Um Sistema de Depuração Reflexivo para uma Linguagem de Extensão

Tomás Guisasola Gorham Roberto Ierusalimschy

tomas,roberto@icad.puc-rio.br

TeCGraf — Departamento de Informática — PUC-Rio

Resumo

Este artigo apresenta uma arquitetura para a implementação de um sistema de depuração para a linguagem Lua, que pode ser adotada por outras linguagens de extensão. O sistema foi construido com base em uma interface de monitoração externa bem definida e no uso extensivo de facilidades reflexivas. O depurador aumenta o grau de reflexividade da linguagem, pois todos os recursos de depuração são oferecidos como funções ao programa sendo depurado. Com isso, outros comandos podem ser construídos a partir dos primitivos, permitindo a configuração e extensão do depurador.

Abstract

This paper presents a design for the implementation of a debugger for the language Lua, which can be adopted by other extension languages as well. The system has been based in a well defined monitoring interface, and in the extensive use of reflexive facilities. The debugger increases the reflexiveness of the language, as all its features are implemented as functions in the program being analized. Therefore, other comands can be built over the primitive ones, allowing the configuration and extension of the debugger.

1 Introdução

Cada vez mais grandes aplicações de software estão adotando linguagens de extensão, como forma de aumentar sua flexibilidade [Rya90, Val91, Fra91, FIF94]. Linguagens de extensão permitem que um usuário configure seu ambiente de formas bastante flexíveis, por exemplo definindo macros, "scripts", ou mesmo a interface gráfica de um programa.

O crescente uso de linguagens de extensão em aplicações levou a propostas de linguagens de extensão genéricas, como por exemplo Tcl [Ous90], Libscheme [Ben94], e Elk [LB]. Essas linguagens são usualmente implementadas como bibliotecas, e podem ser ligadas a qualquer aplicação que precise de uma linguagem de extensão. Tais linguagens têm como proposta fundamental serem pequenas e abrangentes o suficiente para servirem a todos os tipos de aplicação sem se prender a nenhum deles. Desta forma, podem ser incorporadas à aplicação hospedeira com o mínimo de custo adicional tanto de desempenho quanto de tamanho.

Lua [FIF94, IdFF] é uma linguagem de extensão genérica desenvolvida na PUC-Rio, e que vem ganhando uma crescente aceitação. Atualmente Lua é usada por várias dezenas de programadores, tanto na PUC quanto fora, nos mais diversos domínios de aplicações, como descrição de interfaces gráficas interativas [FFG95], descrição de objetos gráficos procedurais [RFGR95], e construção de bases de informação para acesso via WWW. Com a crescente utilização de Lua, vem surgindo uma necessidade igualmente crescente de ferramentas de apoio ao desenvolvimento. Entre estas ferramentas, uma das principais é um depurador interativo.

O desenvolvimento de um depurador interativo para uma linguagem de extensão exige soluções diferentes daquelas adotadas em linguagens de programação de uso geral. A dificuldade de se criar um depurador interativo para Lua (ou para qualquer linguagem de extensão) é conseguir o equilíbrio entre o benefício adquirido com a nova ferramenta e o custo adicional que ela traz para todas as aplicações que usam a biblioteca.

Neste trabalho, apresentamos a solução adotada no desenvolvimento de um depurador para Lua. Esta solução se baseou em duas idéias: a construção de uma *interface de monitoração* e o uso de facilidades reflexivas.

A interface de monitoração permite que uma biblioteca externa monitore a execução de um programa pelo interpretador Lua. Através de um mecanismo de *ganchos*, a biblioteca externa pode definir rotinas que são chamadas em determinados pontos durante a interpretação de um trecho de código. Além disso, a interface oferece funções de consulta ao estado interno do interpretador, como sua pilha de execução e valores de variáveis locais.

Fazendo uso desta interface de monitoração e da API padrão da linguagem, um depurador pode ser totalmente implementado como uma biblioteca em separado, que só precisa ser ligada à aplicação durante a fase de depuração. Esta mesma interface permite a implementação de outros tipos de ferramentas, como por exemplo analisadores de desempenho ("profile").

Lua, por ser uma linguagem interpretada, oferece naturalmente várias facilidades reflexivas, como por exemplo a possibilidade de executar trechos de código armazenados como dados (strings). Lua tem seu grau de reflexividade aumentado através de funções pré-definidas que acessam suas estruturas internas. Várias facilidades que, em outras linguagens, necessitam de tratamento especial, como persistência e "clonagem" de objetos, podem ser implementadas em Lua de maneira genérica.

A implementação do depurador procurou tirar o máximo de proveito da reflexividade da linguagem. Em especial, todos os comandos de depuração são fornecidos como funções na própria linguagem, e a interação com o programador é feita através da linguagem. Essa arquitetura traz diversos benefícios, como:

- toda a funcionalidade do depurador fica diretamente disponível para o programa, que pode até definir novas funções de depuração.

- O usuário tem toda a flexibilidade da linguagem diretamente à mão. Facilidades para impressão de expressões complexas ou alteração no valor de variáveis não precisam ser fornecidas pelo depurador.

O restante deste trabalho está organizado da seguinte forma. A Seção 2 mostra alguns tipos de depuradores que foram estudados e usados como referência para o desenvolvimento deste sistema. A Seção 3 apresenta uma visão geral da linguagem Lua e suas características mais importantes. A Seção 4 descreve brevemente o sistema de monitoração implementado no interpretador da linguagem. O depurador propriamente dito está apresentado na Seção 5, onde são descritos a sua arquitetura e os recursos oferecidos. Finalmente, a última seção resume os resultados e tece algumas conclusões sobre o trabalho.

2 Depuradores

Os depuradores para linguagens compiladas são programas que controlam a execução de outros programas já compilados em código de máquina. Eles implementam todas as funcionalidades que oferecem ao usuário, desde um meio de associar os nomes das variáveis a seus valores, localizados em posições de memória, até avaliar expressões que usam estas variáveis.

Esses depuradores oferecem meios para o usuário avaliar expressões, indexar vetores, acessar campos de estruturas e recuperar valores apontados por ponteiros, dentre outros. Para executar essas tarefas, são necessárias informações sobre os tipos dos dados que estão sendo manipulados, os nomes dos campos desses tipos e os nomes das variáveis utilizadas pelo programador que foram descartadas depois da geração de código final pelo compilador. Além destas informações, o depurador precisa criar um meio para o usuário poder representar as expressões que deseja acessar, assim como os vetores e os campos de estruturas. Este meio é geralmente uma nova linguagem, mais simples que a linguagem fonte, mas com construções sintáticas semelhantes, para que o usuário se sinta confortável com elas.

Depuradores para linguagens interpretadas podem utilizar um recurso inexistente para os depuradores de linguagens compiladas: a presença do interpretador. Por outro lado, em geral todas as facilidades de depuração são implementadas diretamente no interpretador, aumentando o seu tamanho. Isso não ocorre com linguagens compiladas, onde informações de depuração são geradas somente quando necessário, e o depurador é um programa independente do programa sendo depurado.

Neste trabalho, foram pesquisadas algumas linguagens interpretadas — principalmente as de extensão — e seus depuradores; no entanto, muito pouco material está disponível.

Das linguagens de extensão, a mais importante para este trabalho é Tcl [Ous90], tanto pela sua popularidade quanto pela semelhança com Lua nos seus objetivos. Tcl (em conjunto com o toolkit Tk) é muito usada na definição de interfaces gráficas e configuração de aplicações. O interpretador da linguagem é uma biblioteca de funções C que pode ser facilmente incorporada a aplicações e que permite que cada aplicação possa estender a linguagem acrescentando novos comandos mais específicos a seus domínios.

Apesar de muito conhecida e utilizada, Tcl carece de facilidades de apoio à depuração. A linguagem oferece apenas uma ferramenta para criar sentinelas em variáveis (chamada *trace*). Com ela o programador pode associar uma função ao evento de mudança de valor da variável. Esta função é ativada automaticamente quando o valor for alterado, como se fosse uma função de tratamento de exceções. Facilidades básicas, como execução passo-a-passo, não são disponíveis.

A linguagem Python [vR93] apresenta um sistema de depuração com muitas das funcionalidades encontradas em depuradores de linguagens compiladas. Python é uma linguagem orientada a objetos projetada para ser usada para prototipagem rápida. Segundo o autor, o interpretador da linguagem pode ser facilmente estendido com novos recursos e usado para configuração e extensão de aplicações. Python é uma linguagem projetada para ser usada como linguagem de extensão, mas o tamanho excessivo de sua implementação deixa um pouco a desejar para uma linguagem com este propósito.

O depurador de Python [Hof95] apresenta os recursos básicos encontrados em outros depuradores: pontos de parada, execução passo-a-passo com ou sem entrada em chamadas de função, mudança de registro de ativação e visualização da pilha de ativação. Para oferecer acesso às variáveis, o depurador aproveita o próprio interpretador da linguagem.

O depurador de Python é um módulo independente da linguagem, que pode ser importado quando o usuário necessitar de seus recursos. Para depurar algum trecho de código ou função Python, o usuário deve chamar uma função própria para isto. Esta função vai fazer o papel do interpretador da linguagem ao executar o trecho de código, mas vai permitir a utilização dos recursos de depuração oferecidos.

Perl [WS91] é uma linguagem idealizada para o tratamento de arquivos de texto, mas que se adequa também para tarefas de gerência de sistemas. Sua versão 5 apresenta características que permitem enquadrá-la como uma linguagem de extensão. Perl 5 oferece a capacidade de definir módulos e pode ser facilmente acoplada em aplicações C ou C++, segundo o autor. Depois de acoplada ela pode chamar ou ser chamada pela aplicação, com o auxílio de um preprocessador.

A versão 5 de Perl apresenta um sistema de depuração bastante poderoso. O depurador de Perl 5 apresenta alguns recursos não encontrados nos outros depuradores. Além dos comandos mais comuns de controle de execução, visualização da pilha etc., ele oferece pontos de parada condicionais e a capacidade de associar trechos de código fonte, chamados *ações*, a linhas.

3 A Linguagem Lua

Lua é uma linguagem de extensão de uso geral e, como tal, foi projetada para ser acoplada a quaisquer tipos de aplicações. As linguagens de extensão são geralmente usadas para configurar, descrever dados ou até para acrescentar novas funcionalidades às aplicações hospedeiras.

Lua é fornecida como uma biblioteca de funções C para ser ligada à aplicação hospedeira. Esta biblioteca oferece uma API que permite à aplicação hospedeira executar um trecho de código Lua, alterar o conteúdo de variáveis globais, ou qualquer outro serviço oferecido. Como ela foi projetada para ser acoplada a outras aplicações, não há noção de programa principal.

A sintaxe de Lua é semelhante à da linguagem Pascal, mas com terminação explícita de comandos. Ela incorpora muitas das facilidades usualmente encontradas em linguagens procedurais (como comandos de controle de fluxo, definição de sub-

```
function map (a, f)
  local i = 1
  local result = {}   -- cria nova lista
  while a[i] do
    result[i] = f(a[i])
    i = i+1
  end
  return result
end
```

Figura 1: Exemplo de função em Lua.

rotinas, atribuições, operadores aritméticos infixados) e ainda oferece mecanismos para extensão da linguagem (vetores associativos, *fallbacks* e algumas facilidades reflexivas).

A Figura 1 mostra uma implementação da função *map*, que recebe uma lista $a_1, \ldots, a_n$ e uma função f, e retorna uma nova lista $f(a_1), \ldots, f(a_n)$.

Lua é uma linguagem dinamicamente tipada. Em Lua, apenas valores possuem tipo, as variáveis não. Além dos tipos comumente encontrados nas linguagens procedurais, como `number`, `string` e `function`, Lua oferece outros três: `nil`, `userdata` e `table`.

Os tipos `number` e `string` não merecem comentários, por se tratarem de tipos bem conhecidos. O tipo `nil` tem um único valor, também chamado `nil`, que tem como característica importante o fato de ser diferente de todos os outros valores. O tipo `userdata` representa um valor arbitrário da aplicação (na verdade, ponteiros C do tipo `void *`), com os quais Lua só realiza duas operações: atribuição e teste de igualdade.

Os tipos `function` e `table` são os principais responsáveis pela flexibilidade de Lua. O tipo `table` implementa vetores associativos, ou seja, vetores que podem ser indexados por quaisquer tipos de dados. Esta é uma característica que dá grande poder de expressão ao tipo e facilita a implementação eficiente de uma multitude de estrutura de dados, como arrays, records, listas, conjuntos, etc. Tabelas são criadas dinamicamente em Lua, o que traz como desvantagem o fato de terem de ser criadas explicitamente. Por outro lado, elas podem referenciar livremente outras tabelas, facilitando a modelagem de estruturas de dados genéricas, inclusive com ciclos.

Funções em Lua são valores de primeira classe, ou seja, como todos os outros valores, podem ser guardadas em variáveis, passadas como argumentos de outras funções e retornadas como resultados. Em particular, tabelas podem ter campos com valores do tipo função, o que possibilita a implementação de mecanismos de módulos e de orientação a objetos na linguagem [Gru98]. Além das funções definidas através da própria linguagem, Lua também manipula funções definidas em C. Para isso, é necessário que estas funções sejam previamente cadastradas pela aplicação hospedeira.

A linguagem Lua oferece algumas facilidades reflexivas que são importantes para a extensão da linguagem. Uma destas facilidades é a capacidade de percorrer tabelas.

Através da função `next`, pode-se obter todos os campos de uma tabela e seus valores. Analogamente, pode-se obter todas as variáveis globais do ambiente de execução de Lua através da função `nextvar`. Estas funções dão à linguagem grande capacidade de manipular seu próprio estado de execução. Com elas é possível, por exemplo, armazenar todo o ambiente de execução de Lua em uma variável e restaurá-lo posteriormente.

Outro mecanismo importante da linguagem é o de *fallbacks*. Este sistema permite a definição de funções a serem invocadas sempre que o interpretador Lua encontra alguma situação em que ele não sabe como proceder, como por exemplo, indexação de um campo inexistente em uma tabela ou operação aritmética aplicada sobre tipos incorretos. Semanticamente, o mecanismo de fallback pode ser comparado a um mecanismo de tratamento de exceção com retomada [YB85], mas pragmaticamente seu uso é bastante diferente. Ao contrário de um mecanismo de exceções, fallbacks são normalmente usadas para a criação de recursos que originalmente a linguagem não oferece ao usuário, como por exemplo herança [Gru98].

4 Suporte à monitoração externa

Esta seção descreve rapidamente o sistema de suporte à monitoração externa implementado no interpretador de Lua. O sistema se baseia num mecanismo de ganchos e num sistema de geração de informações de depuração.

O maior desafio para a implantação de um depurador em Lua é conseguir um equilíbrio entre os benefícios ganhos com esta nova ferramenta e o custo adicional imposto por sua implementação. Neste sentido, a interface de monitoração implementada respondeu bem às espectativas, mostrando-se pequena, eficiente e modular.

Para a implementação desta interface foram feitas, basicamente, três mudanças no interpretador Lua. Primeiramente, pequenas alterações na pilha permitiram o percurso e a visualização da cadeia dinâmica. Para acesso a variáveis locais e informações sobre linha de execução, foi necessária a geração de informações de depuração adicionais. Finalmente, ganchos foram introduzidos em pontos chave do interpretador.

Todas estas modificações foram feitas com o cuidado de não se contrapor à filosofia de desenvolvimento da linguagem. Para alcançar este objetivo, foram definidas duas metas para o desenvolvimento do sistema de suporte: acrescentar o mínimo possível de código ao interpretador, e não alterar o tempo de execução dos programas que não utilizam os novos recursos de monitoração.

A Figura 2 apresenta uma listagem do arquivo `luadebug.h`, que é o arquivo de cabeçalho da interface de monitoração do interpretador. A variável `lua_linehook` define o tratador do gancho de mudança de linha. A função apontada por essa variável é chamada toda vez que o interpretador Lua detecta uma mudança na linha de código sendo executada. A variável `lua_callhook` define um tratador de chamadas, que é ativado cada vez que o interpretador executa uma chamada ou um retorno de subrotina. Apenas estes pontos são suficientes para implementar os recursos de depuração normalmente encontrados nos depuradores. O tratador de mudança de linha de execução é fundamental por permitir a execução passo-a-passo dos programas e a implementação de pontos de parada, os recursos de depuração

```
#include "lua.h"

typedef lua_Object lua_Function;
typedef void (*lua_LHFunction) (int line);
typedef void (*lua_CHFunction)
               (lua_Function func, char *file, int line);

extern int lua_debug;
extern lua_LHFunction lua_linehook;
extern lua_CHFunction lua_callhook;

lua_Function lua_stackedfunction (int level);
int lua_currentline (lua_Function func);
lua_Object lua_getlocal (lua_Function func, int number, char **name);
int lua_setlocal (lua_Function func, int number);
void lua_funcinfo (lua_Object func, char **filename, int *linedef);
char *lua_getobjname (lua_Object o, char **name);
```

Figura 2: Arquivo `luadebug.h`

mais básicos. Os tratadores de chamada e retorno de função são importantes para fazer o controle das funções que entram e saem da pilha.

A função `lua_stackedfunction` retorna um "handle" de acesso ao registro de ativação correspondente ao nível dado. A função `lua_currentline` retorna a linha corrente da função ativa indicada. As funções `lua_getlocal` e `lua_setlocal` permitem a manipulação de variáveis locais nos diversos registros de ativação. Finalmente, as funções `lua_funcinfo` e `lua_getobjname` fornecem informações extras sobre uma dada função, como onde ela foi definida, seu nome, etc.

5 O depurador: uma biblioteca

Conforme já comentado, o depurador foi totalmente implementado como uma biblioteca, apoiada na interface de monitoração, descrita na seção anterior, e na API padrão de Lua. Sua arquitetura consta de dois módulos: um módulo é responsável pela definição dos *tratadores*, que são as rotinas chamadas pelos ganchos da interface de monitoração; o outro módulo define e deixa à disposição para o interpretador Lua um conjunto de funções de depuração.

O módulo de tratadores tem uma arquitetura dirigida por eventos. Os eventos correspondem aos três ganchos definidos na interface: mudança de linha de execução, chamada de função, e retorno de função. Portanto, é necessário criar um mecanismo de comunicação entre estes eventos para que eles cooperem. A solução adotada foi a utilização de um conjunto de variáveis globais que indicam o estado do depurador e as ações que ele deve realizar.

Os tratadores são os únicos pontos onde a execução do programa Lua pode

break - cria um ponto de parada.

cont - continua a execução.

delbreak - elimina um ponto de parada ou uma ação.

display - define expressão a ser mostrada toda vez que o programa for interrompido.

doalways - define uma ação a ser executada antes de cada linha do programa.

doat - define uma ação a ser executada em um determinado ponto do programa.

down - desce o registro de ativação corrente.

file - retorna o nome do arquivo fonte corrente.

goto - continua execução até um determinado ponto do programa.

level - define registro de ativação corrente.

line - retorna linha corrente.

list - lista um trecho de um arquivo.

Next - executa próxima linha do programa.

show - mostra pontos de parada definidos.

step - executa próxima linha do programa (entrando em sub-rotinas).

stop - interrompe execução do programa.

undisplay - elimina expressão do "display".

up - sobe o registro de ativação corrente.

where - exibe pilha de execução.

Figura 3: Lista de comandos do depurador.

ser interrompida e, conseqüentemente, são estes os momentos em que o usuário poderá interagir com o depurador. Isto limita os possíveis estados da pilha e do próprio interpretador a algumas situações bem conhecidas. As funções de controle da execução apenas modificam o estado do depurador de maneira a sinalizar aos tratadores quando a execução deve ser interrompida. Elas nunca interrompem a execução, pois isto poderia ter conseqüências indesejadas.

Os tratadores têm, basicamente, o mesmo mecanismo de funcionamento, que é atualizar algumas variáveis de controle, executar eventuais ações (trechos de código Lua definidos pelo usuário), avaliar as condições de parada e, eventualmente, permitir a interação do usuário.

O conjunto de funções oferecidas pelo depurador é mostrado na Figura 3. Como estão à disposição do interpretador, estas funções podem ser chamadas tanto pelo usuário, através da linha de comandos de interação, quanto diretamente pelo programa em execução.

A disponibilidade desses comandos na forma de funções Lua dá ao depurador uma grande flexibilidade, sem aumentar sua complexidade. Por exemplo, não existe um comando explícito para definir um ponto de parada condicional. Esse recurso pode ser facilmente obtido através do comando

```
doat("if condition then stop() end", line)
```

```
function goto (line, file)
  local action = format("stop(); delbreak(%d, %q)", line, file)
  doat (action, line, file)
  cont ()
end
```

Figura 4: Definição da função `goto`.

A função `doat` define uma ação — descrita através de uma string com código Lua — a ser executada em um determinado ponto do programa. Cada vez que o programa passar pela linha indicada, a ação `"if condition then stop() end"` é executada; a condição é calculada e, caso seja verdadeira, a função `stop` interrompe a execução.

Outro exemplo da flexibilidade do depurador e da sua capacidade de extensão é a possibilidade de se criar novas funções de depuração. Na Figura 4, mostramos como o comando `goto` poderia ser definido. O comando `goto` continua a execução até um determinado ponto definido pelo usuário, indicado por uma linha de um arquivo. O comando pode ser definido como uma função da linguagem criando-se uma ação na linha do arquivo indicado e continuando a execução. Esta ação, quando alcançada, deve se autodestruir. A ação é formada por uma chamada à função `stop`, que força a interrupção da execução, e por uma chamada à função `delbreak`, que irá destruir esta ação.[1] Depois de definir esta ação, a função `goto` avisa aos tratadores que a execução deve continuar, chamando a função `cont`.

5.1 Short-cuts

Apesar do uso de Lua como linguagem de interação do depurador aumentar em muito sua flexibilidade, alguns comandos podem ter uma sintaxe excessivamente longa. Por exemplo, toda chamada de função em Lua deve ser indicada por uma lista de parâmetros, mesmo que vazia. Um comando `cont` deve ser digitado como

```
ldb> cont()
```

Para contornar esse problema, o depurador oferece um mecanismo de *short-cuts*. Através de uma função `setshortcut`, o usuário pode associar comandos a caracteres. Quando a linha de comandos começa com o caracter `'`, o depurador, ao invés de passar a linha para o interpretador Lua, executa o comando associado ao caracter seguinte ao `'`. Assim, por exemplo, após a execução do comando

```
setshortcut("c", "cont()")
```

basta o usuário digitar

```
ldb>'c
```

para continuar a execução do programa. Como o comando `setshortcut` é uma função Lua, vários podem ser agrupados em um arquivo de configuração que, executado no início da seção de depuração, configura o depurador de acordo com as preferências pessoais do usuário.

[1] A função `format` retorna uma string formatada de acordo com regras similares a `printf` de C.

6 Conclusões

O desenvolvimento de um depurador interativo para uma linguagem de extensão tem um conjunto de características distintas de depuradores para linguagens convencionais. Por um lado, as restrições de tamanho são bem mais severas. Por outro lado, o depurador conta com o interpretador da linguagem, o que facilita em muito sua implementação. Na construção do depurador de Lua, esta primeira característica foi contornada através da construção de uma interface de monitoração, enquanto a segunda foi explorada ao máximo através de reflexividade.

A interface de monitoração foi adotada como forma de separar o interpretador e o depurador em bibliotecas distintas. Ela permite que um módulo externo acompanhe a interpretação de um programa Lua, chamando ganchos em ocasiões específicas (chamada e retorno de funções, mudança de linha sendo executada), e oferecendo funções de consulta e manipulação da pilha interna do interpretador. Essa interface se mostrou extremamente compacta e flexível. Ela compreende apenas dois ganchos e seis funções, e sua implementação exigiu alterações pequenas no interpretador (em torno de uma centena de linhas de código extra). A interface se mostrou completa para a implementação de todos os recursos usualmente oferecidos por depuradores, como execução passo-a-passo, manipulação de ambientes locais, código vigia, etc. Além disso, a interface se mostrou suficiente para a implementação de outras ferramentas. Esta flexibilidade foi validada com a implementação de um profiler simples para Lua, que calcula o tempo de execução de funções e conta o número de vezes que cada linha é executada. Finalmente, a possibilidade de manipulação do conjunto de variáveis locais de uma função permite o uso desta mesma interface na construção de rotinas para calcular o *closure* de um trecho de código, permitindo estender a semântica da linguagem de maneira não possível apenas com a API padrão.

A implementação do depurador, além de utilizar a interface de monitoração, utilizou amplamente a API de Lua. Em especial, a linguagem de interação entre o usuário e o depurador é Lua, e os recursos de depuração são todos oferecidos na forma de funções Lua que podem ser utilizados em conjunto com as outras funções e construções da linguagem. Essas duas características dão ao depurador uma grande flexibilidade com baixo custo. O uso de Lua como linguagem de interação permite ao usuário visualizar o resultado de expressões complexas, alterar valor de variáveis, etc, sem necessidade de suporte explícito do depurador. Por outro lado, a disponibilidade dos recursos de depuração como funções em Lua permite o uso da própria linguagem para a extensão das funcionalidades de depuração.

Os custos decorrentes da utilização do depurador foram bastante satisfatórios. Por ser uma biblioteca, seu tamanho não influi na aplicação depois da fase de depuração. Mesmo quando acoplado à aplicação, o depurador só altera o tempo de execução do código Lua se houver alguma condição de parada ativada. Caso contrário, o programa executa com eficiência normal. A perda de desempenho decorrente de um ou dois pontos de parada em linhas quaisquer de um programa fica abaixo de 50% [?].

O depurador foi totalmente desenvolvido utilizando-se o sistema de programação literária *NoWeb*, e será incorporado à distribuição da linguagem Lua, encontrada no www-site `http://www.inf.puc-rio.br/~roberto/lua.html`.

Referências Bibliográficas

[Ben94] B. W. Benson Jr. libscheme: Scheme as a C Library. Em *Proceedings of the 1994 USENIX Symposium on Very High Level Languages*, páginas 7–19. USENIX, October 1994.

[FFG95] W. Celes Filho, L. H. Figueiredo, e M. Gattass. Edg: Uma ferramenta para criação de interfaces gráficas interativas. Em *SIBGRAPI 95*, páginas 241–248, 1995.

[FIF94] L. H. Figueiredo, R. Ierusalimschy, e W. Celes Filho. The design and implementation of a language for extending applications. Em *XXI Semish*, páginas 273–284, Caxambu, 1994.

[Fra91] N. Franks. Adding an extension language to your software. *Dr. Dobb's Journal*, 16(9):34–43, 1991.

[Gru98] Grupo de Tecnologia em Computação Gráfica - TeCGraf/PUC-Rio. *Reference Manual of the Programming Language Lua*, 1998.

[Hof95] C. Hoffmann. Python Quick Reference. `ftp://ftp.python.org.edu:/pub/python/doc/QuickRef.ps.gz`, September 1995.

[IdFF] R. Ierusalimschy, L. H. de Figueiredo, e W. Celes Filho. Lua—an extensible extension language. to appear in *Software: Practice & Experience*.

[LB] O. Laumann e C. Bormann. Elk: The Extension Language Kit. `ftp://ftp.cs.indiana.edu:/pub/scheme-repository/imp/elk-2.2.tar.gz`, Technische Universität Berlin, Germany.

[Ous90] J. Ousterhout. Tcl: an embeddable command language. Em *Proc. of the Winter 1990 USENIX Conference*. USENIX Association, 1990.

[RFGR95] N. Rezende, W. Filho, M. Gatass, e E. Russo. PREGO: Um editor de objetos gráficos procedurais. Em *SIBGRAPI 95*, páginas 309–310, Outubro 1995.

[Rya90] B. Ryan. Scripts unbounded. *Byte*, 15(8):235–240, 1990.

[Val91] R. Valdés. Little languages, big questions. *Dr. Dobb's Journal*, 16(9):16–25, 1991.

[vR93] G. van Rossum. An introduction to Python for UNIX/C programmers. Em *Proc. of the NLUUG najaarsconferentie*. Dutch UNIX users group, 1993. (`ftp://ftp.cwi.nl/pub/python/nluug-pape r.ps`).

[WS91] L. Wall e R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 103 Morris Street, Suite A, Sebastopol, CA, january 1991.

[YB85] S. Yemini e D. Berry. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2), Abril 1985.