Pontifícia Universidade Católica
do Rio de Janeiro

Anolan Yamilé Milanés Barrientos

Suporte de linguagens de programação para migração heterogênea de computações

Tese de Doutorado

Tese apresentada ao Programa de Pós–graduação em Ciências da Computação do Departamento de Informática da PUC-Rio como requisito parcial para obtenção Do título de Doutor em Ciências da Computação

Orientador : Prof. Noemi Rodriguez
Co–Orientador: Prof. Roberto Ierusalimschy

Rio de Janeiro
julho de 2008

Pontifícia Universidade Católica
do Rio de Janeiro

Anolan Yamilé Milanés Barrientos

Suporte de linguagens de programação para migração heterogênea de computações

Tese apresentada ao Programa de Pós–graduação em Ciências da Computação do Departamento de Informática do Centro Têcnico Científico da PUC-Rio como requisito parcial para obtenção Do título de Doutor em Ciências da Computação. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Noemi Rodriguez
Orientador
Departamento de Informática — PUC-Rio

Prof. Roberto Ierusalimschy
Co–Orientador
Departamento de Informática — PUC-Rio

Prof. Andre Luís de Medeiros Santos
Centro de Informática – Universidade Federal de Pernambuco

Prof. Edward Hermann Haeusler
Departamento de Informática — PUC-Rio

Prof. Marco Túlio de Oliveira Valente
Instituto de Informática — PUC Minas

Prof. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática — PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro Têcnico Científico — PUC-Rio

Rio de Janeiro, 11 de julho de 2008

Anolan Yamilé Milanés Barrientos

Ficha Catalográfica

Barrientos, Anolan Yamilé Milanés

Suporte de linguagens de programação para migração heterogênea de computações / Anolan Yamilé Milanés Barrientos; orientador: Noemi Rodriguez; co–orientador: Roberto Ierusalimschy. — Rio de Janeiro : PUC-Rio, Departamento de Informática, 2008.

v., 97 f: il. ; 29,7 cm

1. Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Migração e persistência de computações. 3. Reificação e instalação de computações. 4. Suporte das linguagens de programação. 5. Linguagem de programação Lua. I. Rodriguez, Noemi. II. Ierusalimschy, Roberto. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

A Paula y Sebastián

Agradecimentos

A minha orientadora Noemi por ter sido uma guia e uma mão amiga durante todos esses anos (mesmo quando não fez economias de puxões de orelha quando necessário). Eu aprendi muito com ela, e espero aprender muito ainda.

A Roberto Ierusalimschy pela ajuda, mas sobretudo pela vontade de estudar que inspira o seu inesgotável caudal de conhecimento.

Aos meus pais, minha irmã e meus irmãos pelo apoio incondicional e por acreditar em mim.

Muitas outras pessoas contribuiram de alguma forma ao feliz término desse trabalho. Em particular quero agradecer a Silvana por revisar atentamente (de novo!) o rascunho da tese. Ao Renato Maia pela valiosa cooperação que melhorou em muito a qualidade do trabalho. E a Thiago e Sergina que me abriram as portas de sua casa durante as longas jornadas nômades da reta final.

Ao pessoal do departamento de Informática da PUC-Rio e do LNCC pelo apoio e a simpatia, e em especial ao professor Bruno Schulze pela ajuda e incentivo.

À professora Lucina Garcia pelo apoio sem o qual toda esta jornada não teria sido possível.

Ao CNPq, à PUC–Rio e ao LNCC, pela oportunidade e os auxílios concedidos.

E enfim, a todos os amigos, presentes e ausentes, que me ajudaram e que tentaram fazer de mim uma pessoa melhor.

Resumo

Barrientos, Anolan Yamilé Milanés; Rodriguez, Noemi; Ierusalimschy, Roberto. **Suporte de linguagens de programação para migração heterogênea de computações**. Rio de Janeiro, 2008. 97p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A migração heterogênea de computações se refere ao movimento de uma computação em execução entre plataformas diferentes. Trata-se de um procedimento difícil, que requer mecanismos de captura e restauração do estado de execução que permitam a identificação da estrutura da computação e seus dados. Estes mecanismos, quando oferecidos, aparecem tradicionalmente na forma de soluções ad-hoc que são difíceis de adaptar aos requisitos de diferentes aplicações. Esta tese discute a necessidade da presença de suporte para captura e restauração de execuções nas linguagens de programação. Este suporte deve ser genérico o suficiente para que sobre ele possam ser implementadas diferentes políticas de captura e restauração, tanto no contexto de migração como no de persistência heterogêneas. Este trabalho estende a linguagem de programação Lua com uma API que permite ao programador reificar estruturas internas da execução em entidades manipuláveis da linguagem, para estudar os mecanismos básicos que uma linguagem deveria oferecer para permitir a implementação de diferentes políticas.

Palavras–chave

Migração e persistência de computações. Reificação e instalação de computações. Suporte das linguagens de programação. Linguagem de programação Lua.

Abstract

Barrientos, Anolan Yamilé Milanés; Rodriguez, Noemi; Ierusalimschy, Roberto. **Language support for the heterogeneous migration of computations**. Rio de Janeiro, 2008. 97p. PhD Thesis — Department of Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The heterogeneous migration of computations allows computations to move between different platforms. It is a difficult procedure, that demands mechanisms for the capture and restoration of the state of the execution allowing for the identification of the structure of the computation and its data. This support, when offered, commonly appears in the form of ad-hoc solutions which are difficult to tailor or adapt to different needs. This thesis discusses the need for this support in current programming languages. This support must allow the implementation of different applications that can profit from the ability of capturing and restoring computations heterogeneously, like migration and persistence. To experiment with this idea, we extend the Lua programming language with an API that allows the programmer to reify the internal structures of execution into manipulable language entities, to explore the basic mechanisms a language should provide in order to support the implementation of different policies.

Keywords

Computation migration and persistence. Reification and installation of computations. Language level support. Lua programming language.

Sumário

Lista de figuras

1
Introdução

A *mobilidade –ou migração– de computações* é comumente definida como o movimento de uma computação entre diferentes dispositivos. Computação aqui se refere a uma execução, seja na forma de processos, threads, agentes móveis, ou registros de ativação. Durante a migração, a execução é enxergada como sendo dados: a migração de computações consiste basicamente na manipulação e transmissão do estado de execução capturado na origem de modo que possa ser restaurado no destino na forma de uma nova computação equivalente. A implementação deste procedimento envolve tanto os mecanismos internos que permitem executar o movimento, quanto o mapeamento desses mecanismos em construções da linguagem ou do sistema.

A migração permite otimizações baseadas na melhoria da localidade ou na mudança dinâmica da topologia da aplicação. Melhorias da localidade podem ser obtidas, por exemplo, ao executar a computação perto dos dados ou de recursos ou serviços especiais, como bancos de dados volumosos ou software licenciado. Mudanças dinâmicas da topologia ocorrem em aplicações de balanceamento de carga, minimização do *downtime* em serviços contínuos, suporte para operação desconectada, entre outros.

Embora largamente pesquisada [Smith88, Eskicioglu90, Nuttall94, MDPW+00], esta técnica não tem sido extensivamente aplicada na prática. Dentre as possíveis causas estão sua relativa complexidade e as dificuldades em obter bom desempenho, assim como aspectos de segurança e até fatores sociológicos [CHK94, MDPW+00]. Por exemplo, aplicações baseadas em agentes móveis são interessantes do ponto de vista acadêmico mas são difíceis de levar na prática por causa dos problemas de segurança que lhes são inerentes. Porém, existem atualmente várias áreas, incluindo as de grades computacionais e computação ubíqua, em que o desempenho da migração fica em um segundo plano quando comparado às capacidades que acrescenta. Nesses contextos, a migração pode permitir, por exemplo, mover computações que estão executando remotamente para devolver o controle da máquina ao seu dono, ou a transferência de aplicações em execução de/para um dispositivo sem fio. Também, aplicações de tempo real podem ser beneficiadas por este

tipo de procedimento.

Um aspecto comum a essas áreas é a sua potencial heterogeneidade. Quando os componentes podem se mover entre plataformas diferentes a migração é chamada de *heterogênea*. Diferentemente da migração heterogênea, a migração chamada de *homogênea* se executa entre plataformas similares. Ela tipicamente se baseia na cópia integral do conteúdo da memória para o nó destino com no máximo alguns ajustes. A migração heterogênea é mais complexa do que a homogênea. Para que a migração heterogênea possa ser executada, precisa-se conhecer a estrutura dos dados, pois cada dado deve ser extraído de forma que, após as devidas traduções, ele possa ser entendido na máquina destino. Isto implica, por um lado, na necessidade de trabalhar no nível da aplicação: apesar das vantagens de implementar a migração no nível de *kernel* em termos de desempenho e transparência (no sentido de que os detalhes do procedimento estão ocultos para o usuário), implementações de migração no nível do kernel dependem do sistema operacional, o que em geral as invalida em ambientes heterogêneos. A migração heterogênea forte de computações é comumente definida como aquela em que o estado de execução também migra [FPV98]. A maioria das linguagens de programação atuais não oferece o suporte para a captura e restauração do estado de execução que a implementação de migração heterogênea forte de computações precisa [MRS08]. Para superar essa limitação, os programadores são forçados, no nível da aplicação, a usar truques que afetam a portabilidade ou o desempenho, ou ambos.

O suporte de linguagens para a captura e restauração heterogênea do estado de execução de computações pode ser caracterizado como a provisão de construções que permitam ambos os procedimentos em um ambiente heterogêneo. Este suporte deve oferecer um substrato comum para a implementação de diferentes tipos de aplicações que precisam de captura e restauração heterogêneas.

Como exemplo, há uma grande similaridade dos requisitos da migração forte e a persistência de execuções. Entretanto, ambos os procedimentos são diferentes no nível da aplicação. A migração requer suporte para coordenação on-line, assincronismo e comunicação em um ambiente distribuído, e, possivelmente, suporte para a manipulação de exceções remotas. As aplicações de persistência, por outro lado, envolvem problemas relacionados com a restauração atemporal da computação. Ainda no contexto mais restrito da mobilidade, podem existir diferenças substanciais entre as aplicações. Estas diferenças estão relacionadas à granularidade dos valores migrados, quanto do estado de execução será transferido (vital para o desempenho da migração), os métodos de serialização e a forma em que a computação será re-vinculada ao novo

contexto local, entre outros. Por exemplo, uma aplicação de balanceamento de carga deve precisar migrar a computação como um todo; já se o objetivo é migrar uma computação para perto dos dados, só um registro de ativação poderia ser suficiente. Na construção do grafo de dependências, algumas referências podem ser anuladas ou então, re-vinculadas no destino. Esses aspectos, freqüentemente fixados nos *frameworks* de migração, são políticas e devem ser considerados como escolhas de implementação da aplicação do ponto de vista do projeto da linguagem. O ideal é que a linguagem ofereça mecanismos flexíveis, e não políticas específicas. Ao acrescentar funcionalidades a uma linguagem de programação de propósito geral, é necessário manter a simplicidade da linguagem e evitar redundâncias e conflitos, mas ao mesmo tempo oferecer suporte para tantas aplicações quanto for possível, privilegiando a generalidade.

Entretanto, uma abordagem escolhida com freqüência para a implementação de mecanismos de suporte a captura e restauração de estado –que pode ser chamada de *opaca* ou *caixa-preta*– consiste em seguir uma semântica predeterminada pelo sistema. Embora fáceis de usar, essas implementações padecem de uma falta de versatilidade que as faz pouco adaptáveis a diferentes aplicações. Para satisfazer esse requisito é necessário oferecer mais controle sobre os procedimentos. Abordagens baseadas na reflexão do estado de execução permitem manipular o estado de execução no nível da linguagem, na forma de valores portáveis e serializáveis. Já que é o programador quem conhece a semântica de cada aplicação, ele deveria poder decidir como ajustar as variáveis que determinam o comportamento dos procedimentos de captura e restauração (através de chamadas diretas às primitivas da linguagem ou através de bibliotecas que implementam diferentes políticas). Em conseqüência, os requisitos de diferentes aplicações podem ser atendidos, aumentam-se as chances de uma restauração bem sucedida, e o desempenho da captura e restauração pode ser controlado, além de facilitar a depuração.

1.1 A tese proposta

Essa tese partiu do estudo das dificuldades associadas à implementação da migração heterogênea forte de computações em execução. Após estudar os trabalhos nessa área concluímos que há necessidade de que as linguagens de programação ofereçam mecanismos que permitam esta implementação. Resolvemos então estudar estes mecanismos. Linguagens de programação devem suportar a implementação de diferentes tipos de aplicações, como é o caso de migração e persistência heterogêneas. Deste modo, este trabalho se dedica ao

estudo do problema chamado de *suporte flexível para a captura e restauração heterogêneas*, e consiste em como produzir uma nova computação – como conseqüência de procedimentos como migração ou persistência de computações, possivelmente heterogêneos– cujo estado de execução seja equivalente ao original.

Acreditamos que o suporte das linguagens deva ser baseado na manipulação explícita da representação das estruturas básicas que definem o estado do programa. A nossa abordagem se baseia em dois mecanismos que chamaremos de *reificação* e *instalação* das estruturas da linguagem. O suporte para reificação e instalação de execuções foi implementado como uma extensão de Lua 5.1 [Ierusalimschy06] chamada de *LuaNua*. Várias facilidades reflexivas de Lua já oferecem algum suporte para captura e restauração. LuaNua estende o conjunto de funcionalidades reflexivas de Lua através da modificação e extensão da biblioteca de depuração.

Abordagens reflexivas, quando permitem a manipulação de estruturas navegáveis e componíveis, são meios poderosos e expressivos de implementar o suporte para várias aplicações. Eles deixam o programador no controle de cada aspecto do procedimento. Entretanto, esta abordagem afeta a usabilidade, sobrecarregando o programador e levando facilmente a erros [Kennedy04, SLW+07]. Espera-se que a API reflexiva seja usada através de bibliotecas intermediárias ("*bibliotecas de facilitadores*") que permitem uma certa customização para determinados conjuntos de aplicações, enquanto as primitivas oferecidas pela linguagem continuam acessíveis para aplicações ou propósitos mais específicos. Ao nos referirmos ao "programador" neste trabalho, estaremos falando tanto do usuário da API quanto do usuário dessas bibliotecas de mais alto nível.

A captura e restauração do estado de execução é uma operação intrinsecamente relacionada com o sistema em que a computação está sendo executada. Este estado pode ser descrito em termos do estado interno e a informação do ambiente que é a visão que a computação tem do sistema. A solução proposta nesta tese consegue lidar com a necessidade de manter o estado interno consistente ainda depois da restauração. Esta solução também permite manipular a visão que a computação tem do sistema, de forma a fazer a restauração possível. Entretanto, isto pode não ser suficiente nos casos em que o ambiente contém entidades especiais como descritores de arquivo e *sockets* abertos. Tratá-los de forma a permitir sua restauração correta requer a utilização de técnicas adicionais como a implementação de *wrappers*. Este problema é tratado em outros trabalhos ([LTBL97, DO99]) e está fora do escopo desta tese.

1.2

Organização do trabalho

O Capítulo 2 introduz os termos que serão usados ao longo do texto, em particular aqueles relacionados à migração heterogênea de computações. Já que a dificuldade fundamental para a implementação dessa técnica está na captura e restauração do estado de execução, discute-se também a forma como as linguagens de programação atuais oferecem esse suporte. Durante o estudo dos trabalhos relacionados, detectamos características comuns entre os métodos atuais em diferentes granularidades e linguagens. Isto motivou a proposta de uma visão diferente para as classificações dos métodos de implementação de migração heterogênea forte de computações, focando no suporte que as linguagens de computação oferecem para a captura e restauração de estado.

No Capítulo 3 se discutem os aspectos relacionados com o projeto de mecanismos de suporte a captura e restauração e a nossa abordagem para a captura e restauração flexível de computações, que é o foco desta tese. A API proposta é apresentada e a semântica operacional dessas funções é definida formalmente.

No Capítulo 4 é validada a efetividade da proposta através de implementações de diferentes exemplos.

O Capítulo 5 comenta detalhes relevantes da implementação de LuaNua. O último Capítulo destina-se às conclusões.

2
Migração heterogênea forte de computações

Este capítulo introduz os conceitos relacionados à migração heterogênea forte de computações e os problemas que os programadores encontram para a implementação desse tipo de migração relacionados à falta de suporte para captura e restauração de estado de execução, assim como soluções comuns a esses problemas. Um estudo mais extenso sobre esse tópico pode ser consultado em [MRS08].

2.1
Preliminares

Uma das dificuldades envolvidas na pesquisa sobre migração está relacionada à sobrecarga de termos utilizados nesta subárea. Por este motivo, nesta seção definiremos a notação a ser usada ao longo deste documento.

Técnicas de migração têm sido estudadas e aplicadas a diferentes níveis de granularidade, como processos [MDPW+00, JC04], threads [Funfrocken98, SMY99, TRVC+00, BHKP+04, JC04], closures [Cardelli95, CJK95], e registros de ativação [HWW93]. É muito comum referir-se às unidades de migração como *agentes móveis*, que podem ser descritos como entidades que podem transportar seu estado de um ambiente à outro. Os agentes móveis podem ser compostos por múltiplas threads que devem mover-se em conjunto, como em Wasp [Funfrocken98], ou podem compartilhar uma thread, como em Aglets [LO98]. Nessa tese usaremos indistintamente os termos computação, unidade de execução (EU) [FPV98], ou unidade de migração [Shub90], exceto quando for necessário especificar a granularidade. Aqui aderiremos à terminologia proposta por Cardelli [Cardelli97, Cardelli99], em que o termo *mobile computation* [1] se diferencia da computação móvel (*mobile computing*) em que o primeiro trata da mobilidade lógica das computações, e o segundo, da mobilidade física dos dispositivos. A migração de computações consiste basicamente na suspensão de uma computação, e o armazenamento e serialização do seu estado que será transferido até um nó remoto em que a computação será reiniciada. O estado inclui o estado interno (em geral, conteúdos do heap, da

[1] na falta de uma tradução mais adequada diferente de "computação móvel" –já bem estabelecida–, usaremos o termo *mobilidade (ou migração) de computações*

pilha, dos registros, variáveis globais) e a informação do ambiente, que pode considerar-se como tudo o que é externo à computação (descritores de arquivo, bibliotecas, identificadores de usuário e de processos). Na realidade, Fuggetta et al. [FPV98] definiram migração e *clonagem remota* como os mecanismos que suportam mobilidade forte. Diferente da clonagem, a migração envolve um último passo adicional, que consiste no cancelamento da computação original. Entretanto, neste trabalho, usaremos o termo migração como sinônimo de mobilidade exceto quando a diferença for relevante.

A migração é comumente chamada de transparente quando os efeitos do movimento estão ocultos ao usuário e à aplicação. Isto só pode ser conseguido tratando todas as referências da computação a objetos e recursos. O suporte para transparência é difícil de implementar sem depender diretamente do sistema operacional. Por este motivo, a transparência é comumente tratada com restrições a esse nível (por exemplo, as conexões podem não ser restauradas, pode ser limitada a restauração da entrada/saída). Em geral, atualmente a transparência não é mais considerada como uma questão crítica, pois se por um lado ajuda em termos de complexidade, pelo outro não permite o controle de erros e as possíveis otimizações que possam ser executadas baseado na localização.

O programa transferido pode capturar e restaurar o seu estado ou então isso pode ser executado por um serviço externo. Estas abordagens serão distinguidas como manipulação *interna* e *externa*, respectivamente.

Por outro lado, a classificação tradicional dos mecanismos de mobilidade em fraca [FPV98] ou forte não resulta clara, pois existem na realidade diferentes "graus" de força. Assume-se comumente que na mobilidade fraca são transferidos o código e opcionalmente alguns dados, enquanto na mobilidade forte o estado de execução também migra, de forma que a computação pode ser reiniciada do ponto em que foi suspensa. Não obstante, é difícil migrar o estado de execução como um todo, e na realidade nem sempre é necessário. Ainda, em alguns sistemas, a suspensão somente é possível quando a pilha estiver vazia, por exemplo, quando está esperando em um loop de eventos, de forma que a diferença entre fraca e forte desaparece. Todavia, é possível encapsular a continuação de uma execução através de dados e código que podem ser migrados usando um mecanismo de migração fraca [MRS08]. Por este motivo, neste trabalho consideraremos como fortes aqueles mecanismos capazes de produzir novas computações cujo estado de execução seja equivalente ao da computação original. Limitações à quantidade de informação contida na pilha de execução não são interessantes.

A disponibilidade de mecanismos de migração forte é desejável em apli-

cações de balanceamento de carga para programas executando computações intensivas e/ou de longa duração. Também na área de agentes móveis, pois a migração fraca contraria o comportamento esperado desses agentes que deveriam continuar a execução automaticamente assim que fossem reativados no destino [BD01]. Entretanto, atualmente muitos sistemas oferecem somente migração fraca. Wang et al. [WHB01] afirmam que a modularidade oferecida pela mobilidade forte comparada à mobilidade fraca tornam a mobilidade forte mais confiável, porque é mais fácil raciocinar sobre o código e depurá-lo.

Qualquer que seja o método de migração, ela deve ser *correta*. Após uma migração correta, o resultado da computação migrada é igual ao resultado da mesma computação não migrada, dada a mesma entrada [Smith97]. Lu e Liu [LL87] estabelecem dois requisitos que devem ser satisfeitos com esse objetivo: os estados da computação na origem e no destino deverão ser iguais, e a computação terá uma visão consistente do sistema (o ambiente) em todo momento. Dessa forma, para se ter uma migração correta não é suficiente restaurar exatamente a informação que descreve a computação original: isto não garante que a semântica do processo não será afetada pelo processo de migração. Tome-se, por exemplo, o caso de uma aplicação que devolva como um dos resultados finais o tempo de execução. Ainda que as máquinas de origem e destino estivessem sincronizadas, o resultado da medição incluiria a latência da migração, levando a um resultado incorreto. Em geral, formas de permitir a migração de processos que interagem com funções ou valores dependentes do sistema são: (i) a manipulação dessas funções de forma a oferecer um ambiente consistente na restauração; (ii) a restrição da suspensão a momentos em que a visão do sistema não inclua esse tipo de funções ou valores. Em efeito, a *migrabilidade* (ou capacidade de uma computação ser migrada corretamente) está relacionada ao estado da computação no instante da migração e ao entendimento dos efeitos dos fatores que podem impedir a migração de forma a poder manipulá-los [Smith97]. Fatores que contribuem na migrabilidade são: restrições da linguagem, análise estática, verificação de código em tempo de execução, suporte para tempo de execução, quantidade de informação de tipos, funcionalidades da ferramenta de migração, e similaridades entre as arquiteturas [Smith97]. Neste caso em que estudamos a captura e restauração de estado para um leque maior de aplicações além da migração, o termo restaurabilidade é mais adequado que aquele de migrabilidade.

A migração pode ser iniciada de dentro da computação (chamada de *pró-ativa*, ou *subjetiva*) ou então de fora da computação(chamada de *reativa*, *objetiva* ou *forçada*). Embora freqüente, a migração subjetiva afeta a separação entre a lógica da aplicação e a da mobilidade [PMOY06]. Por outro lado, na

migração objetiva aparecem vários problemas, derivados do fato de que nesse caso a migração está fora do controle do programador da aplicação. Além de problemas relacionados com autoridade, sobre quem teria direito a migrar a computação [CG98], outros requisitos podem surgir se a aplicação precisar de informação relativa à localização ou a mudança dela. Uma questão fundamental é que a computação deve ser interrompida de forma portável e somente quando estiver em estado consistente, ou seja, em uma forma reiniciável, estável e coerente [MP96]. Uma instrução de código fonte ou *bytecode* pode ser composta de várias instruções de código de máquina, cujo número pode diferir entre arquiteturas devido às diferenças em conjuntos de instruções. As interrupções não devem ocorrer enquanto uma instrução está sendo executada (exceto quando as operações restantes não têm efeitos colaterais) ou então a correção da restauração será afetada.

A consistência, então, é garantida permitindo a interrupção somente em determinados pontos, chamados de *pontos de ônibus* [SJ95], *pontos de migração* [TH91], *poll points* [FCG00, CS02], *pontos de preempção* [SH98] ou *pontos de adaptação* [JC04]. Neste texto eles serão chamados de pontos de migração. A justificativa desses pontos vem de que uma computação procedural pode ser modelada como uma progressão através de uma seqüência de estados bem definidos que são pontos na execução em que o estado de uma computação é equivalente ao de qualquer outra implementação da computação [BSS94].

A definição de pontos lógicos em que a migração é permitida tem outras motivações além da migração objetiva, como resolver o problema das diferentes localizações que o ponto de execução corrente poderia ter no segmento de texto em diferentes arquiteturas. Nesse caso, eles atuam como etiquetas para simular contadores de programa. O posicionamento desses pontos precisa ser cuidadoso, pois em excesso, geram uma sobrecarga em espaço e tempo de execução, e em número reduzido, podem provocar uma latência excessiva na resposta a requisições de suspensão. Sendo que os métodos de suspensão usualmente dependem da plataforma, a estratégia mais popular se baseia em *pooling*, que é executado nos pontos de migração.

2.2 Captura e restauração do estado de execução

A migração oferece uma ilusão de que a computação está sendo movida, quando na verdade se está criando uma nova computação no destino, a partir de informações parciais do estado interno e o ambiente da computação migrante, que irá executar no novo contexto local. A captura e restauração dessas informações do estado de execução da computação são problemas chave

nesse nível. Mais complexa que a migração em ambientes homogêneos, a migração heterogênea implica na necessidade de informação detalhada sobre cada dado devido às diferenças entre as plataformas em conjuntos de instruções e representações de dados.

Nesse sentido, a virtualização oferece uma alternativa interessante devido a sua crescente popularidade, que permitiria ter uma base instalada o suficientemente ampla para uma determinada aplicação. Os monitores permitem implementar sistemas de migração de forma homogênea, assim, o VMMigration [CFH+05] foi implementado sobre o Xen Virtual Machine Monitor. Esta abordagem tem também as vantagens de permitir sandboxing e a execução de vários sistemas operacionais de uma vez. Entretanto, este tipo de implementação efetivamente precisa da disponibilidade de uma determinada máquina virtual e tem uma granularidade alta e inflexível.

2.2.1 Captura e serialização

O passo de captura e serialização heterogêneas é o que segue à suspensão da execução. O estado capturado deve ser o mínimo requerido para gerar uma migração correta: isto permite melhorar o desempenho da migração e influi na migrabilidade, diminuindo a chance de se encontrar problemas relacionados à dependência do sistema. Ainda, depois da captura, a aplicação pode passar por processos de manutenção, atualização das entidades capturadas ou das dependências. Dessa forma, quanto menor for o volume capturado, menor será o esforço para adaptá-lo a novas implementações ou a chance da recuperação não poder ser feita caso essas adaptações não tenham sido previstas. Nesse sentido, é interessante aproveitar os recursos disponíveis no contexto local do destino executando as substituições apropriadas. A correção dessas substituições é fundamentalmente semântica, portanto é o programador da aplicação quem deve decidir permití-las ou não. Isto também permite evitar a criação de dependências remotas desnecessárias, que afetam a escalabilidade e a tolerância a falhas.

Relevante nessa etapa é o conceito de *reflexão*, que é a capacidade de um sistema de programação de observar e mudar o seu próprio comportamento. A migração heterogênea precisa do conhecimento do tipo de cada valor para poder executar a extração e serialização dos segmentos de memória corretamente. Em linguagens com funcionalidades reflexivas este problema pode ser resolvido diretamente. Entretanto, a maioria das linguagens compiladas não mantém informação de tipos em tempo de execução. Ainda, linguagens comuns como ANSI-C são *type-unsafe*, o que implica que é possível que o tipo

dinâmico de um valor em algum momento seja diferente de seu tipo declarado estaticamente. Soluções possíveis são restringir as características não seguras da linguagem (*unsafe features*) ou então modificar os compiladores de forma a lidar com essas características [SH98]. No entanto, ambas as soluções podem resultar em um comportamento fora do padrão da linguagem. Por outro lado, o acesso à pilha freqüentemente é restrito por questões de segurança. Como se verá, a falta de reflexão também afeta a restauração.

O fato de estarmos considerando ambientes de memória distribuída, implica na necessidade de transferir o estado todo da computação, incluindo os objetos alcançáveis dela, ou vinculá-la a referências remotas. Em conseqüência, uma semântica específica para identidade é necessária já que as computações podem se mover para fora e dentro do ambiente original e ainda serem consideradas a mesma computação.

A transferência do grafo de dependências da computação pode ser feita através de referências ou cópia. As referências podem ser remotas ou podem ser resolvidas localmente no destino através de um descritor. Resolver as referências no contexto local é uma solução atraente, mas requer informação do contexto no destino e, em conseqüência, alguma coordenação entre as plataformas de execução origem e destino. As referências remotas podem comprometer a escalabilidade e a tolerância a falhas. Por outro lado, a captura das dependências através de cópia está limitada pela complexidade e tamanho do fecho transitivo. O problema nesse caso consiste em achar a mínima quantidade de entidades que deveriam migrar com a computação, garantindo a correção do procedimento. Duas abordagens podem ser assumidas: chamadas de *rasa* (*shallow*) como em JavaGo [SMY99], ou *profunda* (*deep*) como na proposta de Tack et al. para a formalização da serialização e minimização de grafos em AliceML [TKS06]. Na primeira abordagem, o programador pode indicar as referencias migráveis explicitamente, sendo que as não migráveis serão anuladas. Além de requerer anotações adicionais na linguagem, essa solução aumenta o trabalho do programador e pode produzir *no-shows* durante a restauração. Por sua vez, a cópia profunda considera que todos os objetos relacionados devem ser migrados. O grafo de objetos é construído transparentemente para o usuário. Implica tipicamente em maior sobrecarga já que mais dados irão ser capturados e transferidos.

Existem sistemas que oferecem ao programador a flexibilidade de decidir como objetos particulares serão serializados, o que pode influir na restaurabilidade da computação. Exemplos dessa abordagem são o Coda [McAffer95] e o Pluto [Sunshine2005]. Eles se baseiam na noção de *descritores de marshalling*, que permitem a especificação do método de *marshalling* para cada objeto.

Para possibilitar a transferência do estado capturado, é necessário que antes da transmissão cada valor seja convertido a uma representação transmissível, dependendo do seu tipo. Este processo é chamado de *serialização*, *pickling* ou *marshalling*, enquanto que a operação oposta é chamada de *de-serialização*, *unpickling* ou *unmarshalling*. A informação a ser transferida, composta pelo estado de execução da computação, seu código e ambiente, pode se expressar na representação usada na origem ou no destino (usado, por exemplo, na estratégia *Receiver Makes Right (RMR)* [ZG95]), ou então na forma de uma representação independente da arquitetura, como no esquema *External Data Representations (XDR)* [RFC4506]. A alternativa de conversão no destino é preferida à conversão na origem porque no caso das duas arquiteturas serem iguais, este passo pode ser evitado. Em geral, a escolha do esquema de conversão é fixado na implementação do pacote de suporte a migração.

No caso do código, a representação independente da plataforma consiste no envio de código fonte ou interpretado. A recompilação do código fonte no destino elimina as implicações das diferenças entre as arquiteturas, mas provoca uma certa demora no reinício devido justamente ao processo de recompilação. O uso de linguagens interpretadas tem as vantagens da adaptação dinâmica e a portabilidade resultante da presença de uma máquina abstrata comum em ambos os lados, mas traz perdas de desempenho causadas pela interpretação quando comparado à execução de código compilado. Por outro lado, a alternativa de usar como representação o código de máquina da arquitetura destino obriga a manter um conjunto de arquivos executáveis para cada possível plataforma. Isto afeta a escalabilidade da aplicação, pois o arquivo executável apropriado para cada plataforma suportada precisa ter sido gerado previamente, e implica em alguma sobrecarga de armazenamento para manter esses arquivos.

2.2.2 De-serialização e restauração

Nesse passo, uma nova computação é criada a partir da informação transferida. Os vínculos aos recursos são reestabelecidos se necessário. Nesse ponto, erros podem surgir como: a detecção de estado incompleto ou inconsistente, impossibilidade de vínculos com recursos locais ou remotos, *overflow* de espaço ou memória, quebra do nó. Nesses casos, a migração deveria ser abortada e a execução original continuar normalmente.

A restauração requer a reconstrução do grafo de objetos e do estado interno da computação (incluindo pilha de chamadas, contadores de programa, variáveis locais) e sua instalação no espaço de memória de forma que a

execução possa ser reiniciada a partir da instrução seguinte à suspensão. O problema do restabelecimento do contador de programa não é trivial: nem todas as linguagens permitem estabelecer um ponto específico para continuar a execução. Também pode ser necessária a definição de contadores de programa lógicos por portabilidade. Java, por exemplo, facilita a execução do código recebido através da carga dinâmica de classes, mas não oferece suporte para continuar a execução a partir da última instrução executada. A migração de threads envolve adicionalmente problemas relacionados à sincronização: threads e locks devem ser restabelecidos numa determinada ordem.

Seguindo a restauração, a última etapa da migração seria o reinício da execução e, em se tratando de migração e não de clonagem, o cancelamento da computação original. As referências dessa computação podem por outro lado estar associadas a outras computações em curso e devem ser tratadas independentemente.

2.2.3 Características específicas das linguagens

O problema da captura e restauração no nível da aplicação está muito relacionado às características da linguagem. Para ilustrar como as características das linguagens influenciam nas implementações que precisam de captura e restauração, mostramos em seguida dois exemplos de linguagens das mais utilizadas para este tipo de implementação.

ANSI-C

A linguagem C oferece mecanismos que permitem ao programador "enganar" o sistema de tipos, mas podem levar a erros de inferência de tipos durante a extração dos dados na etapa de captura que podem fazer com que o programa não possa ser restaurado. Entretanto, esses não são os únicos fatores que tornam um programa C não restaurável. Em um estudo sobre migração no contexto do desenvolvimento do sistema TUI [Smith97, SH98], Smith and Hutchinson determinaram os problemas que provocam não restaurabilidade em programas escritos em ANSI-C em 4 classes:

- conflito de tipos: ocorre quando uma posição de memória tem mais de um tipo associado, de forma que o sistema não vai saber como extrair o dado corretamente. As características que provocam este problema são: *unions*, *casting de ponteiros*, não correspondência de parâmetros, reuso de armazenamento variável e falta de checagem de tipos na compilação independente;

- falta de informação de tipos: o compilador não gera ou não consegue gerar informação de tipos suficiente, por exemplo, em ponteiros de tipo *void*, listas de argumentos de tamanho variável e nas funções *setjmp* e *longjmp*;
- falsa identificação: tentativa de migrar dados que não são parte do programa. Ocorre, por exemplo, em casos de *casting* de valores inteiros a ponteiros, ponteiros que se referem a localizações de memória ilegais, *dangling pointers* e ponteiros não inicializados.
- incorreta conversão de valores: ocorre quando o dado é convertido incorretamente no destino. É devido a diferenças com a origem em arquitetura (operador *sizeof*, perdas na conversão de formatos) e conjunto de caracteres (significado de *char*).

Os problemas relacionados com a tipagem insegura da linguagem (*casting*, *unions*) tem sido evitados em algumas implementações [SH98, CS02] através de restrições na linguagem, ou seja, programando em estilo *type-safe*. Os programas que violam estas características são declarados não migráveis. Na realidade, este método é muito restritivo: programas *type-unsafe* podem ser migrados desde que estas características sejam detectadas e tratadas [Smith97]. Por outro lado, programas escritos em linguagens *type-safe* que façam chamadas ou usem variáveis dependentes do sistema no momento da migração podem ser não-migráveis.

Java

Apesar da linguagem Java ser uma das mais comumente usadas para a implementação de sistemas de agentes móveis, a informação sobre o estado interno que a máquina virtual disponibiliza é restrita [IKKW02].

O estado da pilha também não é portável: na maioria das JVMs ela é implementada como uma estrutura de dados nativa, ou seja, uma estrutura C [BHKP+04]. Isto faz com que a informação contida na pilha seja dependente da arquitetura. Para representar o estado da pilha em um formato independente da plataforma, é necessário um passo de tradução durante a serialização, assim como a operação contrária para a de-serialização. Isto implica em traduzir os valores das variáveis locais e operandos a valores Java, para o qual é necessário acesso ao tipo dos valores. No entanto, a informação de tipos está embutida no bytecode dos métodos que guardam os dados na pilha e não está disponível em tempo de execução. Usando a Java Debugging Architecture, é possível extrair o estado de execução [IKKW02]. Entretanto, não existe forma de restaurar o contador de programa. Existe um grupo de publi-

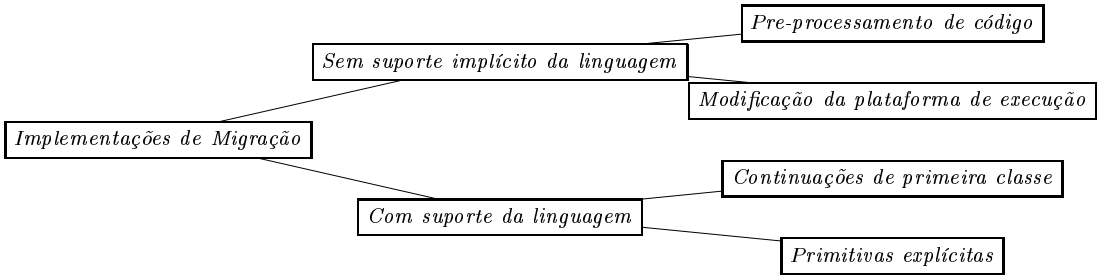

Figura 2.1: Métodos de implementação de migração heterogênea forte de computações

cações dedicadas à solução dos problemas relativos à migração de threads de Java [Funfrocken98, IKKW02, TRVC+00, BHKP+04]. As técnicas comumente descritas têm desvantagens no desempenho da serialização ou na portabilidade.

2.3 Classificação do suporte das linguagens para captura e restauração de estado

Como discutido em [MRS08], com relação ao suporte das linguagens de programação para a captura e restauração do estado de execução, as implementações podem ser divididas em dois grupos: os que se baseiam ou não em linguagens com este suporte, como mostra a figura 2.1.

2.3.1 Implementações sobre linguagens sem suporte para captura e restauração de estado

A seguir detalhamos os métodos mais utilizados para a implementação de captura e restauração de estado em aplicações de migração de computações em linguagens sem o necessário suporte. Finalizamos o capítulo analisando esses métodos e os problemas derivados da falta desse suporte.

Pré-processamento do código do usuário

A abordagem de *pré-processamento do código do usuário*, também chamado de *transformação fonte-fonte*, é um mecanismo de manipulação interna que consiste em modificar o programa do usuário inserindo fragmentos de código que permitem ao programa salvar o próprio estado de execução e reiniciar a computação por si próprio. O programa de usuário pode estar em formato fonte, compilado ou bytecode. O estado da computação é abstraído ao nível da linguagem, garantindo assim portabilidade.

O método mais usado é o que chamaremos de *stack unwinding* [SMY99, FCG00, BD01], que explicamos a seguir. A transformação inclui a inserção, em

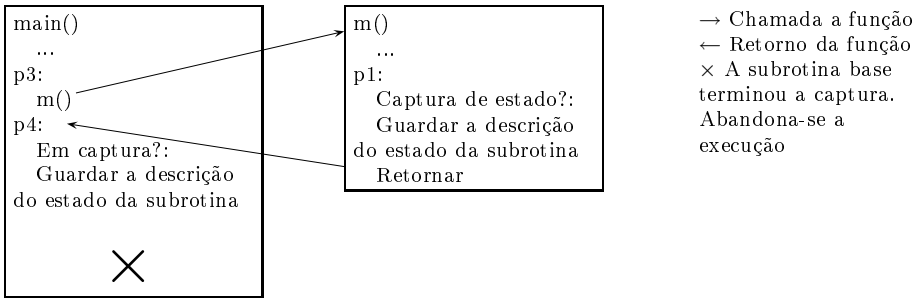

Figura 2.2: Mecanismo nativo de retorno de subrotinas: captura de estado

cada função, de código que registra todas as variáveis locais, parâmetros e a atualização do contador lógico de instruções, numa variável local ou objeto de *backup* criado para este propósito. A restauração geralmente requer a execução parcial da aplicação. Os estados que a aplicação atravessa durante o processo de captura e restauração são:

Execução Normal → Captura de Estado → Restauração de Estado → Execução Normal

A *captura* consiste em guardar o estado do procedimento corrente, retornar à subrotina que fez a chamada e repetir o procedimento recursivamente até chegar na subrotina principal, em que a aplicação termina. Este procedimento é ilustrado na figura 2.2. Na restauração, como aparece na figura 2.3, o código inserido no início do programa detectará o estado de restauração e cada subrotina da pilha armazenada será recursivamente chamada e restaurados os dados de seu frame local. Depois será executado um salto até o ponto de migração em que foi feita a captura da subrotina corrente. Para este mecanismo funcionar, é necessário que exista um ponto de migração antes e depois de cada chamada a subrotina. Uma variante desse método consiste no uso dos mecanismos

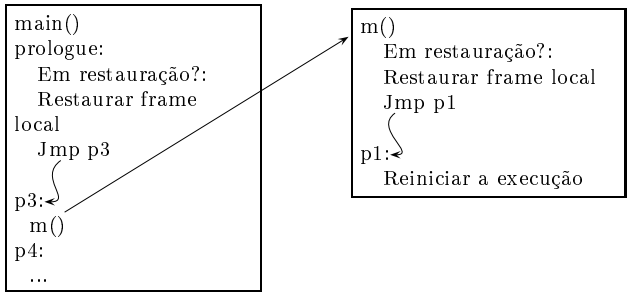

Figura 2.3: Mecanismo nativo de retorno de subrotinas: restauração de estado

de tratamento de exceções da linguagem. O compilador insere expressões *try-catch* no código fonte que executam a rotina de captura quando é lançado o evento correspondente. A captura da pilha se baseia na propagação do erro

ou exceção até o nível base. Esse método admite a captura de continuações parciais e delimitadas (fragmentos da computação) [SMY99].

Problemas particulares emergem em sistemas que oferecem persistência de threads. Na recuperação, todas as threads ativas durante a captura devem ser reiniciadas criando novas threads que serão inicializadas com os conteúdos dos seus pares na origem. Problemas de sincronização podem emergir durante a restauração devido à imprevisibilidade da situação de cada thread no momento da captura de estado. Este problema é tratado de formas diferentes, que vão desde restringir as computações a somente uma thread até oferecer anotações que indiquem que a thread está pronta para migrar.

Outro problema que deve ser tratado é o referente a como capturar e restaurar os valores temporários (resultados parciais) de operações interrompidas. Por exemplo, em:

d = f(x) + g(x)

se a computação é interrompida enquanto g(x) está sendo calculada, o resultado de f(x) deve ser capturado de sua localização temporária, e restaurado no destino para permitir continuar a computação. Isto implica que o número, tipo e localização desses temporários deve ser conhecido. Uma solução para este problema é inserir essa informação dentro do código durante a compilação. Outra abordagem é evitá-lo, dividindo a expressão e fazendo os temporários explícitos. Naturalmente, existe também a opção de não permitir a transformação durante a avaliação de uma expressão.

Este método pode ser enxergado como uma forma de *Continuation Passing Style (CPS)*. A esencia do CPS consiste em fazer explícitos aspectos comumente implícitos na continuação, como retornos de procedimentos e valores temporários. O CPS produz constantemente um valor mais a continuação da execução. Entretanto, obriga a adotar um estilo de programação que é considerado inconveniente e pode levar a erros.

Em comparação com a instrumentação de código fonte, trabalhar com o bytecode tem como vantagem maior disponibilidade do código, um conjunto de instruções mais amplo e melhor desempenho. O requisito de disponibilidade é uma séria desvantagem no pre-processamento de código fonte no caso de bibliotecas e código legado.

Modificação/extensão da plataforma de execução

A modificação/extensão da plataforma de execução é uma abordagem baseada em manipulação externa. O código do usuário não precisa ser modificado, e a abordagem é válida tanto para código nativo quanto interpretado. Entretanto, existem várias diferenças na forma de tratar ambas as represen-

tações. Linguagens interpretadas mantêm valores temporários na pilha, que são transferidos com o restante dos dados durante a migração. Por outro lado, na abordagem baseada em código nativo (ou a modificação do compilador), a informação relativa à localização e tipo dos valores temporários não está diretamente disponível. Esta informação poderia ser inserida pelo compilador se for convenientemente modificado. Basicamente, a abordagem de modificação/extensão da plataforma de execução no caso de código nativo consiste em modificar o compilador de forma a gerar informação para fazer os programas restauráveis. Isto requer também a implementação de uma plataforma de execução que execute as tarefas relacionadas com a migração/persistência da computação.

Em linguagens interpretadas, é a máquina virtual que é modificada de forma a executar as tarefas de migração/persistência. Nesse caso não é necessária uma plataforma adicional para este tipo de requisito. Um problema comum a várias máquinas virtuais é que a informação de tipos está embutida no bytecode dos métodos que guardam os dados na pilha, fazendo a inferência de tipos difícil. Quando se interpreta uma instrução, é possível saber o tipo dos dados envolvidos. Por este motivo, uma abordagem possível para a determinação do tipo dos valores na pilha é estender o interpretador de forma a manter uma estrutura paralela contendo esta informação. No entanto, esta solução não é compatível com *JIT* (*Just-In-Time compilation*) e gera sobrecarga em tempo de execução para a aplicação. A alternativa é analisar (*parse*) o bytecode somente na hora da serialização. O custo desta forma, é transferido como latência de serialização, fazendo com que este método seja mais aconselhável em aplicações em que estas operações sejam pouco freqüentes.

A grande desvantagem da abordagem de modificação/extensão da plataforma de execução é a portabilidade, pois todas as plataformas de execução envolvidas na aplicação podem precisar de modificações.

2.3.2 Linguagens com suporte para captura e restauração de estado

O suporte das linguagens para a captura e restauração heterogênea do estado de execução de computações pode ser caracterizado como a provisão de construções que permitam ambos procedimentos em um ambiente heterogêneo. Este suporte pode ser expressado tanto através de mecanismos como continuações de primeira classe –tipicamente oferecidas por linguagens funcionais– quanto por primitivas explícitas para manipulação de estado que permitam a construção de continuações.

A provisão de continuações de primeira classe permite a manipulação de continuações como estruturas explícitas que podem ser tratadas como valores. Quando este suporte se estende à extração e instalação heterogêneas da estrutura da continuação, a implementação de persistência e migração heterogêneas é direta. Usar continuações tem a vantagem de permitir implementar migração forte através de mecanismos de migração fraca. No entanto, continuações são do domínio quase exclusivo das linguagens funcionais, cujas limitações de desempenho são conhecidas. A alternativa de oferecer primitivas para manipulação de estado será provavelmente a escolha no caso de linguagens orientadas a objeto ou estruturadas, onde as continuações não são usualmente estruturas explícitas.

Discussão

As categorias de implementação, no caso de linguagens sem suporte implícito, podem ser comparadas em termos de sobrecarga em tempo de execução, sobrecarga representada pela migração, sobrecarga em espaço de armazenamento, portabilidade, completude e facilidade de manutenção.

O pré-processamento de código tem a propriedade da garantia de portabilidade. Por outro lado, implica mudanças no fluxo de programação, maiores sobrecargas de espaço e tempo de execução quando comparado a outros métodos, causadas pelas instruções inseridas. A sobrecarga da migração é também alta, devido ao processo de reconstrução da pilha, que simula uma execução normal.

A abordagem baseada na modificação da plataforma de execução exibe melhor desempenho. Nessa abordagem, não é necessário modificar o código do usuário, dessa forma não existem requisitos de disponibilidade de código. A grande desvantagem desse método é a perda de portabilidade.

Finalmente, exceto na transformação de código fonte, a manutenção do código apresenta problemas em todos os métodos. Esses métodos dependem de versões de bytecode e detalhes da plataforma de execução com os quais geralmente as linguagens não têm compromisso de compatibilidade entre versões. A completude também pode ser afetada: por exemplo, as cláusulas *try-catch* não podem ser migradas quando a implementação se baseia nos mecanismos de tratamento de exceções da linguagem.

Poucos trabalhos em serialização/de-serialização focam no problema da captura e restauração de execuções em ambientes heterogêneos no nível da linguagem. Como mostra esta revisão, o suporte para captura e restauração atualmente é majoritariamente opaco e não oferece formas de customização para requerimentos específicos. Deste estudo resulta claro que o suporte

das linguagens à captura e restauração é uma necesidade real quando se implementam aplicações que precisam de captura e restauração heterogênea de computações, como é o caso da migração.

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

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 (*):

$$\langle\langle \sigma, \langle S, E, b\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \langle\langle \sigma, \langle b\ S, E, C, D\rangle, A\rangle, \Sigma\rangle \tag{3-7}$$

$$\langle\langle \sigma, \langle S, E, x\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \langle\langle \sigma, \langle v\ S, E, C, D\rangle, A\rangle, \Sigma\rangle \quad (*) \tag{3-8}$$

onde $v = \Sigma(E(x))$

$$\langle\langle \sigma, \langle b_n ... b_1\ S, E, prim_{o^n}\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \langle\langle \sigma, \langle v\ S, E, C, D\rangle, A\rangle, \Sigma\rangle \tag{3-9}$$

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*).

$$create\ \langle x\ C'\rangle E' ::$$
$$\langle\langle\sigma, \langle\langle x\ C'\rangle E'.S, E, create.C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \tag{3-14}$$
$$\langle\langle\sigma, \langle\sigma'.S, E, C, D\rangle, A\rangle, \Sigma[\sigma' \leftarrow \langle\langle x\ C'\rangle E', \emptyset, \emptyset, \emptyset\rangle]\rangle$$
$$onde\ \sigma' \notin dom(\Sigma)$$
$$resume\ \sigma'\ v ::$$
$$\langle\langle\sigma, \langle\sigma'\ v\ S, E, resume\ C, D\rangle, A\rangle, \Sigma\rangle \longmapsto \tag{3-15}$$
$$\langle\langle\sigma', \langle v\ S', E', C', D'\rangle, \langle\sigma, \langle S, E, C, D\rangle, A\rangle\rangle, \Sigma[\sigma' \leftarrow nil]\rangle$$
$$onde\ \langle S', E', C', D'\rangle = \Sigma(\sigma')$$
$$yield\ v ::$$
$$\langle\langle\sigma, \langle v\ S, E, yield\ C, D\rangle, \langle\sigma', \langle S', E', C', D'\rangle, A\rangle\rangle, \Sigma\rangle \longmapsto \tag{3-16}$$
$$\langle\langle\sigma', \langle v\ S', E', C', D'\rangle, A\rangle, \Sigma[\sigma \leftarrow \langle S, E, C, D\rangle]\rangle$$

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

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.

4
Reificação em Lua

A manipulação explícita da representação das computações permite ao programador implementar diversas políticas e introduzir estruturas não diretamente oferecidas pela linguagem. Para estudar quais as funcionalidades reflexivas que deve oferecer uma linguagem com suporte a captura e restauração heterogêneas, desenvolvemos uma extensão da linguagem Lua que oferece este suporte, chamada de LuaNua. A linguagem Lua foi escolhida pelas funcionalidades reflexivas que já oferece, além de outras características que são descritas neste capítulo. Este capítulo também mostra o poder do suporte a captura e restauração heterogêneas baseado em manipulação, através de exemplos das várias estruturas e aplicações que podem ser implementadas.

4.1
Breve introdução a Lua

Lua é uma linguagem interpretada, procedural e dinamicamente tipada. Os valores Lua podem ser de tipo *boolean*, *lightuserdata*, *number*, *string*, *table*, *function*, *thread* e *userdata*. A linguagem possui coleta de lixo. O suporte para concorrência é oferecido através de *co-rotinas*, que implementam suporte para multithreading cooperativo. A transferência de controle está baseada nas primitivas resume/yield. As tabelas Lua implementam arrays associativos, ou seja, elas podem ser indexadas usando qualquer valor da linguagem, exceto nil.

Lua possui escopo léxico. Isto significa que funções aninhadas tem acesso total às variáveis locais das funções mais externas. Funções que referenciam variáveis livres no contexto léxico são chamadas de *closures*. As closures são valores de primeira classe em Lua. Ou seja, elas podem ser armazenadas em estruturas de dados, passadas como argumentos a outras funções e retornadas como valores de outras funções. As funções podem ser escritas tanto em Lua quanto em C. Lua é baseada em *protótipos*. Quando uma função é compilada se gera um protótipo que contém o bytecode das instruções da função, seus valores constantes e informação de debug. Quando uma função é inicializada, se cria uma nova closure que contém uma referência ao respectivo protótipo, uma referência ao *ambiente* e um array de referências às variáveis locais das

funções mais externas, chamados em Lua de *upvalues*.

Metatabelas são tabelas regulares de Lua que permitem mudar o comportamento de um valor em operações não definidas (por exemplo, para somar tabelas). Para cada uma das operações contempladas, Lua define uma chave específica (por exemplo, __add para a soma), chamada de evento. Quando Lua executa uma dessas operações sobre um valor, verifica se este valor tem uma metatabela com o evento correspondente. Nesse caso, o valor associado com a chave (chamado de metamétodo) determina como Lua executa a operação. Tabelas e userdata têm metatabelas individuais, enquanto os outros tipos somente têm uma metatabela por tipo. Lua oferece o método *getmetatable* para consultar a metatabela de qualquer valor. A metatabela de tabelas pode ser mudada através da função *setmetatable*. Outros valores precisam da biblioteca de debug ou a API C para fazer isto. Metatabelas são estruturas convenientes para implementar reflexão. Elas não somente permitem interceptar certos comportamentos como oferecem a possibilidade de modificá-los em tempo de execução.

Lua oferece primitivas que permitem reificar e instalar funções Lua representadas como strings de bytes. Entretanto, esta representação não é facilmente navegável e não é portável, podendo requerer tradução em caso das arquiteturas origem e destino serem diferentes. Lua não dá suporte à serialização de co-rotinas, mas oferece mecanismos para incorporar bibliotecas de terceiros facilmente. A biblioteca Pluto [Sunshine2005] permite a serialização profunda de valores Lua. Apesar da serialização poder ser feita de forma opaca, Pluto permite especificar o método de serialização para cada objeto, oferecendo um certo grau de controle sobre o procedimento e permitindo, por exemplo, que mesmo entidades dependentes do sistema possam ser serializadas. Outra opção para serializar entidades que incluem valores não serializáveis é o *rebinding*: pode ser definido um descritor para o objeto, que será substituído na de-serialização. Problemas da serialização oferecida em Pluto são (i) a granularidade alta, pois Pluto não oferece serialização no nível de registros de ativação (ii) a representação retornada pela serialização é uma string de bytes, que não é facilmente navegável, nem manipulável. Finalmente, a serialização em Pluto não é heterogênea. Por exemplo, não oferece tradução quando a *endianness* das arquiteturas origem e destino são diferentes. Entretanto, já que a composição da representação é identificável –ou distinguível– em tipo e valor, as transformações necessárias podem ser implementadas de forma relativamente simples, modificando as funções de de-serialização de Pluto.

Já que o nosso interesse está no desenvolvimento de um suporte flexível para a captura e restauração de estado, procuramos vias de aumentar as ca-

pacidades reflexivas da linguagem de forma a permitir estas operações. Lua já disponibiliza informação de tipos, o que é necessário para a correta extração dos dados. Lua permite a carga dinâmica de código fonte ou bytecode. Também oferece outras funcionalidades reflexivas, que incluem o acesso ao ambiente, à metatabela dos objetos, ao nome das variáveis locais e a upvalues existentes. O fato de ser uma linguagem interpretada garante portabilidade para a informação representada no nível da linguagem. Alem disso, a simplicidade da concorrência em Lua evita a necessidade de tratar problemas de sincronização. As co-rotinas Lua são construções *stackful*, permitindo suspender (e reiniciar) a execução em um nível arbitrário de chamadas a funções. Ter as co-rotinas como valores de primeira classe permite o tratamento homogêneo de valores e execuções na serialização, já que as co-rotinas encapsulam execuções. Internamente, a informação sobre a execução está concentrada na pilha, organizada em frames correspondentes a cada função ativa. Finalmente, Lua oferece parte da sua funcionalidade através de bibliotecas. LuaNua é basicamente uma extensão do conjunto de recursos reflexivos de Lua com um grupo de funcionalidades que discutimos a seguir.

4.1.1 LuaNua, o projeto

Neste trabalho iremos nos concentrar na implementação de funcionalidades reflexivas na linguagem, que permitam a manipulação do estado de execução. Em particular, o requisito de conexão causal colocado em [Maes87] não é uma prioridade nesta proposta. A modificação da representação é feita offline: ela não implica necessariamente na modificação da execução. Esta representação pode ser vista como um *snapshot* do estado da computação cuja consistência cessa na medida que a computação progride. É necessária uma operação explícita para sincronizar a representação da computação com a execução. Esta escolha vem do compromisso contraditório entre a eficiência da representação e a sua capacidade de permitir-nos raciocinar sobre o sistema. Nós precisamos da capacidade de manipular e inspecionar os componentes da representação, sem afetar o desempenho da execução. Esta visão de atualização offline é utilizada também em uma implementação de Smalltalk chamada de StrongTalk [BBG+02]. Em StrongTalk, as modificações reflexivas não modificam entidades da execução diretamente, mas criam cópias frescas nas quais são executadas as devidas modificações, que depois serão instaladas atomicamente em lugar da entidade original. O fato das modificações não serem aplicadas diretamente sobre a entidade objeto evita a necessidade de atualizar todas as dependências relacionadas com cada modificação executada e permite atua-

lizações transacionais e em batch. Isto facilita a implementação ao garantir consistência e é adequado para os alvos do nosso trabalho.

Para poder manipular a representação interna das computações como dados na linguagem, precisamos de uma estrutura de dados que facilite a composição e navegação. As tabelas Lua têm as vantagens de que, como mencionado, podem ser indexadas com qualquer tipo de dado, facilitando a construção da representação, e podem ser percorridas. Os valores reificados em LuaNua foram representados como tabelas Lua. Isto permite manipular facilmente os valores e construir as representações progressivamente sob controle do programador. As tabelas então, podem ser devidamente serializadas usando mecanismos da linguagem.

Para satisfazer o requisito de controle (veja a subseção 3.2.1), a representação devolvida por *content* contém somente valores atômicos (numbers, booleans, strings), e referências a abstrações estruturadas (tables, functions, upvalues, prototipos, threads, userdata). Isto permite controlar a extensão do fecho transitivo.

A operação de reificação de co-rotinas consiste em reificar todos os frames, um por um, já que dessa forma pode controlar-se o procedimento (o programador poderia não estar interessado na co-rotina toda). Dessa forma, podem ser suportadas diferentes granularidades: altas granularidades podem ser construídas a partir de granularidades finas por composição. Na instalação, pode ser instalado tanto um único registro de ativação como uma co-rotina inteira, através da composição de registros de ativação. A reificação e instalação são operações simétricas. As informações (por exemplo, o *status*) que somente fazem sentido para a co-rotina inteira devem ser extraídas e incorporadas pelo usuário de forma independente, usando as funcionalidades oferecidas com esse objetivo.

A instalação precisa ter a capacidade de modificar execuções correntes, e não somente criar novas computações. Isto é necessário nos casos em que somente algumas co-rotinas da aplicação são modificadas, pois a criação de uma nova co-rotina gera uma nova referência que não irá coincidir com o valor referenciado pelas outras entidades do programa. Por este motivo, o install deve receber como parâmetro a co-rotina sobre a qual será feita a instalação, e deverá ter a capacidade de modificar ou acrescentar o conteúdo de uma pilha qualquer.

A reificação de uma co-rotina pode ser iniciada a partir de uma outra co-rotina, estando a co-rotina suspensa, ou no contexto da própria co-rotina. Quando executada apartir de outra co-rotina, a representação conterá exatamente o conteúdo da pilha depois da suspensão. Quando executada na própria

co-rotina, a pilha irá conter também a execução da reificação que, dependendo da implementação, pode conter vários registros e funções C que não podem ser capturadas. Implementações baseadas em bibliotecas de facilitadores deveriam estar cientes desse fato e evitar reificar esses registros.

A reificação acontece através da chamada explícita à função de reificação. Isto implica que se precisa de um trecho de código chamando a operação de reificação: este pode ser inserido diretamente no código do programa, ou então através do mecanismo de *hooks* de Lua, que permite registrar uma função que será chamada em situações especiais durante a execução do programa. A inserção de pontos de captura no código é um problema estudado pela área de Aspectos é está fora do escopo desse trabalho.

A transferência de controle (reinício da execução) depois da instalação pode ser executada através das primitivas resume/yield. Isto é um ponto importante na instalação e um diferencial na implementação de reificação/instalação em Lua. A restauração do ponto de execução durante o processo de instalação usando LuaNua é baseada nos mecanismos de suspensão/reativação de co-rotinas. Para conseguir reativar uma co-rotina, é necessário que ela esteja suspensa, ou seja, que tenha chamado a função yield, ou dito de outra forma, que tenha uma chamada a yield no topo da pilha. Esta situação pode ser criada artificialmente em representações de co-rotinas não suspensas instalando, através da própria API proposta, um registro de ativação correspondente a uma chamada yield nessa posição.

Durante a implementação de LuaNua foi acrescentado um grupo de funções auxiliares. Essas não pertencem à API genérica apresentada no capítulo anterior porque sua necessidade vem das características da linguagem Lua. Essas funções são:

newthread devolve uma nova co-rotina vazia;

setstatus muda o status de uma thread.

getopenupvals devolve uma tabela com os upvalues abertos referenciados pela co-rotina

openupvals abre os upvalues contidos na lista;

gettrail devolve uma lista que contém o caminho das chamadas desde a mainthread até a co-rotina requisitada

A seguir se mostram exemplos de como LuaNua permite reificar e instalar diferentes tipos de dados e a flexibilidade que este método oferece. O código apresentado é código Lua real. Os comentários em Lua começam

com "--"; será usado o símbolo --> para indicar saída. A segunda etapa da serialização (ou seja, conversão da representação em transmissível ou *streaming*) pode ser resolvida usando mecanismos da linguagem e por isso não será tratada nesse texto. Apesar dos exemplos seguintes não tratarem explícitamente a parte de migração, todos foram executados salvando o estado em um arquivo e restaurando a execução a partir desse arquivo em outra instância do interpretador, o que mostra a viabilidade dessa abordagem para a implementação tanto de migração quanto de persistência.

4.2 Explorando a API

O primeiro exemplo mostra como reificar e instalar uma função usando a API proposta. A função *inc* simplesmente recebe um parâmetro e retorna o seu valor incrementado:

```
local function inc(counter)
  return counter + 1
end
```

Através da API proposta, esta função pode ser reificada como segue:

```
-- reificando a função inc na tabela tinc
local tinc = debug.content(inc)
print(tinc) --> {p = 0x532920}
```

debug.content retornou o protótipo da função, cuja referência é salva no campo p da tabela tinc (usualmente a reificação de uma função deveria devolver outros campos, mas este exemplo é simples e esses campos são nulos). Em seguida, o protótipo da função deve ser reificado com uma nova chamada a debug.content:

```
local proto = debug.content(tinc.p)
```

Agora a tabela *proto* contém a representação do protótipo que inclui o bytecode da função. Já que todos os valores que compõem a representação são atômicos, a reificação termina aqui.

Depois dos dados serem persistidos ou transmitidos, podem ser instalados. Primeiramente se instalam os valores internos não atômicos no espaço de memória. Neste caso, o único valor desse tipo é o protótipo da função (tipo "proto"):

```
local tinc = {p = debug.install(proto,'proto')}
local newinc = debug.install(tinc,'function')
```

– Agora a função instalada esá pronta para ser executada:

```
print(newinc(1)) -->2
```

Pode-se ver pelo exemplo que o procedimento de reificação/instalação seletiva oferece ao programador um controle fino sobre a composição da representação. Por outro lado, também se pode ver que a reificação faz com que valores que usualmente estão ocultos na implementação de Lua (como é o caso dos protótipos, de tipo "proto") sejam visíveis agora na linguagem como qualquer tipo oficial de Lua. Estes novos valores devem ser tratados de forma similar aos oficiais – por exemplo, o print teria que imprimir o tipo e referência do valor – e operações não permitidas sobre estes valores deveriam lançar erros.

4.2.1 Reificando/instalando execuções

Um exemplo mais interessante é a captura e restauração de computações em execução. Lua provê co-rotinas assimétricas, que são controladas através de chamadas ao módulo *coroutine*. Uma co-rotina é definida através da invocação de *create* com uma função inicial como parâmetro. A co-rotina criada pode ser (re)iniciada invocando *resume*, e executa até que *yield* seja invocado. Por exemplo, a função *count* é um iterador que, para cada número de 1 a 5, imprime o número e faz um yield:

```
-- definição da função
local function count()
  for i = 1,5 do
    print("Numero",i)
    -- manda esse numero de volta ao ativador
    coroutine.yield(i)
  end
end
```

Para se criar uma co-rotina que execute essa função, se invoca *create* com *count* como parâmetro:

```
local coro = coroutine.create(count)
```

Suponha que se deseja executar *count* até que produza o número 3, e depois capturar a co-rotina suspensa. Nesse caso pode-se escrever o seguinte código:

```
local i
repeat
  -- resume retorna o status e
```

```
  -- os valores devolvidos por yield
  _,i = coroutine.resume(coro)
until (i == 3)
capture(coro)
```

Voltamos a atenção agora para a implementação de *capture*. *Capture* contém uma chamada à função (*save*) mais as operações necessárias para serializar a informação reificada. *Save* é uma função auxiliar que construímos, que inspeciona o tipo do argumento e o reifica recursivamente, se não for atômico. Internamente, uma co-rotina Lua executa em uma pilha organizada em registros de ativação, cada um correspondendo a uma função ativa. É possível reificar uma co-rotina Lua compondo seus registros de ativação reificados.

Pode-se iterar sobre os frames que compõem a pilha, chamando a primitiva *content* para cada nível e reificando cada representação estruturada (ou seja, não atômica). Este procedimento cria uma tabela com a representação das entidades solicitadas e seus componentes, que depois pode ser convertida em strings de bytes.

A reificação de co-rotinas em *save* começa criando uma nova tabela:

```
local thr = {}
```

Então salvamos os atributos da co-rotina, como seu status:

```
thr.status = coroutine.status(coro)
```

Agora iteramos sobre os níveis válidos da pilha:

```
repeat
  level = level + 1
  thr[level] = debug.content(coro, level)
```

Já que a representação da co-rotina contém dados não atômicos, seu conteúdo deve também ser reificado. Para isso, invocamos *save* passando como parâmetros o valor a ser reificado e a tabela dos valores já reificados (a tabela *saved*), de forma que os valores são reificados somente uma vez, ainda que apareçam várias vezes (pois podem ser referenciados por vários valores). O *level* é incrementado para mover-se para o registro de ativação seguinte.

```
repeat
  level = level + 1
  thr[level] = debug.content(coro, level)
  thr[level] = save(thr[level],saved)
until (thr[level]==nil)
```

Para instalar uma co-rotina equivalente, devemos carregar a representação salva em *thr* e reconstruir a estrutura da co-rotina. Para isto precisamos criar uma nova co-rotina, executar o install em cada nível e mudar o status da co-rotina para suspensa depois de terminada a instalação (assumindo que todos os valores aninhados já foram reificados):

```
local ncoro = debug.newthread()

for i = #thr, 0, -1 do
  ncoro = debug.install(thr[i], ncoro, 0)
end

debug.setstatus(ncoro,thr.status)
print(coroutine.status(ncoro)) --> suspended
```

Agora a co-rotina pode ser reiniciada. A saída esperada é o próximo número na iteração:

```
coroutine.resume(ncoro) --> Numero 4
```

4.2.2 Controlando a extensão do grafo

Para melhorar o desempenho e a restaurabilidade, o programador pode omitir dados não essenciais antes da transmissão ou persistência. Este é o caso, por exemplo, de variáveis cujo conteúdo não será mais usado.

Consideremos o exemplo de um trecho de código em que uma função recebe um valor de entrada qualquer, o processa e imprime o resultado. Depois retorna o controle ao chamador.

```
local function myprint()
  local stop, data = false

  while not stop do
    data = input_data() -- recebe dados
    print(process_data(data)) -- processa e imprime
    stop = coroutine.yield() -- retorna o controle
  end
end
```

Se a computação for capturada depois que a função *myprint* invocou yield, todos os dados, incluindo a variável local *data*, estarão tomando parte do seu estado e serão serializados (note que *data* poderia conter um dado de qualquer tipo). Entretanto, o valor contido em *data* já foi processado

e será descartado, de forma que o valor corrente pode ser substituído, por exemplo, com um valor nulo. Esta possibilidade não é interessante somente para minimizar a quantidade de informação a ser migrada/persistida, mas também para eliminar dados não serializáveis que a variável poderia conter, como userdata, por exemplo. Com este objetivo, aplicamos a função *setlocal* da API de depuração Lua sobre a co-rotina corrente. Esta função permite setar o valor das variáveis em uma determinada pilha, nível e índice dentro do nível. Outro método possível é modificar diretamente o valor na pilha capturada, mas o método setlocal é preferível por ser de mais alto nível e consequentemente, mais seguro:

```
debug.setlocal (coro, 1, 2, nil)
```

Agora já podemos capturar a co-rotina:

```
local t = save(coro,saved)
local ncoro = rebuild(t)
print(debug.getlocal (ncoro, 1, 2)) --> data    nil
```

As funções *save* e *rebuild* foram implementadas como parte da biblioteca *seriallib* que é a biblioteca de mais alta ordem que temos construído para facilitar os procedimentos de reificação e instalação.

4.2.3 Tratamento de funções C

Lua permite a fácil integração com código C em programas Lua. Embora isto ofereça muitas vantagens, por outro lado complica a captura: a captura do estado de execução em programas com chamadas C envolve a captura da pilha de execução C. Entretanto, não existe um método compatível com Ansi-C que permita restaurar a parte do programa escrita em C. É possível suspender co-rotinas contendo código C, desde que a suspensão não aconteça entre fronteiras de chamadas C. O tratamento neste trabalho é semelhante, ou seja, a reificação de programas incluindo código C pode ser feita desde que a pilha não contenha chamadas a C, pois elas não poderiam ser capturadas. Na realidade, referências a funções C – assim como referências a outros dados não portáveis como userdata – podem ser substituídas na reificação por descritores de serialização e revinculadas no destino usando esses descritores. Neste texto utilizamos esta abordagem através da tabela *saved*, que deve conter, por exemplo, uma referência à função print, que temos utilizado amplamente nos exemplos. A dificuldade envolvida nesse método está na necessidade de identificar as funções e o fato delas serem anônimas em Lua. Pode ser necessário construir a tabela das funções ativas no ambiente com seus respectivos nomes

para conseguir fazer o mapeamento reverso. O caracter usualmente cíclico do ambiente pode complicar esta tarefa. Por outro lado, nem todas as funções poderão ser encontradas nessa tabela. Por exemplo, a função *pairs* devolve um iterador (uma função C), uma tabela e um nil. O iterador devolvido por pairs não pode ser serializado e é uma função anônima, somente foi identificado através de testes sobre a função reificada e após o estudo da documentação da função pairs. A solução utilizada nesse trabalho foi dar um nome á função (pairsx=pairs) e incorporá-la na tabela de funções que não seriam serializadas.

A solução baseada em substituição tem limitações, como as situações em que valores não locais estão sendo referenciados pelo valor a ser substituído, e funções C que estão manipulando dados ativos no lado C. Estes casos não podem ser devidamente restaurados sem um tratamento adicional.

4.2.4 Compartilhamento e minimização

Em Lua 5.1 é possível atribuir valores às variáveis não locais dos closures (upvalues). Entretanto, não é possível restaurar o vínculo dessas variáveis com a pilha e entre os closures que usam estas variáveis. Como exemplo, consideremos o seguinte código que devolve duas funções (*inc* e *dec*) que incrementam e decrementam um contador, respectivamente, e retornam seu valor:

```
local function count()
 local counter = 1
  local function inc()
    counter = counter + 1
    return counter
  end
  local function dec()
    counter = counter - 1
    return counter
  end
  return inc, dec
end

local inc, dec = count()
print(dec()) --> 0
print(inc()) --> 1
print(dec()) --> 0
```

Após a execução desse código, obtemos duas funções inc e dec que compartilham o valor de counter. Podemos persistir os closures que compartilham o

upvalue counter usando a biblioteca de serialização de Lua através das funções *string.dump* e *loadstring*:

```
local copyinc = loadstring(string.dump(inc))
local copydec = loadstring(string.dump(dec))
debug.setupvalue(copyinc,1,select(2, debug.getupvalue(inc,1)))
debug.setupvalue(copydec,1,select(2, debug.getupvalue(dec,1)))

print(copydec()) --> -1
print(copyinc()) --> 1
print(copydec()) --> -2
```

Entretanto, verificamos que apesar do valor do upvalue poder ser restaurado corretamente, o programa está se comportando como se existissem duas variáveis isoladas. Este problema da versão atual de Lua pode ser resolvido através dos mecanismos de reificação e instalação propostos. A reificação de upvalues permite manter o compartilhamento através da inserção do upvalue na estrutura dos closures correspondentes, pois os valores são reificados como valores de primeira classe, compondo a representação dos closures respectivos. As funções do exemplo poderiam ser reificadas da seguinte forma:

```
-- reificando a função dec na tabela table tdec
local tdec = debug.content(dec)
print(type(tdec))
--> table
-- imprimindo os conteudos da tabela (key and value)
table.foreach(tdec,print)
--> p       proto: 0x532920
--> upvals  table: 0x533cd0

-- guardando o numero de upvalues. Os upvalues podem ter valor nulo
tdec.nups = debug.getinfo(dec,"u").nups

-- reificando os upvalues na tabela tdec.upvals
for key, value in pairs(tdec.upvals) do
  print(type(value))
  tdec.upvals[key]=debug.content(value)
  print(key,tdec.upvals[key]) -- valores dos upvalues
end
--> upval
--> 1       1
print(type(tdec.p))
--> proto
```

```
-- reificando o prototipo da função em tdec.p
tdec.p=debug.content(tdec.p)
table.foreach(tdec.p,print)
--> upvalues        table: 0x55a1c0
--> nups    1
--> maxstacksize    2
--> numparams       0
--> k       table: 0x559ec0
--> code    table: 0x559c80
--> is_vararg       0
...
-- reificando a função inc em tinc
local tinc = debug.content(inc)
tinc.p = debug.content(tinc.p)
```

O upvalue da função inc não precisa ser reificado pois já foi reificado como upvalue de dec. Já que, nesse caso, todos os valores retornados são atômicos, a reificação termina aqui. Na representação de um closure Lua existem os campos *env* e *isC*. Eles não são devolvidos pela função content pois já podem ser extraídos usando a API Lua, mas precisam ser incluídos na representação antes da instalação.

Depois dos dados serem persistidos ou transmitidos, estão prontos para serem instalados:

```
-- Instalando tdec na tabela fdec--
local fdec={};fdec.upvals={}
local uv
for key=1, tdec.nups do
  -- instalando os upvalues
  fdec.upvals[key]=debug.install(tdec.upvals[key],'upval')
end
fdec.p = debug.install(tdec.p,'proto')

--a instalação de tinc em finc é idéntica exceto pelo upvalue:
finc.upvals[1]=fdec.upvals[1]

newinc = debug.install(finc,'function')
newdec = debug.install(fdec,'function')
-- setando o ambiente
setfenv(newinc, tinc.env)
setfenv(newdec, tdec.env)

-- Testando o compartilhamento
```

```
print(newdec()) --> -1
print(newinc()) --> 0
print(newdec()) --> -1
```

Desse exemplo podemos concluir que a reificação permite:

- Minimizar a informação a ser transferida. Por exemplo, já que inc e dec compartilham um upvalue, a reificação do upvalue somente precisa ser feita uma vez;
- manter o compartilhamento. Isto se deve à capacidade de manipulação e composição que o procedimento oferece.

4.2.5 Migração fraca

A pesar da grande vantagem desse método residir na capacidade de reificar o estado, ele também permite a captura de código sem estado que caracteriza a migração fraca. Esta funcionalidade já está disponível no Lua 5.1 através das funções antes mencionadas *string.dump* e *load*. A vantagem dessa proposta consiste em que, nesse caso, a representação é portável e pode ser manipulada antes da instalação do protótipo no destino.

4.2.6 Continuações parciais

Em algumas situações, pode ser conveniente realizar a migração parcial de uma thread. Pode ser interessante, por exemplo, migrar uma thread para perto dos dados que ela está acessando (quando tanto o custo de migrar os dados quanto o tempo de acesso remoto forem elevados), e, para diminuir o custo dessa migração, pode se desejar transferir apenas os registros de ativação envolvidos no acesso a esses dados. A seguir discutimos um mecanismo para implementar essa migração parcial.

O mecanismo chamado de migração de computações (computation migration) proposto no trabalho de Hieb et al.[HWW93] foi definido como a migração parcial de uma thread ativa para perto dos dados que ela acessa. O desempenho da migração de computações é melhor que o da migração de dados (em que os dados são movidos até o computador que irá processá-los) "quando as escritas são caras ou dominam as leituras". Dessa forma, precisam-se mecanismos que permitam escolher uma ou outra alternativa dependendo do perfil da aplicação. A migração abrange um grupo de registros de ativação, enquanto o restante da computação é ativado quando retorna o resultado. Repare que

levar um registro de ativação é diferente de levar o código correspondente pelo fato dele carregar o estado de execução corrente e possivelmente alterá-lo.

No seguinte exemplo implementaremos um mecanismo semelhante. Neste exemplo capturaremos parte de uma execução, que consiste em uma sucessão de chamadas, que se suspende para executar a captura, gerada pelo código a seguir:

```
local function f2(a)
  local b = a + 1
  capture()
  return b
end

local function f1(a)
  local b = f2(a) + 1
  print(b)
  return b
end

local coro = coroutine.create(function(p)
  local a = f1(p)
  print("resultado",a)
end)

coroutine.resume(coro,5)
```

A idéia consiste em migrar a última chamada à função de forma a ser executada remotamente, e trazer o resultado de volta, passando-o então como parâmetro à execução restante. No final é obtido o mesmo resultado que no caso da execução sem quebras. O procedimento começa com a reificação da co-rotina *coro*. A tabela *reified_ coro* contém a descrição da representação da co-rotina com todos os seus valores internos reificados (exceto as funções C).

```
local reified_coro = save(coro, saved)
```

Agora construímos duas representações dividindo o conteúdo da tabela de forma a deixar o yield e a última função Lua chamada (f2) em uma tabela e o restante em outra. As co-rotinas que serão construídas precisarão estar em estado suspenso para poder reiniciar a execução, de forma que inserimos um yield no topo da segunda tabela. (O estado suspenso da thread não aparece aqui porque é atribuído pela função rebuild da biblioteca).

```
local reified_coro1 = {
  type = reified_coro.type,
```

```
   ci={
   [0]=reified_coro.ci[0],
   [1]=reified_coro.ci[1]}
 }
 local reified_coro2 = {
   type=reified_coro.type,
   ci={[0]=yield,
   [1]=reified_coro.ci[2],
   [2]=reified_coro.ci[3]}
 }
```

onde *yield* é uma tabela que descreve a estrutura do registro de ativação da chamada a yield.

```
local yield = {
  func = coroutine.yield,
  savedpc = -1,
  nvargs = 0,
  high = 0,
  nresults = -1,
  size = 21,
  nregs = 0
}
```

Agora com a estrutura necessária para criar as duas novas co-rotinas, podemos instalar e executar remotamente a co-rotina superior. Repare que caso esta co-rotina seja executada remotamente, deveria ser transportado somente o registro 1 e acrescentado o yield no destino (é uma informação redundante e contém uma função C – a função yield – que não é portável) seguindo o procedimento anterior.

```
 local installed_coro1 = rebuild(reified_coro1, saved)
 local _, result = coroutine.resume(installed_coro1)
 print("Resultado parcial",result) --> Resultado parcial 5
```

Depois de obtido o resultado (remoto ou não) da co-rotina contendo os registros superiores, é passado como parâmetro do resume da co-rotina contendo o restante da computação. O resultado final é equivalente ao da computação quando executada sem quebras:

```
 local installed_coro2 = rebuild(reified_coro2, saved)
 print(coroutine.resume(installed_coro2, result)) --> true    7
 print(coroutine.resume(coro)) --> true    7
```

Um detalhe sutil nesse procedimento está na conservação do estado da co-rotina. No caso em que o estado da computação é modificado durante a execução parcial remota (seja o estado global ou as variáveis não locais, ou upvalues), uma implementação correta implicaria na migração da sub-computação em lugar do resultado. Ou seja, a co-rotina deveria ser migrada de volta antes de ela terminar de executar, e não somente o resultado, de forma que estas modificações possam ser incorporadas à co-rotina origem. Este procedimento não será ilustrado aqui por consistir basicamente na migração de uma co-rotina mais o binding das variáveis modificadas.

4.2.7 Continuações

As primitivas da API permitem capturar execuções parciais na forma de co-rotinas. Entretanto, para capturar a execução, pode ser necessário capturar a continuação toda, que pode estar formada pela composição de várias co-rotinas ativadas em uma determinada ordem.

Continuações podem ser implementadas em Lua usando co-rotinas assimétricas [MI09]. A reificação e aplicação de co-rotinas assimétricas permite a sua instalação em diferentes instâncias do interpretador, hosts ou instantes de tempo. No entanto, a assimetria implica no retorno obrigatório à co-rotina chamadora. O retorno consiste no retorno da chamada a resume (uma função C) quando a co-rotina é suspendida ou termina a execução. Mas a pilha C não poder ser restaurada de uma forma portável.

Entretanto, a pilha de chamadas pode ser facilmente restaurada quando as co-rotinas retornam a uma mesma co-rotina e não estão relacionadas entre si (são execuções independentes). Esta configuração é tipica de escalonadores como o mostrado a seguir, tomado de um exemplo de uso da biblioteca LOOP [Maia08]. O escalonamento no exemplo é efetuado na função *run*, que percorre a lista *self* que contém as threads ativas, as executa e retira as threads mortas da lista. Depois de percorrer a lista toda, *run* retorna. O *run* é executado dentro de um loop (parte de uma função da biblioteca LOOP que não é mostrada aqui) até não ter mais trabalho para fazer.

```
Scheduler = oo.class({}, UnorderedArray)

function Scheduler:run()
  while #self > 0 do
    local i = 1
    repeat
      local thread = self[i]
      coroutine.resume(thread)
```

```
      if coroutine.status(thread) == "dead" then
        self:remove(i)
      end
      i = i + 1
    until i > #self
  end
end

function thread(name)
     return coroutine.create(function()
        for step=1, 3 do
          print(string.format("%s: step %d of 3", name, step))
          coroutine.yield()
        end
     end)
end
```

Este programa cria três threads (A, B e C). Cada thread executa um loop que imprime uma mensagem e suspende a execução.

```
scheduler = Scheduler{ thread("A"), thread("B"), thread("C") }
local threads = scheduler:run()
-->A: step 1 of 3
-->B: step 1 of 3
-->C: step 1 of 3
-->A: step 2 of 3
-->B: step 2 of 3
-->C: step 2 of 3
-->A: step 3 of 3
-->B: step 3 of 3
-->C: step 3 of 3
```

Dois exemplos de políticas que podem ser implementadas nesta configuração são: a captura não antecipada do estado da aplicação (por exemplo, para realizar a manutenção do servidor) e a captura prevista do estado das co-rotinas que estaria inserida no próprio código do escalonador, como na implementação de um escalonador distribuído que cria co-rotinas a serem executadas em outros nós.

Uma maneira simples de restaurar a execução deste programa consiste em capturar as threads A, B e C e reiniciar a execução recriando o escalonador no destino e registrando nele as threads transferidas, que seriam recriadas da seguinte forma:

```
-- em threads está a lista de threads reconstruídas:
  scheduler = Scheduler(threads)
  scheduler:run()
```

Desvantagens do método são o fato de precisar do código do escalonador junto com todas as bibliotecas relacionadas no destino e de conhecimento sobre seu funcionamento. Para o usuário que deseja implementar a migração/persistência, não se trata simplesmente de colocar um *capture* que captura a aplicação toda, mas de capturar cada uma das threads – que ele teria que identificar – e registrá-las em um novo escalonador no destino. Isto implica conhecer o processo de registro e saber todo o que o escalonador precisa para executar. As vantagens, no entanto, são importantes, pois este método é muito simples e permite minimizar a informação transferida, aproveitando a possível presença das bibliotecas relacionadas no destino. Escalonadores poderiam ser escritos que embutissem a funcionalidade de captura de threads independentes, para migração (poderiam ser executadas remotamente) ou persistência.

Outras estruturas de chamadas acrescentam problemas adicionais. Por exemplo, na figura 4.1 *co*1 transfere o controle para *co*2, que por sua vez transfere o controle para *co*3, em que a execução é capturada (na figura as setas contínuas indicam operações de *resume* e as ponteadas, de *yield*).

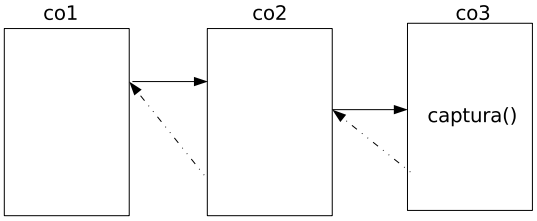

Figura 4.1: Fila de chamadas

Neste caso, depois da restauração, uma tentativa de reiniciar co3 chamando a resume com co3 como parâmetro provocaria o retorno do controle à co-rotina chamante (a co-rotina que efetua a restauração) em lugar da co-rotina co2. Precisamos neste caso de métodos que permitam restaurar corretamente a fila de chamadas, contornando a ausência da chamada original a resume durante a restauração. Neste texto, explicamos dois métodos que podem ser utilizados para esse fim. O primeiro consiste em inserir um escalonador para dirigir o reinício do programa (que transforma a execução em uma semelhante à mostrada anteriormente), enquanto o segundo força a criação da fila seguindo uma disciplina CPS, através da inserção de novos frames no topo das pilhas das co-rotinas envolvidas. Estes métodos são mostrados a seguir.

Inserção de um dispatcher

Para simular o efeito do yield/resume das co-rotinas assimétricas usaremos um método similar ao utilizado em [Moura04] para a implementação de co-rotinas simétricas com assimétricas. Ao contrário das co-rotinas assimétricas, a troca de contexto nas co-rotinas simétricas (através de uma operação *transfer*) é independente da lista de chamadas até a co-rotina atual, que não precisaria ser reconstruída após a restauração. A idéia justamente é criar o efeito de um transfer entre as co-rotinas contidas na fila de chamadas. Dessa forma, somente é necessário saber qual a próxima co-rotina a ser reiniciada.

O método consiste na utilização de um *dispatcher* que age como intermediário nas transferências de controle entre duas co-rotinas. Em lugar de seguir

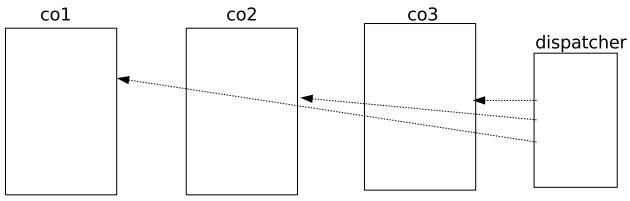

Figura 4.2: Restauração da fila de chamadas usando dispatcher

exatamente a ordem de transferência da fila de chamadas original em que co3 retornaria à co2 e esta retornaria à co1, aqui co3 retorna o controle ao dispatcher, que em seguida ativa co2 e quando esta retorna, co3. O resultado obtido é o mesmo em ambos os casos. O dispatcher também deve receber o valor de retorno das co-rotinas e repassá-lo para a co-rotina seguinte, como mostrado na figura 4.2.

```
local resultado = coroutine.resume(co3)
resultado = coroutine.resume(co2, resultado)
coroutine.resume(co1,resultado)
```

```
-> Coroutine co3
-> Coroutine co2
-> Coroutine co1
```

Restauração da fila de chamadas usando Continuation Passing Style

A idéia neste caso consiste em gerar uma fila de chamadas como mostra a figura 4.3. O usuário reinicia a co-rotina co1 que por sua vez reinicia a co-rotina co2 que continua a execução. A fila seria reconstruida usando uma chamada como esta:

```
coroutine.resume(co1,
[[coroutine.resume(co2,"coroutine.resume(co3)")]])
```

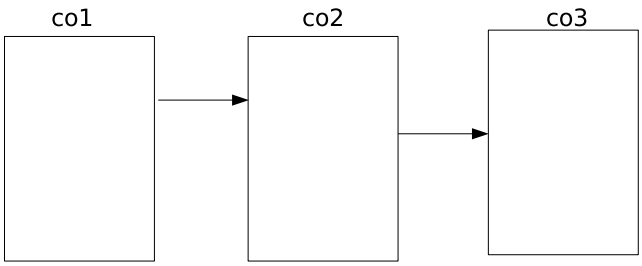

Figura 4.3: Restauração da fila de chamadas usando CPS

Desta forma, a fila de ativação é corretamente reconstruída e não se precisa de instâncias intermediárias nem intervenções posteriores para as co-rotinas retornarem corretamente, isto é, seguindo a fila inicial. Para isto, precisamos que no topo da pilha haja um yield e uma função f esperando a continuação do processo de restauração da fila de chamadas. A função f reiniciaria a co-rotina passada como parâmetro, que seria a seguinte da fila, passando por sua vez como parâmetro do resume a próxima continuação da restauração da fila. Podemos construir essa situação de duas formas que se diferenciam pela forma em que este código se insere na aplicação: uma é através do programa, e a outra é por imposição desde o metanível.

Na primera opção, o código necessário para executar estas operações seria inserido na aplicação diretamente ou através de uma função contendo todas as operações necessárias para executar a captura. Este código começaria assim:

```
if coroutine.resume(co)=="migrando" then ...
```

Neste caso, seria necessário um tag especificando que o programa se encontra em estado de restauração. No entanto, isto implica em escurecer o código da aplicação, atenta contra a ortogonalidade da captura e restauração, dificulta a compreensão do código, aumenta a chance de erros, afeta o desempenho da aplicação durante a execução (ainda que não seja migrado) e não permite captura objetiva.

Outra opção precisaria de uma intervenção *à la "Deus ex machina"*. Vamos nos colocar no metanível para inserir no topo das pilhas de cada co-rotina o código necessário para reiniciar a execução de forma que as co-rotinas sejam ativadas na ordem correta. Este código pode ser o correspondente à função execresume mostrada a seguir:

```
local function execresume ()
  local command = coroutine.yield()
  loadstring(command)()
end
```

cuja representação interna no ponto da chamada ao yield é a seguinte:

```
records.execresume = {
func = execresume,
nresults = -1, -- -1: retorna todos; 0: nada; #: # resultados
high =  1,
savedpc= 2,
nvargs = 0,
nregs = 0,
size = 4
}
```

Os índices dos registros da tabela que contém as co-rotinas (thr) se deslocam para inserir um novo registro:

```
thr.co1.ci[2]=thr.co1.ci[1]
thr.co2.ci[2]=thr.co2.ci[1]
```

A representação da função é inserida como um frame na representação da pilha:

```
thr.co1.ci[1]=records.execresume
thr.co2.ci[1]=records.execresume
```

Cada pilha tem que ser suspensa, ou seja, depois de reconstruída, ter seu status mudado para "suspended" e um frame yield acrescentado no topo das pilhas. Depois disso elas podem ser instaladas:

```
  coros = rebuild(thr)
```

Finalmente, a fila de chamadas é reiniciada:

```
coroutine.resume(coros.co1,
[[coroutine.resume(coros.co2,"coroutine.resume(coros.co3)")]])
-->Coroutine co3
-->Coroutine co2
-->Coroutine co1
```

Apesar de este método estar bem na linha da separação entre o nível objeto e o meta-nível, evitando a poluição de código inerente ao método anterior (poluição esta que afeta o desempenho da aplicação), também tem desvantagens. A quantidade justa de causalidade imposta é um ingrediente difícil de gerenciar para o programador: por um lado, é um método fácil e poderoso de obter o objetivo desejado, pelo outro, aparece de repente no meio da execução, sem controle de erros e quebrando as invariantes da linguagem. Seu uso deve estar limitado, então, às situações em que ele seja estritamente necessário, como no caso que acabamos de descrever.

4.2.8 Implementando continuações multi-shot como construção da linguagem

As continuações multi-shot, diferentemente das continuações one shot, implicam em uma operação inerente de cópia do estado [BWD96], motivo pelo qual não podem ser implementadas diretamente através dos mecanismos de co-rotinas de Lua. A API proposta permite manipular ambos os tipos de continuações como estruturas da linguagem (o desempenho, no entanto, não seria um ponto alto desta alternativa), através da reificação/instalação da computação e todas as dependências.

4.2.9 Aplicação em outros frameworks

A solução proposta permite a extensão de frameworks já implementados usando Lua que seriam beneficiados com a capacidade de transmitir estado serializado. Este é o caso da proposta de Skyrme et al. [SRI08] que consiste em uma biblioteca Lua para a implementação de multithreading cooperativo em ambientes de memória compartilhada. O modelo implementado se baseia na execução independente de fluxos de código Lua cuja execução é coordenada por um escalonador. A comunicação entre os processos Lua é realizada exclusivamente através da troca de mensagens contendo valores atômicos. Isto implica na impossibilidade de transmitir execuções entre estados Lua. Esta limitação, entretanto, pode ser resolvida através dos mecanismos que a API proposta nessa tese oferece. De fato, utilizando a API temos transmitido mensagens contendo vários tipos de dados, entre eles co-rotinas.

4.2.10 Liberdade demais? Detecção de acessos a variáveis globais

A grande vantagem da API proposta é o grande nível de controle que oferece sobre a execução. No entanto, isto também permite facilmente alterar a computação em formas não previstas. Alguns programadores argumentam a favor deste tipo de facilidades (de fato, atualmente alguns programadores Lua modificam o bytecode gerado pela máquina virtual) que dessa forma é possível gerar código mais eficiente. Esta prática não é pouco habitual em linguagens como Java. De fato existem aplicações para facilitar a edição do bytecode de forma a ter acesso ao conjunto estendido de operações da máquina (por exemplo, ao GOTO).

Entretanto, é possível também contrariar desnecessariamente a programação bem comportada. Como exemplo, mostramos um método que permite detectar acessos a variáveis globais dentro do código reificado. Isto é necessário

caso o programador desejar proteger o ambiente destino de alterações decorrentes da restauração da computação, mantendo em aparência o ambiente original da função. Os detalhes do procedimento são mostrados a seguir.

A detecção se baseia na busca dos opcodes *GETGLOBAL* e *SETGLOBAL* [Man06, IFC05] no array de instruções (code) do protótipo da função. A figura 4.4 mostra a estrutura de instruções desse tipo em Lua.

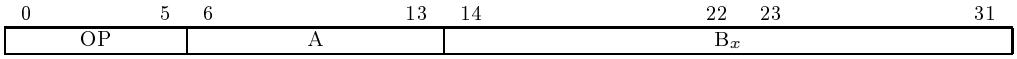

Figura 4.4: Instrução Lua

Uma instrução GETGLOBAL(opcode 5) copia no registro A o valor da variável global cujo nome está na B_x-ésima constante (tabela K). (GETGLOBAL R(A)=G[K(B_x)]). Entretanto, SETGLOBAL (opcode 7) copia na variável global cujo nome está na B_x-ésima constante, o valor do registro A. (SETGLOBAL G[K(B_x)]:=R(A)).

Depois de detectado o acesso a globais, o programador pode optar por várias alternativas, dependendo do resultado esperado. Uma delas é usar a função *setfenv*, que estabelece como ambiente da função a tabela passada como parâmetro. Aqui mostramos para propósitos de ilustração de capacidades, o tratamento que consiste em converter o acesso a variáveis globais em acesso a upvalues através das primitivas de reificação. A transformação consiste em acrescentar a variável global na lista de upvalues, converter o acesso a global em acesso a upvalue GETGLOBAL R(A)=G[K(B_x)] , opcode 5, A=1, B=0 (GETUPVAL, R(A)=U[B], opcode 4, A=1, B=1 $\rightarrow$ 0x800044), e eliminar a string "global" da lista de constantes. Dessa forma obtem-se uma função equivalente à função count, equivalência na realidade definida pela semântica desejada pelo programador. Por exemplo, a função:

```
global = 1
local counter = 1
local function x()
  counter = counter + global
  return counter
end
```

através dessas transformações:

```
--function reification
local t = debug.content(x);
t["p"] = debug.content(t.p)
```

```
-- nups: number of upvalues
t.p.nups = t.p.nups + 1
-- creating a new upvalue containing the value of global
-- (global is the last one)
t.upvals[t.p.nups] = debug.install (global, "upval")
t.p.upvalues[t.p.nups] = "global"
t.p.k[1] = nil --was "global"

for k,v in pairs(t.p.code) do
   -- GETGLOBAL instruction detected
   if v % 32 == 5 then
        -- replace with GETUPVAL instruction (opcode 4)
        t.p.code[k] = (t.p.nups - 1)*2^23 + 1*2^6 + 4
   end
end

t.p = debug.install(t.p,'proto')
local f = debug.install(t,'function')
```

equivalente com:

```
local counter = 1
local global = 1
local function x()
  counter = counter + global
  return counter
end
```

que em determinados contextos é equivalente à função inicial.

Na realidade, não é imprescindível fazer uso da API com este objetivo, pois a linguagem oferece as funcionalidades necessárias e ainda, é além disso, seu uso nesse caso é perigoso, pois modifica diretamente o bytecode, o que pode provocar erros não controlados (Lua assume que o bytecode gerado está certo). Isto leva a avaliar a possibilidade de impedir o acesso direto ao bytecode das funções (através do empacotamento, por exemplo), favorecendo os métodos oferecidos pela linguagem para a modificação do ambiente.

4.3 Biblioteca de facilitadores

Ao longo deste capítulo usamos repetidamente as funções oferecidas por uma biblioteca que implementamos para facilitar o uso da API de reificação.

Esta biblioteca oferece as funções save and rebuild, que executam a captura e restauração profunda de dados Lua exceto userdata, lightuserdata e funções C. Ambas as funções utilizam uma tabela para registrar os valores já processados, de forma a aproveitar os resultados e evitar duplicação. Esta lista pode ser usada também para evitar o processamento de dados específicos, se a inicializarmos, por exemplo, com funções C mapeadas a seus respectivos valores. Isto permite a revinculação usando os valores especificados na tabela, com independência do tipo. Por exemplo, a tabela global (_G) é uma boa candidata a ser revinculada no destino, pela sua disponibilidade e porque contém uma grande quantidade de informação que inclui funções C e userdata.

A função *save* devolve diretamente o argumento se ele for atômico. Senão, e ele já tiver sido reificado, devolve a referência à tabela armazenada. Se for um novo valor, inicia o processo registrando a nova tabela e efetuando o procedimento de reificação correspondente ao tipo do valor. Depois a tabela obtida, é por sua vez, reificada.

```
function save(value, saved)
  local ttype = type(value)
  if ttype=="number" or ttype=="string" or
  ttype=="boolean" or ttype=="nil" then
    return value
  end
  saved = saved or {}
  if saved[value] then
    return saved[value]
  else
    local s = {}
    saved[value] = s
    if ttype=="function" and debug.getinfo(value,"S").what=="C" then
      error("while saving. Function "..debug.name(value)..
      " cannot be serialized (C function)")
    end
    if ttype=="proto" then value = reifyproto(value)
    elseif ttype==...
    ...
    end
    for k,v in pairs(value) do
      k = save(k, saved)
      s[k] = save(v, saved)
    end
    return s
  end
```

```
  end
```

A função rebuild devolve o valor se for atômico. Caso contrário procura o valor já reificado. Se o valor ainda não tiver sido instalado, começa a instalação.

```
function rebuild(t,register)
  register = register or {}
  if type(t)=="table" then
    if t.id and register[t.id] then
      t = register[t.id]
      return t
    end
    if t.type=="upval" then
    ...
  end
  return t
end
```

Um detalhe importante na instalação de upvalues é o fato de que eles podem conter referências a si próprios. Por exemplo, na função:

```
local function f()
  print("printing from nested upval");
  return f
end
```

que pode ser representada da seguinte forma:

```
t={}
t["type"]="function"
t["upvals"]={}
t["upvals"][1]={}
t["upvals"][1]["id"]="0x52b040"
t["upvals"][1]["type"]="upval"
t["upvals"][1]["value"]=t --autoreferencia
t["p"]={}
t["p"]["type"]="proto"
t["p"]["nups"]=1
...
```

Repare que nesse caso o valor do upvalue é a própria função que o contém (t["upvals"][1]["value"]=t). Por esse motivo, precisamos criar uma referência ao upvalue com um valor qualquer, e registrá-la, antes de começar a instalação do valor contido, como mostra o seguinte fragmento extraído da função *rebuild*:

```
    ...
elseif t.type=="upval" then
  local uv = debug.install(0,'upval')
  register[t.id] = uv
  t.value = rebuild(t.value, register)
  t = debug.install(t.value, uv)
```

A implementação destas bibliotecas deve levar em conta o fato de que as informações capturadas podem sobreviver às distribuições do programa em que foram geradas. Ou seja, a restauração pode acontecer em uma versão diferente do programa. É recomendável, portanto, incluir algum tipo de versionamento que seria conferido antes da restauração, devido à estreita ligação entre a implementação da linguagem (que não oferece garantias de compatibilidade entre versões) e a representação das suas entidades. Esta informação deveria ser adicionada à informação capturada, seja através desta biblioteca ou da biblioteca de serialização. A capacidade de manipulação (e minimização) da representação facilita a resolução do problema de versionamento, pois o volume de informação que pode gerar conflito é menor e pode ser tratado antes da restauração.

5
Implementação

Este capítulo descreve a implementação de *LuaNua*, uma versão de Lua 5.1 que implementa a API de suporte para captura e restauração do estado de execução de computações baseado nas primitivas para *reificação* de valores (captura) e para a sua *instalação* (restauração). A seguir descrevemos alguns aspectos internos de Lua que são importantes para entender o restante do capítulo.

5.1
Internals

Um estado Lua representa o estado da máquina virtual, dessa forma para Lua somente pode existir um estado (ou *global_State*). Entretanto, em C é possível criar e manter vários estados simultâneos (máquinas virtuais), todos eles invisíveis entre si. Um estado pode conter várias threads (várias pilhas, representadas por lua_States). Quando se cria um estado (através de uma chamada a lua_newstate), também é criada uma thread dentro desse estado, chamada de *mainthread*. A mainthread não é coletável, sendo liberada junto com o estado. Threads adicionais podem ser criadas através da função lua_newthread, que devolve um ponteiro a um lua_State e coloca a thread na pilha. Na realidade, co-rotinas são threads com a interface Lua. Co-rotinas de um mesmo estado (apontam ao mesmo global_State) compartilham a tabela de globais.

Co-rotinas em Lua são objetos de primeira classe de tipo *thread*. As co-rotinas permitem gerenciar tarefas de forma cooperativa, podendo elas suspender a própria execução e serem reiniciadas posteriormente. Co-rotinas suspendidas estão em estado *suspended*. Co-rotinas que terminam a execução por retorno da função principal ou se acontecer um erro não protegido, estão em estado *dead* e não podem ser reiniciadas. O interpretador Lua, ao executar um resume, invoca recursivamente a função principal do interpretador. A nova invocação usará a pilha da co-rotina para executar as chamadas e retornos da co-rotina resumida. Um *yield* provocará o retorno à invocação prévia do interpretador (a que fez a chamada a *resume*), deixando a pilha da co-rotina

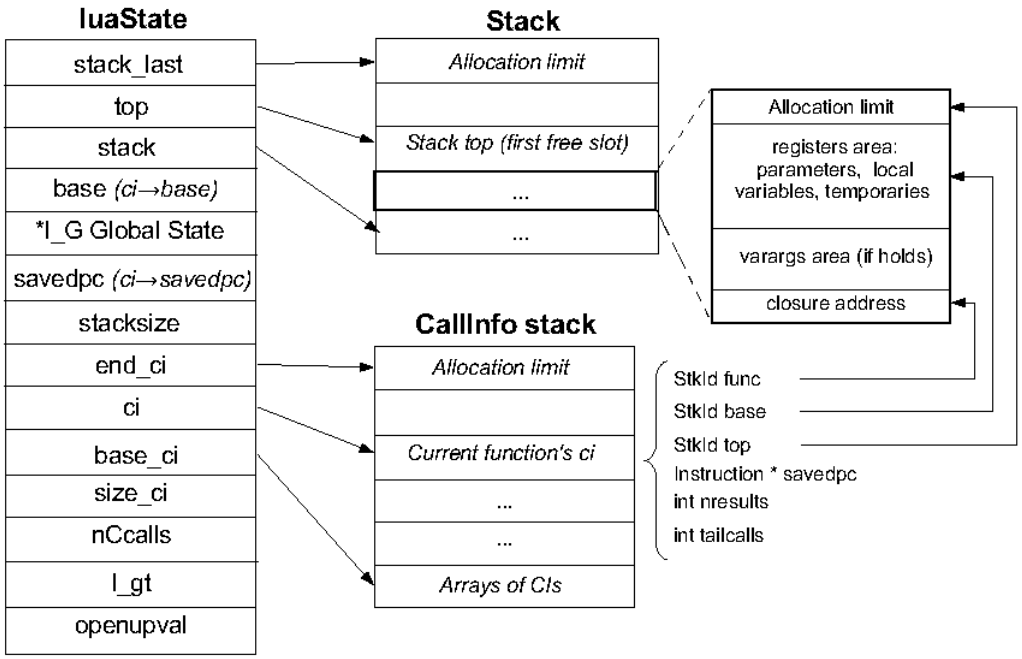

Figura 5.1: Estado Lua

com as chamadas pendentes. Não existe um operador para destruir corotinas explícitamente: como os otros valores da linguagem, elas são eventualmente coletadas quando não estiverem mais sendo referenciadas.

Como mostra a figura 5.1, cada co-rotina (lua_State) tem duas pilhas: a pilha de chamadas (CallInfo Stack) e a pilha de objetos (the Stack). A pilha de objetos tem um slot (ou registro de ativação) para cada função ativa. Cada slot armazena os parâmetros da função (variáveis por causa do vararg), a função em si (apontada por ci→func) e os registros temporais. A outra pilha é um array em que se guardam ponteiros para posições na pilha do início e fim de slot e a posição da função (um frame por função).

Todas as variáveis globais vivem como campos em tabelas Lua que são chamadas de tabelas de ambiente ou simplesmente *ambiente*. Cada função tem o seu próprio ambiente, dessa forma os acessos a globais nessa função irão se referir a esse ambiente. O ambiente inicial de cada função é herdado da função que a criou. Lua permite consultar e modificar o ambiente, através das funções getfenv e setfenv, respectivamente.

Upvalues em Lua são estruturas de dados que concentram as referências de funções internas do aninhamento a variáveis locais das funções mais externas. Os upvalues são usados para implementar closures. Os closures definem o espaço em que vivem as variáveis locais das funções. Variáveis locais externas podem ser acessadas pelas funções mais internas. Isto é válido mesmo quando a variável sai do escopo. Nesse caso, o upvalue, que anteriormente

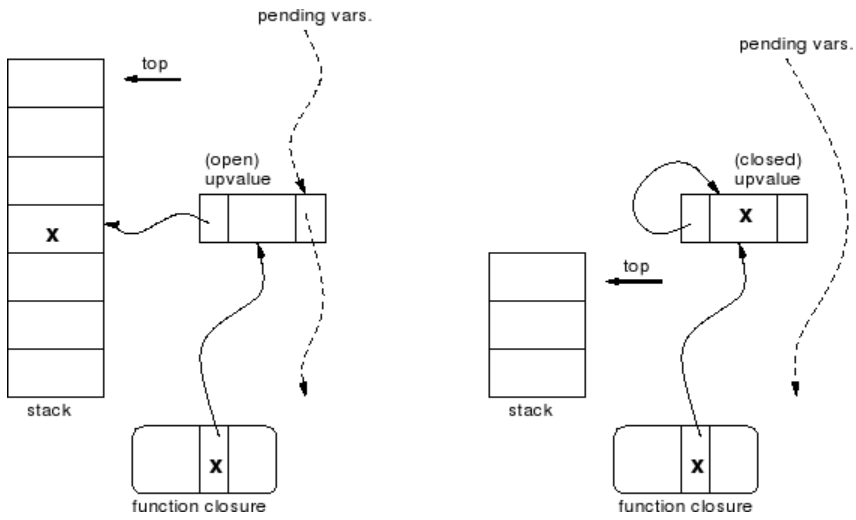

Figura 5.2: Upvalues Lua [IFC05]

referenciava a variável na pilha, agora vai conter uma cópia da variável (chama-se de fechamento do upvalue: a variável é copiada da pilha para o heap), como mostrado na figura 5.2. O upvalue é fechado quando a função que declarou a variável como local retorna, ou quando a thread em que a closure foi criada é coletada. Este processo é transparente para as funções que acessam a variável através do upvalue. Os upvalues começam usualmente abertos, garantindo o compartilhamento com as variáveis locais na pilha.

As facilidades oferecidas pelas metatabelas Lua são um método interessante para a implementação de bibliotecas de mais alto nível. Este método é utilizado em Pluto na forma de um evento `__persist` que permite definir a forma que um determinado valor seria persistido.

5.2 Implementação de LuaNua

A implementação de LuaNua consistiu na extensão e modificação das bibliotecas de depuração e serialização da linguagem Lua, aproveitando várias funcionalidades que Lua já oferece. Este capítulo discute aspectos relevantes da implementação baseado na formalização apresentada na seção 3.4.

5.2.1 Criação de corotinas

A regra 3-18 descreve a criação de uma nova co-rotina vazia e o retorno da referência que o representa. A função newthread cria uma thread vazia com status inválido e a retorna na pilha:

```
static int db_newthread(lua_State *L) {
  lua_State *L1 = lua_newthread(L);
```

```
  lua_setstatus(L1, LUA_ERRRUN);
  return 1;
}
```

5.2.2 Reificação

A forma em que os valores são reificados depende do tipo:

nil, boolean, number, string e table : são reificados como eles próprios.

protótipos : devolve a informação contida no bytecode, mas na forma de tabela.

closures : retorna o protótipo da função e a lista de upvalues apontados pelo closure. Os campos *isC* e número de upvalues podem ser extraídos usando a função *getinfo* e o ambiente com a função *getfenv*.

upvalues : são reificados fechados. A reificação dos upvalues retorna o valor referenciado pelo campo que aponta ao valor do upvalue.

lightuserdata e userdata : não são suportados: uma tentativa de reificação lançará um erro de tipo.

co-rotinas : A reificação de registros é formalizada na regra 3-20. Esta regra descreve a operação de reificação de um registro de ativação como a localização do frame correspondente ao nível requisitado dentro da co-rotina e a extração do seu conteúdo. A primitiva *content* recebe uma co-rotina e um nível e devolve os valores correspondentes à chamada contida nesse nível. Estes valores incluem um closure, uma lista variável de argumentos e uma lista de registros.

Na implementação da reificação de registros de ativação, o nível é utilizado para localizar o registro correspondente:

```
CallInfo *ci = L1->ci - level;
```

Desse registro se extraem o closure, a tabela de argumentos variáveis, a tabela de registros, o contador de programa (*currentpc(L1, ci)*), o tamanho do ci ($ci \rightarrow top$ - $ci \rightarrow func$) e os resultados esperados a serem devolvidos pela chamada ($ci \rightarrow nresults$). Como mostra a figura 5.1, o closure se encontra na posição apontada pelo ponteiro $ci \rightarrow func$. A tabela de funções variáveis, caso exista, é preenchida com o conteúdo das posições situadas entre a posição imediatamente seguinte a $ci \rightarrow func$ e a anterior à $ci \rightarrow base$.

```
int nvargs = ci->base - (ci->func+1);
```

A tabela de registros contém os valores situados desde a posição *ci→base* até a posição anterior ao ci→func do registro de ativação seguinte ou ao topo da pilha no caso de tratar-se do último slot ocupado.

```
int nregs = ((ci != L1->ci) ? (ci+1)->func-1 : L1->top-1)
          - ci->base + 1;
```

Já que a pilha pode conter valores nulos, são incluídos dois campos contendo o tamanho de ambas as tabelas. Por último, para admitir a instalação de frames em qualquer ordem, precisamos de um campo (*ci→func* - *L1→stack*) especificando a posição de início deste registro de ativação na pilha. Este campo não é necessário se a instalação se executa inserindo sempre o registro seguinte em direção ao topo da pilha.

Como parte da API se oferece a função *getopenupval* que permite a reificação da lista de upvalues abertos. Esta função devolve uma tabela que contém a lista de upvalues abertos que apontam a valores na pilha. Esta lista é parte da definição da estrutura da thread (*L→openupval*) e consiste em uma lista encadeada de upvalues. Esta lista é percorrida para extrair de cada upvalue a posição da pilha que está sendo apontada e o conteúdo dessa posição:

```
for (uv=gco2uv(L1->openupval); uv!=NULL; uv=gco2uv(uv->next)) {
  pos = uv->v - L1->stack;
```

5.2.3 Instalação

A instalação é realizada da seguinte forma, dependendo do tipo:

nil, boolean, number, string e table : a instalação é uma operação sem efeito.

protótipos : são preenchidos a partir da tabela que contém a sua representação. Esta representação inclui os campos maxstacksize, nups, numparams, k, sizek, code, is_vararg, upvalues, maxstacksize, locvars, lineinfo, lastlinedefined, linedefined e source. Embora vários desses campos ofereçam na realidade informação de depuração, que poderia ser omitida aos efeitos da execução, eles permitem validar o código (utilizando a função *lua_checkcode*). O campo sizek é necessário pelo fato de que a tabela k pode conter valores nulos.

closures : Precisam de um protótipo para serem inicializados. O ambiente pode ser adiciondo posteriormente, através da função *setfenv*. Os upvalues podem ser inicializados com valor *nil* e prenchidos posteriormente, permitindo assim evitar loops infinitos durante a instalação (pois upvalues podem conter a própria closure, ainda não instalada).

upvalues : Upvalues abertos podem apontar para valores em qualquer lugar da pilha, e inclusive em outras pilhas. A instalação de upvalues nessa implementação segue a solução implementada em Pluto. A idéia consiste em instalar os upvalues fechados e abrí-los somente quando chamada a função *openupvals*, após o fim da instalação. Os upvalues são extraídos da tabela e inseridos na lista encadeada L→openupval. O ponteiro uv→v que aponta ao valor do upvalue é reposicionado apontando à nova pilha no offset da posição que ele apontava na pilha original (repare que esta posição pode ser manipulada). Finalmente, o upvalue é marcado como não coletável e retirado da lista do coletor de lixo (se percorre o gc raiz para achar o upvalue e desligá-lo do gc).

co-rotinas : A regra 3-21 estabelece que a instalação da representação de um registro de ativação em uma co-rotina consiste em inserí-la no nível requisitado da co-rotina. Isto se traduz na localização do frame correspondente na pilha e a atribuição dos valores contidos na tabela no frame localizado. A instalação de um registro é efetuada através da primitiva install. A instalação completa requer do prenchimento da pilha, mais o status e a lista de upvalues abertos referenciados pela co-rotina.

Alterações do código podem levar a erros não recuperáveis.Por outro lado, como discutido na subseção 4.2.10, a capacidade de modificar o código permite modificar o comportamento da execução própriamente dita, acrescentando ou tirando aspectos que não foram previstos no momento da programação. Existe então um compromisso entre o grau de flexibilidade e a consistência do sistema. Nessa implementação, o código é devolvido como um array de números, facilitando a manipulação para propósitos de experimentação, mas lembrando que deve dar-se preferência as boas práticas de programação em detrimento daquelas que requeram a modificação manual do código.

5.2.4 Suspensão da computação

Vários exemplos do capítulo 4 mostraram a necessidade de suspender uma corotina desde o metanível, ou seja, por fora da execução normal. Isto implica em mudar o status da corotina e instalar um registro de execução

equivalente a uma chamada a yield. O status da co-rotina pode ser mudado usando a função *setstatus*, que tem os argumentos thread e o novo status.

5.3 Modificações ao interpretador Lua

As primitivas da API foram implementadas como funções da biblioteca debug. A política é oferecer as funcionalidades necessárias para captura e restauração de computações focado em aplicações de migração e persistência. Em vários casos existe uma forma baixo nível de executar determinada operação, como por exemplo, suspender uma corotina. Entretanto, recomenda-se o uso de funções de mais alto nível em prol da consistência da aplicação e a maior facilidade de uso da biblioteca.

Ao todo, as funções acrescentadas são as seguintes:

content retorna o próprio valor se for atômico e a representação do valor em formato de tabela caso contrário;

install instala uma representação no espaço de memória, devolve um valor equivalente ao representado;

fields retorna os campos que compõem a estrutura de um valor de um determinado tipo;

name no caso de valores estruturados, devolve o endereço em que está armazenado o valor, gerando uma identidade única no sistema;

As funções seguintes são específicas da implementação em Lua.

newthread devolve uma nova corotina vazia;

setstatus muda o status de uma thread;

getopenupvals cria uma tabela com os upvalues abertos referenciados pela corotina

openupvals abre os upvalues contidos na lista;

gettrail devolve uma lista que contém o caminho das chamadas desde a mainthread até a corotina requisitada.

Algumas funções auxiliares internas também foram acrescentadas, como a função setuvvalue, para copiar valores de tipo upvalue.

Em LuaNua a captura de co-rotinas é realizada a nível de registros de ativação, para satisfazer o requisito de controle sobre o grafo de dependências

e a granularidade da computação. O fato da instalação não ser atômica, pois os registros de ativação que compõem a nova thread são instalados individualmente, gera algumas dificuldades. Em primeiro lugar, existem variáveis que correspondem à co-rotina toda (como é o caso do status e os upvalues abertos) que devem ser setadas pelo programador antes da execução. Em segundo lugar, a API é diferente a dos outros tipos de dados (mas similar à de reificação). As vantagem do método são a simetria com a reificação, e que permite a atualização de registros de ativação de co-rotinas.

Alguns dos valores necessários para a instalação já são devolvidos por Lua, como: o environment, número de upvalues, e se a função é "C" ou "Lua". Eles não precisam ser retornados pela função content na reificação.

A geração de representações manipuláveis (e portáveis) de protótipos está baseada nas funções de serialização/de-serialização (bibliotecas dump e undump). As modificações consistiram em permitir-lhes operar com tabelas (lembre-se que funções em Lua podem ser serializadas/de-serializadas na forma de uma string através das funções *string.dump* e *loadstring*).

A manipulação do estado de execução traz novos problemas na implementação da linguagem. Alguns deles são enumerados a seguir:

1. Reificação de valores opacos: Em Lua, a reificação do estado de execução implica na reificação de upvalues e protótipos, que são normalmente invisíveis para o usuário. A linguagem precisa tratar os seus valores de forma homogênea, portanto, o respectivo tratamento deverá ser incorporado. Se trata de ser corretamente tratados pelas funções de Lua (o print, por exemplo, deveria imprimir o tipo e a referência) e pelos mecanismos de tratamento de erros (podem acontecer erros não tratados se eles forem indexados, por exemplo).

2. Integridade da informação restaurada: Lua é uma linguagem bem comportada, dessa forma falhas de segmentação não são permitidas. Entretanto, inúmeros erros podem ser cometidos durante a restauração. As funções de restauração deveriam garantir a verificação dos valores submetidos para instalação. Na construção de protótipos esta verificação é feita através da função *lua_checkcode*.

3. Coleta de lixo: Valores criados unicamente para a instalação deveriam depois serem liberados. A solução a este problema está nas mãos do programador.

Como já foi dito, LuaNua foi implementada através da modificação das bibliotecas Lua. Na realidade, as facilidades que Lua oferece para a integração

de bibliotecas permitiriam implementá-la como uma nova biblioteca. As exceções estão no controle das operações efetuadas sobre os novos valores (upvalues e protos) que aparecem agora na linguagem. Por exemplo, upvalues e protos não podem ser indexados, e a linguagem deveria tratar este tipo de operação. Já que isto não foi feito nesse trabalho, atualmente essas operações geram falhas de segmentação. Nesse sentido, já que quando invocada com parâmetros não atômicos a função *print* deve imprimir uma string contendo o tipo e o endereço do valor, a função *lua_topointer* (da biblioteca lapi.c) teve que ser modificada para acrescentar o retorno dos valores de protótipo e upvalue.

```
LUA_API const void *lua_topointer (lua_State *L, int idx) {
  StkId o = index2adr(L, idx);
  switch (ttype(o)) {
    case LUA_TTABLE: return hvalue(o);
    case LUA_TFUNCTION: return clvalue(o);
    case LUA_TTHREAD: return thvalue(o);
    case LUA_TPROTO: return prvalue(o); --> acrecimo
    case LUA_TUPVAL: return uvvalue(o); --> acrecimo
    case LUA_TUSERDATA:
    case LUA_TLIGHTUSERDATA:
      return lua_touserdata(L, idx);
    default: return NULL;
  }
}
```

Entretanto, a implementação por fora da linguagem foge da intenção inicial, que trata justamente do suporte da linguagem para as operações de reificação e instalação. Isto traria de volta os problemas comuns às implementações baseadas na modificação da plataforma de execução relatadas no capítulo 2 (por exemplo, a falta de um compromisso de suporte em cada atualização da linguagem).

5.3.1 Validade da representação reificada em outras implementações da linguagem

Uma das desvantagens do método proposto é a sua estreita relação com a implementação da linguagem. Já que a implementação da linguagem não está padronizada, não existem garantias de que exista uma representação genérica (além do bytecode) que permita as diferentes implementações interagirem. De fato isto pode não ser possível.

Por exemplo, Lua2IL [MI05] é um transformador de bytecodes Lua para bytecodes da .NET Common Language Runtime (CLR). Já que o

Lua2IL compila bytecodes Lua, a implementação de um mecanismo de reificação/instalação de protótipos seria compatível entre as duas implementações. Entretanto, o modelo de execução do Lua2IL é bem diferente ao de Lua. Lua2IL usa a pilha do própio CLR (Common Language Runtime) como pilha de controle, em lugar de uma pilha própria como é o caso de Lua. Por este motivo, as corotinas são preemptivas. No Lua2IL não é possível de especificar a posição em que a execução vai continuar. A alternativa de um interpretador, como KahLua[KahLua], uma Virtual Machine implementada em Java que interpreta um subconjunto de Lua, seria mais interessante. Entretanto, KahLua encontra-se em desenvolvimento e ainda não inclui co-rotinas, que são indispensáveis neste trabalho, aos efeitos da restauração do ponto de execução.

5.4 Comentários finais

A implementação de LuaNua considerou as várias funcionalidades reflexivas de Lua de forma a evitar duplicidade na extração de informações. Entretanto, funcionalidades adicionais deveram ser implementadas, as quais foram acrescentadas à biblioteca de depuração. Consideramos que a biblioteca de depuração é apropriada para estas novas funcionalidades porque ela não é comumente utilizada para programas típicos, pois permite quebrar "invariantes" da linguagem [Ierusalimschy06] como acontece ao utilizar essa proposta. Também, as funcionalidades implementadas como parte dessa biblioteca são enxergadas como pertencendo ao nível meta, evitando confusões entre computações pertencentes ao domínio e computações reflexivas.

6
Considerações finais

O trabalho feito em torno do tema de captura e restauração de estado é extenso. Linguagens reflexivas como LISP, Prolog, e Smalltalk oferecem este suporte há muito tempo. Esta tese, entretanto, apresenta requisitos específicos que não podem ser satisfeitos por grande parte destes trabalhos.

Linguagens que oferecem mecanismos opacos, *dump*, ou caixa-preta são simples de usar mas, por outro lado, limitam a flexibilidade dos mecanismos de captura e restauração, pois fixam aspectos que afetam a forma como a linguagem conseguiria satisfazer os requisitos de diferentes aplicações. Diferente dessas abordagens, o foco deste trabalho está na flexibilidade para a adaptação a requisitos específicos, que implica oferecer controle sobre aspectos como a granularidade e a extensão da informação a ser restaurada.

Por exemplo, a proposta de Hsieh et al. [HWW93], chamada de *migração de computações*, foi projetada especificamente para migrar o registro de ativação no topo de uma computação. A proposta aborda migração (mas não clonagem) de computações, dessa forma, quando o procedimento de migração está na base da pilha, a informação de *binding* é enviada à continuação e a thread original é destruída. Caso contrário, o *stub* cliente espera pelo resultado e o envia ao chamador.

Outros trabalhos suportam tanto migração quanto persistência, mas na forma de mecanismos "caixa-preta" que seguem uma semântica pré-determinada do tipo *dump*. Este é o caso da proposta de Tack et al. [TKS06], que apresenta uma formalização, aplicada no contexto da linguagem AliceML, para serialização e minimização de grafos para qualquer tipo de dados. A serialização em AliceML é transparente e segue uma abordagem profunda, ou seja, é capturado completamente o fecho transitivo de todos os objetos referenciados pelo valor.

A intervenção manual tem sido defendida tanto no contexto de persistência quanto nos sistemas distribuídos. Sewell et al. [SLW+07] defendem esta abordagem para controlar a interação entre instâncias de diferentes versões do mesmo programa coexistindo em um sistema distribuído. O trabalho explora as funcionalidades necessárias para a programação distribuída tipada de alto

nível na linguagem de programação Acute. Essas funcionalidades incluem o suporte para interação segura entre programas compilados em separado, e leva em conta a coerência na revinculação em caso de problemas de versionamento. Acute provê construções para a serialização arbitrária de valores em bytestrings com ênfase em tipagem e versionamento. A serialização de uma execução pode ser implementada chamando uma primitiva que converte uma computação em um valor (*thunkify*) e depois serializando o resultado. Entretanto, a thunkificação é uma operação atômica. A intervenção manual em Acute consiste em dar dicas ao compilador. Por exemplo, podem ser inseridas marcas para sinalizar os limites da captura. Também é disponibilizada uma linguagem para especificar restrições de versionamento, entre outros.

No contexto das aplicações persistentes, a intervenção manual tem sido estudada como solução para problemas de manutenção e versionamento, relacionados com a revinculação de computações armazenadas, depois que uma aplicação é atualizada. Um exemplo é o projeto da linguagem de programação E [Miller06]. E é uma linguagem puramente baseada em objetos destinada à computação persistente distribuída que foi implementada sobre a linguagem Java. A serialização em E é atômica e superficial: objetos não serializáveis são serializados como referências quebradas em lugar de ser lançada uma exceção. E é baseado em eventos, e a captura somente é possível quando a pilha está vazia. E propõe uma combinação de persistência ortogonal transparente (para tolerância a falhas) e persistência manual (para atualizações). E promove a separação de mecanismos e políticas como meio para permitir a implementação de diferentes aplicações.

Poucos trabalhos abordam o problema da reflexão de comportamento com granularidade fina. Um deles é Reflex [TNCC03], uma extensão de Java que oferece reflexão com granularidade fina de entidades Java. Por restrições da linguagem, as transformações de bytecode que permitem reificar execuções Java são executadas em tempo de carga (não pode ser modificado durante a execução). Geppetto [RDT08] é uma implementação de Reflex para Squeak que não tem essa limitação já que Squeak é uma linguagem dinâmica (ou seja, admite carga dinâmica). Até onde sabemos, este trabalho é o único que a propos integrar as facilidades reflexivas necessárias para a captura e restauração flexíveis na própria linguagem.

6.1 Experiências da implementação em Lua

A implementação da API proposta em Lua foi facilitada por várias características da linguagem. O fato das co-rotinas serem valores de primeira

classe facilita a captura de execuções nesse nível de granularidade. Os mecanismos de suspensão/reinicio da co-rotinas permitiram satisfazer a necessidade de um meio de reiniciar a execução a partir de determinado ponto. Também, o fato delas serem *stackful* permite reiniciar a execução inclusive de operações aninhadas. Entretanto, o fato das co-rotinas serem assimétricas e por tanto, precisarem do retorno à co-rotina chamante, e o mecanismo de ativação estar baseado na pilha C, complicou a restauração de aplicações compostas por várias co-rotinas ativas. A necessidade de restaurar uma fila de chamadas não portável pode ser superada através de métodos como os mostrados no capítulo 4.

A implementação faz uso extensivo da expressividade das tabelas Lua. As tabelas prencheram perfeitamente os requisitos de composição e navegação para a estrutura de dados que receberia as representações e permitiram apresentar estas representações – a serem analisadas e manipuladas – de forma conveniente para o programador graças à capacidade de poderem ser indexadas usando qualquer valor da linguagem.

Boa parte das funcionalidades necessárias já são providas pela linguagem. Entre elas estão a capacidade de testar tipos, atribuir valores a variáveis na pilha, e verificar o tipo de uma função (Lua ou C). A implementação foi muito facilitada pela organização de Lua estar muito bem definida em bibliotecas e a facilidade com que a linguagem pode ser estendida através de novas funções.

A comunicação entre a linguagem e a parte C (através da pilha) também é muito simples. O fato das informações relativas às chamadas estarem concentradas em registros de ativação na pilha de execução, além de ter um array de ponteiros à pilha que facilita a localização desses registros e as informações contidas, permitiu capturar facilmente todas as informações necessárias para a reificação das co-rotinas.

A abertura de upvalues foi um aspecto difícil e não muito bem resolvido nesta implementação. Os upvalues fechados, ao serem abertos e portanto, colocados para apontar a pilha, precisam ser retirados da lista do coletor de lixo. Já que a lista não é duplamente encadeada, deve ser percorrida a lista toda (no pior caso) para achar cada upvalue e retirá-lo, o que resulta em uma operação pouco eficiente.

Um aspecto que permitiu a implementação foi o fato de Lua ser uma linguagem de código aberto. Entretanto, não existe muita documentação dos internals da linguagem, de forma que boa parte do trabalho se baseou nas informações obtidas através da interação com os membros da equipe de desenvolvimento e no estudo do pacote de serialização Pluto.

A validação do trabalho realizado é complexa, pois se baseia em critérios

subjetivos relacionados a utilidade dos mecanismos propostos. Uma avaliação mais completa precisaria de um tempo maior de amadurecimento e interação com um pool de usuários. Entretanto, contatos com implementadores e uma revisão no forum de Lua [1] permitem reparar em limitações que podem ser resolvidas através da disponibilidade dos mecanismos propostos, representam problemas reais nas implementações. Este é o caso, por exemplo, do compartilhamento de upvalues após a restauração, e a captura e restauração portável de funções Lua e co-rotinas (por exemplo, para a implementação de continuações multi-shot, troca de mensagens). Atualmente, as aplicações que precisam dessas funcionalidades são com freqüência implementadas através da biblioteca Pluto, descrita anteriormente. Outra funcionalidade que tem se mostrado de interesse é a reificação da fila de chamadas até determinada co-rotina (a função aqui chamada de *gettrail*).

6.2 O que precisa uma linguagem para oferecer o suporte necessário para captura e restauração de estado das computações

Ao implementar o suporte necessário para satisfazer os requisitos colocados na subseção 3.2.1, detectamos um grupo de funcionalidades que a linguagem precisaria oferecer com esse objetivo. Estas funcionalidades são:

- Suporte a reificação das computações minimizando a granularidade;
- Suporte a instalação de representações como novas computações ou modificações a computações existentes;
- Suporte a composição de computações gerando uma continuação portável;
- Uma estrutura de dados que permita navegação e composição;
- Um mecanismo que permita restaurar a execução apartir de determinado ponto desde o metanível;
- Para algumas aplicações, um método de suspensão da execução.

A restauração da execução apartir de determinado ponto desde o metanível é trivial em programas escritos usando CPS. Em outros casos, a ausência de operações tipo *goto* devido a sua inconveniência dificulta esta tarefa. Mecanismos de suspensão/ativação de computações baseados em multithreading cooperativo resolvem bem este problema, diferente daqueles em que a troca de contexto é preemptiva.

[1] http://bazar2.conectiva.com.br/mailman/listinfo/lua

Suportar a composição de computações trata o fato de que computações podem estar formadas por computações de menor granularidade. Por exemplo, uma computação em Lua pode estar formada por várias co-rotinas em determinado momento, a recomposição de essa execução implica na restauração da fila de ativações das co-rotinas instaladas. Outras linguagens podem precisar da restauração de mecanismos de sincronização (mutexes, etc).

O método de suspensão da execução não foi colocado como imprescindível porque, para capturar a execução, esta não precisa necessariamente ser suspensa.

6.3 Conclusão

O grande problema da migração heterogênea de computações é que a migração heterogênea deveria dispor de um modelo abstrato da computação que pudesse ser restaurado em qualquer ambiente. Entretanto, a realidade é que a representação da computação está perto demais da implementação, e em geral, não pode ser abstraída até esse ponto.

A migração heterogênea de computações é um problema difícil. Além das dificuldades inerentes ao problema em si, estão as decorrentes da falta de suporte das linguagens de programação atuais, que contribuem em boa parte à complexidade e aos problemas de desempenho relatados [MRS08]. As linguagens de programação deveriam oferecer suporte para as operações de captura e restauração de computações de forma a permitir a implementação de diferentes aplicações que precisam da capacidade de manipular computações, como é o caso da migração e persistência de execuções. Isto facilitaria o trabalho dos programadores, assim como diminuiria as situações em que esta restauração não pode ser realizada, aumentando a portabilidade. O nível da linguagem é o nível adequado para estas operações. Da mesma forma, as políticas específicas de cada domínio de aplicação pertencem ao nível de aplicação e estão fora do escopo do projeto da linguagem. Assim se elimina a necessidade de diferentes notações para políticas diferentes.

Este trabalho foi focado no estudo das funcionalidades necessárias para oferecer este suporte de forma flexível que permita acomodar os requerimentos específicos de cada aplicação. A captura e restauração de estado baseados na reificação e instalação que manipula representações navegáveis e componíveis do estado de execução permite ao programador controlar variáveis como a granularidade, a quantidade de estado a ser transmitida, e a forma em que a computação pode ser revinculada ao novo contexto de execução. Variadas aplicações podem ser implementadas através dessa abordagem.

Este método tem desvantagens, pois o trabalho do programador aumenta e em consequência, a probabilidade de erros. Isto sugere a implementação de bibliotecas de mais alto nível para facilitar o procedimento de captura e restauração para os cenários mais comuns, baseada nas primitivas de reificação e instalação da linguagem.

Entre os resultados relevantes deste trabalho estão (i) o levantamento da necessidade de suporte a captura e restauração heterogêneas nas linguagens de programação atuais, (ii) a proposta de uma API reflexiva que oferece este suporte, (iii) a definição formal da semântica operacional das operações dessa API, (iv) a validação desta API através da implementação de LuaNua, que habilita a linguagem Lua para a programação de migração e persistência de computações, tudo isto dentro do contexto da pesquisa de quais funcionalidades uma linguagem deveria oferecer para facilitar a implementação de migração e persistência de computações. O capítulo 4 mostrou que a API proposta tem um poder expressivo suficiente para satisfazer os requisitos colocados, assim como permitir a manipulação de estruturas não previstas originalmente na linguagem, como as continuações multi-shot. Os exemplos apresentados servem como prova de completude da linguagem, pois ilustram as variadas aplicações que a linguagem permite implementar.

A partir deste ponto Lua encontra-se habilitada para o desenvolvimento de aplicações de migração. Pretendemos como trabalho futuro estudar os problemas relacionados ao desenvolvimento de sistemas distribuídos baseado nas ferramentas de reificação e instalação de computações que Lua oferece.

Acreditamos que a capacidade de reificar e instalar computações é uma necessidade das linguagens atuais que pretendem oferecer suporte para aplicações de migração e persistência de computações de uma forma flexível. Boa parte das dificuldades enumeradas em [MRS08] relacionadas à migração heterogênea de computações podem ser resolvidas através destas funcionalidades, facilitando a implementação destas técnicas e permitindo assim que seu potencial seja re-descoberto.

Referências Bibliográficas

[BBG+02] BAK, L.; BRACHA, G.; GRARUP, S.; GRIESEMER, R.; GRISWOLD, D. ; HOLZLE, U.. **Mixins in Strongtalk**. In: PROCEEDINGS OF THE "INHERITANCE WORKSHOP"AT ECOOP'02, 2002. 4.1.1

[BD01] BETTINI, L.; NICOLA, R. D.. **Translating strong mobility into weak mobility**. Lecture Notes in Computer Science, 2240:182–197, 2001. 2.1, 2.3.1

[BHKP+04] BOUCHENAK, S.; HAGIMONT, D.; KRAKOWIAK, S.; PALMA, N. D. ; BOYER, F.. **Experiences implementing efficient Java thread serialization, mobility and persistence**. Software: Practice & Experience, 34(4):355–393, 2004. 2.1, 2.2.3

[BSS94] VON BANK, D. G.; SHUB, C. M. ; SEBESTA, R. W.. **A unified model of pointwise equivalence of procedural computations**. ACM Trans. Program. Lang. Syst., 16(6):1842–1874, 1994. 2.1

[BWD96] BRUGGEMAN, C.; WADDELL, O. ; DYBVIG, R. K.. **Representing control in the presence of one-shot continuations**. In: PLDI '96: PROCEEDINGS OF THE ACM SIGPLAN 1996 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, p. 99–107, New York, NY, USA, 1996. ACM. 4.2.8

[CFH+05] CLARK, C.; FRASER, K.; HAND, S.; HANSEN, J. G.; JUL, E.; LIMPACH, C.; PRATT, I. ; WARFIELD, A.. **Live migration of virtual machines**. In: NSDI'05: PROCEEDINGS OF THE 2ND CONFERENCE ON SYMPOSIUM ON NETWORKED SYSTEMS DESIGN & IMPLEMENTATION, p. 273–286, Berkeley, CA, USA, 2005. USENIX Association. 2.2

[CG98] CARDELLI, L.; GORDON, A. D.. **Mobile ambients**. In: FOUNDATIONS OF SOFTWARE SCIENCE AND COMPUTATION STRUCTURES: FIRST INTERNATIONAL CONFERENCE, FOSSACS '98. Springer-Verlag, Berlin Germany, 1998. 2.1

[CHK94] CHESS, D.; HARRISON, C. ; KERSHENBAUM, A.. **Mobile agents: are they a good idea?** Technical Report RC 19887, IBM Research Division, T.J. Watson Research Center, 1994. 1

[CJK95] CEJTIN, H.; JAGANNATHAN, S. ; KELSEY, R.. **Higher-order distributed objects**. ACM Transactions on Programming Languages and Systems (TOPLAS), 17(5):704–739, 1995. 2.1

[CS02] CHANCHIO, K.; SUN, X.-H.. **Data collection and restoration for heterogeneous process migration.** Software: Practice & Experience, 32(9):845–871, 2002. 2.1, 2.2.3

[Cardelli95] CARDELLI, L.. **A language with distributed scope**. In: PROCEEDINGS OF THE 22ND ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES (POPL '95), p. 286–297, New York, NY, USA, 1995. ACM. 2.1

[Cardelli97] CARDELLI, L.. **Mobile Computation**. In: Jan Vitek; Christian Tschudin, editors, MOBILE OBJECT SYSTEMS: TOWARDS THE PROGRAMMABLE INTERNET, volumen 1222, p. 3–6. Springer-Verlag: Heidelberg, Germany, 1997. 2.1

[Cardelli99] CARDELLI, L.. **Abstractions for Mobile Computation**. In: SECURE INTERNET PROGRAMMING, p. 51–94, 1999. 2.1

[DM95] DEMERS, F.-N.; MALENFANT, J.. **Reflection in logic, functional and object-oriented programming: a short comparative study**. In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI. 3.2

[DO99] DOUGLIS, F.; OUSTERHOUT, J.. **Transparent process migration: design alternatives and the sprite implementation**. p. 57–86, 1999. 1.1

[Eskicioglu90] ESKICIOGLU, M. R.. **Design issues of process migration facilities in distributed systems**. In: IEEE COMPUTER SOCIETY TECHNICAL COMMITTE ON OPERATING SYSTEMS AND APPLICATION ENVIRONMENTS NEWSLETTER, volumen 4, p. 3–13, 1990. 1

[FCG00] FERRARI, A.; CHAPIN, S. ; GRIMSHAW, A.. **Heterogeneous process state capture and recovery through Process Introspection**. Cluster Computing, 3(2):63–73, 2000. 2.1, 2.3.1

[FF06] FELLEISEN, M.; FLATT, M.. **Programming languages and lambda calculi**. 3.4

[FPV98] FUGGETTA, A.; PICCO, G. P. ; VIGNA, G.. **Understanding Code Mobility**. IEEE Transactions on Software Engineering, 24(5):342–361, 1998. 1, 2.1

[FW84] FRIEDMAN, D. P.; WAND, M.. **Reification: Reflection without metaphysics**. In: LFP '84: PROCEEDINGS OF THE 1984 ACM SYMPOSIUM ON LISP AND FUNCTIONAL PROGRAMMING, p. 348–355, New York, NY, USA, 1984. ACM. 3.2

[Funfrocken98] FÜNFROCKEN, S.. **Transparent migration of Java-based mobile agents**. In: PROCEEDINGS OF THE SECOND INTERNATIONAL WORKSHOP ON MOBILE AGENTS (MA '98), p. 26–37. Springer-Verlag, 1999. 2.1, 2.2.3

[HWW93] HSIEH, W. C.; WANG, P. ; WEIHL, W. E.. **Computation migration: enhancing locality for distributed-memory parallel systems**. In: PROCEEDINGS OF THE FOURTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING (PPOPP '93), p. 239–248, New York, NY, USA, 1993. ACM. 2.1, 4.2.6, 6

[Henderson80] HENDERSON, P.. **Functional Programming**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1980. 3.4

[IFC05] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; FILHO, W. C.. **The Implementation of Lua 5.0**. Journal of Universal Computer Science, 11(7):1159–1176, 2005. (document), 4.2.10, 5.2

[IKKW02] ILLMANN, T.; KRUEGER, T.; KARGL, F. ; WEBER, M.. **Transparent migration of mobile agents using the Java Platform Debugger Architecture**. In: PROCEEDINGS OF THE 5TH INTERNATIONAL CONFERENCE ON MOBILE AGENTS (MA '01), p. 198–212. Springer-Verlag, 2002. 2.2.3

[Ierusalimschy06] IERUSALIMSCHY, R.. **Programming in Lua, Second Edition**. Lua.Org, 2006. 1.1, 5.4

[JC04] JIANG, H.; CHAUDHARY, V.. **Process/thread migration and checkpointing in heterogeneous distributed systems**. In: PROCEEDINGS OF THE 37TH ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES (HICSS'04) - TRACK 9, p. 10pp. IEEE, 2004. 2.1

[KahLua] KARLSSON, K.. **KahLua, a Lua Virtual Machine for Java**. http://code.google.com/p/kahlua/, 2008. 5.3.1

[Kennedy04] KENNEDY, A. J.. **Functional pearl: Pickler combinators**. Journal of Functional Programming, 14(6):727–739, 2004. 1.1

[LL87] LU, C.; LIU, J. W. S.. **Correctness criteria for process migration**. Technical Report NASA-CR-182939, University of Illinois, 1987. 2.1

[LO98] LANGE, D. B.; OSHIMA, M.. **Programming and deploying Java(TM) mobile agents with Aglets(TM)**. Addison-Wesley Professional, 1 edition, 1998. 2.1

[LTBL97] LITZKOW, M.; TANNENBAUM, T.; BASNEY, J. ; LIVNY, M.. **Checkpoint and migration of UNIX processes in the Condor distributed processing system**. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997. 1.1

[Landin64] LANDIN, P.. **The mechanical evaluation of expressions**. Computer Journal, 6(4):308–320, 1964. 3.4

[MDPW+00] MILOJICIC, D.; DOUGLIS, F.; PAINDAVEINE, Y.; WHEELER, R. ; ZHOU, S.. **Process migration**. ACM Computing Surveys, 32(3):241–299, 2000. 1, 2.1

[MI05] MASCARENHAS, F.; IERUSALIMSCHY, R.. **Running Lua Scripts on the CLR through Bytecode Translation**. Journal of Universal Computer Science, 11(7):1275–1290, jul 2005. 5.3.1

[MI09] MOURA, A. L.; IERUSALIMSCHY, R.. **Revisiting coroutines**. ACM Transactions on Programming Languages and Systems, 2009. Aceito para publicação. 4.2.7

[MJD96] J. MALENFANT, M. J.; DEMERS, F.-N.. **A tutorial on behavioral reflection and its implementation**. In: G., K., editor, PROCEEDINGS OF THE REFLECTION '96 CONFERENCE, p. 1–20, 1996. 3.2

[MP96] MILOJICIC, D. S.; PAINDAVEINE, Y.. **Process vs. task migration**. In: PROCEEDINGS OF THE 29TH HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES (HICSS'96), volumen 1, p. 636, Washington, DC, USA, 1996. IEEE. 2.1

[MRS08] MILANÉS, A.; RODRIGUEZ, N. ; SCHULZE, B.. **State of the art in heterogeneous strong migration of computations**. Concurrency and Computation: Practice and Experience, 20(13):1485–1508, 2008. 1, 2, 2.1, 2.3, 6.3

[Maes87] MAES, P.. **Concepts and experiments in computational reflection**. In: OOPSLA '87: CONFERENCE PROCEEDINGS ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS, p. 147–155, New York, NY, USA, 1987. ACM. 3.2, 4.1.1

[Maia08] MAIA, R.. **LOOP: Lua Object-Oriented Programming**. http://loop.luaforge.net/, 2008. Last visited on 20/10/2008. 4.2.7, A

[Man06] MAN, K.. **A No-Frills Introduction to Lua 5.1 VM Instructions. Version 0.1**. http://luaforge.net/docman/view.php/83/98/ ANoFrillsIntroToLua51VMInstructions.pdf, 2006. Last visited on 06/01/2008. 4.2.10

[McAffer95] MCAFFER, J.. **Meta-level architecture support for distributed objects**. In: IWOOOS '95: PROCEEDINGS OF THE 4TH INTERNATIONAL WORKSHOP ON OBJECT-ORIENTATION IN OPERATING SYSTEMS, p. 232. IEEE, 1995. 2.2.1

[Miller06] MILLER, M. S.. **Robust composition: towards a unified approach to access control and concurrency control**. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006. 3.1, 6

[Moura04] MOURA, A. L.. **Revisitando co-rotinas**. PhD thesis, PUC-Rio, 2004. 4.2.7

[Nuttall94] NUTTALL, M.. **Survey of systems providing process or object migration**. Technical Report 94/10, Imperial College Research Report, 1994. 1

[PMOY06] PESCHANSKI, F.; MASUYAMA, T.; OYAMA, Y. ; YONEZAWA, A.. **MobileScope: A programming language with objective mobility**. Available at http://www-desir.lip6.fr/~ pesch/data/pesch-ijwmc-2006-final.ps, 2006. 2.1

[RDT08] RÖTHLISBERGER, D.; DENKER, M. ; Í. TANTER. **Unanticipated partial behavioral reflection: Adapting applications at runtime**. Computer Languages, Systems and Structures, 34(2-3):46–65, 2008. 3.2, 6

[RFC4506] EISLER, M.. **XDR: External data representation standard**, 2006. 2.2.1

[SH98] SMITH, P.; HUTCHINSON, N. C.. **Heterogeneous process migration: the Tui system**. Software: Practice & Experience, 28(6):611–639, 1998. 2.1, 2.2.1, 2.2.3

[SJ95] STEENSGAARD, B.; JUL, E.. **Object and native code thread mobility among heterogeneous computers**. In: PROCEEDINGS OF THE 15TH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, p. 68–78, 1995. 2.1

[SLW+07] SEWELL, P.; LEIFER, J. J.; WANSBROUGH, K.; NARDELLI, F. Z.; ALLEN-WILLIAMS, M.; HABOUZIT, P. ; VAFEIADIS, V.. **Acute: High-level programming language design for distributed computation**. J. Funct. Program., 17(4-5):547–612, 2007. 1.1, 6

[SMY99] SEKIGUCHI, T.; MASUHARA, H. ; YONEZAWA, A.. **A simple extension of Java language for controllable transparent migration and its portable implementation**. In: PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON COORDINATION LANGUAGES AND MODELS (COORDINATION '99), p. 211–226. Springer-Verlag, 1999. 2.1, 2.2.1, 2.3.1, 2.3.1

[SRI08] A. SKYRME ; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Exploring Lua for Concurrent Programming**. In: ANAIS DO XII SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO (SBLP), 2008. 4.2.9

[Shub90] SHUB, C. M.. **Native code process-originated migration in a heterogeneous environment**. In: PROCEEDINGS OF THE 1990 ACM ANNUAL CONFERENCE ON COOPERATION (CSC '90), p. 266–270. ACM, 1990. 2.1

[Smith84] SMITH, B. C.. **Reflection and semantics in LISP**. In: PRINCIPLES OF PROGRAMMING LANGUAGES (POPL84), January 1984. 3.2

[Smith88] SMITH, J. M.. **A survey of process migration mechanisms**. ACM SIGOPS Operating Systems Review, 22(3):28–40, 1988. 1

[Smith97] SMITH, P. W.. **The Possibilities and Limitations of Heterogeneous Process Migration**. PhD thesis, Department of Computer Science, The University of British Columbia, October 1997. 2.1, 2.2.3

[Sunshine2005] SUNSHINE-HILL, B.. **Pluto: Heavy-duty persistence for lua**. http://luaforge.net/projects/pluto/, 2005. last visited on 05/09/2008. 2.2.1, 4.1

[TH91] THEIMER, M.; HAYES, B.. **Heterogeneous process migration by recompilation**. In: 11TH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, p. 18–25, 1991. 2.1

[TKS06] TACK, G.; KORNSTAEDT, L. ; SMOLKA, G.. **Generic pickling and minimization.** Electronic Notes in Theoretical Computer Science, 148(2):79–103, 2006. 2.2.1, 3.1, 6

[TNCC03] TANTER, É.; NOYÉ, J.; CAROMEL, D. ; COINTE, P.. **Partial behavioral reflection: spatial and temporal selection of reification**. In: OOPSLA '03: PROCEEDINGS OF THE 18TH ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, p. 27–46, New York, NY, USA, 2003. ACM. 6

[TRVC+00] TRUYEN, E.; ROBBEN, B.; VANHAUTE, B.; CONINX, T.; JOOSEN, W. ; VERBAETEN, P.. **Portable support for transparent thread migration in Java**. In: PROCEEDINGS OF THE JOINT SYMPOSIUM ON AGENT SYSTEMS AND APPLICATIONS / MOBILE AGENTS (ASA/MA), p. 29–43, 2000. 2.1, 2.2.3

[TVP02] TANTER, E.; VERNAILLEN, M. ; PIQUER, J.. **Towards Transparent Adaptation of Migration Policies**. In: 8TH ECOOP WORKSHOP ON MOBILE OBJECT SYSTEMS (EWMOS 2002), Málaga, Spain, June 2002. 3.2

[WHB01] WANG, X.; HALLSTROM, J. ; BAUMGARTNER, G.. **Reliability through strong mobility**. In: PROCEEDINGS OF THE 7TH ECOOP WORKSHOP ON MOBILE OBJECT SYSTEMS: DEVELOPMENT OF ROBUST AND HIGH CONFIDENCE AGENT APPLICATIONS (MOS '01), p. 1–13, 2001. 2.1

[ZG95] ZHOU, H.; GEIST, A.. **'Receiver Makes Right' data conversion in PVM**. In: PROCEEDINGS OF THE 14TH INTERNATIONAL CONFERENCE ON COMPUTERS AND COMMUNICATIONS, p. 458–464, 1995. 2.2.1

A
Biblioteca de serialização

Este arquivo é uma extensão à biblioteca de serialização do LOOP [Maia08]. LOOP (Lua Object-Oriented Programming) é um conjunto de pacotes para a implementação de diferentes modelos de programação orientada a objeto em Lua. Devido às limitações da versão corrente de Lua, não é possível manter o compartilhamento de upvalues na restauração, nem capturar co-rotinas. Estas limitações podem ser resolvidas através do uso dos mecanismos de reificação e instalação de LuaNua. A seguir se lista o arquivo que extende a biblioteca de serialização do LOOP para permitir o acesso a estes mecanismos.

```
local _G = _G
local getmetatable = getmetatable
local setmetatable = setmetatable
local getfenv = getfenv
local setfenv = setfenv
local package = package
local assert = assert
local select = select
local pairs = pairs
local pcall = pcall
local ipairs = ipairs
local loadstring = loadstring
local rawget = rawget
local rawset = rawset
local require = require
local tostring = tostring
local tonumber = tonumber
local error = error
local type = type

local debug = debug
local string = require "string"
local table = require "loop.table"
local oo = require "loop.simple"
local coroutine = require "coroutine"
```

```
local Serializer = require "loop.serial.Serializer"

local print = print
module "loop.serial.LuaNuaSerializer"

oo.class(_M, Serializer)

__mode = "k"

function value(self, id, type, ...)
local value = self[id]
if not value then
if type == "proto" then
value = debug.install(..., "proto")
elseif type == "upval" then
value = debug.install(0, "upval")
elseif type == "function" then
value = debug.install(..., "function")
elseif type == "thread" then
value = debug.newthread()
else
return Serializer.value(self, id, type, ...)
end
self[id] = value
else
return Serializer.value(self, id, type, ...)
end
return value
end

--------------------------------------------------------------------------------

function serialthread(self, thread, id)
self[thread] = self.namespace..":value(..id..")"
self:write(self.namespace,":setup(")
self:write(self.namespace,":value(",id,",'thread'),")
        local i = 0
        local ci = {}
        repeat
        ci[i] = debug.content(thread, i)
        i = i + 1
until (ci[i-1]==nil)
```

```
self:serialize(ci)
self:write(",")
self:serialize(coroutine.status(thread))
self:write(",")
self:serialize(debug.getopenupvals(thread))
self:write(")")
end

function serialupvalue(self, upvalue, id)
self[upvalue] = self.namespace..":value("..id..")"

-- serialize contents
self:write(self.namespace,":value(",id,",'upval')")
end

-- Recebe um proto
function serialproto(self, proto, id)
self[proto] = self.namespace..":value("..id..")"

-- serialize contents
self:write(self.namespace,":value(",id,",'proto',")
local content = debug.content(proto)
self:serialize(content)
self:write(")")
end

function serialfunction(self, func, id)
self[func] = self.namespace..":value("..id..")"
local content = debug.content(func)

if content.isC==1 then --Alarm, it should never happen
error ("C functions cannot be serialized")
else
self:write(self.namespace,":setup(")

-- serialize contents
self:write(self.namespace,":value(",id,",'function',")
self:serialize(content)
self:write(")")


-- serialize environment
```

```
local env
if self.getfenv then
env = self.getfenv(func)
if env == self.globals then env = nil end
end
self:write(",")
self:serialize(env)

local nups = debug.getinfo(func,"u").nups
self:write(",",nups)
        self:write(",",id) --> Debug

-- serialize upvalues contents
for _, upval in ipairs(content.upvals or {}) do
self:write(",")
self:serialize(debug.content(upval))
end

self:write(")")
end
end

_M["upval"]    = serialupvalue
_M["proto"]    = serialproto
_M["function"] = serialfunction
_M["thread"] = serialthread
```

Na biblioteca Serializer original somente foi modificada a função setup, mostrada a seguir:

```
function setup(self, value, ...)
local type = type(value)
if type == "function" then
if self.setfenv then self.setfenv(value, ... or self.globals) end
local nups = select(2, ...)
local setupvalue = self.setupvalue
if setupvalue then
for i=1, nups do
setupvalue(value, i, select(3+i, ...) or nil)
end
end

elseif type == "thread" then
```

```
local ci = select(1, ...)
for i = #ci,0,-1 do
value = debug.install(ci[i], value, 0)
end

local status = select(2, ...)
local openupvals = select(3, ...)
if openupvals then debug.openupvals(openupvals,value) end
debug.setstatus(value, status)
else
local loader = getmetatable(value)
if loader then loader = loader.__load end
if loader then loader(value, ...) end
end
return value
end
```