Inferência de anotações para evitar impasses numa linguagem intermédia polimórfica

Roberto Silva, Francisco Martins, and Vasco T. Vasconcelos

LaSIGE & Universidade de Lisboa
Portugal

Resumo A linguagem intermédia MIL é apropriada para programação baixo nível de aplicações concorrentes com uso de memória partilhada. O seu sistema de tipos polimórfico garante que programas MIL bem tipificados não possuem condições de corrida nem impasses. A primeira propriedade é conseguida através da imposição de uma disciplina no uso de trincos através de tipos singulares; a segunda é alcançada através de anotações polimórficas sobre a ordem pela qual os trincos são fechados em cada bloco de código. O sistema de tipos recusa programas cujos fios de execução dependam ciclicamente uns dos outros no fecho de trincos. No entanto, a introdução de anotações sobre trincos pode introduzir uma complexidade desnecessária no processo de geração de código.
Este artigo propõe um algoritmo para inferir as anotações polimórficas sobre trincos. A inferência é feita através da recolha de restrições locais sobre a ordem pela qual os trincos são fechados em cada bloco de código. As restrições são passadas a um SMT que averigua a sua consistência. Implementámos o algoritmo e experimentámo-lo extensivamente em programas MIL, nomeadamente em código de alguma dimensão obtido da compilação de uma linguagem de objetos concorrentes.

1 Introdução

A utilização da informática na resolução de problemas de maior dimensão e complexidade implica uma necessidade permanente de *hardware* com maior capacidade de processamento. Esta situação fez com que, num passado recente, os fabricantes tenham abandonado o modelo tradicional de Von Neumann e adotado sistemas com múltiplos processadores ou com múltiplos núcleos por processador.

No entanto, é fundamental que o *software* acompanhe a evolução do *hardware*, de forma a tirar partido do poder de computação extra oferecido pelos sistemas com processamento paralelo. Torna-se imperioso o desenvolvimento de novas linguagens de programação concorrentes que ajudem os programadores a lidar com as dificuldades adicionais que são colocadas pela programação com múltiplos fios de execução. O tema não é novo. Problemas decorrentes de condições de corrida e de impasse remontam aos primórdios da informática. A novidade está na necessidade generalizada em recorrer à programação concorrente, necessidade essa que antes se restringia a um grupo limitado de profissionais altamente competentes.

A linguagem intermédia MIL é apropriada para o desenvolvimento baixo nível de sistemas concorrentes com uso de memória partilhada. O seu sistema

de tipos garante que programas MIL bem tipificados não possuem condições de corrida nem impasses [14,15]. A primeira propriedade é conseguida através da imposição criteriosa de uma disciplina no uso de trincos através da utilização de tipos singulares; a segunda é alcançada através de anotações polimórficas sobre a ordem pela qual os trincos devem ser fechados (ou tomados ou adquiridos) [3]. O sistema de tipos verifica as anotações polimórficas e recusa programas cujos fios de execução dependam ciclicamente uns dos outros na obtenção de trincos.

Todavia, a anotação de programas pode complicar o processo de geração de código, pois os conceitos de trinco e de ordem para apanhar o trinco podem não fazer parte da linguagem fonte. A fim de solucionar este problema, propomos neste artigo um algoritmo para inferir as anotações polimórficas referentes à ordem pela qual os trincos devem ser fechados num programa MIL. Esta inferência é efetuada através da recolha de restrições locais sobre a ordem pela qual os trincos devem ser tomados em cada bloco de código. As restrições são depois passadas a um SMT que averigua a sua consistência.

Coffman et al. classificam o problema de impasse em três categorias: deteção e recuperação, impedimento e prevenção [7]. Na primeira categoria, deteção e recuperação, existem alguns trabalhos que verificam se programas atingem impasses, em tempo de execução. Cunningham et al. inferem trincos numa linguagem orientada a objetos, mas usam um mecanismo em tempo de execução para detetar quando é que a aquisição de um trinco por parte de um fio de execução atinge um impasse [9]. Java PathFinder [5] e Driver Verifier [2] identificam violações no uso de trincos através de testes de *software*. Agarwal et al. apresentam um algoritmo que deteta impasses em tempo de execução em programas que usam trincos, semáforos ou variáveis condicionais [1].

O nosso trabalho assenta sobre a terceira categoria, prevenção. Flanagan e Abadi apresentam uma linguagem funcional com referências mutáveis [11]. De forma a prevenir impasses, os trincos são introduzidos na verificação de tipos através de tipos singulares. Com esta introdução pode antecipar-se uma possível presença de impasses, bem como garantir a ausência de condições de corrida nos programas. A prevenção de impasses também tem sido estudada no âmbito de linguagens orientadas por objetos. Para a linguagem Java, Boyapati et al. utilizam uma variante destes tipos, realizando uma inferência parcial de anotações, mas não daquelas relacionadas com a ordem dos trincos [4]. Suenaga propõe uma linguagem concorrente funcional idêntica à linguagem de Flanagan e Abadi, exceto na estrutura de blocos [13]. Tal como o MIL, esta linguagem inclui primitivas separadas para a obtenção e libertação de trincos.

As contribuições deste artigo são as seguintes:

- Um algoritmo para inferir as necessárias anotações polimórficas sobre a ordem pela qual os trincos devem ser fechados num programa MIL, a fim de assegurar que os programas não atingem impasses;
- Implementação em Z3 de um resolvedor das restrições geradas pelo algoritmo; teste do algoritmo através da sua aplicação a um grande leque de programas MIL.

O artigo encontra-se organizado da seguinte forma. A secção seguinte define a linguagem MIL por intermédio de um exemplo bem conhecido da literatura em concorrência: o jantar dos filósofos. Na secção 3 discute-se os problemas da inferência das anotações necessárias, em particular, como é que a informação sobre as restrições flui através do grafo de chamada dos blocos de código. A secção 4 apresenta detalhadamente o algoritmo ilustrando-o com exemplos. A secção 5 trata da resolução das restrições em Z3. A última secção apresenta as conclusões do artigo e o trabalho que nos propomos realizar num futuro próximo.

2 A linguagem MIL

Esta secção apresenta de modo informal, e por intermédio de um exemplo, a sintaxe e semântica da linguagem de programação MIL—*Multithreaded Intermediate Language*. A bandeira do MIL é a sua semântica estática imposta por um sistema de tipos, e que garante, em tempo de compilação, que programas bem tipificados não acedem a posições de memória inválidas e são livres de condições de corrida e de impasses.

A linguagem MIL destina-se a programar uma máquina abstrata com vários processadores, cuja memória principal é partilhada. Cada processador dispõe de um conjunto de registos, uma memória local para instruções e um conjunto de trincos fechados, e corre um fio de execução de cada vez; os registos são globais a este fio de execução. A memória principal é dividida em duas partes: um amontoado e uma piscina para os fios de execução suspensos. O amontoado armazena blocos de dados e blocos de código: os primeiros são representados por tuplos e são partilhados entre os vários processadores; os segundos são compostos por uma assinatura e por um conjunto de instruções. A assinatura indica quais os tipos esperados para os registos e o conjunto de trincos fechados requeridos no instante em que se salta para, ou se lança um fio de execução com, o bloco de código. A piscina de fios de execução contém os fios de execução que esperam um processador livre.

A figura 1 ilustra uma implementação do famoso problema do jantar de filósofos [12] escrito em MIL. O programa é composto por quatro blocos de código identificados pelas etiquetas main, levantarGarfoEsquerdo, levantarGarfoDireito e comerEFilosofar. A primeira linha contém a assinatura do bloco main, assinatura essa que não indica restrições nem ao nível do tipo dos registos nem dos trincos requeridos, o que se justifica por este ser o ponto de entrada da execução do programa. As assinaturas dos restantes blocos de código são polimórficas nos tipos dos registos e dos trincos. Por exemplo, a linha 11 contém a assinatura do bloco levantarGarfoEsquerdo, onde é requerido que aquando de um salto para, ou do lançamento de um fio de execução com levantarGarfoEsquerdo, os registos r1 e r2 deverão conter dois trincos, que neste caso se encontram abstraídos universalmente. Já as definições de levantarGarfoDireito (linha 17) e de comerEFilosofar (linha 23) exigem adicionalmente a posse do trinco e1, na primeira, e dos trincos e2 e d2, na segunda. As variáveis sobre trincos aparecem artificialmente renomeadas para simplificar a explicação do algoritmo (secção 4).

```
1  main::{}
   main = {
3    g1::Lock    r3 := new lock g1    unlock r3                    -- primeiro garfo
     g2::Lock    r4 := new lock g2    unlock r4                    -- segundo grafo
5    g3::Lock    r5 := new lock g3    unlock r5                    -- terceiro grafo
     r1 := r3    r2 := r4    fork levantarGarfoEsquerdo[g1][g2] -- primeiro filosofo
7    r1 := r4    r2 := r5    fork levantarGarfoEsquerdo[g2][g3] -- segundo filosofo
     r1 := r5    r2 := r3    fork levantarGarfoEsquerdo[g3][g1] -- terceiro filosofo
9    done
   }
11 levantarGarfoEsquerdo :: ∀ e::Lock.∀ d::Lock.{r1:lock e, r2:lock d}
   levantarGarfoEsquerdo = {
13   r3 := getSetLock r1
     if r3 == 0 jump levantarGarfoDireito[e][d]
15   jump levantarGarfoEsquerdo[e][d]
   }
17 levantarGarfoDireito :: ∀ e1::Lock.∀ d1::Lock.{r1:lock e1, r2:lock d1} requires {e1}
   levantarGarfoDireito = {
19   r3 := getSetLock r2
     if r3 == 0 jump comerEFilosofar[e1][d1]
21   jump levantarGarfoDireito[e1][d1]
   }
23 comerEFilosofar :: ∀ e2::Lock.∀ d2::Lock.{r1:lock e2, r2:lock d2} requires {e2,d2}
   comerEFilosofar = {
25   -- comer
     unlock r1 -- pousa o garfo esquerdo
27   unlock r2 -- pousa o garfo direito
     -- filosofar
29   jump levantarGarfoEsquerdo[e2][d2]
   }
```

Figura 1. O jantar dos filósofos escrito em MIL

O bloco de código main começa por declarar três trincos com os nomes g1, g2 e g3 que representam os três garfos que os filósofos partilham (linhas 3–5). Analisando a linha 3 em detalhe, o trinco g1 é declarado (g1::**Lock**), de seguida é reservada memória no amontoado para o guardar, ficando a referência para a memória guardada no registo r3 (r3 := **new lock** g1). Por último o trinco é aberto (**unlock** r3), pois a instrução **new** cria um trinco fechado e neste exemplo podemos liberta-lo de imediato. As linhas 6–8 lançam os fios de execução referentes aos três filósofos. Por exemplo, no caso do filósofo 1, são carregados nos registos r1 e r2 as referências respeitantes aos trincos g1 e g2 e de seguida a instrução **fork** lança um novo fio de execução que irá executar o bloco levantarGarfoEsquerdo. Note-se que os parâmetros de tipo e e d são instanciados com os trincos g1 e g2, respetivamente. No instante do lançamento de cada fio de execução os tipos dos registos em main coincidem com os tipos esperados pelo

bloco levantarGarfoEsquerdo. Para tal, verifique-se que aquando do lançamento do fio de execução do terceiro filósofo (linha 8) o registo r1 refere o trinco g3, enquanto o registo r2 refere o trinco g1, que está de acordo com a instanciação dos argumentos de tipo levantarGarfoEsquerdo[g3][g1].

Uma bloco de instruções termina ou com a instrução **done** (linha 9) ou com a instrução **jump** (linha 15). A primeira termina o fio de execução, deixando o processador disponível para executar outro fio que aguarde na piscina de fios de execução; a segunda continua a execução no bloco indicado na instrução de salto.

Os blocos de código levantarGarfoEsquerdo e levantarGarfoDireito fecham os trincos e e d1, que representam os garfos esquerdo e direito do filósofo. O bloco levantarGarfoEsquerdo tenta fechar o trinco e utilizando a instrução **getSetLock** (linha 13). Esta instrução, de forma indivisível, obtém o valor do trinco indicado pelo registo r1 e fecha-o. O teste na linha 14 verifica se o trinco estava aberto anteriormente. Se tal for o caso, isso significa que o fio de execução fechou o trinco e o controle passa para o bloco levantarGarfoEsquerdo. Caso contrário, volta a tentar fechar o trinco utilizando o método da espera ativa (linha 15). O bloco levantarGarfoDireito é semelhante ao levantarGarfoEsquerdo, exceto que requere que o trinco e1 já tenha sido fechado, tenta fechar o trinco d1 e quando tal acontecer salta para o bloco de código comerEFilosofar.

Por último, no bloco comerEFilosofar o filósofo come, pousa os garfos, libertando para tal os trincos referidos por r1 e r2 (linhas 26 e 27), pensa e inicia uma nova ronda (linha 29).

3 Anotações para detetar impasses

Um impasse ocorre quando um programa não consegue progredir porque os seus fios de execução necessitam de recursos que estão detidos por outros fios de execução e vice-versa. A figura 1 apresenta um caso simples de um programa que, quando executado, poderá eventualmente atingir um impasse. Na verdade, há um escalonamento que permite que cada filósofo possa apanhar o garfo à sua esquerda (levantarGarfoEsquerdo) e ficar indefinidamente à espera de apanhar o garfo à sua direita (ciclo de espera ativa em levantarGarfoDireito). Esta situação é facilmente evitada substituindo a instrução **fork** levantarGarfoEsquerdo[g3][g1] da linha 8 por **fork** levantarGarfoEsquerdo[g1][g3], ou seja, trocando as voltas (ou antes, os braços) ao terceiro filósofo.

Mas como determinar, sem executar o programa da figura 1, que este tem um escalonamento que pode levar a um impasse, e que a variante que propomos nunca atinge um impasse? O método que seguimos em [15] consiste em decorar as variáveis de trinco com anotações (locais e polimórficas) de ordem, conduzindo a uma ordem (parcial) global para o fecho dos trincos. Por sua vez, o sistema de tipos verifica que de facto o programa fecha os trincos por esta ordem. A tarefa de anotar um programa pode complicar a geração de código, porque em alguns casos a informação sobre a ordem de fecho de trincos está ausente do processo

```
2  main = {
     g1::Lock({},{})        r3 := new lock g1     unlock r3          -- primeiro garfo
4    g2::Lock({g1},{})      r4 := new lock g2     unlock r4          -- segundo grafo
     g3::Lock({g1,g2},{})   r5 := new lock g3     unlock r5          -- terceiro grafo
6    r1 := r3          r2 := r4                r6 := levantarGarfoDireito [g1]
     fork levantarGarfoEsquerdo[g1][g2]                              -- primeiro filosofo
8    r1 := r4          r2 := r5                r6 := levantarGarfoDireito [g2]
     fork levantarGarfoEsquerdo[g2][g3]                              -- segundo filosofo
10   r1 := r5          r2 := r3                r6 := levantarGarfoDireito [g3]
     fork levantarGarfoEsquerdo[g3][g1]                              -- terceiro filosofo
12   done
   }
14 levantarGarfoEsquerdo::∀ e::Lock({},{}).∀ d::Lock({e},{}).{r1:lock e,r2:lock d,
     r6:∀ l::Lock({e},{}).{r1:lock e,r2:lock l} requires {e}}
16 levantarGarfoEsquerdo = {
     r3 := getSetLock r1
18   if r3 == 0 jump r6[d]
     jump levantarGarfoEsquerdo[e][d]
20 }
   levantarGarfoDireito ::
22   ∀ e1::Lock({},{}).∀ d1::Lock({e1},{}).{r1:lock e1,r2:lock d1} requires {e1}
   levantarGarfoDireito = {
24   r3 := getSetLock r2
     if r3 == 0 jump comerEFilosofar[e1][d1]
26   jump levantarGarfoDireito [e1][d1]
   }
28 comerEFilosofar ::
     ∀ e2::Lock({},{}).∀ d2::Lock({},{}).{r1:lock e2,r2:lock d2} requires {e2,d2}
30 comerEFilosofar = {
     ...
32   r6 := levantarGarfoDireito [e2]
     jump levantarGarfoEsquerdo[e2][d2]
34 }
```

Figura 2. O jantar dos filósofos com uma indireção

de compilação (isto foi-nos dado observar na escrita de um compilador de uma linguagem de objetos concorrentes em MIL, secção 5).

A figura 2 apresenta uma generalização do bloco levantarGarfoEsquerdo (cf. figura 1), além de tornar explícitas as anotações polimórficas sobre os trincos. O objetivo é ilustrar a dificuldade em inferir estas anotações. Note-se que este simples programa de 30 linhas contém 20 anotações relacionadas com ordens de trincos (linhas 3–5, 14-15, 22 e 29). Nesta versão, o bloco levantarGarfoEsquerdo recebe no registo r6 o endereço da sua continuação, usado na instrução **jump** (linha 18) em lugar do nome explícito do bloco levantarGarfoDireito , tal como acontece com o código na figura 1 (linha 14).

Voltemos a nossa atenção para o bloco main. Cada declaração de trinco (linhas 3–5) é decorada com dois conjuntos, por exemplo g2::**Lock**({g1},{}), que

contêm os trincos que devem ser fechados antes—{g1}—e depois—{}—de g2. Assim, é possível estabelecer uma relação de ordem parcial entre g2 e os demais trincos visíveis neste ponto do programa. As declarações polimórficas de blocos de código são também anotadas com conjuntos com igual semântica (*vide* linhas 14–15, 22 e 29). Na ausência de uma dada anotação, o algoritmo que propomos na secção seguinte, anota a variável de trinco, não com conjuntos concretos, mas com variáveis—L0, L1,...—sobre conjuntos de trincos. Será tarefa do algoritmo concretizar estas variáveis de modo a que o programa passe no sistema de tipos.

A ordem pela qual os trincos são fechados é determinada quando um fio de execução, tendo na sua posse alguns trincos fechados, tenta fechar o próximo trinco. Neste ponto do programa ficamos com a indicação de que os trincos já fechados têm de ser menores do que o trinco que se está a tentar fechar. Pretendemos recolher esta informação (na forma de restrições) e depois verificar se a informação é consistente, ou seja, se as restrições recolhidas não levam a concluir que para fechar um trinco é necessário ter já este mesmo trinco fechado.

Tomando o exemplo da figura 2, note-se que na linha 25 pretende-se fechar o trinco d1 tendo já o trinco e1 fechado (veja-se a menção **requires**{e1} na assinatura do bloco de código, na linha 22). Mas repare-se que e1 e d1 não são trincos concretos do programa; antes, correspondem a parâmetros de tipo que são substituídos em tempo de execução pelos trincos g1, g2 e g3. Esta informação é obtida através da análise do corpo dos blocos de código e é expressa em termos da informação local, na maior parte das vezes referente a parâmetros polimórficos.

4 O algoritmo de inferência de anotações

A figura 3 apresenta o pseudo-código do nosso algoritmo. Este efetua duas passagens sobre o código fonte. Na primeira passagem associa duas novas variáveis Li1 e Li2 a cada trinco li declarado e gera restrições que relacionam Li1, Li2 e li. Estas variáveis são definidas sobre conjuntos de trincos e denotam os trincos a apanhar antes (Li1) e depois (Li2) do trinco em questão. As restrições geradas capturam estes factos: o trinco li é apanhado após todos os trincos em Li1 e antes de qualquer trinco em Li2. Estas duas restrições, $\mathsf{Li1} < \mathsf{li}$ e $\mathsf{li} < \mathsf{Li2}$, são escritas abreviadamente como $\mathsf{Li1} < \mathsf{li} < \mathsf{Li2}$. Adicionalmente, requeremos que os trincos de Li1 e de Li2 correspondam a trincos que estejam no âmbito do bloco ou da instrução em causa. Caso contrário, a solução encontrada pode ser inválida por mencionar trincos declarados noutro ponto do programa.

A primeira passagem no exemplo da figura 2, e numerando os conjuntos sequencialmente (começando em L0), resulta que na linha 3 são associadas a g1 as variáveis L0 e L1 e as restrições $\mathsf{L0} < \mathsf{g1} < \mathsf{L1}$, $\mathsf{L0} \subseteq \emptyset$ e $\mathsf{L1} \subseteq \emptyset$. Na linha 14, por exemplo, são associadas ao trinco polimórfico d as variáveis L8 e L9 e as restrições $\mathsf{L8} < \mathsf{d} < \mathsf{L9}$, $\mathsf{L8} \subseteq \{e\}$ e $\mathsf{L9} \subseteq \{e\}$, pois o trinco e encontra-se em âmbito. A tabela 1 apresenta todas as restrições geradas na primeira passagem.

Na segunda passagem o algoritmo processa cada bloco de código e comporta-se de três formas distintas consoante se trate de um teste (2.1), da instanciação

1. Analisar cada bloco do programa: (primeira passagem)
   1.1 **para cada** trinco li definido na assinatura de um bloco de código (da forma $\forall$ li :: **Lock**.t), no corpo de um bloco (da forma li :: **Lock**), ou numa instrução **unpack** (da forma li , r:=**unpack** v), associar-lhe duas novas variáveis sobre conjuntos de trincos Li1 e Li2 e incluir as seguintes restrições: Li1 $<$ li $<$ Li2, Li1 $\subseteq$ *Trincos*, Li2 $\subseteq$ *Trincos*, em que *Trincos* representa o conjunto de trincos conhecidos no âmbito do bloco ou da instrução em causa.
2. Analisar cada bloco do programa: (segunda passagem)
   2.1 **Se** a instrução é um salto condicional sobre o trinco li (da forma **if** r==0 **jump** v) com v do tipo {...} **requires** G, **então** gerar a restrição: G $<$ li.
   2.2 **Se** a instrução inclui a aplicação de um trinco li a um valor v (da forma v[li]), em que o tipo de v é $\forall$m::**Lock**.t, e m está associado aos conjuntos L1 e L2, **então** gerar as restrições: L1 $<$ li $<$ L2.
   2.3 **Se** a instrução é um salto (da forma **jump** v) **ou** o lançamento de um fio de execução (da forma **fork** v), com v do tipo {r1: t1 ,..., rn: tn} **requires** G, comparar os tipos ti que são da forma $\forall$ li :: **Lock**.ui, com os tipos dos registos correspondentes conhecidos até ao momento (da forma $\forall$mi::**Lock**.si). Se li está associado aos conjuntos Li1 e Li2 e mi aos conjuntos Mi1 e Mi2, **então** gerar as restrições: Li1 = Mi1, Li2 = Mi2.

Figura 3. Algoritmo para colecionar restrições

de um valor polimórfico (2.2) ou de uma instrução de salto ou de lançamento de um fio de execução (2.3). Em relação a (2.1), a recolha de restrições ocorre na instrução **if**. Neste ponto do programa ficamos com a indicação de que os trincos já fechados têm de ser menores do que o trinco que está a tentar fechar-se. Esta regra pode ser aplicada em duas situações no exemplo da figura 2. Na linha 18 pretende-se obter o trinco e sem que tenha sido fechado qualquer outro trinco anteriormente. Na linha 25 pretende-se obter o trinco d1 tendo já o trinco e1 fechado (cf. **requires** {e1} na assinatura do bloco de código na linha 22). As restrições adicionadas são, respetivamente, {} $<$ e e {e1} $<$ d1.

Em relação à aplicação de valores (2.2) a restrição adicionada regista o facto do trinco li respeitar a ordem do fecho dos trincos associado a m, ou seja, ser fechado depois dos trincos denotados por Li1 e antes dos de Li2. Esta regra pode ser aplicada por diversas vezes no exemplo que apresentamos. Por exemplo, na linha 6 (r6:= levantarGarfoDireito [g1]) são adicionadas as restrições L12 $<$ g1 $<$ L13, em que L12 e L13 são as variáveis associadas ao trinco polimórfico e1 declarado na linha 21. A tabela 2 contém todas as restrições geradas de acordo com esta regra, assinaladas por (2.2).

O tratamento das instruções **jump** e **fork** (2.3) é mais complicado porque há que determinar o tipo dos registos antes da execução destas instruções, que é calculado pelo sistema de tipos (que omitimos neste artigo). Vamos ilustrar a aplicação desta regra à linha 7. O tipo dos registos relevantes para o lançamento

Tabela 1. Restrições geradas durante a primeira passagem do algoritmo

Linha	Restrições
3	L0 < g1 < L1, L0 ⊆ ∅ e L1 ⊆ ∅
4	L2 < g2 < L3, L2 ⊆ {g1} e L3 ⊆ {g1}
5	L4 < g3 < L5, L4 ⊆ {g1, g2} e L5 ⊆ {g1, g2}
14, 15	L6 < e < L7, L6 ⊆ ∅, L7 ⊆ ∅, L8 < d < L9, L8 ⊆ {e},L9 ⊆ {e}, L10 < l < L11, L10 ⊆ {e, d} e L11 ⊆ {e, d}
22	L12 < e1 < L13, L12 ⊆ ∅, L13 ⊆ ∅, L14 < d1 < L15, L14 ⊆ {e1}, e L15 ⊆ {e1}
29	L16 < e2 < L17, L16 ⊆ ∅, L17 ⊆ ∅, L18 < d2 < L19, L18 ⊆ {e2}, e L19 ⊆ {e2}

Tabela 2. Restrições geradas durante a segunda passagem do algoritmo

Linha	Restrições	Alínea
6, 8, 10	L12 < g1 < L13, L12 < g2 < L13 e L12 < g3 < L13	2.2
7	L6 < g1 < L7, L8[g1/e] < g2 < L9[g1/e],	2.2
	L10[g1/e][g2/d] = L14[g1/e1] e L11[g1/e][g2/d] = L15[g1/e1]	2.3
9	L6 < g2 < L7, L8[g2/e] < g3 < L9[g2/e],	2.2
	L10[g2/e][g3/d] = L14[g2/e1] e L11[g2/e][g3/d] = L15[g2/e1]	2.3
11	L6 < g3 < L7, L8[g3/e] < g1 < L9[g3/e],	2.2
	L10[g3/e][g1/d] = L14[g3/e1] e L11[g3/e][g1/d] = L15[g3/e1]	2.3
18	L10 < d < L11 e {} < e	2.2, 2.1
19	L6 < e < L7, L8[e/e] < d < L9[e/e],	2.2
	L10[e/e][d/d] = L14[e/e] e L11[e/e][d/d] = L15[e/e]	2.3
25	L16 < e1 < L17, L18[e1/e2] < d1 < L19[e1/e2] e {e1} < d1	2.2, 2.1
26	L12 < e1 < L13 e L14[e1/e1] < d1 < L15[e1/e1]	2.2
32	L12 < e2 < L13	2.2
33	L6 < e2 < L7, L8[e2/e] < d2 < L9[e2/e],	2.2
	L10[e2/e][d2/d] = L14[e2/e1] e L11[e2/e][d2/d] = L15[e2/e1]	2.3

do fio de execução são r1:**lock** g1, r2:**lock** g2 e r6:∀ d1::**Lock**.{r1:**lock** g1 ....} . Como o tipo do registo r6 de levantarGarfoEsquerdo, após a instanciação por g1 e g2 é r6:∀ l ::**Lock**.{r1:**lock** g1 ...} há que gerar as restrições L10[g1/e][g2/d]=L14[g1/e1] e L11[g1/e][g2/d]=L15[g1/e1], em que L10 e L11 são as variáveis associadas ao trinco polimórfico l do bloco levantarGarfoEsquerdo e L14 e L15 são as associadas ao trinco polimórfico d1 do bloco levantarGarfoDireito .

Por último, as restrições recolhidas são passadas a um SMT que afere da sua consistência. Caso sejam consistentes, o SMT indica um modelo que instancia cada um dos conjuntos associados aos trincos, obtendo deste modo uma anotação válida; caso contrário, o SMT responde negativamente e o programa não tem qualquer anotação possível.

5 Implementação sobre Z3

Os programas MIL declaram um conjunto finito de trincos que pode ser determinado em tempo de compilação. Portanto, os conjuntos (Li) que pretendemos determinar são finitos e definidos sobre um conjunto de trincos também finito.

No SMT que utilizámos, o Z3 [10], representamos os trincos e os conjuntos de trincos usando a teoria *vetores de bits*. O tamanho do vetor de *bits* coincide com o número de trincos declarado no programa. A cada trinco é atribuído um *bit* diferente, uma potência de 2. Um conjunto de trincos é representado por um vetor onde os *bits* correspondentes aos trincos que pertencem ao conjunto têm valor 1 e os restantes 0. O programa da figura 2 contém dez trincos: g1, g2, g3, e, d, l, e1, d1, e2 e d2. Os trincos são representados por constantes. Por exemplo, em Z3 atribuímos ao trinco g1 do bloco main a constante #b0000000100, enquanto que ao trinco d do bloco levantarGarfoEsquerdo atribuímos #b0001000000. Outra qualquer escolha de constantes para os trincos seria válida, desde que sejam usados valores distintos para cada trinco. Para representar o conjunto L0 usamos a definição (**declare−const** L0 (_ BitVec 10)), que declara uma constante com o nome da variável (L0) e o seu tipo (um vetor de *bits* de tamanho coincidente com o número de trincos do programa).

Para representar as restrições há que implementar predicados e funções que nos permitam escrever, por exemplo, que um trinco é menor do que um conjunto de trincos ou efetuar uma substituição de um trinco por outro num dado conjunto, de acordo com a representação que escolhemos para os trincos e para os conjuntos de trincos. O excerto seguinte define o predicado subset e as funções remove e add, que são utilizados para definir a função de substituição subs.

```
(define−fun subset ((e (_ BitVec 10)) (S (_ BitVec 10))) Bool
   (= (bvand e S) e))
(define−fun remove ((l (_ BitVec 10)) (S (_ BitVec 10))) (_ BitVec 10)
   (bvand (bvxor All  l) S))
(define−fun add ((l (_ BitVec 10)) (S (_ BitVec 10))) (_ BitVec 10)
   (bvadd l S))
(define−fun subs ((l(_ BitVec 10)) (m(_ BitVec 10)) (S(_ BitVec 10))) (_ BitVec 10)
    ( if  (subset  m S) (add l  (remove m S)) S))
```

Para cada predicado e função, a primeira linha define a sua assinatura e a linha seguinte o seu corpo, escrito numa notação prefixa. As expressões bvand, bvxor e bvadd fazem parte da teoria do Z3 para vetores de *bits*. Abordamos somente a função subs que implementa a substituição de m por l em S (S[m/l]). A assinatura indica que a função tem três parâmetros l, m e S todos do tipo BitVec 10, tal como o retorno da função, que é definido a seguir aos três parâmetros. O corpo do predicado remove m e adiciona l a S, caso m pertença a S. Caso contrário, comporta-se como a função identidade. Omitimos as funções lockLessThanSet, setLessThanLock e setLessThanSet por falta de espaço. As funções e predicados que definimos são geradas para cada programa, pois o seu tipo depende do número de trincos do programa e o Z3, como a generalidade dos SMT, não aceita definições polimórficas. As funções cujo código omitimos definem também de forma explícita (com recurso a expressões condicionais) a associação entre trincos e variáveis sobre conjuntos, pois o Z3 não tem modo de especificar relações de dependência funcional entre diferentes valores que permitam estabelecer esta associação de forma compacta e eficiente.

Tabela 3. O algoritmo numa bancada de trabalho

Programa	Anot	MIL(loc)	#trincos	#restrições	Z3(loc)	Tempo(s)	Mem(Mb)	SAT?
Filósofos3	Sim	35	10	110	221	0,20	3,20	Sim
Filósofos2	Sim	35	10	110	221	0,21	3,29	Não
LibPi	Não	144	12	125	244	0,41	3,63	Sim
Filósofos1	Não	30	9	73	176	0,49	3,91	Não
MOOL1	Não	200	25	151	399	2,93	8,01	Sim
ProdCons	Não	270	50	398	821	54,44	52,50	Sim
MOOL3	Sim	380	74	658	1273	70,12	23,24	Sim
MOOL2	Não	380	74	510	1125	127,51	95,71	Sim

As restrições geradas são adicionadas usando a expressão **assert**. Exemplificamos como escrever em Z3 as restrições geradas pelo exemplo da secção anterior. A restrição `L0` $<$ `g1` é representada por (**assert** (setLessThanLock L0 #b0000000100)), onde `g1` foi substituido pela constante respetiva; a restrição `L2` $\subseteq$ {`g1`} por (**assert** (subset L2 #b0000000100)); e, por último, a restrição {`e1`} $<$ `d1` é representada por (**assert** (or (subset #b0000100000 L14) (subset #b0000010000 L13))). O trinco `e1` está associado às variáveis `L12` e `L13` e `d1` às variáveis `L14` e `L15`. Para `e1` ser menor do que `d1` tem de ser verdade que `d1` $\in$ `L13` ou `e1` $\in$ `L14`. O Z3 irá explorar todo o espaço de procura para tentar encontrar um modelo que satisfaça as asserções indicadas. Se não encontrar, responde `unsat`. Infelizmente, esta resposta não é suficiente para dar informação ao programador sobre os trincos envolvidos no impasse.

A tabela 3 apresenta o resultado de várias experiências que conduzimos numa máquina Windows 7 com processador *quad core* Intel i7 a 1,73 MHz e 8 GBytes de memória. Entre os vários programas, Filósofos1 e Filósofos2 correspondem às figuras 1–2; Filósofos3 tem a alteração que indicámos na secção 3 e que elimina o impasse. LibPi é uma biblioteca de apoio a um compilador do cálculo-π em MIL [8]. ProdCons corresponde ao problema tradicional do Produtor-Consumidor. MOOL1 e MOOL2 correspondem a código MIL gerado por um compilador por nós desenvolvido para uma linguagem de objetos concorrentes [6], sobre um programa contendo 2 classes e 60 linhas de código e outro contendo 3 classes e 144 linhas de código. MOOL3 é idêntico ao MOOL2, mas apresenta-se totalmente anotado. Da tabela podemos observar que o crescimento, tanto em tempo quanto em memória, parece ser exponencial no número de trincos, o que não será de estranhar. Verificamos também que a análise de um programa contendo anotações (MOOL3) é mais rápida do que a de o mesmo programa sem anotações (MOOL2), tal como esperado.

6 Conclusão

Neste artigo apresentámos um algoritmo para inferir anotações polimórficas sobre trincos de modo a assegurar que um programa MIL passa no sistema de tipos descrito em [15]. Por sua vez, o sistema de tipos garante a ausência de impasses para programas bem tipificados. O algoritmo recolhe restrições locais

sobre a ordem do fecho de trincos e passa-as para um SMT para verificação da sua consistência. Testámos intensamente o algoritmo com vários programas MIL, programas desenvolvidos diretamente em MIL mas também resultantes de um compilador de uma linguagem de objetos concorrentes em MIL. Estes testes, embora nos transmitam confiança de que o algoritmo funciona, não são suficientes para provar a correção e completude do algoritmo. A prova formal destes resultados será o foco do nosso trabalho futuro.

Agradecimentos. Os autores agradecem os excelentes comentários dos revisores anónimos. Este trabalho foi suportado pela FCT através do projeto Advanced Type Systems for Multicore Programming (PTDC/EIA-CCO/122547 /2010) e do programa multianual LaSIGE (PEst-OE/EEI/UI0408/2011).

Referências

1. R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *PADTAD'06*, pages 51–60. ACM, 2006.
2. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.
3. A. Birrell. An introduction to programming with threads. Technical Report 35, Digital Systems Research Center, 1989.
4. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. *SIGPLAN Not.*, 37(11):211–230, 2002.
5. G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder - second generation of a Java model checker. In *Workshop on Advances in Verification*, 2000.
6. J. Campos and V. T. Vasconcelos. Channels as objects in concurrent object-oriented programming. In *PLACES'10*, volume 69 of *EPTCS*, pages 12–28, 2011.
7. E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.
8. T. Cogumbreiro, F. Martins, and V. T. Vasconcelos. Compiling the pi-calculus into a multithreaded typed assembly language. *ENTCS*, 241:57–84, 2009.
9. D. Cunningham, S. Drossopoulou, and S. Eisenbach. Lock inference proven correct. In *FTfJP'08*, 2008.
10. L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
11. C. Flanagan and M. Abadi. Types for safe locking. In *ESOP'99*, volume 1576 of *LNCS*, pages 91–108. Springer, 1999.
12. C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, 1985.
13. K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *APLAS'08*, volume 5356 of *LNCS*, pages 155–170. Springer, 2008.
14. V. T. Vasconcelos and F. Martins. A multithreaded typed assembly language. In *Multithreading in Hardware and Software: Formal Approaches to Design and Verification*, pages 133–141, 2006.
15. V. T. Vasconcelos, F. Martins, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded typed assembly language. In *PLACES'09*, volume 17 of *EPTCS*, pages 95–109, 2010.