Um sistema de prova para as primitivas não bloqueantes LL/SC

Diogo S. Serra António Ravara

CITI e Departamento de Informática
FCT, Universidade Nova de Lisboa

Resumo Na programação concorrente, as *locks* controlam por serialização o acesso a recursos partilhados. Para tirar melhor partido da concorrência, surgiram mecanismos não bloqueantes como o par de primitivas `LL/SC` (`load-link/store-conditional`) ou a menos expressiva mas muito utilizada `CAS` (`compare-and-swap`). Apresenta-se um sistema de prova composicional, baseado em lógica de separação concorrente e suportando raciocínio "rely-guarantee", para uma linguagem imperativa concorrente com memória partilhada, estruturas de dados mutáveis e o par `LL/SC`. Desenvolve-se uma semântica operacional estrutural para a linguagem e propõe-se uma noção de validade que permite garantir programas sem corridas pelos recursos e que são correctos face a objectivos especificados como invariantes. Provam-se correctos programas de inserção e remoção numa lista, ilustrando a expressividade da proposta.

1 Introdução

Contexto e problema. Os programas concorrentes usam hoje novas formas de garantir a coerência das estruturas de dados partilhadas, recorrendo a primitivas "lock-free" em vez das tradicionais (baseadas em semáforos, monitores ou "locks"), para tirar partido das arquitecturas *multi-core* e para que haja um uso intensivo de recursos partilhados. Os algoritmos não bloqueantes usam uma abordagem optimista e tentam aceder aos recursos partilhados apesar das possíveis interferências, "recuando" em caso de inconsistência. Como explicado por Herlihy [4], consegue-se assim melhores performances. Há várias primitivas linguísticas para suportar programação não bloqueantes. Uma das mais usadas é a `CAS`– "compare-and-swap" – que verifica se o valor de dada variável se alterou desde a última vez que, no processo ou "thread" em questão, foi lida; se não se alterou, então a variável é modificada. Foi já proposto um sistema de prova para programas com esta primitiva [3], mas um problema de expressividade fez-nos optar por outra. O `CAS` não detecta modificações conflictuais desde que o valor inicial da variável testada seja reposto antes do teste. É necessário ter tantos testes `CAS` quanto a dimensão da estrutura que se pretende manter coerente. Escolhemos por isso estudar o par de primitivas `LL/SC`– "load-link" e "store-conditional", que não sofre do problema referido, pois funcionam como uma transacção: o load-link lê a variável e o store-conditional só escreve nela se o seu valor se manteve inalterado desde o último load-link. As primitivas `LL/SC`

capturam assim melhor a natureza optimista da programação não bloqueante, funcionando analogamente às transacções: `LL` marca o início; `SC` faz o "commit" ou o "rollback".

O nosso objectivo é desenvolver um sistema de prova axiomático, no estilo de Floyd e Hoare [2, 5], para uma linguagem imperativa usual com memória partilhada e estruturas de dados mutáveis, com as primitivas `LL`/`SC`, e com concorrência por um comando de composição paralela (de comandos sequenciais). As regras de inferência estão desenhadas para permitir raciocínio modular e local, facilitando construir provas de correcção. Os exemplos apresentados no fim do artigo ilustram isso.

Contribuições. Não existindo nem uma linguagem com as características escolhidas, nem um sistema de prova com os requisitos colocados, vários trabalhos foram inspiradores. As contribuições deste trabalho são as seguintes: (1) uma *semântica operacional estruturada* (indutiva e orientada pela sintaxe [9]) que captura rigorosamente o efeito no estado de dado programa das primitivas `LL`/`SC`; (2) uma *linguagem de asserções e sua semântica*, estendendo a Lógica de Separação [10] e distinguindo o que acontece em pedaços disjuntos de memória – os recursos locais do programa e os recursos partilhados pelo ambiente, como em [12]; (3) um sistema de prova com *regras de inferência* para cada construtor da linguagem e "framing rules" que permitem estender uma prova local a um ambiente sem interferência (conseguindo-se assim fazer provas locais e modulares); e (4) uma noção de *validade* que garanta que um programa que não falha (*i.e.*, que não acede a memória não alocada) não modifica variáveis em uso por outros, e que se executado num estado que satisfaz dada pré-condição, se terminar (sem abortar), estabelece certa pós-condição [11].

2 Trabalho relacionado

Tanto quanto sabemos, apresenta-se aqui o primeiro sistema axiomático de prova de programas imperativos concorrentes com estruturas de dados mutáveis usando as primitivas não bloqueantes `LL`/`SC`. Vários ingredientes vêm directamente de, ou foram inspirados por, outras propostas.

A lógica de Floyd-Hoare. No final dos anos sessenta, surgem propostas pioneiras para formalizar sistemas de prova para raciocinar sobre o comportamento de sistemas e programas. Floyd propõe o uso de asserções lógicas para verificar programas [2], e Hoare define um sistema de prova para garantir a correcção de programas sequenciais imperativos face ao seu objectivo, especificado em Lógica de Primeira Ordem [5].

Os ingredientes essenciais para a definição de um sistema axiomático de prova da correcção parcial de programas à la Hoare são: uma sintaxe precisa, uma semântica rigorosamente definida, uma linguagem de asserções e um sistema de prova para garantir propriedades de correcção (em relação a objectivos especificados na linguagem de asserções). O sistema apresentado neste artigo segue à risca este programa de trabalho.

Lógica de separação. Introduzida por Reynolds em 2002 [10] , é uma extensão da lógica de Hoare destinada a provar correctos programas que manipulam estruturas de dados mutáveis, *i.e.*, com ponteiros. A manipulação de memória, directa ou indirecta, é muito comum nas linguagens utilizadas hoje em dia, sendo por isso necessário ter primitivas lógicas para raciocinar sobre esta componente. As novas primitivas são a constante `emp` que declara uma "heap" vazia, um operador de *separação* que declara que o "heap" pode ser dividido em duas porções disjuntas, cada uma satisfazendo um dos argumentos do operador, e o operador *points to* que declara que o "heap" é singular e o endereço contém o valor indicado.

Note-se que no cenário considerado há execuções de programas que podem abortar se se tenta realizar operações em endereços de memória que não estão no domínio do "heap". Outro aspecto é que a execução já não é determinista, pois a alocação de memória devolve um qualquer bloco contíguo de endereços.

Um triplo de Hoare válido garante agora adicionalmente que o programa não aborta. Para se construirem provas composicionais é importante fazer raciocínio "local", *i.e.*, focado nos endereços de memória usados pelo programa. A regra de enquadramento ("frame") permite extender o "heap" considerado com blocos de endereços que não interferem com os inicialmente considerados.

Lógica de separação concorrente. Introduzida por O'Hearn em 2007 [1, 8], pretende provar correctos programas concorrentes com estruturas de dados mutáveis partilhados. A noção de recurso é central, havendo primitivas para a sua criação, aquisição e uso exclusivo.

Interessa agora garantir que os programas não têm "corridas pelos recursos" (data races) – em que um programa modifica uma variável ou endereço de memória usado por outro – as regras do sistema de prova consideram um invariante sobre o estado partilhado que é um quarto argumento do juízo de Hoare.

Uma limitação desta lógica é não permitir nenhum tipo de interferência. Em 1981 Jones propôs o raciocínio "rely-guarantee" para provar correctos programas concorrentes com interferência [6, 7]. A ideia é que se o ambiente onde o programa executa respeitar a condição "rely", então o programa respeita a relação "guarantee" entre os "heaps" inicial e final. O método não se baseia na abordagem que aqui seguimos e não permite manipular o "heap". A lógica `RGSep` proposta por Vafeiadis [12] visa combinar "rely-guarantee" com a lógica de separação concorrente, obtendo-se uma forma de controlar a interferência que dispensa as primitivas para manipular e controlar recursos. As asserções da lógica de separação verificam-se no "heap" local e as fórmulas sobre o ambiente são verificadas no estado partilhado.

Neste trabalho adoptamos o raciocínio "rely-guarantee", tornando-o local, e introduzimos asserções específicas para tratar estado partilhado. São ainda necessários elementos para lidar com a natureza optimista da concorrência não bloqueante.

$C_b ::=$		Basic Commands	$C_s ::=$		Sequential Commands
	$\texttt{skip}$	(skip)		C_b	(basic command)
	$V := E$	(assignment)		$\texttt{if } B \texttt{ then } C_s \texttt{ else } C_s$	(conditional)
	$V := \texttt{alloc}(E, \ldots, E)$	(allocate)		$\texttt{while } B \texttt{ do } C_s$	(while)
	$V := [V]$	(load)		$C_s; C_s$	(sequential composition)
	$V := \texttt{LL}(V)$	(load-link)			
	$[V] := V$	(store)			
	$V := \texttt{SC}(V, V)$	(store-conditional)			
	$\texttt{disp}(V)$	(dispose)			

$C_c ::=$		Concurrent Commands	$C_t ::=$		Threaded Command
	C_s	(sequential command)		$C_c @ \mathsf{ThreadId}$	(Single thread)
	$C_c \parallel C_s$	(parallel composition)		$C_t \parallel C_c @ \mathsf{ThreadId}$	(Multiple threads)

Figura 1. Sintaxe da linguagem de programação

$$\mathit{Config} \triangleq C_t \times [\mathsf{ThreadId} \rightharpoonup \wp(\mathit{Address})] \times (\mathit{Store} \times \mathit{Heap}) \cup \{\texttt{abort}\}$$

Figura 2. Configurações

3 Sintaxe e semântica da linguagem de programação

Sintaxe. A Figura 1 apresenta a linguagem: os comandos básicos, gerados por C_b, incluindo construtores para manipular "store" e "heap", e as primitivas LL/SC para controlar a concorrência; a produção C_s inclui os comandos imperativos usuais e a C_c permite a composição concorrente dos comandos anteriores. Como é necessária informação local para modelar o comportamento das primitivas LL/SC, não permitir o aninhamento da composição paralela facilita as definições.

Semântica operacional. A Figura 2 apresenta as configurações, manipuladas pela função de transição que modela a execução de programas. A Figura 3 contém as regras mais significativas. Para executar LL/SC, como alguns programas podem ter uma visão actualizada do conteúdo de um endereço de memória e outros não, é necessário saber que endereços não foram modificados desde terem sido pela última vez "load-linked". A semântica das asserções em RGSep usa um parâmetro adicional para separar a memória partilhada da local, mas como os programas não bloqueantes não usam esta distinção, permitindo acessos concorrentes, a ideia não resolve o nosso problema. Como os "threads" são um conceito utilizado em programação concorrente, que paraleliza programas mas permite assumir recursos partilhados, são um conceito útil no nosso cenário. Por isso as nossas configurações têm um componente que mapeia "threads" em conjuntos de endereços de memória que não foram modificados desde terem sido pela última vez "load-linked". Esta opção permite especificar, na definição de validade, de que endereços um "thread" tem sempre uma visão actualizada.

$$\rightsquigarrow : \mathit{Config} \times \mathit{Config} \qquad \text{Program transition}$$

$$\frac{s(y) = a \wedge a \in \mathsf{dom}(h) \wedge l(i) = l_i}{\langle [x := \mathtt{LL}(y)]@i, l, (s,h)\rangle \rightsquigarrow \langle \mathtt{skip}@i, l[i \mapsto l_i \cup \{a\}], (s[x \mapsto h(a)], h)\rangle} \quad \text{(load-link)}$$

$$\frac{s(y) \notin \mathsf{dom}(h)}{\langle [x := \mathtt{LL}(y)]@i, l, (s,h)\rangle \rightsquigarrow \mathtt{abort}} \quad (\text{load-link}_A)$$

$$\frac{s(z) = a \wedge a \in \mathsf{dom}(h) \wedge a \in l(i)}{\langle [x := \mathtt{SC}(y,z)]@i, l, (s,h)\rangle \rightsquigarrow \langle \mathtt{skip}@i, l \downarrow a, (s[x \mapsto 1], h[a \mapsto s(y)])\rangle} \quad (\text{store-conditional}_T)$$

$$\frac{s(z) = a \wedge a \in \mathsf{dom}(h) \wedge a \notin l(i)}{\langle [x := \mathtt{SC}(y,z)]@i, l, (s,h)\rangle \rightsquigarrow \langle \mathtt{skip}@i, l, (s[x \mapsto 0], h)\rangle} \quad (\text{store-conditional}_F)$$

$$\frac{s(z) \notin \mathsf{dom}(h)}{\langle [x := \mathtt{SC}(y,z)]@i, l, (s,h)\rangle \rightsquigarrow \mathtt{abort}} \quad (\text{store-conditional}_A)$$

Figura 3. Semântica operacional: regras para LL/SC

Relativamente a outras abordagens, a diferença fundamental da semântica proposta é que os comandos que modificam a "heap" devem invalidar a visão que o "thread" tem dos endereços modificados, e a composição paralela deve criar um novo "thread" com o seu próprio conjunto de localizações e fundir threads que acabam de executar. Para se raciocinar sobre "data races" (corridas pelos recursos partilhados – localizações do "heap"), dada a natureza optimista da programação não bloqueante, um programa não aborta ao haver acessos concorrentes e conflictuais a uma localização. A Figura 3 apresenta as cinco regras das primitivas LL/SC. A primeira define o comportamento normal do *load-link*: copia o valor no "store" de uma variável para o conteúdo da memória da localização da outra, juntando essa localização ao conjunto das que o "thread" tem uma visão actualizada. As transições *store-conditional*$_T$ e *store-conditional*$_F$ colocam condicionalmente y no endereço de memória z, guardando o resultado em x; ou têm sucesso (o "thread" tem uma visão actualizada da localização z), removendo a localização dos conjuntos associados aos "threads" e colocando o valor de x a 1, ou falha e coloca o valor de x a 0. As restantes regras (*load-link*$_A$ e *store-conditional*$_A$) fazem os comandos abortar se tentam manipular endereços de memória não alocados. As regras usam definições auxiliares: além dos operadores $f[x \mapsto v]$, $\mathsf{dom}(f)$, $\mathsf{cdom}(f)$ e $\rceil$, o operador $l \downarrow a$ remove a localização a de todos os conjuntos de endereços de memória associados aos "threads" (em l).

$$\downarrow \;\; : ([\mathsf{ThreadId} \rightharpoonup \mathfrak{P}(\mathit{Address})] \times \mathit{Address}) \to [\mathsf{ThreadId} \rightharpoonup \mathfrak{P}(\mathit{Address})]$$
$$l \downarrow a \triangleq \{t \mapsto (l_t \setminus \{a\}) \mid t \mapsto l_t \in l\}$$

Para raciocinar sobre corridas pelos recursos do "store", usam-se as funções $\mathsf{assigns}(c_t)$ e $\mathsf{reads}(c_t)$, que referem o conjunto de variáveis que dado "thread" *num passo* pode escrever ou ler. Estas funções diferem de $\mathsf{writes}(c)$ e $\mathsf{fv}(c)$ porque estas tomam em consideração toda a execução de um programa.

$$
\begin{array}{lll|lll}
P_{sl} ::= & & \text{Asserções da Lógica de separação } P ::= & & \text{Assertions} \\
\quad | & B & \text{(boolean expression)} & | & P_{sl} & \text{(local assertion)} \\
\quad | & P_{sl} \wedge P_{sl} & \text{(conjunction)} & | & {}^{V}\boxed{P_{sl}} & \text{(shared assertion)} \\
\quad | & \neg P_{sl} & \text{(negation)} & | & P * P & \text{(separation)} \\
\quad | & \texttt{emp} & \text{(empty)} & | & \exists_V\, P & \text{(existential quantification)} \\
\quad | & P_{sl} * P_{sl} & \text{(local separation)} & & & \\
\quad | & E_1 \mapsto E_2 & \text{(points to)} & & & \\
\quad | & \exists_V\, P_{sl} & \text{(existential quantification)} & & &
\end{array}
$$

$$
\begin{array}{rl}
e \mapsto - \triangleq \exists_x\, e \mapsto x, \text{where } x \notin \mathsf{fv}(e) & \text{(anonymous)} \\
e \mapsto e_1, \ldots, e_n \triangleq e \mapsto e_1 * \ldots * e + n - 1 \mapsto e_n & \text{(multi-valued)}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{Resource} \subset \mathfrak{P}(P_{sl}) & \mathrm{res} : \mathsf{Resource} \to V \\
\{\} \in \mathsf{Resource} & \mathsf{res}(\{\}) \triangleq \varnothing \\
\Gamma \in \mathsf{Resource} \Rightarrow \Gamma \uplus \{\exists_v\, x \mapsto v * p\} \in \mathsf{Resource} & \mathsf{res}(\Gamma \uplus \{\exists_v\, x \mapsto v * p\}) \triangleq \mathsf{res}(\Gamma) \cup \{x\}
\end{array}
$$

Figura 4. Sintaxe da linguagem de asserções

4 Lógica: sintaxe, semântica e sistema de prova

Tendo uma forma rigorosa de descrever a execução de programas, define-se agora a linguagem das propriedades das execuções, para dizer o que se verifica num estado. Como o sistema de prova lida com programas que manipulam "store" e "heap", usa-se a lógica de separação. Dada à natureza optimista da programação não bloqueante não se distinguem recursos: nem a lógica de separação concorrente nem `RGSep` são directamente aplicáveis.

Asserções. A Figura 4 apresenta a gramática das asserções (a linguagem para escrever propriedades dos estados). A primeira produção define as asserções da lógica de separação; a segunda gera as novas fórmulas, sendo o elemento essencial as asserções partilhadas – escritas numa caixa e com a indicação de uma variável – que dizem que, se o endereço de memória associado à variável não foi alterado, então a propriedade (sobre o seu valor) verifica-se. Antes de apresentar a semântica, é necessária uma definição sintáctica: a lógica de separação concorrente usa um componente nas asserções para lidar com a parte partilhada do "heap" – o contexto dos recursos, Γ. Considera-se então um contexto para gerir a interferência e raciocinar sobre dados partilhados. Os contextos são conjuntos de asserções, cada uma dizendo o que se verifica numa parte do "heap" controlada pela variável referida, clarificando o que se verifica no estado partilhado.

Em resumo, as asserções da lógica de separação permitem raciocinar sobre dados da "store" e do "heap"; os contextos usam lógica de separação concorrente, adaptada para que variáveis controlem o acesso aos recursos partilhados. As asserções nas caixas falam das propriedades de um endereço de memória não modificado.

$$
\begin{array}{l}
\mathcal{P} : P \times \mathfrak{P}\,(\mathit{Address}) \times (V \rightharpoonup \mathit{Heap}) \times \mathit{State} \to \{\top, \bot\} \\
\mathcal{P}[\![p_{sl}]\!]_{(\delta, h_\Gamma, s, h_l)} \triangleq \mathcal{P}_{sl}[\![p_{sl}]\!]_{(s,h_l)} \\
\mathcal{P}[\![^{x}\boxed{p_{sl}}]\!]_{(\delta, h_\Gamma, s, h_l)} \triangleq h_l = \varnothing \wedge h_\Gamma = h'_\Gamma \uplus \{x \mapsto h_x\} \wedge \big(s(x) \in \delta \Rightarrow \mathcal{P}_{sl}[\![p_{sl}]\!]_{(s,h_x)}\big) \\
\mathcal{P}[\![p_1 * p_2]\!]_{(\delta, h_\Gamma, s, h_l)} \triangleq h_l = h_{l_1} \uplus h_{l_2} \wedge h_\Gamma = h_{\Gamma_1} \uplus h_{\Gamma_2} \wedge \mathcal{P}[\![p_1]\!]_{(\delta, h_{\Gamma_1}, s, h_{l_2})} \wedge \mathcal{P}[\![p_2]\!]_{(\delta, h_{\Gamma_2}, s, h_{l_2})} \\
\mathcal{P}[\![\exists_x p]\!]_{(\delta, h_\Gamma, s, h_l)} \triangleq \exists_v\, \mathcal{P}[\![p]\!]_{(\delta, h_\Gamma, s[x \mapsto v], h_l)}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{R} : \mathsf{Resource} \times (V \rightharpoonup \mathit{Heap}) \times \mathit{Store} \to \{\top, \bot\} \\
\mathcal{R}[\![\{\}]\!]_{(h,s)} \triangleq h = \varnothing \\
\mathcal{R}[\![\Gamma \uplus \{\exists_v\, x \mapsto v * p\}]\!]_{(h,s)} \triangleq h = h_\Gamma \uplus \{x \mapsto h_x\} \wedge \mathcal{R}[\![\Gamma]\!]_{(h_\Gamma, s)} \wedge \mathcal{P}_{SL}[\![\exists_v x \mapsto v * p]\!]_{(s,h_x)}
\end{array}
$$

Figura 5. Semântica das asserções

Validade. A Figura 5 apresenta a semântica das asserções e dos contextos: o predicado $\mathcal{P}$ define quando se verifica dada asserção. Tem como argumentos: (1) um conjunto de localizações de memória partilhadas que estão actualizadas, para verificar as propriedades a si associadas; (2) uma função que associa a variáveis as porções do "heap" que controlam, para verificar as asserções partilhadas; e (3) a "store", partilhada mas com uso disjunto (para impedir corridas), e o "heap" local, para referir dados manipulados exclusivamente por um "thread".

À semelhança do `RGSep`, as propriedades da lógica de separação são tratadas como originalmente, usando a "store" e o "heap" local. A quantificação existencial no nível "superior" é a usual, verificando se existe alguma "store" que torna a propriedade verdadeira. As asserções partilhadas forçam o "heap" local a ser vazio para impedir a sua extensão, pois o construtor só lida com recursos partilhados. A separação parte "heaps" partilhados (além dos locais), de forma a garantir composicionalidade.

Os contextos dos recursos são conjuntos de propriedades que falam sobre a parte partilhada do "heap". Ao contrário das asserções partilhadas, que podem não se verificar se um endereço foi modificado por outro "thread", estes contextos são *invariantes* que referem propriedades que se devem verificar sempre no estado partilhado e a que todos os "threads" devem obedecer. Como a avaliação do contexto vazio se faz com o "heap" vazio, todo o "heap" partilhado tem que ser considerado, enquanto que nas asserções partilhadas se pode ignorar partes do estado partilhado não relevantes que estão consideradas no contexto. O passo indutivo verifica dada propriedade na partição do "heap" partilhado, controlada por certa variável. Como na literatura só se refere um estado do programa ou contextos arbitrários de recursos, propõe-se uma nova definição.

$$
\begin{array}{l}
\Gamma \vDash \delta_1\{p\}c\{q\}\delta_2 \text{ if and only if for all } l, i, l_i, s, h = h_s \uplus h_l \text{ and } h_\Gamma \in \text{partitions}(h_s) \\
\quad \text{whenever } \mathcal{R}[\![\Gamma]\!]_{(h_\Gamma, s)}, \mathcal{P}[\![p]\!]_{(l_i, h_\Gamma, s, h_l)} \text{ and } l_i \subseteq \text{eval}_{\mathcal{E}}(\delta_1, s) \text{ then} \\
\qquad \text{not } \langle c@i, l[i \mapsto l_i], (s,h)\rangle \rightsquigarrow^* \texttt{abort} \text{ and} \\
\qquad \text{if } \langle c@i, l[i \mapsto l_i], (s,h)\rangle \rightsquigarrow^* \langle \texttt{skip}@i, l', (s',h')\rangle \text{ then there exists } h'_s, h'_l \text{ and } h'_\Gamma \\
\qquad\quad \text{such that } h' = h'_s \uplus h'_l, h'_\Gamma \in \text{partitions}(h'_s) \text{ and} \\
\qquad\quad\quad \mathcal{R}[\![\Gamma]\!]_{(h'_\Gamma, s')}, \mathcal{P}[\![q]\!]_{(l'(i), h'_\Gamma, s', h'_l)} \text{ and } l'(i) \subseteq \text{eval}_{\mathcal{E}}(\delta_2, s')
\end{array}
$$

$\text{partitions} : \mathit{Heap} \to \mathfrak{P}(V \rightharpoonup \mathit{Heap})$
$\text{partitions}(h) \triangleq \{h_\Gamma \in V \rightharpoonup \mathit{Heap} \mid \text{disjoint}(\mathsf{cdom}(h_\Gamma)) \wedge h = \bigcup \mathsf{cdom}(h_\Gamma)\}$

$\text{disjoint} : \mathfrak{P}(\mathit{Heap}) \to \{\top, \bot\}$
$\text{disjoint}(h) \triangleq \forall_{s_1 \in h} \forall_{s_2 \in h}\, s_1 \neq s_2 \Rightarrow \mathsf{dom}(s_1) \cap \mathsf{dom}(s_2) = \varnothing$

$\text{eval}_\mathcal{E} : [\mathfrak{P}(E) \times \mathit{Store}] \to \mathfrak{P}(\mathit{Value})$
$\text{eval}_\mathcal{E}(\delta, s) \triangleq \{\mathcal{E}[\![e]\!]_s \mid e \in \delta\}$

O predicado $\vDash$ usa um contexto de recursos Γ que declara o que se verifica na parte partilhada do "heap", um conjunto δ_1 de variáveis que foram "load-linked", uma pré-condição p que declara o que se verifica no início da execução do programa c (que se está a verificar), uma pós-condição q que se deve verificar após a execução do programa, bem como um conjunto final δ_2 de variáveis "load-linked". A definição, inspirada pela de Vafeiadis [13], diz que se o programa começa a ser executado com um dado conjunto de variáveis "load-linked" cujo valor conhecido está actualizado, com dado "store" e "heap" que podem ser partidos de forma que parte do "heap" satisfaça o contexto dos recursos e as asserções partilhadas e outra parte satisfaça as propriedades locais da pré-condição, então o programa não falha e termina numa configuração que se pode separar como previamente (mas considerando agora a pós-condição) e que tem no conjunto final de variáveis "load-linked" as que se mantêm actualizadas.

Em resumo, as asserções partilhadas permitem raciocinar sobre "threads" com a premissa que dado endereço não foi alterado, enquanto os invariantes de recursos se verificam sempre na partição do "heap" partilhada, que todos os "threads" devem preservar.

Regras de inferência. O sistema de prova que aqui se apresenta permite raciocinar sobre programas de forma composicional, orientada pela sintaxe. Introduz uma noção de validade *sintáctica* – o predicado $\vdash$ – definido indutivamente por regras – a Figura 6 apresenta as mais relevantes. Tal como a validade semântica, toma um contexto de recursos Γ, um conjunto de variáveis previamente "load-linked", δ, uma pré-condição p, um programa c que se quer verificar, uma pós-condição q e um conjunto final δ' de variáveis "load-linked".

Como grande parte das linguagens (de programação e de asserções) é comum com a lógica de separação concorrente ou a `RGSep`, muitos dos elementos formais são idênticos. Lida-se adicionalmente com asserções partilhadas para tratar a natureza optimista das primitivas `LL`/`SC`, e todas as regras proíbem a atribuição a variáveis usadas no contexto dos recursos, pois isso pode conduzir a conflitos. Note-se que todas as regras (excepto `skip`) usam "heaps" e conjuntos de variáveis "load-linked" mínimos, pois podem ser alterados pelas regras de composição (dando assim carácter local às regras). Pode-se portanto focar apenas na porção do "heap" relevante para o programa (a sua "pegada" de memória), que se estende se necessário. Lida-se então com as alterações locais produzidas pelos sub-programas, obtendo-se uma prova do programa todo pela composição das regras. A ideia central do sistema de prova é a transferência de conhecimento entre

$$\frac{\Gamma \uplus \{\exists_v\, x \mapsto v * f\} \vdash \delta\{p\}\, c\, \{q\}\delta'}{\Gamma \vdash \delta\{p * \exists_v\, x \mapsto v * f\}\, c\, \{q * \exists_v\, x \mapsto v * f\}\delta'} \quad \text{if } x \notin \delta' \qquad \text{(shared resource)}$$

$$\frac{}{\Gamma \vdash \{z\}\{^{z}\boxed{p[e/x] \wedge y \mapsto e}\}x := [y]\{^{z}\boxed{p \wedge y \mapsto e}\}\{z\}} \quad \text{if } x \neq y \text{ and } x \notin \mathsf{fv}(\Gamma, e) \qquad \text{(shared load)}$$

$$\frac{}{\Gamma \uplus \{\exists_v\, y \mapsto v * f\} \vdash \{\}\{\mathtt{emp}\}\, x := \mathtt{LL}(y)\, \{^{y}\boxed{f[x/v]}\}\{y\}} \quad \text{if } x \neq y \text{ and } x \notin \mathsf{fv}(\Gamma, f) \qquad \text{(load-link)}$$

$$\frac{p' * q' \Rightarrow f[y/v]}{\begin{array}{c}\Gamma \uplus \{\exists_v\, z \mapsto v * f\} \vdash \\ \{z\}\{p * p' * {}^{z}\boxed{q * q'}\}\, x := \mathtt{SC}(y, z)\, \{(x = 1 \wedge p * q) \vee (x = 0 \wedge p * p')\}\{\}\end{array}} \quad \text{if } x \neq z \text{and } x \notin \mathsf{fv}(\Gamma, f) \qquad \text{(store-conditional)}$$

$$\frac{\Gamma \vdash \{x\}\{^{x}\boxed{p}\}\, c\, \{^{x}\boxed{q}\}\{x\}}{\Gamma \vdash \{x\}\{^{x}\boxed{p * f}\}\, c\, \{^{x}\boxed{q * f}\}\{x\}} \quad \text{if } \mathsf{fv}(f) \cap \mathsf{writes}(c) = \varnothing \qquad \text{(shared expand)}$$

$$\frac{\Gamma \vdash \delta\{p\}\, c\, \{q\}\delta'}{\Gamma \vdash (\delta \cup \{x\})\, \{p * {}^{x}\boxed{f}\}\, c\, \{q * {}^{x}\boxed{f}\}\, (\delta' \cup \{x\})} \quad \text{if } x \in \mathsf{res}(\Gamma) \setminus [\delta \cup \mathsf{fv}(c)] \text{ and } \mathsf{fv}(f) \cap \mathsf{writes}(c) = \varnothing \qquad \text{(shared frame)}$$

$$\frac{\Gamma \vdash \{\}\{p_1\}\, c_1\, \{q_1\}\{\} \quad \Gamma \vdash \{\}\{p_2\}\, c_2\, \{q_2\}\{\}}{\Gamma \vdash \{\}\{p_1 * p_2\}\, c_1 \parallel c_2\, \{q_1 * q_2\}\{\}} \quad \begin{array}{c}\text{if } \mathsf{fv}(p_1, q_1, c_1) \cap \mathsf{writes}(c_2) = \mathsf{fv}(p_2, q_2, c_2) \cap \mathsf{writes}(c_1) = \varnothing \\ \text{and } \mathsf{frees}(c_1) = \mathsf{frees}(c_2) = \varnothing\end{array} \qquad \text{(parallel)}$$

Figura 6. Regras de inferência

as asserções partilhadas e locais quando se executam as primitivas não bloqueantes. O facto dos juízos recorrerem a conjuntos de variáveis "load-linked" permite manter a coerência dos programas, com blocos de instruções a comportarem-se como transacções. A lógica proposta exprime invariantes em "heaps" de dimensão arbitrária, distinguindo as transacções sucedidas das que falham e lidando com programas que lêm concorrentemente localizações partilhadas.

$\{\exists_v\, head \mapsto v * \texttt{list}(v)\} \vdash$
$\{\}\{\texttt{emp}\}$
$node := \texttt{alloc}(\texttt{nil});$
$\{\}\{\exists_v\, node \mapsto v\}$
$res := 0;$
$\{\}\{(res = 0 \wedge \exists_v\, node \mapsto v) \vee (res = 1 \wedge \texttt{emp})\}$
$\texttt{while}\ res \neq 1\ \texttt{do}$
$\{\}\{\exists_v\, node \mapsto v\}$
$first := \texttt{LL}(head);$
$\{head\}\{\exists_v\, node \mapsto v *\ ^{head}\boxed{\texttt{list}(first)}\}$
$[node] := first$
$\{head\}\{node \mapsto first *\ ^{head}\boxed{\texttt{list}(first)}\}$
$res := \texttt{SC}(node, head)$
$\{\}\{(res = 0 \wedge node \mapsto first) \vee (res = 1 \wedge \texttt{emp})\}$
$\{\}\{(res = 0 \wedge \exists_v\, node \mapsto v) \vee (res = 1 \wedge \texttt{emp})\}$
$\{\}\{\texttt{emp}\}$

Figura 7. Inserção numa lista

Exemplos: inserção e remoção numa lista. Para ilustrar a utilização do sistema de prova, mostram-se correctos programas que adicionam e removem nós de uma lista. Uma lista ou é vazia (representada pelo valor `nil`), caso em que o "heap" tem que ser vazio, ou é uma sequência não vazia de valores, sendo acedida por um endereço de memória que aponta depois para o resto da lista:

$$\text{list}(i) \triangleq (i = \texttt{nil} \wedge \texttt{emp}) \vee (i \neq \texttt{nil} \wedge \exists_j\, i \mapsto j * \text{list}(j))$$

O programa para inserção de um dado endereço de memória no início da lista está na Figura 7. O contexto dos recursos garante que o endereço *head* contém a lista, e que, começando a executar com um "heap" vazio e sem variáveis "load-linked", termina também assim. A prova apresentada omite alguns passos. Primeiro é alocado um endereço de memória, sendo a sua localização guardada numa variável local (*node*). O program entra em ciclo até conseguir inserir um nó na lista: começa por carregar por "load-link" a variável partilhada *head* que contém a cabeça da lista na variável local *first*, fazendo o nó alocado apontar para essa localização, e depois tenta guardá-lo no estado partilhado.

A outra operação importante é a remoção de elementos da lista, implementada na Figura 8. A especificação é semelhante à da operação de inserção. O programa está em ciclo até conseguir remover um elemento da lista ou mantê-la vazia. A libertação de memória não é segura porque se o programa liberta o primeiro nó, outro que tente aceder ao segundo vai abortar.

5 Conclusões

Definimos um sistema de prova para garantir a correcção de programas imperativos, com estruturas de dados mutáveis, concorrentes não bloqueantes usando

$\{\exists_v\, head \mapsto v * \texttt{list}(v)\} \vdash$

$\{\varnothing\}\{\texttt{emp}\}$
$res := 0;$
$\{\}\{\texttt{emp} \wedge res = 0\}$
$\{\}\{(res = 0 \wedge \texttt{emp}) \vee (res = 1 \wedge (first = \texttt{nil} \Rightarrow \texttt{emp}) \wedge (first \neq \texttt{nil} \Rightarrow \exists_v\, first \mapsto v))\}$
$\texttt{while}\ res \neq 1\ \texttt{do}$
  $\{\}\{\texttt{emp}\}$
  $first := \texttt{LL}(head);$
  $\{head\}\{^{head}\boxed{\texttt{list}(first)}\}$
  $\texttt{if}\ first \neq \texttt{nil}\ \texttt{then}$
    $\{head\}\{^{head}\boxed{\texttt{list}(first)} \wedge first \neq \texttt{nil}\}$
    $\{head\}\{^{head}\boxed{\exists_j\, first \mapsto j * \mathrm{list}(j)}\}$
    $second := [first]$
    $\{head\}\{^{head}\boxed{first \mapsto second * \mathrm{list}(second)}\}$
    $\{head\}\{^{head}\boxed{((first = \texttt{nil} \Rightarrow \texttt{emp}) \wedge (first \neq \texttt{nil} \Rightarrow \exists_v\, first \mapsto v)) * \mathrm{list}(second)}\}$
  $\texttt{else}$
    $\{head\}\{^{head}\boxed{\texttt{list}(first)} \wedge first = \texttt{nil}\}$
    $second := first$
    $\{head\}\{^{head}\boxed{\texttt{list}(second)} \wedge first = \texttt{nil}\}$
    $\{head\}\{^{head}\boxed{((first = \texttt{nil} \Rightarrow \texttt{emp}) \wedge (first \neq \texttt{nil} \Rightarrow \exists_v\, first \mapsto v)) * \mathrm{list}(second)}\}$
  ;
  $\{head\}\{^{head}\boxed{((first = \texttt{nil} \Rightarrow \texttt{emp}) \wedge (first \neq \texttt{nil} \Rightarrow \exists_v\, first \mapsto v)) * \mathrm{list}(second)}\}$
  $res := \texttt{SC}(second, head)$
  $\{\}\{(res = 0 \Rightarrow \texttt{emp}) \wedge (res = 1 \Rightarrow ((first = \texttt{nil} \Rightarrow \texttt{emp}) \wedge (first \neq \texttt{nil} \Rightarrow \exists_v\, first \mapsto v)))\}$
;
$\{\}\{(first = \texttt{nil} \Rightarrow \texttt{emp}) \wedge (first \neq \texttt{nil} \Rightarrow \exists_v\, first \mapsto v)\}$
$\texttt{if}\ first \neq \texttt{nil}\ \texttt{then}$
  $\{\}\{\exists_v\, first \mapsto v\}$
  $\texttt{disp}(first)$
  $\{\}\{\texttt{emp}\}$
$\texttt{else}$
  $\{\}\{\texttt{emp}\}$
  $\texttt{skip}$
  $\{\}\{\texttt{emp}\}$
$\{\}\{\texttt{emp}\}$

Figura 8. Remoção duma lista

as primitivas LL/SC. A linguagem é mínima, tendo os construtores imperativos básicos, a composição paralela e as ditas primitivas. O acesso às estruturas de dados partilhadas é feito pelos seus endereços. A semântica operacional considera um novo elemento nas configurações para saber se um thread tem uma visão ac-

tualizada dos endereços de memória. A sintaxe das asserções permite especificar propriedades do estado partilhado. A noção de validade, que define a semântica das asserções, foi não trivialmente adaptada do trabalho relacionado, garantindo o bom comportamento dos programas e permitindo raciocinar sobre os valores assumidos pelas variáveis "load-linked". As regras de inferência permitem provas locais, modulares, estruturadas e orientadas pela sintaxe, capturando a natureza optimista das primitivas `LL`/`SC`.

O sistema apresentado não permite libertar memória. No entanto, uma vez que a memória só pode ser transferida (de local para partilhada) usando "store-conditional", pode-se juntar uma componente à semântica operacional e aos juízos de prova para controlar essas localizações e só não permitir a libertação de memória partilhada. Faltam também mecanismos para raciocinar sobre mudanças no estado partilhado. É também importante desenvolver uma prova formal da correcção do sistema de prova.

Para terminar, fica um agradecimento especial ao Luís Caires, pelo seu grande apoio a este trabalho, também suportado pela FCT, projecto "CITI/FCT/UNL 2011-2012" — PEst-OE/EEI/UI0527/2011.

Referências

1. S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007.
2. R.W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19:19–31, 1967.
3. A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *Proceedings of the Symposium on Principles of Programming Languages*, volume 44, pages 16–28. ACM, 2009.
4. Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, February 2008.
5. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
6. C.B. Jones. *Development methods for computer programs including a notion of interference.* PhD thesis, PhD thesis, Oxford University, UK, 1981.
7. C.B. Jones. *Systematic software development using VDM.* Prentice Hall, 1990.
8. P.W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.
9. G. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61(0):3–15, 2004.
10. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
11. D. Serra. A proof system for lock-free concurrency. Technical report, MSc thesis, Universidade NOVA de Lisboa, 2013.
12. V. Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, UK, 2007.
13. V. Vafeiadis. Concurrent separation logic and operational semantics. *Electronic Notes in Theoretical Computer Science*, 276:335–351, 2011.