LegoShell: Linguagem de Computações[1]

Rogério Drummond

DCC - UNICAMP
CP 6065
13.081 Campinas, SP, Brasil

Resumo

A LegoShell é uma linguagem de especificação de comandos complexos chamados computações. Ela estende o conceito de pipe do Unix, que é restrito a conexões lineares de fluxo de dados, para grafos bidimensionais. Além de conectores pipe, a LegoShell suporta conectores com semântica de caixa-de-correspondência e conectores com semântica de disseminação ("broadcast"). Computações complexas podem ser abstraídas em programas e usadas em outras computações como qualquer outro programa. Ela ainda preserva as características polimórficas da linguagem de programação possibilitando a especificação de computação polimórfica.

Abstract

LegoShell is a language for the specification of complex comands, called computations. The Unix pipe concept is extended from the linear data flow conections to bidimensional graphs. Besides the pipe conector, the LegoShell suports mail-box and broad cast conectors. Complex computations can be abstracted in to programs and used as such in any computations. The LegoShell also preserves the polimorphic properties of the programming language allowing the specification of poliphormic computations.

1 Introdução

As linguagens de comando são responsáveis por tornar acessíveis ao usuário as facilidades suportadas pelo sistema operacional. Quanto mais hergonômica a interface oferecida ao usuário melhor a linguagem de comandos. Para atingir este objetivo são criadas várias abstrações de alto nível que escondem os detalhes de uma interação direta com o sistema operacional.

[1]Este trabalho foi parcialmente financiado pelo CNPq, FAPESP e SID-Informática através do projeto ESTRA.

As linguagens de comando do Unix [Rit74] atingem estes objetivos de forma simples, sintética e elegante. O Unix, bem como, os computadores evoluíram substancialmente nos últimos anos. As "workstation" são o padrão de hardware atual oferecendo alta capacidade gráfica, rede local, vasta memória e alto desempenho. Os sistemas operacionais em geral e o Unix em particular, incluem facilidades de distribuição do sistema de arquivos e da execução (TCP/IP [Dod80], NFS [San85], RPC [Bir84], Sprite [Hil86], RFS [Rif86], Andrew [Mor86]). O gerenciamento de janelas distribuídas também sofreram grandes avanços com o X-Windows [Sch86], News [Sun86] e outros.

As linguagens de comando (interface sistema operacional-usuário) tem que evoluir para tirar proveito destas novas facilidades. As primeiras inovações foram idealizadas na Xerox, por Alan Kay [Kay77] e implantadas no Finder do Apple Macintosh, que introduziu uma interface gráfica ao sistema de arquivos.

Estas idéias foram estendidas com algum suporte para integração de ferramentas [Ing88, Bea89]. Os mecanismos de comunicação entre processos (streams, sockets e RPC) tem sido usados para deixar transparente a distribuição dos objetos de uma computação. Esta comunicação ainda se restringe a conexões lineares (um para um) herdadas da proposta original do pipe do Unix.

A LegoShell inova ao introduzir conectores do tipo muitos-para-muitos a nível da interface com o usuário. Como estas conexões não são diretamente suportadas pelo núcleo do Unix, sua implementação depende da elaboração de um sistema de execução que simula novas chamadas ao sistema.

2 Estrutura do A_HAND

A LegoShell é parte do projeto A_HAND (Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados). O A_HAND visa ser um ambiente adequado ao desenvolvimento de sistemas de software grandes e complexos. Ele é composto de um conjunto de ferramentas integradas que cobrem o espectro de programação (programming in the large, programming in the small e programming in the many). O A_HAND provê gerenciamento de versões, manutenção de programas, esquemas de abstração, catalogação e recuperação de objetos de software e editores sensíveis a sintaxe/estrutura dos programas. O sistema suporta desenvolvimento concorrente e distribuído e os programas gerados pelo ambiente são potencialmente distribuídos. O conjunto de requisitos que o A_HAND pretende satisfazer estão descritos em [Dru87].

A LegoShell aqui descrita, é uma das três linguagens do A_HAND. Ela se situa entre a linguagem de programação Cm e a linguagem de comandos propriamente dita (CO^2Shell). O presente trabalho se concentra em descrever os aspectos funcionais da linguagem. Ele não se estende em discussões sobre a escolha da sua semântica (veja [Sar88]) nem sobre o seu editor e ferramentas periféricas como controle de versões e manutenção de programas (veja [Vic89]). Estes aspectos, essenciais ao A_HAND, serão superficialmente mencionados se auxiliarem a por em perspectiva a LegoShell.

O Cm é uma derivação da linguagem C que é modular, suporta tipos abstratos de dados, herança mútipla, polimorfismo paramétrico e permite verificação rigorosa de tipos em tempo de compilação. Ele não suporta ligação dinâmica de tipos e consequentemente não precisa de garbage collection. As classes em Cm correspondem a construtores de tipos que podem ser utilizados na especificação de tipos em Cm, LegoShell ou *CO²Shell*. Suas características indicam que será tão eficiente quanto a linguagem C. Seu objetivo principal é de possibilitar altos níveis de reutilização (via herança e polimorfismo) sem perder a verificação rigorosa de tipos.

A LegoShell será a linguagem onde programas (classes em Cm) são acoplados para especificar uma computação. Ela é uma extensão do conceito de pipe do Unix. Enquanto o pipe permite somente a interligação linear entre programas, a LegoShell possibilita interligações bidimensionais. A LegoShell ainda se mantém dentro do paradigma de "streams" que acompanha com sucesso e sucessivos melhoramentos o sistema Unix desde a sua concepção. Pode-se dizer que o uso da palavra Shell no nome da LegoShell é um abuso de linguagem, uma vez que ela não possibilita a especificação de comandos sequenciais e fluxo de controle (for, if etc.).

O último nível das liguagens corresponde a uma linguagem de comandos orientada para objetos chamada *CO²Shell*. Sua especificação ainda não está terminada. Em poucas palavras ela segue a linha das linguagens de comando do Unix. É textual, interpretada e suporta estruturas de controle e variáveis. Em adição ela suporta ligação dinâmica de tipos que podem ser qualquer tipo abstrato derivado de classes em Cm, computações ou "conectores" da LegoShell. A *CO²Shell* servirá a dois propósitos no A_HAND: linguagem de comandos e linguagem de prototipagem.

3 Objetos Básicos

A LegoShell suporta um conjunto de objetos que podem ser interligados por meio de conectores. Os objetos são arquivos, programas e dispositivos periféricos. A Figura 1 mostra o subconjunto dos objetos e conectores discutido neste trabalho.

Os arquivos (Fig. 1.a) tem portas de entrada e saída de acordo com a sua permissão de leitura e escrita. Estas portas são portas virtuais. Elas tornam-se portas reais quando são ligadas a um conector. Se um arquivo pode ser lido, ele pode ser lido múltiplas vezes. Assim, sempre que uma porta de saída é utilizada, uma nova porta virtual (de saída) é criada. Por outro lado, os arquivos podem ser escritos por um único processo a cada vez e, consequentemente cada arquivo tem no máximo uma porta de entrada. O ambiente garante que cada arquivo esteja aberto no máximo uma vez para escrita. Ele ainda é responsável por mostrar (de forma gráfica) o estado de um arquivo (i.e. aberto ou fechado para leitura e/ou escrita).

Todos os outros objetos apresentam (graficamente) somente as portas reais. Alguns deles podem ter portas virtuais mas que não são mostradas graficamente.

Os programas (Fig. 1.b) tem uma representação parecida com os arquivos. Suas portas, no entanto, são sempre portas reais e correspondem às portas declaradas na classe em Cm que implementa o programa. Estas portas são diferenciadas pelos seus nomes dados no programa Cm. O botão no canto direito superior possibilita que o programa seja ativado, suspenso ou desativado. O botão de nome **par** quando acionado possibilita a especificação dos parâmetros da classe e argumentos de execução.

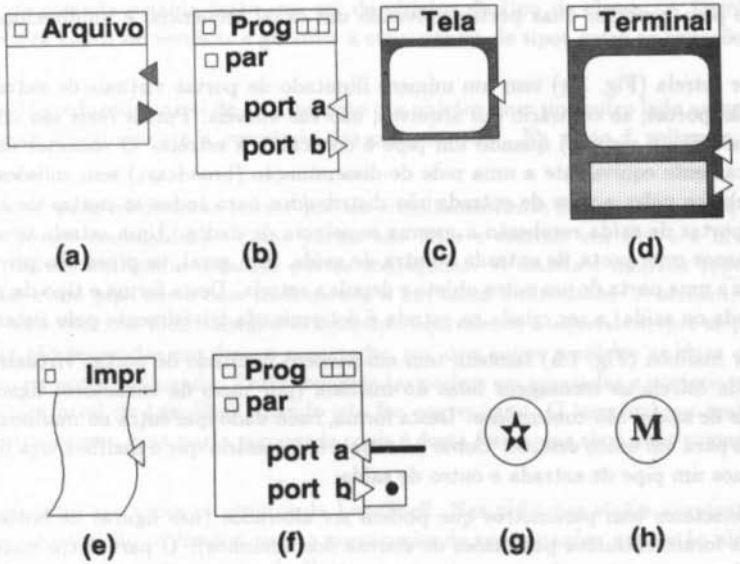

Figura 1

A Figura 1.c mostra um vídeo. Como dispositivo de saída, ele tem uma porta de entrada. A cada vídeo estará associado a uma janela quando a computação a que ele pertence for ativada. O nome do vídeo, que deve ser fornecido pelo usuário, será o identificador da janela associada.

O terminal (Fig. 1.d) é associação de um vídeo com um teclado. Como em todos os sistemas desktop, quando uma janela está ativa o teclado físico da máquina fica associado ao teclado virtual do dispositivo terminal da LegoShell. O vídeo e o terminal podem ser reconfigurados alterando-se os seus parâmetros. Um terminal pode ecoar ou não a sua entrada, os caracteres de controle podem ter interpretações locais etc. Os parâmetros do vídeo e terminal são alterados por meio do botão **par** destes periféricos. Estes botões não foram incluídos nas figuras por uma questão de claresa.

Outros dispositivos como impressora, plotter, scanner etc. tem representação icônica e podem ter seus parâmetros alterados. A Figura 1.e ilustra uma impressora. É importante notar que os periféricos da LegoShell são na realidade "drivers" inteligentes dos dispositivos periféricos físicos. Os ícones identificam a classe do periférico e o seu nome identifica um periférico específico dentro daquela classe.

São três os tipos de conectores da LegoShell. O conector mais simples é a tubulação ou pipe (Fig. 1.f porta a). Sua semântica é semelhante a de um "stream" do Unix. O pipe é utilizado para conectar duas portas provendo um canal sequencial e unidirecional de dados.

O conector estrela (Fig. 1.g) tem um número ilimitado de portas virtuais de entrada e saída. Estas portas, ao contrário dos arquivos, não são visíveis. Portas reais são criadas (e neste caso ficam visíveis) quando um pipe é conectado à estrela. O conector estrela é semanticamente equivalente a uma rede de disseminação (broadcast) sem colisões. Os dados recebidos pelas portas de entrada são distribuídos para todas as portas de saída. Todas as portas de saída receberão a mesma sequência de dados. Uma estrela tem que ter pelo menos uma porta de entrada e outra de saída. Em geral, os pipes são primeiro conectados a uma porta de um outro objeto e depois a estrela. Desta forma o tipo da porta real (entrada ou saída) a ser criada na estrela é determinada trivialmente pelo sistema.

O conector mailbox (Fig. 1.h) também tem um número ilimitado de portas virtuais. Ao contrário da estrela as mensagens lidas do mailbox (por meio de conectores ligados a suas portas de saída) são consumidas. Desta forma, cada dado que entra no mailbox será distribuído para um único destino. Como na estrela é necessário que o mailbox seja ligado a pelo menos um pipe de entrada e outro de saída.

Os três conectores tem parâmetros que podem ser alterados (nas figuras os botões de parâmetros foram excluídos por razões de clareza dos desenhos). O parâmetro mais importante destes conectores é o tamanho do seu buffer interno. Conectores sem buffer (tamanho igual a zero) produzirão sistemas síncronos. Nestes sistemas o leitor e o escritor de um conector executarão estas operações ao mesmo tempo. O tamanho dos buffers poderá ser tão grande quanto um segmento da máquina (veja a seção 9 para maiores detalhes). A representação gráfica dos conectores síncronos e assíncronos (com buffers nulos e não nulos respectivamente) é distinta realçando suas propriedades. Nas figuras deste trabalho usamos uma única representação.

O conector pipe é funcionalmente equivalente a estrelas e mailboxes que estejam ligadas e uma (e só uma) entrada e uma (e só uma) saída. Mantivemos o pipe como um conector em separado para simplificar o desenho de uma computação.

Um buraco negro tem uma porta virtual que pode ser instanciada em porta de entrada ou de saída conforme a porta em que ele for conectado. Ele serve para "fechar" uma porta real cujo acesso deva ser negligenciado. Um buraco negro pode ser considerado um dispositivo como o "/dev/null" do sistema Unix. Com ele é possível criar computações completas (vide seção 4).

As portas podem ainda ser caracterizadas com respeito a quantidade de dados que são

enviados a cada operação de entrada e saída. Elas podem transferir dados byte a byte (como no pipe do Unix), blocos de tamanho fixo (512 ou 1024 bytes dependente da implementação) e blocos de tamanho variável.

As portas são instâncias da classe pré-definida **port** que tem parâmetros tipo da porta (entrada/saída) e stream type (byte, bloco, ...). Uma outra classe pré-definida será disponível em Cm que possibilita a declaração de portas tipadas. Esta classe, será uma subclasse da classe **port**. As portas derivadas desta classe terão um tipo associado e as operações de entrada e saída terão que ser de objetos do tipo de classe. A LegoShell, neste caso será capaz de verificar e garantir a consistência de tipos entre as conexões das portas.

Portas tipadas reduzem o nível de acoplamento dos objetos, mas por outro lado aumentam sensivelmente a segurança e a corretude das computações. Na seção 7 voltamos neste assunto.

Na realidade é possível declarar portas que são simultaneamente de entrada e saída (chamadas de portas conjugadas). Estas portas são lidas e escritas em Cm e a nível da LegoShell devem ser ligadas a outras portas conjugadas. A estrela e mailbox suportam estas portas e um pipe neste caso corresponde a um canal bidirecional. A semântica associada a uma conexão bidirecional é exatamente equivalente a aquela em que as portas conjugadas são separadas em duas e conectadas por dois pipes paralelos as duas outras portas de uma porta conjugada. Portas conjugadas podem ser separadas e posteriormente reagrupadas a nível da LegoShell quando isto for conveniente. O terminal, na realidade será apresentado com uma porta conjugada, pois é desta forma que ele é mais comumente utilizado.

Em essência estes são todos os objetos da LegoShell. Nas próximas seções apresentamos como estes objetos são utilizados para a construção de computações que estão além do alcance do limitado mundo do pipe do Unix.

4 Computações

Uma computação é um programa em LegoShell. O A_HAND proverá um editor específico para a manipulação de computações que já está em desenvolvimento [Arias89]. Apresentamos a seguir alguns exemplos simples de computações somente para ilustrar alguns aspectos da linguagem. Apresentaremos estas computações já prontas, sem discorrer sobre o processo de sua construção.

A primeira computação (Fig. 2) pressupõe a existência de um programa Sort que ordena os dados de sua porta de entrada produzindo-as na sua porta de saída. O programa Merge produz em sua porta de saída a intercalação de sequências ordenadas oriundas de suas duas portas de entrada. Cada programa e dispositivo de uma computação executa como um processo independente.

O conector mailbox é utilizado para distribuir o conteúdo do arquivo de entrada para as

duas instâncias do programa Sort. Cada uma receberá um conjunto de dados distinto. Elas estão competindo pelo mesmo recurso (dados do arquivo de entrada) e o mailbox arbitra a competição. O mailbox lê do arquivo sob demanda dos programas Sort, enviando cada dado ao Sort que o requisitou a mais tempo. Os programas são objetos ativos e consequentemente suas portas de entrada e saída também são ativas. Ou seja, elas fazem requisições aos conectores associados. As portas de saída de arquivos são passivas, ou seja, elas esperam por requisições para produzir um dado. Se o buffer do conector mailbox for diferente de zero o mailbox fará leituras no arquivo até encher o seu buffer.

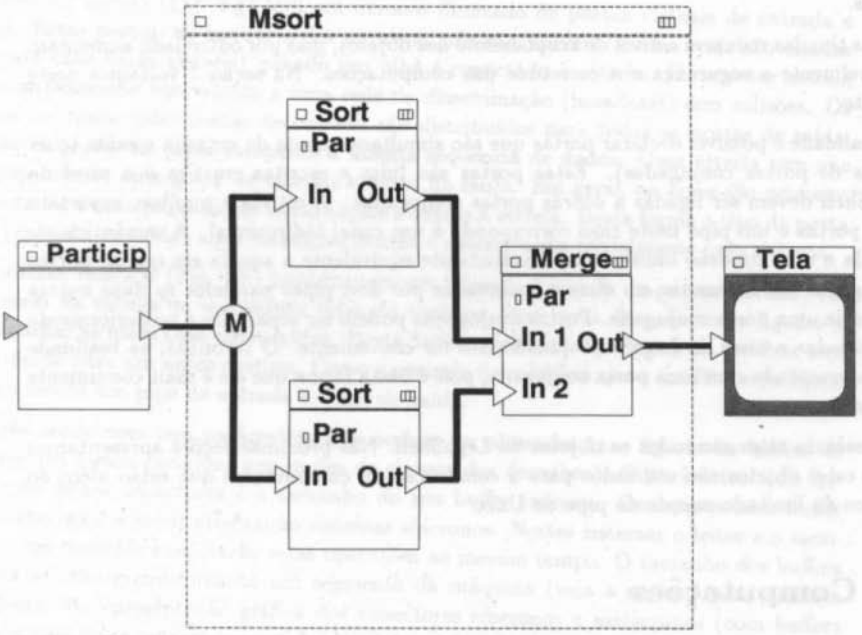

figura 2

Faremos uma pequena digressão para explicar as restrições nos buffers dos conectores pipe. Os conectores tipo pipe que interligam dois outros conectores, ou um conector e um arquivo ou programa (como os pipes de entrada do Sort) tem sempre buffer de tamanho zero não sendo possível alterá-los. Um conector pipe só pode ter buffer maior do que zero quando ele interliga dois extremos ativos (i.e. programas, na maioria dos casos). Os conectores com buffers maiores do que zero tem uma representação gráfica diferente dos de tamanho zero.

De volta ao exemplo o programa Merge requisitará dados de suas duas portas e (ativamente) os produzirá em sua porta de saída. Os conectores que interligam o Sort ao

Merge podem ter buffer maior do que zero e neste caso servem para equilibrar variações de velocidade dos processos.

Uma computação é dita completa se todas as suas portas reais estão devidamente conectadas. Somente computações completas podem ser executadas. O buraco negro deve ser utilizado para se alcançar uma computação completa com portas que devam ser ignoradas.

5 Distribuição

A LegoShell foi idealizada com intuito de aproveitar dos recursos da geração atual de sistemas computacionais. Sendo o ambiente atual, aquele composto de estações de trabalho interligadas em rede, é natural que se aproveite das facilidades de distribuição possível nestes sistemas.

Como a LegoShell executará sobre o Unix, os arquivos em uma computação já são, de forma transparente, distribuídos na rede (pelo NSF, RFS ou Sprite). Possibilitamos ainda que cada um dos programas possa executar em uma máquina diferente na rede local. Para tal usaremos os mecanismos do próprio Unix (streams ou sockets). O exemplo acima poderia ter cada um dos seus componentes (as duas instâncias do Sort, o Merge e os arquivos de entrada e saída) residindo em uma estação diferente. A estação onde um programa vai executar é um parâmetro da computação associado à instância do programa. Ele não é um parâmetro do programa.

6 Abstrações

Uma das características mais importantes do A_HAND é a regularidade entre as suas várias linguagens e conceitos fundamentais, que são recorrentes nas diversas ferramentas do ambiente. A LegoShell provê o mesmo esquema de abstração que a linguagem Cm (a menos de herança de tipos de dados que não cabe neste contexto). Assim é possível abstrair uma computação em um programa. A computação descrita acima incluindo os mailbox, os dois Sort e o Merge pode ser abstraida como um programa "MSort". A Figura 2 mostra a computação a ser abstraída dentro de uma caixa pontilhada. O resultado é um ícone com duas portas uma de entrada e outra de saída que correspondem aos pipes que foram cortados (aqueles que ficam entre o que foi selecionado como parte da abstração e o que ficou fora desta seleção). A Figura 3 exemplifica o uso do MSort. Sua composição interna é apresentada em tamanho reduzido a título de ilustração.

Uma abstração pode ser realizada sobre uma computação incompleta, e neste caso as portas (reais) em aberto devem ser conectadas à borda da abstração onde uma porta é automaticamente criada. Na realidade é criada uma porta lógica que é implementada pela porta real do objeto dentro da abstração.

O importante é que fica transparente para o usuário se uma ícone de programa está im-

plementada em Cm ou se está implementada como uma computação em LegoShell. Ao se requisitar uma edição de um programa o ambiente ativa o editor apropriado (LegoShell ou Cm). Como é comum no Unix, o A_HAND possibilita a "programação" em sua linguagem de comandos. As aspas em programação se justificam uma vez que a LegoShell é bastante restrita em termos de computação. Como brevemente descrito no início deste trabalho, a maior parte da programação a nível de linguagem de comandos ficará a cargo da *CO²Shell*.

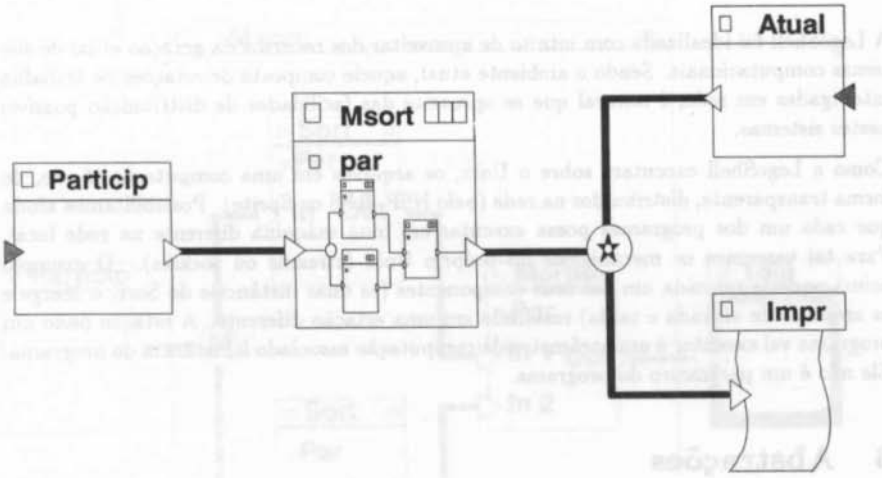

Figura 3

A semelhança da LegoShell com o Cm não se limita a sua capacidade de abstração como mostramos na próxima seção.

7 Polimorfismo

As classes Cm, definidas pelo usuário, são na realidade construtores de tipos. É necessário neste ponto discriminar os parâmetros de classe e parâmetros de execução (Fig. 4). Os parâmetros de classe são aqueles que selecionam um tipo dentre os inúmeros tipos de uma classe. A classe Sort, por exemplo, só se torna um tipo quando seus parâmetros de classe (**int** N_buf e **type** T) são especificados. Após a especificação do tipo a LegoShell está pronta para instanciar um objeto deste tipo. Os parâmetros de execução são aqueles que especificam alternativas de execução e são parâmetros da instância. O objeto (do tipo já definido) só é realmente instanciado quando ele é ativado para execução. O objeto criado

é um objeto anônimo uma vez que não está associado a uma variável como normalmente estaria em um programa. A LegoShell da um nome a instância para diferenciá-la das demais instâncias do mesmo tipo.

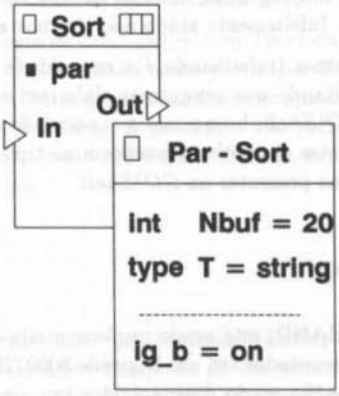

Figura 4

Da mesma forma que cada programa (classe Cm) pode ser parametrizada e alguns dos seus parâmetros podem ser tipos, uma abstração de uma computação pode ser igualmente parametrizada. É possível fazer com que o MSort tenha parâmetros. O MSort pode manter o polimorfismo das classes Cm declarando parâmetros **int** N_buf e **type** T e passando-os como os valores dos parâmetros atuais do Merge e do Sort.

O MSort pode ser utilizado em outras computações que podem ser ainda abstraidas como programas, e assim indefinidamente. A LegoShell oferece a possibilidade de níveis ilimitados de abstrações preservando as propriedades polimórficas do Cm. Como as portas podem ter seus tipos definidos por parâmetros da classe ou de uma computação abstraida, elas preservam suas propriedades polimórficas. Neste caso a corretude das computações é garantida e o nível de acoplamento não é prejudicado uma vez que os tipos podem facilmente ser alterados por meio dos parâmetros da computação.

8 Limitações

O nível de concorrência suportado pela LegoShell (e pela *CO²Shell*) é muito grosso para uma série de aplicações. No estágio atual, o Cm não suporta concorrência. Assim o ambiente sofre da falta de um mecanismo de concorrência com granularidade mais fina. Os ensaios neste sentido na LegoShell foram todos frustados. Apesar de viável, uma

simulação de chamada remota de procedimento, requer uma dependência muito grande entre os programas em Cm e a configuração em LegoShell. O resultado é que a LegoShell não é adequada a este tipo de concorrência que será futuramente incorporado no Cm [Cal89]. Outra limitação da LegoShell é a falta de mecanismos de fluxo de controle. Existem propostas recentes de linguagens de fluxo de dados com mecanismos semi-gráficos para fluxo de controle [Pu89]. Infelizmente ainda não tivemos acesso a esta literatura.

Outro aspecto que ainda estamos trabalhando é a capacidade de criar computações de forma não interativa, possibilitando que programas elaborem computações. Estas facilidades estarão presentes na *CO²Shell*, bem como a capacidade de alterar dinamicamente uma computação. Note que estas operações pertencem ao tipo abstrato de dados "computação" que será um dos tipos presentes na *CO²Shell*.

9 Implementação

A LegoShell, como todo o A_HAND, está sendo implementada em ambiente Unix. Parte dos trabalhos iniciais foram executados em um Digirede 8000/II com um Unix-like. Hoje a maior parte dos trabalhos estão sendo desenvolvidos em uma estação Intergraph 220 com Unix System V.3. O BSD Unix seria muito mais adequado. Esperamos terminar a implementação em cima do Open-look, ou ao menos seguindo suas diretrizes de interface.

Definimos um tipo abstrato de dados chamados *computação* que contém as operações para a manipulação dos objetos da LegoShell (conectar, alterar parâmetros, executar, ...). A implementação da LegoShell consiste de duas partes: seu editor e o sistema de execução.

O editor é um front-end interativo para o tipo *computação*. Ele é responsável pela verificação de consistência de uma computação, bem como pela geração de código. Ele está sendo desenvolvido na Interpro usando X-windows 11.3, e será facilmente portável para outros sistemas Unix com X-windows.

No topo das chamadas ao sistema do Unix estamos implementando uma pequena camada de funções que compõem o sistema de execução da LegoShell e do Cm. As computações e as classes Cm serão traduzidas para as funções desta camada. As linguagens do A_HAND tirarão todo o proveito possível do sistema Unix. Citamos alguns detalhes abaixo a título de ilustração.

Os conectores terão seus buffers implementados em segmentos individuais e seu tamanho só é limitado pelo espaço de endereçamento virtual da máquina por segmento. O sistema de gerenciamento de memória será responsável pela paginação destes segmentos. Os conectores mailbox e estrela tem implementação trivial. Um mailbox é um simples produtor-consumidor. Um conector estrela será implementado por um único buffer na estrela, evitando a coexistência de múltiplas cópias. A sua implementação é trivial no lado dos produtores e nos lado dos consumidores cada um terá um ponto de retirada independente. Basta controlar a posição do ponto de retirada que esteja mais atrasada. A porção do buffer que deve ser preservada vai da posição mais atrasada à posição da

entrada mais recente no buffer.

As conexões remotas, que interligam objetos distribuídos é realizada por meio dos streams Este mecanismo é altamente eficaz em manter a transparência da distribuição. A implementação do mailbox distribuído pode apresentar algum problema devido a bufferização e latência inerente aos mecanismos de comunicação.

Um *protótipo parcial* do sistema de execução está em funcionamento [Van89]. Uma versão do Cm sem parâmetros foi implementada neste protótipo. Ele inclui a definição e implementação de uma máquina RISC segmentada, de um conjunto de macros para a tradução do Cm e da classe pré-definida **port**. O protótipo não toca nos problemas de sistemas distribuídos. Ele foi desenvolvido em MS-DOS a título de experiência. A versão definitiva será exclusivamente para Unix.

10 Exemplo

A Figura 5 mostra uma computação que controla o **Estoq** de um supermercado. Cada **Reg** registradora do supermercado é composta de um terminal, um leitor ótico de código de barras, um programa **Reg** e uma impressora de notas fiscais de compra (não incluída na figura).

Os programas **Reg** enviam os dados de cada ítem vendido para o conector estrela que os distribui para os programas **Estoq** e **Finan**. Os gerentes de **Estoq** e **Finan** tem um terminal onde podem ver a situação corrente do supermercado e alterar o funcionamento dos programas **Estoq** e **Finan** respectivamente. O programa **Estoq** gera pedidos de compra para o programa **Finan** que efetua as ordens de compra, listadas em uma impressora. O gerente de compras recebe os pedidos pela impressora e informa os recebimentos por meio de terminal Gcomp. O **Estoq** e a situação financeira corrente do supermercado estão sempre à disposição dos respectivos gerentes.

Os programas **Reg**, **Estoq** e **Finan** podem ser computações LegoShell ou classes em Cm. Cada instância do programa **Reg** terá a identificação da **Reg** registradora que ele controla especificada por meio de parâmetros.

Todas as conexões entre objetos são conexões lógicas. Se os componentes do sistema estão interligados fisicamente em rede local, então nenhuma outra linha física tem que ser implantada. Os terminais podem corresponder a terminais reais ou a janelas em um terminal gráfico. Finalmente esta computação pode ser executada em uma ou mais máquinas.

O mais importante é que todas estas variantes ficam transparentes para os programas Caixa, Estoque e Finanças.

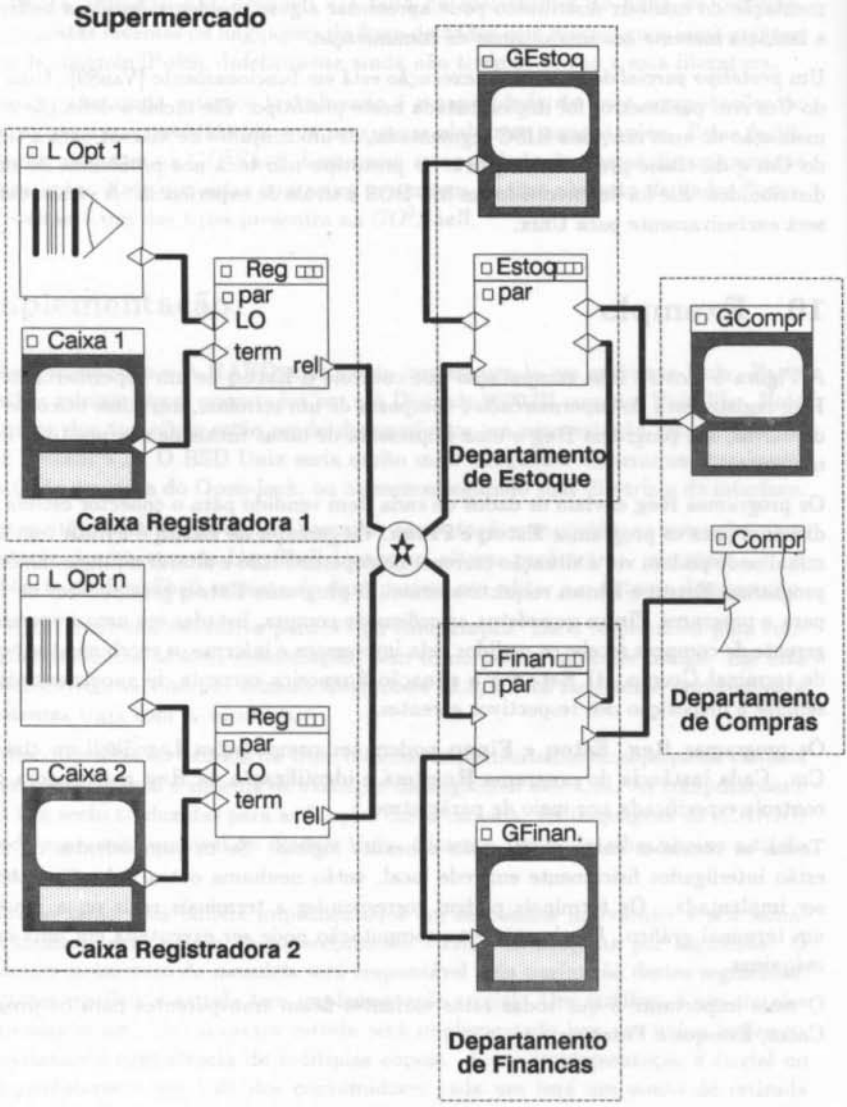

Figura 5

11 Conclusão

Apresentamos uma linguagem de especificação de computações que é bastante simples e com alto poder de expressão quando associada com uma linguagem de programação polimórfica. Note que a LegoShell tem o mérito de preservar e manter o polimorfismo da linguagem de programação. As qualidades de uma e outra ficam realçadas pelo perfeito entrosamento das duas linguagens. Ambas foram definidas concorrentemente para um fim comum. Muitas das características de uma linguagem são decorrência da outra.

As computações possíveis na LegoShell são inerentemente bidimensionais e uma representação gráfica é essencial. Com os baixos preços das estações de trabalho com monitores de alta resolução, linguagens de fluxo de dados como a LegoShell com representações gráficas devem aparecer com bastante frequência nos próximos anos. O sistema Fabrik da Apple Computers [Ing88] é um exemplo desta tendência.

Várias das metodologias de desenvolvimento atuais são baseadas em fluxo de dados. A LegoShell (com portas tipadas) associada ao Cm possibilita um mapeamento quase direto da especificação nestas metodologias para código executável. Este não é o objetivo da LegoShell e este mapeamento não é necessariamente completo.

Agradecimentos

A Lego Shell foi idealizada em 1986 juntamente com o Cm. Ainda em 1986 Rodrigo Figueiredo (na época no núcleo da SID-Informática) sugeriu a elaboração de um ambiente de desenvolvimento que vestisse as duas linguagens e acelerasse o processo de desenvolvimento de software. Muitos foram os que colaboraram no refinamento das idéias iniciais e nos esforços de implementação.

Hernan P. Arias, Alícia di Sarno e Carlos Furuti participaram das discussões visando a especificação da LegoShell. Recentemente, Calton Pu e Roger Hoover sugeriram extensões que não estão descritas no presente trabalho. Ricardo Anido, Hernan P. Arias, Jeff Henslin e Angela Viel colaboraram na elaboração deste texto e das figuras.

Referências

[Ari89] Arias, Hernan P. *Editor para a LegoShell.* DCC, Unicamp (em preparação).

[Bea89] Beaudouin-Jafon, M. *User Interface Support for the Integration of Software Tools: an Iconic Model of Interaction.* ACM SESPDE Proceedings, SIGPLAN Notices, 24, 2 (novembro 1989), pp. 143-152.

[Bir84] Birrell, A. e Nelson, B. *Implementing remot procedure calls.* TOCS, 2, 1, (fevereiro 1984), pp. 39-59.

[Cal89] Calsavarra, A. *Cm distribuído.* DCC, Unicamp (tese de mestrado em preparação).

[Dod80] *Dod Standard - Transmission Control Protocol*, DARPA, Arlington (janeiro 1980).

[Dru87] Drummond, R. e Liesenberg, H. *Requisitos para um Ambiente de Desenvolvimento de* PROGRAMAS. I Encontro IBM de Ciência e Tecnologia da Informática. Rio de Janeiro, RJ, (novembro 1987).

[Dru88] Drummond, R e Silva, F. Q. B. *Manual de Referência - Linguagem Cm*, DCC, Unicamp, (março 1988).

[Gif88] Gifford, D. e Glasser, N. *Remote Pipers and Procedures for Efficient Distributed Communication.* ACM TOCS 6, 3 (agosto 1988), pp. 258-283.

[Hil86] Hill, M., et all. *Design decisions in SPUR.* IEEE Computers, 19, 11 (novembro 1986), pp. 8-22.

[Ing88] Ingalls, D., et all. *Fabrik: A Visual Programming Environment.* ACM OOPSLA 88 Conference Proceeding, SIGPLAN Notices, 23 (novembro 1988), pp. 176-190.

[Kay77] Kay, A. e Goldberg, A. *Personal Dynamic Media*, Computer (março 1977).

[Ker78] Kernighan, B. W. and Ritchie D. M. *The C Programming Language.* Prentice-Hall, Inc., USA (1978).

[Mor86] Morris, J., et all. *Andrew: A distributed personal computing environment.* CACM, 29, 3 (março 1986), pp. 184-201.

[Pu89] Pu, C. *Comunicação Pessoal*, DCC, Unicamp (julho 1989).

[Rif86] Rifkin, A. et all. *RFS archetectural overview.* Anais da USENIX 1986, Summer Conference, USENIX Association, Berkeley, Ca (1986), pp. 248-259.

[Rit74] Ritchie, D. e Thompson, K. *The UNIX time-sharing system.* CACM 17, 7 (julho 1974), pp. 105-117.

[San85] Sandberg, R. et all. *Design and implementation of the Sun network filesystems.* Anais do USENIX 1985, Summer Conference, USENIX Association, Berkeley, Ca (1985), pp. 119-130.

[Sar88] di Sarno, A. e Drummond, R. *LegoShell: Linguagem de Configuração de Programas.* DCC, Unicamp (dezembro 1988).

[Sch86] Scheifler, R. e Gettys, J. *The X Window System*, ACM Transaction on Graphics, 5, 2 (abril 1986), pp. 79-109.

[Sil88] da Silva, F.Q.B., Liesenberg, H. e Drummond, R. *Programação em Cm.* DCC, Unicamp (março 1988).

[Sun86] *News Preliminary Technical Overview.* Sun MicroSystems Inc. (outubro 1986).

[Van89] Vanine M. *Sistemas de Execução para Ambientes de Desenvolvimento Distribuído.* DCC, Unicamp (em preparação).

[Vic89] Victorelli, E., Magalhães, G. e Drummond, R. *Mecanismo de Gerenciamento de Versões e Configurações do A_HAND.* Anais do III Simpósio Brasileiro de Engenharia de Software, Recife, PE (outubro 1989).