3
Suporte das linguagens para captura e restauração de estado

No capítulo anterior, concluímos que o suporte das linguagens de programação para captura e recuperação de estado é necessário para a implementação de diferentes aplicações de migração e persistência em ambientes heterogêneos. As dificuldades relatadas, relacionadas com a captura e restauração de execuções, se devem tanto à não disponibilidade de um modelo portável da execução no nível da linguagem quanto à impossibilidade de instalar o modelo de uma execução na forma de uma nova execução ou de alterações a uma computação existente. Esta seção apresenta a proposta dessa tese para oferecer esses mecanismos.

3.1
Uma abordagem reflexiva para a captura e restauração

Acreditamos que o suporte das linguagens à captura e restauração de estado deve ser baseado na manipulação explícita da representação das estruturas básicas que definem o estado de uma computação. Ao oferecer ao programador primitivas de manipulação do estado, a linguagem lhe permite construir procedimentos customizados de captura e restauração. Alguns exemplos da necessidade de flexibilidade nessas implementações são:

- Tipicamente as linguagens ou sistemas de migração oferecem captura com granularidades altas (no nível de thread ou processos). Enquanto aplicações de balanceamento de carga provavelmente requeram granularidades altas (abrangem o programa todo), aplicações que pretendem executar computações perto dos dados podem precisar movimentar somente um registro de ativação;
- Uma escolha comum da variável "extensão da captura", consiste em extrair o fecho transitivo completo. Em conseqüência, um grande volume pode ser extraído ainda que seja desnecessário – por exemplo, quando as dependências já se encontram no destino;
- A execução é tipicamente representada no nível da linguagem na forma de strings de bytes. Essa representação não é facilmente manipulável:

não facilitaria, por exemplo, alterações da representação da execução do exemplo anterior de forma a revinculá-la com o novo contexto local no destino;

– O mecanismo pode gerar representações em que a estrutura da computação pode ser distinguida ou não. Se a estrutura não for distinguível (ou seja, os diferentes valores e seus tipos não poderiam ser identificados dentro da estrutura, como acontece na migração ou persistência homogêneas) esse mecanismo não permitiria implementar uma restauração heterogênea.

– Deseja-se extrair o grafo de dependências completo (cópia profunda) como em [TKS06] ou desconsiderar as referências (quebrá-las) como em [Miller06].

Já que o programador conhece a semântica da aplicação, ele deveria ter controle sobre essas variáveis. Entretanto, não são comuns as linguagens de programação seqüenciais com suporte flexível à captura e restauração heterogêneas. O suporte das linguagens à captura e restauração é tradicionalmente oferecido através de mecanismos que seguem uma semântica predeterminada. Na maioria dos casos, se trata de implementações *ad-hoc* focadas para determinado domínio de aplicação e que não permitem a manipulação da representação no nível da linguagem. O nível de controle que o programador pode ter sobre o procedimento de captura está muito relacionado à capacidade da representação de admitir composição: se a captura é atômica (alta granularidade), o usuário não pode controlar o procedimento (exceções a essa regra são as implementações que admitem como parâmetro uma função que especifica o método de captura). Se a granularidade for pequena e a representação é componível, é possível fazer cópias parciais e dessa forma controlar a extensão do fecho transitivo capturado. Entretanto, usualmente a granularidade da captura é fixada pelos frameworks de persistência ou migração. Eles podem permitir a captura de uma thread ou processo, várias threads ou registros de ativação.

Esta tese propõe uma abordagem baseada na combinação de operações de reflexão que chamamos de *reificação* e *instalação*. Estas operações devolvem a representação da execução como uma estrutura de dados manipulável no nível da linguagem, e permitem a sua instalação de volta no espaço de memória, respectivamente. Quando implementadas de forma que as representações sejam navegáveis e componíveis, resultam em ferramentas poderosas permitindo a implementação tanto de diferentes políticas como, inclusive, de construções da linguagem, como é o caso das continuações. Em uma representação navegável, cada entidade que a compõe pode ser alcançável a partir de alguma outra

entidade. A componibilidade permite por um lado controlar a extração do fecho transitivo, já que a partir de uma granularidade menor é possível construir granularidades maiores, e a modificação e instalação seletiva da representação. A capacidade de manipular o estado no nível da linguagem que estas operações permitem é o que provê a flexibilidade da proposta, permitindo mudanças seletivas no comportamento do programa durante a execução da computação.

Nesta proposta, o procedimento comumente chamado de serialização está dividido em dois passos. O primeiro consiste na reificação (e instalação) das estruturas da execução. O segundo passo, que chamaremos de *serialização* (e o procedimento inverso de *de-serialização*), consiste na conversão a strings de bytes para permitir a posterior transmissão ou persistência.

A seção seguinte introduz o conceito de reflexão e aspectos relacionados com nossa proposta.

3.2 Reflexão computacional

A reflexão computacional é uma abordagem que permite a uma computação tomar conhecimento do próprio estado (introspecção), possivelmente afetando-o [Smith84, Maes87]. As linguagens podem oferecer funcionalidades reflexivas, como no presente trabalho, ou ser reflexivas, quando o programa pode observar e mudar tanto o próprio código quanto todos os aspectos relacionados com a linguagem [MJD96].

Do ponto de vista de um mecanismo reflexivo, o sistema pode ser visto como dividido em duas partes: o nível meta, reflexivo, ou de gerenciamento, em que a computação vê a si própria do ponto de vista de uma entidade externa, e o nível base ou de objeto, que implementa a aplicação. Estes níveis estão *causalmente conectados*: a aplicação tem acesso à própria representação e a modificação dessa representação afeta a continuação da execução. A conexão causal é uma necessidade da consistência da reflexão. Para observar a computação, o controle deve passar do nível base ao meta-nível: com esse objetivo a aplicação precisa ser instrumentada para acrescentar pontos de transferência de controle ou a linguagem pode oferecer mecanismos para executar esta transferência. Depois de observada e/ou modificada a execução, o controle deve retornar ao nível base para continuar a computação. Dessa forma, o nível base pode ser controlado pelo meta-nível, em que podem ser implementadas diferentes políticas para modificar uma aplicação [TVP02].

Para que o meta-nível tenha acesso à informação relativa à execução, é necessário que o estado de execução do nível base seja reificado: o modelo da sua execução deve ser extraído na forma de uma representação manipulável pelo

meta-nível. O termo *reificação* se refere a disponibilizar, ao programa que está executando, as estruturas de dados do interpretador [FW84]. Chamaremos de instalação a operação oposta, que consiste em instalar valores obtidos através de operações de reificação no espaço de memória.

Dependendo de que parte da computação é refletida, podem identificar-se dois tipos de reflexão: a *estrutural* e a de *comportamento*. Elas se diferenciam em que no primeiro caso se acessa a estrutura estática do programa, e no segundo é refletido o seu comportamento dinâmico [MJD96]. Nesse trabalho, estamos fundamentalmente interessados na reflexão de comportamento.

A reificação é diferente da abordagem das linguagens de mais alta ordem que tratam as computações como valores de primeira classe [DM95]. De fato, os valores reificados são também entidades de primeira classe mas, diferentemente daqueles, seu conteúdo pode ser inspecionado e manipulado no nível da linguagem.

A operação reflexiva mais frequentemente encontrada é a introspecção. Ela permite a implementação de depuradores, *profilers* e ferramentas de monitoramento. Entretanto, ela não é suficiente para implementar captura e restauração: para isto precisamos também modificar a computação. A intercessão (ou *atualização reflexiva*) é a operação contrária à introspecção e se refere a mudar a estrutura da execução através de reflexão.

A reflexão que não precisa de planejamento prévio é chamada de *não antecipada* [RDT08]. Este tipo de reflexão é interessante para aplicações que precisam de adaptação dinâmica como depuração online e monitoramento. A reflexão implica em um custo em termos de eficiência devido à reificação quando se passa do nível base ao meta-nível e no processo de instalação no sentido contrário. Isto motiva a implementação de *reflexão parcial*, em que os usuários selecionam o que será reificado e quando.

Na seção seguinte são apresentados os requisitos da nossa proposta e como eles podem ser satisfeitos através de reflexão.

3.2.1 Requisitos

A API de reificação e instalação deve satisfazer os seguintes requisitos:

1. Equivalência: O resultado da reificação deve ser uma representação observacionalmente equivalente aos dados representados.

   Entretanto, esta equivalência é instantânea e não precisa seguir a evolução da computação.

2. Isolamento: Apesar de equivalentes, a representação reificada e a abstração em si são independentes: modificações na representação reificada não afetarão a computação em execução até a execução de uma operação explícita de instalação.

   Esta abordagem "transacional" permite evitar inconsistências que possam surgir devido à quebra de invariantes durante modificações reflexivas (entretanto, a representação finalmente instalada deve respeitar estas invariantes). Não existe um compromisso de conexão causal entre a execução é a representação fora o momento da captura e da instalação da representação.

3. Controle: o usuário pode controlar as variáveis da reificação/instalação (extensão do grafo, granularidade, etc).

   A reificação de valores estruturados não pode ser atômica: ela deve se basear na construção da representação reificada por composição da representação dos seus membros, reificados. A representação deve ser navegável, de forma a permitir alcançar os membros da representação e manipulá-los.

4. Portabilidade: uma representação pode ser instalada em diferentes plataformas.

3.3 API

As funcionalidades genéricas que formam parte da nossa API são implementadas pelas funções content e install. O nosso modelo de reificação é baseado no uso da função *content* e a instalação é provida pela primitiva *install*.

- *content(valor, [nível])* recebe um valor Lua como argumento, e retorna a representação da sua estrutura. Esta primitiva aceita um segundo parâmetro na reificação de co-rotinas que é o nível do registro de ativação.
- *install(repr, tipo/valor,[nível])* recebe como argumento a representação e o tipo ou um valor do tipo a ser reconstruído. Da mesma forma que o *content*, *install* recebe um parâmetro *nível* na instalação de co-rotinas. Uma invocação a essa função deve retornar um valor do tipo especificado, em caso de sucesso.

A API também oferece duas funções genéricas auxiliares: *name* e *fields*. *name* retorna um identificador único para valores estruturados (aqui a unicidade é garantida somente na plataforma de execução) para fins de identidade.

Cada valor deve ser univocamente identificado para permitir a restauração de valores compartilhados e para identificar valores que estão sendo recebidos de volta no seu ambiente original. Um aspecto a levar em conta, que é conseqüência do não compromisso com a consistência entre valores e representações fora das chamadas às primitivas da biblioteca, é o fato de que em Lua as referências são imutáveis durante o tempo de vida dos dados, mas podem ser reutilizadas depois de que esses forem coletados. Então, se usarmos como identificador único um número que representa a referência do valor, pode acontecer que entre duas serializações de uma computação, um mesmo identificador se refera a valores diferentes.

A função *fields* retorna a descrição da representação de qualquer tipo, para fins de documentação. O único argumento é o nome do tipo.

A seguir se descreve a semântica operacional das primitivas propostas.

3.4 Semântica operacional

Com o objetivo de oferecer ao usuário um conjunto de operações bem definidas, especificamos a seguir a semântica operacional das primitivas de reificação e instalação da API proposta. Esta semântica pode ser definida através de operações executadas em uma máquina que interpreta uma linguagem especificada formalmente sobre uma determinada representação do estado. Nossa definição está baseada em uma extensão da máquina SECD [Landin64] (escolhida pela sua similaridade com a linguagem Lua) com suporte para atribuição e múltiplos estados. Henderson [Henderson80] oferece uma extensa descrição da máquina SECD e seu conjunto de instruções. A notação usada é tomada de [FF06].

A SECD é uma máquina baseada em pilha, de onde as funções obtém os seus parâmetros. O estado da máquina pode ser descrito através do conteúdo de 4 registros (de onde se origina o nome da máquina):

- S (stack): armazena os resultados temporários quando se computa o valor de expressões. Equivale a um registro de ativação ou *frame*;
- E (environment): armazena os valores vinculados a variáveis durante a avaliação;
- C (control list): armazena o bytecode do programa que está sendo executado;
- D (dump): se usa como pilha para armazenar os valores de outros registros quando se chama uma nova função.

O efeito de uma instrução pode definir-se através dos estados desses registros antes e depois da execução da instrução.

C = ∅ | b C | x C | ap C | $prim_{o^n}$ C
S = ∅ | v S
E = é uma função de variáveis a valores $\{\langle x, v\rangle, ...\}$
D = ∅ | $\langle S, E, C, D\rangle$
v = b | $\{\langle\langle x\ C\rangle E\rangle\}$

Figura 3.1: Sintaxe da linguagem

Os conjuntos S E C e D se definem como mostra a figura 3.1, onde b é uma constante, x é uma variável, ap é uma aplicação, $prim_{o^n}$ são operações primitivas de aridade n, $\{\langle\langle x\ C\rangle E\rangle\}$ são closures (pares de termos abertos e ambientes E).

A relação $\longmapsto$ define uma transição de um passo no estado da máquina SECD. As regras de transição fundamentais nessa máquina são as seguintes:

$$\langle S, E, b\ C, D\rangle \longmapsto \langle b\ S, E, C, D\rangle \quad (3\text{-}1)$$

$$\langle S, E, x\ C, D\rangle \longmapsto \langle E(x)\ S, E, C, D\rangle \quad (3\text{-}2)$$

$$\langle b_n ... b_1\ S, E, prim_{o^n}\ C, D\rangle \longmapsto \langle \delta(o^n, b_1, b_2, ..., b_n)\ S, E, C, D\rangle \quad (3\text{-}3)$$

$$\langle S, E, \langle x\ C'\rangle C, D\rangle \longmapsto \langle\langle\langle x\ C'\rangle E\rangle\ S, E, C, D\rangle \quad (3\text{-}4)$$

$$\langle v\langle\langle x\ C'\rangle E'\rangle\ S, E, ap\ C, D\rangle \longmapsto \langle\emptyset, E'[x \leftarrow v], C', \langle S, E, C, D\rangle\rangle \quad (3\text{-}5)$$

$$\langle v\ S, E, \emptyset, \langle S', E', C', D\rangle\rangle \longmapsto \langle v\ S', E', C', D\rangle \quad (3\text{-}6)$$

Estas regras estabelecem que:

Regra 3-1: a avaliação de um dado retorna o dado no registro S.

Regra 3-2: a avaliação de uma variável retorna no registro S o valor da variável no ambiente E.

Regra 3-3: a aplicação de uma operação primitiva sobre uma lista de n valores retorna um valor no registro S.

Regra 3-4: a avaliação de uma abstração λ retorna uma closure.

Regra 3-5: a aplicação de uma função sobre um parâmetro gera a criação de um novo registro de ativação na pilha e o armazenamento dos valores prévios no dump D. No ambiente E, o parâmetro x é substituído pelo valor do argumento.

Regra 3-6: No final da execução de uma chamada a função, o registro de ativação prévio é restaurado e o valor de retorno é inserido no topo da nova pilha.

Para este trabalho precisamos estender a máquina SECD com atribuições (Lua é uma linguagem com estado) e múltiplas co-rotinas. Uma co-rotina é representada pelos 4 registros SECD. Para armazenar estas co-rotinas acrescentamos um registro que chamaremos de *storage*(Σ). O *storage* é uma função que mapea localizações a valores.

Para formalizar a transferência de controle, precisamos de um registro adicional na máquina $SECD\Sigma$ que será chamado de *pilha de ativação* (A). O registro A irá armazenar as co-rotinas ativas na ordem da ativação: no topo da pilha está a co-rotina que está executando no momento. No restante do texto, o topo de A será usado como os registros da máquina, ou seja, estamos definindo a semântica das operações como transições de $\langle A, \Sigma\rangle$ em $\langle A', \Sigma'\rangle$ onde:

A $= \emptyset \mid \langle \sigma, thr, A\rangle$, thr $= \langle S, E, C, D\rangle$

Dessa forma, a definição de nossa configuração modifica e extende a definição na Figura 3.1 com os conjuntos:

S $= \emptyset$ **|** v S | σ S

E = uma função de identificadores a localizações $\{\langle x, \sigma\rangle, ...\}$

C $= \emptyset$ **|** b C | x C | ap C | $prim_{o^n}$ C | set C | newthread C **|** reify C | install C | resume C | yield C

D $= \emptyset \mid \langle S, E, C, D\rangle$

v = b | $\{\langle\langle x\ C\rangle E\rangle\}$

Σ = uma função de localizações a valores $\{\langle \sigma, v\rangle, ...\}$

A $= \emptyset$ **|** $\langle \sigma, \langle S, E, C, D\rangle, A\rangle$

A seguir, as regras básicas de transição modificadas (*):

$$\langle\langle\sigma, \langle S, E, b\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \tag{3-7}$$
$$\langle\langle\sigma, \langle b\ S, E, C, D\rangle, A\rangle, \Sigma\rangle$$
$$\langle\langle\sigma, \langle S, E, x\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \tag{3-8}$$
$$\langle\langle\sigma, \langle v\ S, E, C, D\rangle, A\rangle, \Sigma\rangle \quad (*)$$
$$onde\ v = \Sigma(E(x))$$
$$\langle\langle\sigma, \langle b_n ... b_1\ S, E, prim_{o^n}\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \tag{3-9}$$
$$\langle\langle\sigma, \langle v\ S, E, C, D\rangle, A\rangle, \Sigma\rangle$$
$$onde\ v = \delta(o^n, b_1, b_2, ..., b_n)$$
$$\langle\langle\sigma, \langle S, E, \langle x\ C'\rangle C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \tag{3-10}$$
$$\langle\langle\sigma, \langle\langle\langle x\ C'\rangle E\rangle S, E, C, D\rangle, A\rangle, \Sigma\rangle$$
$$\langle\langle\sigma, \langle v\langle\langle x\ C'\rangle E'\rangle\ S, E, ap\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \tag{3-11}$$
$$\langle\langle\sigma, \langle\emptyset, E'[x \leftarrow \sigma'], C', \langle S, E, C, D\rangle\rangle, A\rangle, \Sigma[\sigma' \leftarrow v]\rangle$$
$$onde\ \sigma' \notin dom(\Sigma) \quad (*)$$
$$\langle\langle\sigma, \langle v\ S, E, \emptyset, \langle S', E', C', D\rangle\rangle, A\rangle, \Sigma\rangle \longmapsto \tag{3-12}$$
$$\langle\langle\sigma, \langle v\ S', E', C', D\rangle, A\rangle, \Sigma\rangle$$
$$\langle\langle\sigma, \langle v\ S, E, set\ x\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \tag{3-13}$$
$$\langle\langle\sigma, \langle v\ S, E, C, D\rangle, A\rangle, \Sigma[\sigma' \leftarrow v]\rangle \quad (*)$$
$$onde\ \sigma' = E(x)$$

A nova regra 3-13 define a atribuição.

A seguir definimos as regras que descrevem a semântica dos operadores de co-rotinas (*create*, *resume* e *yield*).

$$
\begin{array}{l}
create\ \langle x\ C'\rangle E' :: \\
\langle\langle\sigma, \langle\langle x\ C'\rangle E'.S, E, create.C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \\
\langle\langle\sigma, \langle\sigma'.S, E, C, D\rangle, A\rangle, \Sigma[\sigma' \leftarrow \langle\langle x\ C'\rangle E', \emptyset, \emptyset, \emptyset\rangle]\rangle \\
onde\ \sigma' \notin dom(\Sigma)
\end{array}
\tag{3-14}
$$

$$
\begin{array}{l}
resume\ \sigma'\ v :: \\
\langle\langle\sigma, \langle\sigma'\ v\ S, E, resume\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \\
\langle\langle\sigma', \langle v\ S', E', C', D'\rangle, \langle\sigma, \langle S, E, C, D\rangle, A\rangle\rangle, \Sigma[\sigma' \leftarrow nil]\rangle \\
onde\ \langle S', E', C', D'\rangle = \Sigma(\sigma')
\end{array}
\tag{3-15}
$$

$$
\begin{array}{l}
yield\ v :: \\
\langle\langle\sigma, \langle v\ S, E, yield\ C, D\rangle, \langle\sigma', \langle S', E', C', D'\rangle, A\rangle\rangle, \Sigma\rangle \longmapsto \\
\langle\langle\sigma', \langle v\ S', E', C', D'\rangle, A\rangle, \Sigma[\sigma \leftarrow \langle S, E, C, D\rangle]\rangle
\end{array}
\tag{3-16}
$$

$$
\begin{array}{l}
\langle\langle\sigma, \langle v\ S, E, \emptyset, \emptyset\rangle, \langle\sigma', \langle S', E', C', D'\rangle, A\rangle\rangle, \Sigma\rangle \longmapsto \\
\langle\langle\sigma', \langle v\ S', E', C', D'\rangle, A\rangle, \Sigma\rangle
\end{array}
\tag{3-17}
$$

A regra 3-14 descreve a criação de uma co-rotina como a criação de um estado e a instalação da closure $\langle x\, C'\rangle E'$ no seu primeiro (e único) registro de ativação.

Regra 3-15: (Re) iniciar uma co-rotina consiste em movê-la do *storage* ao topo de A. O argumento se coloca no topo de S.

Regra 3-16: A operação *yield* armazena a co-rotina corrente no *storage* e a elimina de A. A pilha previa é restaurada nos registros da máquina.

A regra 3-17 é uma extensão da regra 3-12 de forma a descrever a terminação de uma co-rotina e o retorno do controle à co-rotina que fez a ativação.

Para formalizar a reificação e instalação de co-rotinas precisaremos de operações de mais baixo nível do que as apresentadas anteriormente. Para isto, podemos aproveitar uma função para a instalação de closures que está embutida na operação de criação de co-rotinas (regra 3-14). Deste modo, iremos decompor esta operação na criação de uma co-rotina vazia e a instalação de

uma closure nessa co-rotina:

$$
\begin{array}{rcll}
& & newthread :: & \\
\langle\langle\sigma, \langle S, E, newthread\ C, D\rangle, A\rangle, \Sigma\rangle & \longmapsto & & (3\text{-}18) \\
& & \langle\langle\sigma, \langle\sigma'\ S, E, C, D\rangle, A\rangle, \Sigma[\sigma' \leftarrow \emptyset, \emptyset, \emptyset, \emptyset]\rangle & \\
& & onde\ \sigma' \notin dom(\Sigma) & \\
& & install\ \langle\langle x\ C'\rangle E'\rangle, \sigma' :: & \\
\langle\langle\sigma, \langle\langle\langle x\ c'\rangle E'\rangle\sigma' S, E, install\ C, D\rangle, A\rangle, \Sigma\rangle & \longmapsto & \langle\langle\sigma, \langle\sigma'\ S, E, C, D\rangle, A\rangle, & (3\text{-}19) \\
& & \Sigma[\sigma' \leftarrow \langle\langle x\ C'\rangle E'\rangle, \emptyset, \emptyset, \emptyset\rangle]\rangle &
\end{array}
$$

A regra 3-18 descreve a criação de uma nova co-rotina vazia e o retorno da referência que a representa. O processo de inicialização de uma co-rotina termina com a inserção da closure passada como parâmetro, no topo da pilha (regra 3-19). Isto é válido tanto quando a operação é executada através do interpretador quando através da operação *install* da API proposta. Na realidade, a operação *install* pode ser extendida ao caso mais geral da instalação de um registro de activação em qualquer nível, considerando o registro $\langle\langle\langle x\ C'\rangle E\rangle, \emptyset, \emptyset, \emptyset\rangle$. Mas antes disso, precisamos definir um grupo de funções auxiliares:

$$
\begin{array}{rcl}
sublista\ n, \langle S, E, C, D\rangle :: & & \\
sublista(0, lista) & = & lista \\
sublista(n+1, \langle S, E, C, D\rangle) & = & sublista(n, D)
\end{array}
$$

$$
\begin{array}{rcl}
put\ n, \langle S', E', C'\rangle, \langle S, E, C, D\rangle :: & & \\
put(0, \langle S', E', C'\rangle, \langle S, E, C, D\rangle) & = & \langle S', E', C', D\rangle \\
put(n+1, \langle S', E', C'\rangle, \langle S, E, C, D\rangle) & = & \langle S, E, C, put(n, \langle S', E', C'\rangle, D)\rangle
\end{array}
$$

$$
\begin{array}{rcl}
findInA\ \sigma, A :: & & \\
findInA(\sigma, \emptyset) & = & \langle\emptyset\rangle \\
findInA(\sigma, \langle\sigma, lista, A\rangle) & = & lista \\
findInA(\sigma, \langle\sigma', lista', \langle\sigma'', lista'', A\rangle\rangle) & = & findInA(\sigma, \langle\sigma'', lista'', A\rangle)
\end{array}
$$

$$
\begin{array}{l}
find\ \sigma, \Sigma, A :: \\
if\ \Sigma[\sigma] \neq nil\ then\ \Sigma[\sigma] \\
else\ findInA(\sigma, A) \\
end
\end{array}
$$

A reificação e instalação de co-rotinas pode ser definida como segue:

$$
\begin{array}{rcll}
 & & reify\ \sigma', n :: & \\
\langle\langle \sigma, \langle \sigma'\ n\ S, E, reify\ C, D\rangle, A\rangle, \Sigma\rangle & \longmapsto & & (3\text{-}20) \\
 & & \langle\langle \sigma, \langle\langle S', E', C'\rangle S, E, C, D\rangle, A\rangle, \Sigma\rangle & \\
 & & onde\ \langle S', E', C', D'\rangle = & \\
 & & sublista(n, find(\sigma', \Sigma, A)) & \\
 & & install\ \langle S', E', C'\rangle, \sigma', n :: & \\
\langle\langle \sigma, \langle\langle S', E', C'\rangle \sigma' n S, E, install\ C, D\rangle, A\rangle, \Sigma\rangle & \longmapsto & \langle\langle \sigma, \langle \sigma' S, E, C, D\rangle, A'\rangle, \Sigma'\rangle & (3\text{-}21) \\
 & & onde\ find(\sigma', \Sigma', A') = & \\
 & & put(n, \langle S', E', C'\rangle, find(\sigma', \Sigma, A)) &
\end{array}
$$

Regra 3-20: A reificação consiste na extração dos registros SEC que correspondem ao *frame* requisitado.

Regra 3-21: Instalar um registro de ativação consiste em copiá-lo no *frame* n da co-rotina correspondente à referência σ.

Usando a formalização apresentada podemos provar que a reificação e instalação são operações simêtricas:

$$
\begin{array}{rcl}
R(I(Rep)) & \equiv & Rep \\
I(R(V)) & \equiv & V
\end{array}
$$

ou seja:

$$
\langle\langle \sigma, \langle \sigma'\ n\ \sigma'\ n\ S, E, reify\ install\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \langle\langle \sigma, \langle S, E, C, D\rangle, A\rangle, \Sigma\rangle \quad (3\text{-}22)
$$

, e o inverso também é válido.

Mostramos que não se produzem modificações quando um registro reifi-

cado é reinstalado no mesmo nível da co-rotina. De 3-20,

$$\langle\langle\sigma, \langle\sigma'\ n\ \sigma'\ n\ S, E, reify\ install\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto$$
$$\langle\langle\sigma, \langle\langle S', E', C'\rangle\ \sigma'\ n\ S, E, install\ C, D\rangle, A\rangle, \Sigma\rangle$$
$$onde$$
$$\langle S', E', C', D'\rangle = sublista(n, \langle S'', E'', C'', D''\rangle),$$
$$\langle S'', E'', C'', D''\rangle = find(\sigma', \Sigma, A)$$

De 3-21, $\langle\langle\sigma, \langle\langle S', E', C'\rangle\ \sigma'\ n\ S, E, install\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto$
$\langle\langle\sigma, \langle\sigma'\ S, E, C, D\rangle, A\rangle, \Sigma\rangle,$
onde σ' referencia a co-rotina $\langle S''', E''', C''', D'''\rangle = put(n, \langle S', E', C'\rangle, find(\sigma', \Sigma, A))$
Então, para satisfazer 3-22, é necessário que

$\langle S'', E'', C'', D''\rangle = \langle S''', E''', C''', D'''\rangle$.

A prova é uma indução trivial no nível do registro de ativação reificado, para uma pilha de tamanho arbitrário.