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

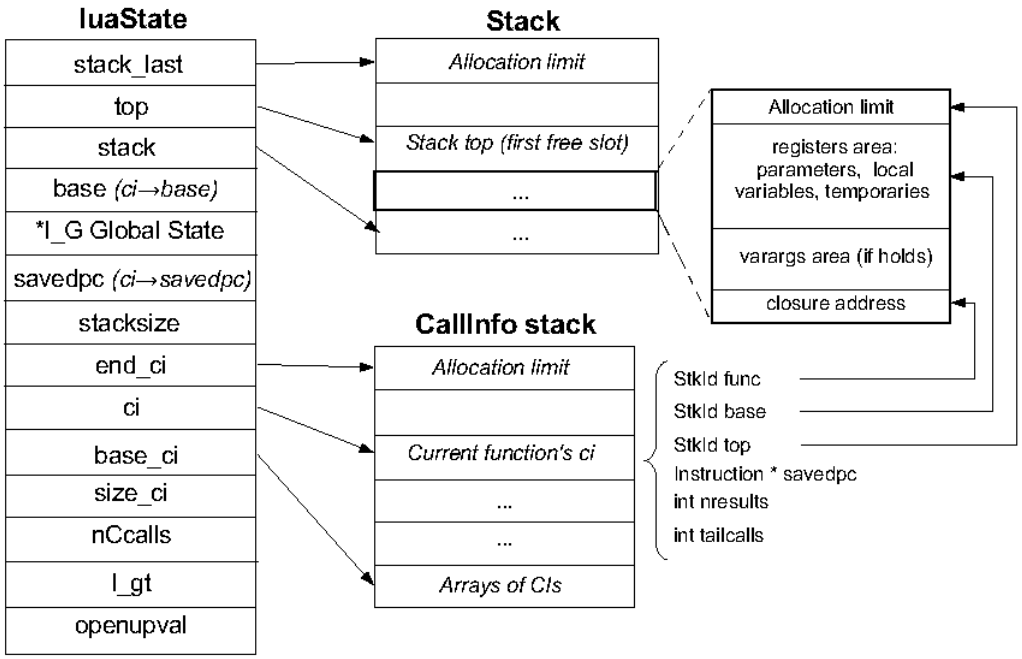

Figura 5.1: Estado Lua

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

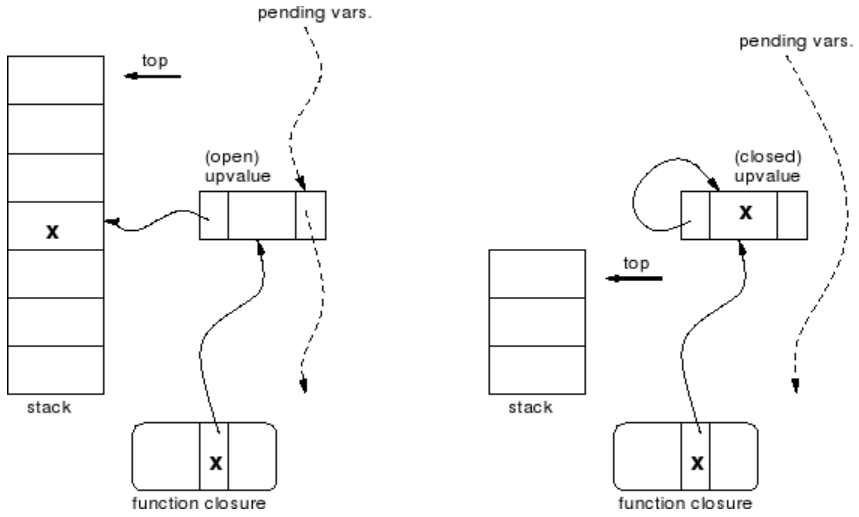

Figura 5.2: Upvalues Lua [IFC05]

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→top - ci→func*) e os resultados esperados a serem devolvidos pela chamada (*ci→nresults*). Como mostra a figura 5.1, o closure se encontra na posição apontada pelo ponteiro *ci→func*. A tabela de funções variáveis, caso exista, é preenchida com o conteúdo das posições situadas entre a posição imediatamente seguinte a *ci→func* e a anterior à *ci→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.