

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 portátil, 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 *re-binding*: 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. Além 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 atômica e 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 a partir 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 e 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 explicitamente 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 está 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 conseqüentemente, 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 carácter 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 anónima, 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 conteúdos 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 idêntica 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 portátil 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,
    nvars = 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 portátil) 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(cororo)) --> 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 suspensa ou termina a execução. Mas a pilha C não poder ser restaurada de uma forma portátil.

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 típica 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
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 *co1* transfere o controle para *co2*, que por sua vez transfere o controle para *co3*, em que a execução é capturada (na figura as setas contínuas indicam operações de *resume* e as ponteadas, de *yield*).

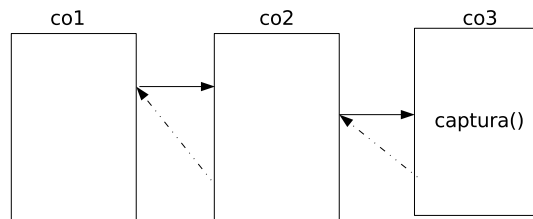


Figura 4.1: Fila de chamadas

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

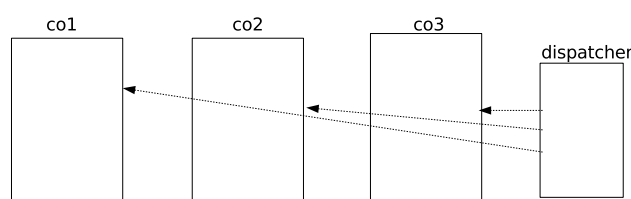


Figura 4.2: Restauração da fila de chamadas usando dispatcher

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 reconstruída usando uma chamada como esta:

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

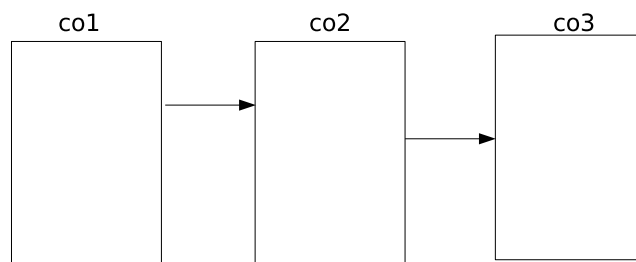


Figura 4.3: Restauração da fila de chamadas usando CPS

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.

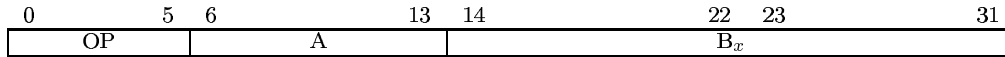


Figura 4.4: Instrução Lua

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 → 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

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
```

```
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
  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.