo estático. Contudo, as modificações ao I/O têm de ser preservadas. Todas as outras expressões têm a semântica esperada.

A regra de semântica de comandos (R-Comando), é semelhante à das expressões, mas a substituição é sobre comandos nos contextos sintáticos dos comandos. O comando $\textsf{while}_E$ (regras (R-Ciclo-T) e (R-Comando-F)) captura a condição do ciclo para poder ser avaliada futuramente, de modo a garantir a semântica esperada de um ciclo **while** (R-Ciclo). Os restantes comandos têm a semântica esperada e não serão discutidos.

Iniciar a execução do programa consiste em carregar todas as funções *top-level* para o ambiente (como associações entre os identificadores das funções e a função na forma lambda), e avaliar a expressão **run**(main()).

6 Sistema de Tipos

O sistema de tipos garante que todo o código impuro é suspenso dentro de um bloco impuro. É ainda garantido que dentro de um bloco impuro e de funções lambda todos os acessos de escrita são feitos em variáveis locais, i.e. o ambiente externo passa a ser *read-only*. Estes acessos de escrita não seriam problema no contexto de um bloco impuro, pois estes nunca são corridos concorrentemente,

$$\Gamma \vdash num : Int_\circ \qquad \Gamma \vdash bool : Bool_\circ \qquad \Gamma \vdash \textbf{nil} : Void_\circ \quad \text{(Int, Bool, Void)}$$

$$\frac{\Gamma, x : \tau \vdash x : \tau \qquad \tau \neq |\tau'|}{\Gamma, x : \tau \vdash x : \tau_\circ} \qquad \Gamma, x : |\tau| \vdash x : \tau_\circ \quad \text{(Identificador)}$$

$$\frac{\Gamma \vdash e_1 : \tau_{\delta_1} \qquad \cdots \qquad \Gamma \vdash e_n : \tau_{\delta_n}}{\Gamma \vdash [e_1, \cdots, e_n] : [\tau]_{\delta_1 \odot \cdots \odot \delta_n}} \quad \text{(Lista)}$$

$$\Gamma \vdash \textbf{read}() : Int_\bullet \quad (\textit{Input})$$

$$\frac{\Gamma \vdash e_1 : [\tau]_{\delta_1} \qquad \Gamma \vdash e_2 : [\tau]_{\delta_2}}{\Gamma \vdash e_1 +\!\!+ e_2 : [\tau]_{\delta_1 \odot \delta_2}} \quad \text{(Concatenação de Listas)}$$

$$\frac{\Gamma \vdash e_1 : [\tau]_{\delta_1} \qquad \Gamma \vdash e_2 : Int_{\delta_2}}{\Gamma \vdash e_1[e_2] : \tau_{\delta_1 \odot \delta_2}} \quad \text{(Acesso Lista)}$$

$$\frac{\Gamma \vdash e : [\tau]_\delta}{\Gamma \vdash \textbf{length}(e) : Int_\delta} \quad \text{(Tamanho Lista)}$$

$$\frac{\Gamma, \overline{x : \tau} \vdash e' : \sigma_\gamma \qquad \Gamma \vdash e_i : \tau_{i\,\delta_i}}{\Gamma \vdash \{\overline{\tau\ x = e}\ e'\} : \sigma_{\gamma \odot \delta_1 \odot \cdots \odot \delta_n}} \quad \text{(Declaração)}$$

$$\frac{|\Gamma| \vdash e : \sigma}{\Gamma \vdash \textbf{impure}\ e : (\textit{Impure}\ \sigma)_\circ} \quad \text{(Bloco Impuro)}$$

$$\frac{\Gamma \vdash e : (\textit{Impure}\ \tau)_\circ}{\Gamma \vdash \textbf{run}(e) : \tau_\bullet} \quad \text{(Execução Impura)}$$

$$\frac{|\Gamma|, x : \tau \vdash e : \sigma_\circ}{\Gamma \vdash \lambda(\tau\ x)\ e : (\tau \to \sigma)_\circ} \quad \text{(Abstração)}$$

$$\frac{\Gamma \vdash e : (\tau \to \sigma)_\circ \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e(e') : \sigma_\circ} \quad \text{(Aplicação)}$$

Figura 4: Regras de tipificação das expressões.

mas no contexto de funções lambda estas escritas violariam a pureza da linguagem. A Listagem 3 apresenta um exemplo de uma função que pode devolver os valores 0 ou 1, dependendo do escalonamento da execução paralela das funções lambda. Isto violaria a propriedade do determinismo que as funções puras têm de garantir, e é rejeitado pelo sistema de tipos.

Na definição do sistema de tipos um ambiente de tipificação é um mapeamento de identificadores para tipos τ, ou para tipos com a anotação de *read-only*, $|\tau|$. A operação $|\Gamma|$ sobre um ambiente Γ é definida por:

$$|\Gamma| = |x_1 : \tau_1, \cdots, x_n : \tau_n| = x_1 : |\tau_1|, \cdots, x_n : |\tau_n|.$$

$$\Gamma \vdash \textsf{skip} : \ \textsf{ok}_\circ \qquad \text{(Comando Vazio)}$$

$$\frac{\Gamma \vdash e : \mathit{Int}}{\Gamma \vdash \textsf{print}\ e\ \ \textsf{ok}_\bullet} \qquad (\mathit{Output})$$

$$\frac{\Gamma \vdash s\ \ \textsf{ok}_\delta \qquad \Gamma \vdash e : \tau_{\delta'}}{\Gamma \vdash s\ \textsf{return}\ e : \tau_{\delta \odot \delta'}} \qquad \text{(Comando)}$$

$$\frac{\Gamma(x_i) = \tau_i \qquad |\Gamma| \vdash e_i : \tau_i\,\delta_i \qquad \tau_i \neq |\tau_i'| \qquad \forall_{i,j}\,\delta_i = \delta_j = \bullet \implies i = j}{\Gamma \vdash x_1, \cdots, x_n = e_1, \cdots, e_n\ \ \textsf{ok}_{\delta_1 \odot \cdots \odot \delta_n}} \qquad \text{(Afectação Paralela)}$$

$$\frac{\Gamma \vdash s_1\ \ \textsf{ok}_{\delta_1} \qquad \Gamma \vdash s_2\ \ \textsf{ok}_{\delta_2}}{\Gamma \vdash s_1\ ;\ s_2\ \ \textsf{ok}_{\delta_1 \odot \delta_2}} \qquad \text{(Sequência)}$$

$$\frac{\Gamma \vdash c : \mathit{Bool}_\delta \qquad \Gamma \vdash s_1\ \ \textsf{ok}_{\delta_1} \qquad \Gamma \vdash s_2\ \ \textsf{ok}_{\delta_2}}{\Gamma \vdash \textsf{if}\ c\ \textsf{then}\ s_1\ \textsf{else}\ s_2\ \ \textsf{ok}_{\delta \odot \delta_1 \odot \delta_2}} \qquad \text{(Condicional)}$$

$$\frac{\Gamma \vdash c : \mathit{Bool}_\delta \qquad \Gamma \vdash s\ \ \textsf{ok}_{\delta'}}{\Gamma \vdash \textsf{while}\ c\ \textsf{do}\ s\ \ \textsf{ok}_{\delta \odot \delta'}} \qquad \text{(Ciclo)}$$

$$\frac{|\Gamma|, x : \sigma \vdash e : \tau_\circ}{\Gamma \vdash \tau\ f(\sigma\ x)\ e\ \ \textsf{ok}_\circ} \qquad \text{(Declaração de Função)}$$

Figura 5: Regras de tipificação dos comandos.

O efeito desta operação é notória na regra (Afectação Paralela), impedindo a afectação a variáveis de tipo $|\tau|$.

O sistema de tipos é definido para as expressões pela asserção $\Gamma \vdash E : \tau_\delta$, definida pelas regras da Figura 4 onde os tipos considerados são o tipo inteiro, booleano, lista, **void**, o tipo das computações impuras e o tipo função. Para os comandos é definido pela asserção $\Gamma \vdash S\ \ \textsf{ok}_\delta$ e pelas regras da Figura 5. A anotação da asserção δ denota a pureza da expressão ou comando a ser tipificado. Os valores da anotação δ são usados para manter o controlo do que é puro ou impuro, através dos valores $\circ$ e $\bullet$ se a expressão for pura ou impura, respectivamente. Estas anotações são omitidos quando não são necessárias.

O ambiente não precisa de guardar estes atributos, pois as referências a identificadores no ambiente são sempre puras, como se pode ver pela regra de tipificação (Identificador). O valor das variáveis é sempre puro, ainda que tenha vindo de uma fonte impura. Nesse caso o que é impuro é a operação que obteve o valor impuro associado à variável. Define-se o operador $\odot$ para combinar atributos de pureza:

$$\delta \odot \delta' = \begin{cases} \circ & \text{se } \delta = \delta' = \circ \\ \bullet & \text{caso contrário} \end{cases}$$

```
Int invalidA()
{
   Int z = −1;

   _, _ = λ(){ z=0; return 0; }(),
          λ(){ z=1; return 0; }();

   return z;
}
```

```
Int invalidB()
{
   Int z = −1;

   _, _ = ( z=0; return 0; ),
          ( z=1; return 0; );

   return z;
}
```

Listagem 3: Programas inválidos rejeitados pelo sistema de tipos.

Esta operação tem como resultado $\bullet$ (impuro) se algum dos seus operandos for impuro, ou seja, $\bullet$ é elemento absorvente de $\odot$. Intuitivamente isto corresponde a dizer que combinar operações impuras com puras dá sempre uma operação impura.

As únicas fontes de impureza nesta linguagem são o comando **print** $\cdot$, a expressão **read**$(\cdot)$ e a expressão **run**$(\cdot)$, como está patente nas suas regras de tipificação. Todo o valor obtido através do tipo *Impure* τ é feito através da expressão **run**$(\cdot)$ que é impura, desta maneira o sistema de tipos obriga a que todos os valores impuros sejam obtidos num bloco impuro.

A afectação paralela tem que evitar a execução imprópria de código em paralelo. Para lidar com isso a regra (Afectação Paralela) obriga a que no máximo uma expressão impura possa aparecer na afectação, e nesse caso a afectação paralela é impura e a sua utilização está limitada a blocos impuros. É ainda necessário evitar que exista qualquer tipo de afectação a variáveis externas dentro da afectação paralela, como exemplificado na Listagem 3. As outras regras de tipificação são típicas e omitiremos a sua explicação. O sistema de tipos garante que todas as funções, sejam elas *top-level* ou funções lambda, são puras.

7 Comentários finais

Tanto quanto sabemos este é o primeiro trabalho a usar uma noção explícita de pureza para paralelizar código numa linguagem imperativa, no entanto existem vários trabalho sobre controlo de interferências, código puro em linguagens imperativas e funcionais, e paralelização de código puro em linguagens funcionais.

Existem vários trabalhos sobre isolar código impuro de código puro em linguagens funcionais através da utilização de *monads* [9], como Simon Peyton Jones explica em [7]. A linguagem *Haskell* também tira facilmente partido de paralelismo de código puro [6]. A linguagem *D*, que é imperativa, também suporta funções puras explicitas e que são verificadas estaticamente pelo compilador [2] mas esta noção não é usada para garantir que código pode ser paralelizado. Existem também trabalhos sobre detecção estática de código puro em linguagens imperativas [12,4,8].

Este trabalho aplica a noção de pureza funcional, que é habitualmente existente em linguagens funcionais, numa linguagem imperativa. Esta estratégia é inspirada na separação entre código puro e código impuro oferecido pela linguagem *Haskell* através do *monad IO*, mas aplicado a uma linguagem com uma noção de estado local e de manipulação desse estado com o objectivo de tornar a paralelização de código fácil. A linguagem *Impure* foi criada de maneira a oferecer as construções habitualmente encontradas em linguagens imperativas, pelo que facilmente modela programas reais. Também são oferecidas ainda outras características mais comuns em linguagens funcionais, como lista como tipo nativo e abstrações lambda, que oferecem desafios adicionais. O sistema de tipos desenvolvido garante que todo o código impuro é protegido e que todo o código executado concorrentemente é de facto puro tendo em conta os efeitos laterais e modificações ao estado local.

Um desafio interessante ainda a explorar é como adicionar referências à linguagem permitindo ainda o seu uso em contexto puro, com as limitações que implica a natureza pura de linguagem. Novos mecanismos de sincronização e controlo de concorrência podem trazer vantagens tanto na escrita de código concorrente como na sua execução.

Referências

1. Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Progr. Lang. Syst.*, 28(2):207–255, 2006.
2. Andrei Alexandrescu. *The D Programming Language*, chapter 5.11.1. Addison Wesley, first edition, 2010.
3. Luís Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theor. Comput. Sci.*, 402(2-3):120–141, July 2008.
4. Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in java. In *Proc. of the 15th ACM conference on Computer and communications security*, CCS '08, pages 161–174. ACM, 2008.
5. Paul et al. Hudak. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992.
6. Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in haskell. Technical report, 1993.
7. Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pages 47–96. Press, 2001.
8. Ravichandhran Madhavan, Ganesan Ramalingam, and Kapil Vaswani. Purity analysis: an abstract interpretation formulation. In *Proc. of the 18th international conference on Static analysis*, SAS'11, pages 7–24, 2011.
9. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.
10. John C. Reynolds. Syntactic control of interference. In *Proc. of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 39–46, 1978.
11. Amr Sabry. What is a purely functional language? *J. Funct. Program.*, 8(1):1–22, January 1998.

12. Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for java programs. In *Proc. of the 6th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 199–215, 2005.