6
Considerações finais

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

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

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

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

A intervenção manual tem sido defendida tanto no contexto de persistência quanto nos sistemas distribuídos. Sewell et al. [SLW+07] defendem esta abordagem para controlar a interação entre instâncias de diferentes versões do mesmo programa coexistindo em um sistema distribuído. O trabalho explora as funcionalidades necessárias para a programação distribuída tipada de alto nível na linguagem de programação Acute. Essas funcionalidades incluem o suporte para interação segura entre programas compilados em separado, e leva em conta a coerência na revinculação em caso de problemas de versionamento. Acute provê construções para a serialização arbitrária de valores em bytestrings com ênfase em tipagem e versionamento. A

serialização de uma execução pode ser implementada chamando uma primitiva que converte uma computação em um valor (*thunkify*) e depois serializando o resultado. Entretanto, a thunkificação é uma operação atômica. A intervenção manual em Acute consiste em dar dicas ao compilador. Por exemplo, podem ser inseridas marcas para sinalizar os limites da captura. Também é disponibilizada uma linguagem para especificar restrições de versionamento, entre outros.

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

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

6.1 Experiências da implementação em Lua

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.3 Conclusão

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

A migração heterogênea de computações é um problema difícil. Além das dificuldades inerentes ao problema em si, estão as decorrentes da falta de suporte das linguagens de programação atuais, que contribuem em boa parte à complexidade e aos problemas de desempenho relatados [MRS08]. As linguagens de programação

deveriam oferecer suporte para as operações de captura e restauração de computações de forma a permitir a implementação de diferentes aplicações que precisam da capacidade de manipular computações, como é o caso da migração e persistência de execuções. Isto facilitaria o trabalho dos programadores, assim como diminuiria as situações em que esta restauração não pode ser realizada, aumentando a portabilidade. O nível da linguagem é o nível adequado para estas operações. Da mesma forma, as políticas específicas de cada domínio de aplicação pertencem ao nível de aplicação e estão fora do escopo do projeto da linguagem. Assim se elimina a necessidade de diferentes notações para políticas diferentes.

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

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

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

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

Acreditamos que a capacidade de reificar e instalar computações é uma necessidade das linguagens atuais que pretendem oferecer suporte para aplicações de migração e persistência de computações de uma forma flexível. Boa parte das dificuldades enumeradas em [MRS08] relacionadas à migração heterogênea

de computações podem ser resolvidas através destas funcionalidades, facilitando a implementação destas técnicas e permitindo assim que seu potencial seja redescoberto.