Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Alex Abate Biral

SWNE
Uma Linguagem Baseada em Predicados

São Paulo
13 de fevereiro de 2016

SWNE
Uma Linguagem Baseada em Predicados

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Marco Dimas Gubitoso

São Paulo
13 de fevereiro de 2016

Resumo

Este documento descreve em termos básicos o SWNE, uma linguagem de computador desenvolvida ao longo do ano de 2015. O objetivo do SWNE é testar a ideia de despacho baseado em predicados. Apesar de não ser nova, esse tipo de indireção apresenta certas dificuldades para ser implementada e usada. Talvez por isso não tenha sido, ainda, mais amplamente usada.

Esta monografia introduz alguns conceitos básicos de design de linguagens de computação, os retomando depois para explicar como o SWNE lida com estes aspectos. O nosso foco aqui é transmitir ao leitor as razões pelas quais criamos o SWNE. Porque achamos que seria interessante criar ainda outra linguagem computacional e que benefícios ela poderá vir a trazer.

Palavras-chave: Despacho Sobre Predicados, Linguagens Computacionais.

Sumário

1 Introdução **1**

1.1 Linguagens Computacionais . . . 1

1.1.1 Indireção . . . 1

1.1.2 Assembly . . . 2

1.1.3 Sub-Rotinas e Funções . . . 3

1.1.4 Despacho Dinâmico . . . 3

1.2 Motivações e Objetivos . . . 4

1.2.1 Prós e Contras . . . 5

1.2.2 Objetivos . . . 6

1.2.3 Estado da Arte . . . 6

2 Conceitos Básicos **7**

2.1 Compilação e Interpretação . . . 7

2.1.1 Açúcar Sintático . . . 8

2.2 Valores e Tipos . . . 8

2.3 Variáveis . . . 9

2.3.1 Valores Mutantes . . . 9

2.3.2 Containers . . . 10

2.4 Contexto . . . 10

2.5 Dinamicidade e Estaticidade . . . 11

2.6 Clausuras . . . 11

2.7 Recursão . . . 12

2.8 Encapsulamento . . . 12

2.9 Memória . . . 13

3 Implementação **15**

3.1 Árvores Semânticas . . . 15

3.2 Ambiente . . . 16

3.3 Variáveis . . . 16

3.4 Predicados . . . 16

3.5 Despacho por Predicados . . . 17

3.5.1 Sistema Lógico . . . 17

3.6 Gramática 18
3.6.1 Declare 18
3.6.2 Predicate 18
3.6.3 Relation 19
3.6.4 Set 19
3.6.5 Lambda 19
3.6.6 Function 19
3.6.7 Call 20

4 Resultados **21**
4.1 Exemplos 21
4.1.1 Fibonacci 21
4.1.2 Asteroides 22
4.2 Melhorias 25
4.2.1 Verificação de Predicados 25
4.2.2 Memória 25
4.2.3 Predicados Estáticos 25
4.2.4 Valores Intrínsecos 25
4.2.5 Compilação 26
4.2.6 Assegurar 26
4.2.7 Pacotes 26

5 Conclusões **27**
5.1 Trabalhos Futuros 27
5.1.1 Processamento de Erros em Tempo de Compilação 27
5.1.2 Despacho Pré-Clausural 28

A Parte Subjetiva **29**
A.1 Disciplinas 29
A.2 Agradecimentos 29

Referências Bibliográficas **31**

Capítulo 1

Introdução

1.1 Linguagens Computacionais

É um pouco difícil responder a pergunta sobre qual foi a primeira linguagem de computação, ou quando esta foi escrita. Isto porque o que é uma linguagem computacional não é tão bem definido. Em 1843, por exemplo, Ada Lovelace publicou junto com a tradução do artigo do matemático Luigi Menabrea, notas descrevendo como a máquina de Babbage poderia ser usada para calcular os números de Bernouli. As descrições de Ada se baseavam em uma "máquina" que não realmente existia, que fora apenas idealizada. Porém, ainda assim, a descrição da máquina definia uma linguagem válida.

Já em 1949 começaram a aparecer os computadores com instruções embutidas, e junto com estes, versões primitivas da linguagem de máquina *assembly*. E com o tempo, vários refinamentos permitiram que se construísse linguagens de altíssimo nível, que não se conformam de maneira tão direta à arquitetura do computador.

Desde então, centenas, talvez milhares, de linguagens de computação foram criadas. Existe, entre o homem e a máquina, um largo "abismo semântico". Instruções que são claras e simples para qualquer humano se tornam ambíguas e ininterpretáveis para um computador. Um programa em Assembly perfeitamente funcional pode facilmente enganar um humano, por ser escrito de maneira não intuitiva e pela sua verbosidade. Programas de computador, porém, tem de ser lidos por ambos. É para tentar fechar este abismo que tantas linguagens foram criadas.

De certa forma, estas mesmas preocupações existem na matemática. Todo programa de computador corresponde, pelo *Isomorfismo de Curry-Howard*, a uma prova matemática. E assim como existem várias maneiras de se escrever a mesma prova, existem várias maneiras de se escrever o mesmo programa. E assim como uma prova pode ser melhor entendida se escrita desta ou daquela forma, a maneira como escrevemos um programa de computador pode torná-lo mais fácil ou difícil de entender.

Neste trabalho, exporemos sobre mais uma linguagem de computador, o SWNE. Para entender por que o SWNE foi feito, porém, seria interessante primeiro mencionar o conceito de indireção.

1.1.1 Indireção

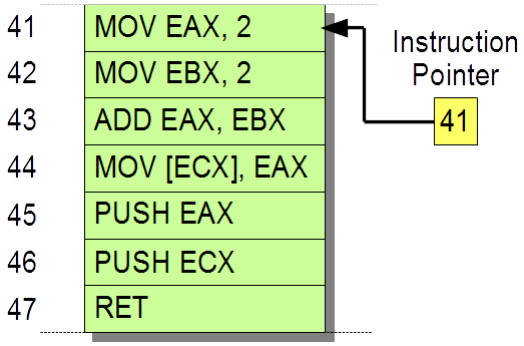
Indireção é quando um programa descreve as ações a serem tomadas não de maneira direta e sequencial, mas através de algum aspecto indefinido, que deve ser definido em outra parte do programa, ou durante a execução, ou mesmo através de alguma recursão dentro do próprio programa.

Um exemplo muito comum disso é o uso de condicionais, cláusulas "IF" e "ELSE", onde o código específico a ser executado depende se um teste feito pelo computador é positivo ou negativo. Outra indireção muito comum é o uso de "loops", como o "WHILE", onde um trecho de código é repetido até que um teste definido pelo programador seja negativo. O uso de variáveis, a leitura de um arquivo (para ser usado de qualquer forma que seja) e mesmo a espera de algum tipo de evento (como o pressionar de uma tecla) são todos exemplos de indireção.

Sem qualquer conceito de indireção, seria impossível fazer qualquer programa interativo, isto é, que interage de alguma forma com o usuário. Todo programa também teria de acabar após um número pré-determinado de ciclos computacionais. Qualquer conceito de recursão também estaria fora de alcance sem nenhuma indireção.

1.1.2 Assembly

Figura 1.1: *Instruções de Assembly.*



Em linguagens de baixo nível, como Assembly, o programa é formado por uma sequência de instruções, que são carregadas na memória do computador. Quando o programa é compilado, cada instrução é traduzida para um código chamado *opcode*. Este código associa bytes (normalmente um número específico dos mesmos) a funções de baixo nível do hardware do computador, tais como acessar uma posição de memória ou somar dois valores. Um registrador especial no processador é o *Apontador de Instrução*. Este registrador aponta para a posição da próxima instrução a ser processada, sendo aumentado por 1 (ou pelo tamanho da instrução) a cada passo.

O apontador não pode ser mudado de maneira direta, mas várias instruções, como o "JMP", permitem que ele seja afetado. É através destas instruções que quase toda indireção num programa de computador é implementada[1]. Não é preciso muito para se construir indireção aqui: a instrução "JE" faz o apontador de instruções "pular" para a posição de memória desejada apenas se dois valores testados são iguais. Com apenas esta instrução, mais algumas instruções algébricas básicas, é possível implementar qualquer indireção no computador.

Porém, apesar de ser suficiente, não é muito fácil de usar nem de ler um código que use apenas instruções de pulo para implementar toda indireção. Portanto, linguagens de alto nível implementam uma série de ferramentas para permitir que o trabalho do programador seja mais fácil.

[1] Interrupções de sistema são uma exceção. Um sistema que suporta interrupções para imediatamente a execução atual, guarda os valores dos registradores na pilha e manda o apontador de instruções apontar para um lugar pré-determinado da memória, geralmente uma sub-rotina do sistema operacional especial que lida com a situação.

1.1.3 Sub-Rotinas e Funções

Um dos tipos de indireção mais comum é o uso de sub-rotinas. Na verdade, existe uma variedade de maneiras diferentes de se implementar sub-rotinas. Chamadas por valor, chamadas por referência, métodos, entre outros. Estes variam em como exatamente os valores dos argumentos são passados ao sub-programa e como este retorna resultados. Mas seja como for, qualquer sub-rotina vai delimitar um trecho útil e reusável de código. Geralmente, este trecho vai precisar de certos parâmetros, sobre os quais ele realizará sua computação.

Sub-rotinas sempre possuem um *contexto* próprio. Isto é, elas possuem valores próprios que não podem ser alterados de fora delas. Isto permite isolar funcionalidades para diferentes funções de maneira encapsulada, sem que sub-rotinas diferentes tenham efeitos colaterais entre sí. De fato, através de sub-rotinas, muitas bibliotecas de funções foram criadas, permitindo que programadores não tivessem de começar sempre do zero, resolvendo os mesmos problemas várias vezes toda vez que começassem um novo projeto.

Figura 1.2: *Parte de uma biblioteca de funções matemáticas da linguagem C. As funções mostradas são todas funções trigonométricas para valores de ponto flutuante.*

```
445
446 /* 7.12.4 Trigonometric functions: Double in C89 */
447 extern float __cdecl sinf (float);
448 extern long double __cdecl sinl (long double);
449
450 extern float __cdecl cosf (float);
451 extern long double __cdecl cosl (long double);
452
453 extern float __cdecl tanf (float);
454 extern long double __cdecl tanl (long double);
455
456 extern float __cdecl asinf (float);
457 extern long double __cdecl asinl (long double);
458
459 extern float __cdecl acosf (float);
460 extern long double __cdecl acosl (long double);
461
462 extern float __cdecl atanf (float);
463 extern long double __cdecl atanl (long double);
464
465 extern float __cdecl atan2f (float, float);
466 extern long double __cdecl atan2l (long double, long double);
467
468 /* 7.12.5 Hyperbolic functions: Double in C89  */
469 extern float __cdecl sinhf (float);
470 #ifndef __NO_INLINE__
471 __CRT_INLINE float __cdecl sinhf (float x)
472   {return (float) sinh (x);}
```

Quando uma função é criada, um nome é associado a sub-rotina. Este pode ser usado para "chamar" a sub-rotina. Quando isso acontece, o contexto é trocado, escondendo variáveis do contexto atual mas permitindo o acesso às variáveis do novo contexto. Os argumentos da sub-rotina são disponibilizados e o código da mesma passa a ser executado pelo computador, até que a sub-rotina termine (geralmente retornando algum resultado) ou até que uma nova sub-rotina seja chamada. Quando o processo termina, o computador volta ao trecho de código que a chamou, com quaisquer resultados que esta precise.

1.1.4 Despacho Dinâmico

O despacho dinâmico é um segundo aspecto de indireção adicionado sobre o mecanismo de sub-rotinas. Quando uma sub-rotina é chamada, ao invés de "pular" imediatamente para

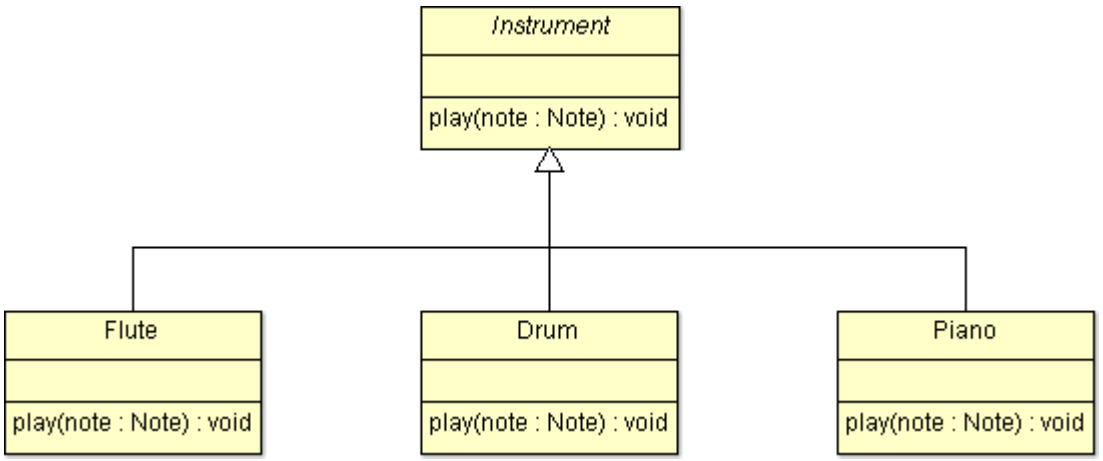
um bloco de código pré-definido, o computador realiza um exame sobre os argumentos e baseado neste exame, escolhe qual o bloco de código realmente deve ser chamado.

O tipo de despacho dinâmico mais comum é o despacho simples, onde o critério utilizado é a classe do primeiro argumento, que em linguagens orientadas a objetos é o "dono" da função. Isto pode ser implementado de maneira bem rápida e simples através de uma tabela de métodos (que é como as funções que pertencem a uma classe de objetos nesse tipo de linguagem são chamadas). Quando um método é compilado, ele é traduzida para um "offset". Quando o programa executa a chamada, o offset é adicionado a tabela, resultando no endereço de código a ser executado.

O objetivo do despacho dinâmico é permitir que métodos diferentes, mas que são semanticamente iguais, possam ser invocados de maneira igual e transparente. Por exemplo, suponha um sistema musical onde existem métodos para tocar instrumentos em uma certa nota. O método possui dois argumentos: o instrumento em si (que, sendo o dono do método, é implícito) e a nota a ser tocada.

Dependendo do instrumento específico a ser tocado, é preciso passar ao sistema sonoro do computador instruções bem diferentes. O formato da onda sonora de uma flauta é diferente do de uma tuba. Porém, a idéia de tocar é a mesma, não importa o instrumento. De fato, gostaríamos de não termos de nos preocupar sequer com qual instrumento específico estamos lidando, mas deixar isso por conta da indireção.

Figura 1.3: *Classes representando instrumentos usando despacho dinâmico para que toquem.*



1.2 Motivações e Objetivos

Como mencionamos anteriormente, o tipo mais comum de despacho dinâmico é o despacho simples. Porém, este não é o único. A generalização mais simples do conceito provavelmente é o *multi-despacho*. Neste, a restrição de apenas a classe do primeiro argumento da função definir o despacho é abolida, e todos os argumentos podem contribuir para a decisão sobre que função real é chamada.

Outra forma de despacho dinâmico é o despacho por classes predicativas. Este é conceitualmente similar ao despacho simples, porém permite que classes "virtuais" classifiquem o objeto em tempo de execução. Isto é, a classe de um objeto não é intrínseca a este, mas pode variar com o tempo conforme algum critério determinado pelo programador.

Outra idéia similar é o *Casamento de Padrões* (Pattern Matching). Aqui, algum padrão é determinado sobre a estrutura dos argumentos recebidos pela função. Por exemplo, um

padrão sobre uma lista poderia casar se a cabeça da lista for par e o restante da lista não fosse vazio. O casamento de padrões ajuda na leitura do código porque cada padrão pode associar partes dos argumentos a variáveis que existem no seu próprio escopo. Porém, os padrões não são usados exatamente como os outros tipos de despacho. Normalmente, a implementação deste despacho requer que todas os casos possíveis, isto é, todos os padrões, sejam definidos em um mesmo lugar e ao mesmo tempo. Isso faz com que este tipo de despacho seja mais similar a certos elementos estruturais, como o "SWITCH".

Todas estas idéias porém, podem ser generalizadas pelo conceito de *Despacho por Predicados*. Aqui temos por predicado qualquer teste que seja computacionalmente possível sobre quaisquer dos argumentos da função. Com isto, podemos criar testes que englobam todos os tipos de despacho mencionados, assim como qualquer outro tipo de despacho imaginável, pois a computação usada para o despacho é arbitrária.

O conceito de despacho por predicados foi introduzido num artigo por Ernst, Kaplan e Chambers Ernst *et al.* (1998) em 1998. Lá, ele é explicado como um elemento a ser adicionado a uma linguagem hospedeira. Desde então, porém, não foram realizados muitos trabalhos sobre o assunto, pelo menos pelo que pudemos averiguar. Este tópico parece ter sido ignorado pela maior parte da comunidade, tanto acadêmica quanto profissional.

1.2.1 Prós e Contras

O uso de predicados como mecanismo de despacho possui algumas sérias desvantagens associados a seu uso. Um dos mais claros é que, sendo a computação dos predicados arbitrária, é possível que cada chamada de função tome um tempo arbitrário para ser processada. Em um programa onde há várias chamadas de funções, como no caso de um algoritmo recursivo, este poderia passar mais tempo processando os despachos do que as funções em si. Fora isso, existe também o problema de que o processo de compilação é NP-completo, o que pode ser muito ruim para projetos grandes.

Porém, estes problemas podem não ser tão ruins quanto aparentam à primeira vista. A questão do tempo de despacho é uma questão de cuidado do programador. Se houvessem ferramentas para evitar testes repetidos, o despacho poderia ser tão rápido quanto um "SWITCH" ou uma série de cláusulas "IF". E a compilação, mesmo que leve um tempo muito longo, seria um custo a ser pago apenas uma vez, não toda vez que o programa é usado.

Pelo lado positivo, despacho por predicados nos permitiria organizar melhor o código em certos projetos, especialmente projetos onde existam regras complexas a serem seguidas que variam de acordo com uma gama grande de fatores. Exemplos destes incluem bancos e hospitais, onde de acordo com detalhes do cliente ou paciente, regras bem diferentes se aplicam.

A maneira mais comum de lidar com estas situações em linguagens orientadas a objetos é o uso de padrões, como o *Visitor* e o *Double Dispatch*. Estes padrões, apesar de resolverem o problema, podem tornar o código mais difícil de ser lido e estendido. Com o despacho por predicados, poderíamos organizar os métodos por predicados específicos sem ter de criar sub-funções auxiliares, o que permitiria que cada função funcionasse como um "portal" para a funcionalidade que representa. Com o uso de um IDE, seria possível apresentar ao usuário as funcionalidades de cada função de maneira dinâmica, permitindo que as diferentes versões da mesma função fossem filtradas por predicados que se aplicam ou não a ela.

1.2.2 Objetivos

O SWNE é uma tentativa de implementar uma linguagem usando o despacho por predicados como mecanismo principal de despacho. Como mencionamos, o conceito de despacho por predicado foi introduzido como algo a ser adicionado à uma linguagem já existente. Ele pressupõe a existência de uma linguagem imperativa, orientada a objetos, com classes e métodos.

O SWNE foi então construído como prova de conceito de que é possível implementar o despacho por predicados não como um mecanismo auxiliar a linguagem, mas sim como forma primária de programação.

Em particular, gostaríamos de mostrar que o uso de predicados é suficiente para criar novas estruturas de dados e realizar os papéis normalmente feitos por tipos, como a verificação estática de consistência do programa.

Um segundo objetivo é que o que for criado possa servir como mais do que uma simples demonstração, mas que possa ser desenvolvido em uma linguagem "verdadeira", isto é, que tenha funcionalidades e bibliotecas suficientes para ser usada em projetos de porte arbitrário de maneira a ser realmente um diferencial positivo a estes.

1.2.3 Estado da Arte

Atualmente, não existem muitas implementações do conceito de despacho por predicados. A que mais se destaca parece ser o JPred Frost e Millstein (2006); Millstein (2004); Millstein *et al.* (2009). Porém, não conseguimos encontrar nenhuma tentativa de implementar o conceito "a partir do zero". Isto é, sem o uso de uma linguagem hospedeira.

Capítulo 2

Conceitos Básicos

Neste capítulo, discutiremos alguns conceitos básicos de design de linguagens computacionais, para podermos depois aplicá-los ao SWNE. Os conceitos aqui explicados serão referenciados no próximo capítulo.

2.1 Compilação e Interpretação

Uma das questões a se decidir quando se implementa uma nova linguagem é se ela será interpretada ou compilada. Na verdade, a compilação é simplesmente um processo que transforma a linguagem da sua estrutura escrita e legível para uma estrutura mais apropriada ao computador. Por isso, ser compilada ou não não é um atributo da linguagem em si, mas sim da implementação. Uma linguagem sem processo de compilação tem de ser interpretada inteiramente em tempo de execução, como é o caso das linguagens de script, como Perl.

Outra questão é se a linguagem é compilada para instruções de nível de hardware ou não. A linguagem C, por exemplo, normalmente é compilada para instruções de processador, enquanto Java usa uma "máquina virtual". Isto é, o resultado da compilação de um programa em Java é uma estrutura que deve ser interpretada por outro programa para ser executada. Este programa é chamado de máquina virtual porque ele funciona como um processador, recebendo instruções e as executando. Porém, por ser um programa e não um processador real, é possível implementar os mesmos comandos e instruções de maneira independente da arquitetura do computador hospedeiro ou do sistema operacional.

Este tipo de troca pode não parecer muito clara. Afinal, que vantagem há em se implementar uma máquina virtual para várias arquiteturas e diferentes sistemas operacionais, ao invés de se fazer o mesmo com um compilador? Porém, na verdade há uma grande diferença entre a complexidade de um compilador e de uma máquina virtual. Um compilador que gere código de máquina precisa seguir várias especificações, tanto da arquitetura destino como do sistema operacional.

Por outro lado, o uso de máquinas virtuais incorrem em um certo custo, já que esta é essencialmente um nível extra de indireção para cada instrução do programa. Este custo pode ser significativo, especialmente se a máquina virtual não é otimizada apropriadamente.

Também existe a possibilidade de uma linguagem ser compilada para outra linguagem. Isto é uma opção muito simples do ponto de vista da implementação da linguagem, pois não requer cuidado em otimizar o código resultante (apesar de ainda ser possível) nem o trabalho de criar e manter uma máquina virtual. Porém, esta opção de certa forma prende a nova linguagem à velha, e enquanto a otimização do código gerado não é impossível, ela pode se tornar mais complicada, ainda mais se este código precisar rodar em computadores com arquiteturas diferentes.

2.1.1 Açúcar Sintático

Açúcar sintático é um conceito de linguagens de programação onde um elemento da linguagem é implementado de maneira indireta, através de um subconjunto de funcionalidades já presente na linguagem. Efetivamente, o elemento implementado não é necessário, mas o uso do açúcar sintático faz com que a linguagem se torne mais rica e fácil de usar para o usuário.

Um exemplo seria a implementação de uma cláusula "FOR" com a cláusula "WHILE" em C[1]. O FOR nesta linguagem nada mais é do que um WHILE acoplado a declaração de variáveis e alguns comandos a serem executados antes de começar um novo loop. É possível, portanto, compilar qualquer cláusula FOR de maneira idêntica a uma WHILE. O código que gera a compilação de um pode gerar a do outro. Como outros aspectos de compilação, açúcar sintático é um elemento de uma implementação específica da linguagem, e não da linguagem em si.

2.2 Valores e Tipos

Quando nos referimos a valores, nos referimos a qualquer elemento "substantivo" da computação. Isto é, qualquer resultado, parcial ou final, de um programa de computador, ou valor auxiliar ao cálculo dos mesmos. Números, textos, caracteres, datas, páginas, posições de memória, todas estas coisas podem ser valores em um programa. Muitas linguagens possuem valores intrínsecos, isto é, que não são definidos por outros termos da linguagem.

Fundamentalmente, objetos normalmente são representados por bytes no nível do hardware, apesar que nem todos os bytes de um computador representam um objeto dentro de um programa. Para cada "tipo" diferente de objeto existe alguma maneira de se ler ou manipular os bytes para que eles sejam interpretados, reconhecidos e isomorfos ao que quer que eles devam representar. Por exemplo, se temos uma string, normalmente haverá alguma maneira para acessar os caracteres individuais da mesma (que também são valores), assim como alguma maneira de se anexar duas strings em uma nova.

Portanto, a representação de cada valor dentro do programa depende do tipo. Em C, existe uma relação direta entre a estrutura dos valores dentro do hardware e os tipos dos mesmos. Por exemplo, se usarmos o comando "sizeof" junto de um tipo, obtemos o número de bytes necessários para representar um valor deste. Além disso, diferentes funções na linguagem são feitas para receber tipos específicos. A soma de inteiros não funciona[2]com valores de ponto flutuante, pois o compilador checa para cada função se o tipo do valor sendo usado está correto. Isso é possível porque a tipagem é estática (explicaremos este conceito na parte de contextos).

Em linguagens imperativas como C, é de praxe que o programador possa criar novos tipos, geralmente os derivando de tipos já existentes e criando um elemento composto. Um exemplo claro de como isso pode ser usado são os números compostos. Um número composto é geralmente representado por dois números reais, um representando a parte real em si e o outro representando a parte imaginária.

Tipos também podem ser usados para tornar o programa mais claro. A checagem de tipos nas funções evita erros básicos no uso das mesmas. O uso cuidadoso de tipos evita então alguns erros que podem aparecer, por exemplo, da associação de dois conceitos similares porém distintos, ou do mal entendimento da estrutura do programa. Mais do que isso,

[1]Não queremos implicar que qualquer implementação de C tem o "FOR" como açúcar. Apenas usamos estas cláusulas porque elas mostram o conceito de maneira simples.

porém, eles também permitem que o código seja mais fácil de ser lido.

2.3 Variáveis

Variáveis são uma outra maneira de indireção no código, e são extremamente importantes para qualquer linguagem imperativas[3]. Uma variável associa um nome a um valor. Porém, este valor pode ser alterado, criando um elemento "mutante" no código. Em linguagens com tipagem forte, variáveis também possuirão tipos. De outra forma seria impossível verificar se os tipos dentro do programa estão corretos.

Também é possível estender o conceito de variáveis para adicionar conceitos um pouco mais avançados. Um exemplo comum disto são vetores, variáveis que guardam uma sequência de valores ao invés de um só, associando a cada um um índice. Em certas linguagens existe também a distinção de apontadores de variáveis, a variável guardando um valor em si, enquanto o apontador guarda uma posição na memória que contém um valor.

O objetivo de termos variáveis é que cada uma delas representa um papel dentro do programa. Um resultado parcial, um valor auxiliar, uma escolha do usuário, etc. Qualquer papel dentro dum programa pode ser representado por uma variável. O conceito de papel é (quase) perpendicular ao de valor. Por exemplo, o número 5, em um determinado momento, pode desempenhar vários papéis dentro de um programa. Ele pode ser o índice de um vetor sendo examinado em um determinado momento, a porcentagem de desconto a ser aplicada às compras do dia, o número de cópias de um item em uma cesta de compras eletrônica, etc. Porém, esta relação não é permanente. O índice do vetor a ser checado vai mudar na próxima vez que o laço for rodado. A porcentagem de desconto vai permanecer a mesma até o fim do dia e o número de cópias no carrinho vai continuar lá até que a compra se realize ou seja desfeita, o que pode demorar um instante ou um ano.

2.3.1 Valores Mutantes

Em linguagens imperativas, os sub-valores de tipo compostos são geralmente implementados de maneira similar à variáveis, de modo que podemos pensar nos sub-campos como variáveis do valor ao invés de variáveis de uma função. Estes sub-campos são chamados de atributos.

Valores mutantes incorporam em si parte do conceito de variáveis. Isto é, parte deles fazem um papel para o valor, e não são partes da essência do mesmo. O nome de um usuário pode mudar durante o programa, sem que o usuário mude. Isto é problemático porque é importante que o valor não possa se tornar inválido. Por exemplo, se duas variáveis representam o mesmo valor mutante, e mudamos algum aspecto deste valor em uma, a outra também deve representar estas alterações.

Além disso, a partir do momento que valores podem mudar, eles também ganham o conceito de ciclo de vida. Um número natural é sempre ele mesmo. Um usuário de uma loja virtual, porém, é criado em um determinado ponto no tempo. A partir de então ele passa a poder mudar e se atualizar, até que eventualmente seja destruído. Chamamos isto de ciclo de vida do valor.

[2]É possível sempre converter um tipo em outro em C, sendo possível criar valores a partir de bytes. Mas esta operação deve ser executada explicitamente. Em alguns casos, e para alguns compiladores, a conversão implícita pode ser permitida, mas esse não é o caso geral.

[3]Linguagens não imperativas também podem possuir variáveis, mas nestes contextos elas podem ter um significado muito diferente. Na linguagem Prolog, por exemplo, uma variável representa um valor qualquer e é usado tanto para definir predicados como para fazer consultas a base de conhecimentos.

O ciclo de vida de um valor como o usuário acima pode ser mais longo do que qualquer execução do programa, requerendo alguma maneira de guardar e recuperar informações. Muitas vezes isto é feito através de bancos de dados, mas também é comum usarmos arquivos.

Normalmente, não existe um suporte linguístico para consistência de valores mutantes. Fica ao cargo do programador implementá-la de alguma maneira. Isto é feito então através de maneiras específicas de se acessar o valor. Por exemplo, é muito comum usar um "D.A.O." (data access object), um sistema pelo qual valores mutantes são sempre acessados e que garante a consistência destes e que eles respeitem o ciclo de vida.

2.3.2 Containers

Como mencionamos, variáveis dão um papel específico a um valor. Isso pode ser referente a um trecho do programa, com uma variável ou a outro valor através de um atributo. Estes papéis então relacionam o valor contido em uma relação de muitos para um. No nosso exemplo anterior, o número cinco (um valor) é usado por diversas partes do programa (muitos). Vendo desta forma, fica claro que precisamos também de alguma forma de criarmos relações de muitos para muitos, isto é, algo similar a uma variável, mas capaz de armazenar vários valores ao invés de apenas um.

Existem vários conceitos de "alto nível" que se enquadram como containers. Filas, listas, conjuntos, árvores, heaps, mapas de hash, entre muitos outros. Normalmente, estes conceitos não são, em sua maioria, implementados como parte da linguagem em si. Ao invés disso, eles normalmente são deixados como responsabilidade de bibliotecas. Porém são raras as linguagens em que pelo menos um tipo de container não exista como aspecto da linguagem em sí.

C, por exemplo, usa vetores (que podem ser declarados estática ou dinamicamente) que estão intimamente ligados a como a arquitetura interna do computador funciona. Quaisquer outros tipos de containers podem ser implementados em bibliotecas separadas da linguagem principal baseado nos vetores (e apontadores). Como os vetores representam diretamente posições contínuas de memória, é possível otimizar como quaisquer outros containers funcionam.

Perl, por outro lado, possui o conceito de hash-maps embutido na própria linguagem e sintaxe. Isto porque os hash-maps estão intimamente ligados à filosofia por trás do Perl. O uso destes containers permite a manipulação de strings de maneira simples e rápida, algo muito importante em Perl.

2.4 Contexto

Um elemento comum a grande parte das linguagens de computação é o uso de contextos estáticos[4]. Um contexto é uma associação de nomes a algum conceito do programa. Normalmente o escopo determina variáveis, mas também é possível termos escopos para tipos, classes, constantes, funções e quaisquer outros elementos que o programa suporte.

Dentro de um mesmo contexto, um nome tem o mesmo significado. Se temos uma variável dentro de um contexto e seu valor é alterado em uma parte do programa; qualquer parte posterior que acesse a variável e esteja no mesmo contexto vai obter o valor modificado, por exemplo. Por outro lado, tentar acessar este mesmo nome de outro contexto resultará ou num erro (caso não exista nenhuma variável com o nome) ou no acesso de uma variável diferente. Por isso, entender bem o contexto é crucial para que o programador entenda bem o que um programa faz. E portanto é crucial que uma linguagem tenha regras claras e simples de contexto para que ela seja útil.

Contextos também podem formar uma hierarquia. Se duas partes do código tem um contexto base em comum, mas não sua derivação, então os nomes definidos pela base são compartilhados, mas não os do contexto, mas os nomes das partes derivadas são inacessíveis da parte do código que não os possuí.

2.5 Dinamicidade e Estaticidade

Até agora, mencionamos várias vezes a idéia de dinamicidade e estaticidade, que um determinado elemento pode ser estático ou dinâmico. Quando falamos de design de linguagens computacionais, estes termos tem um significado bem definido.

Mencionamos que existem várias maneiras de se compilar um programa, e que a maneira específica a ser usada não é uma propriedade da linguagem em si, mas sim de sua implementação. Porém, linguagens frequentemente são escritas levando em conta como elas serão compiladas ou interpretadas. No caso de C, por exemplo, temos vários aspectos, como pragmas, macros, headers, etc, que tem como intuito expresso interagir com o compilador.

Por causa disso, é comum separar o que sabemos sobre o programa em duas categorias: "tempo de compilação" e "tempo de execução". Estes dois conceitos são equivalentes às nossas duas categorias, estática e dinâmica. Então, quando dizemos que um programa possui despacho estático, queremos dizer que a função específica usada por uma chamada de função (ou de método) é conhecida em tempo de compilação.

Geralmente falando, quanto mais sabemos sobre um programa em tempo de compilação, ou seja, quão mais estático ele é, maior a nossa capacidade de detectar erros lógicos e de ajudar o usuário a montar o programa. Porém, quão mais dinâmica for nossa linguagem, maior será sua flexibilidade e capacidade de ser elegante. Isto é, sua capacidade de permitir que o programador crie seu programa de maneira sucinta e clara.

2.6 Clausuras

Clausuras são uma maneira de se tratar funções como valores, ou objetos, de primeira classe. Normalmente, funções tomam um papel sintático nas linguagens. Quando uma função é declarada para o sistema, o próprio compilador cuida para que ela seja corretamente chamada quando seu nome é chamado em alguma parte do código. O processo de criar e chamar funções é algo estritamente lidado pelo compilador.

Porém, se funções são um objeto de primeira classe, então elas são algo compreendido como um valor do próprio programa. Uma função pode retornar como resultado, por exemplo, uma nova função que ela acabou de criar. Esta pode ser guardada em uma variável e usada como se usaria qualquer outro valor.

Isto permite um modo bem diferente de programar, pois agora funções podem ser usadas como argumentos para outras funções. Por exemplo, podemos ter uma função iteradora, que recebe um vetor e uma função para ser chamada sobre cada valor do vetor.

Outro aspecto interessante de clausuras é a possibilidade de currying, onde uma função não precisa receber em um mesmo momento todos os seus argumentos. Ao invés disso, podemos passar a função uma lista de argumentos parciais e recebemos então como resultado uma nova função que recebe os argumentos remanescentes. Num exemplo simples, a função

[4]Um contexto dinâmico é um onde a definição de nomes é feita somente em tempo de execução do código. Isto é, é impossível saber durante a compilação se um nome usado pertence ou não ao escopo. Isto é possível fonte de vários problemas e erros e por isso é muito pouco utilizado.

"add(x, y)" soma dois números. Usando currying, podemos fazer "add(5)" e receber como resultado uma função que some 5 a qualquer argumento que receba.

2.7 Recursão

Recursão é quando usamos um determinado conceito, como um tipo, uma função ou uma variável, dentro da definição do mesmo. Um exemplo clássico disto é o número de Fibonacci.

Definição (Número de Fibonacci)**.** *Fibonacci é uma sequência de números naturais F_n onde:*

$$\begin{aligned} F_n &= 1 \ se \ 0 \leq \ n \leq \ 1 \\ &= F_{n-1} + F_{n-2} \ caso \ contrário. \end{aligned}$$

Na definição acima, a definição dos números da sequência depende dos números anteriores. Recursões podem ser usadas para definir sequências como esta, padrões, conjuntos, etc. Elas nos permitem definir conjuntos infinitos de maneira simples e finita.

Na computação, o uso da recursão geralmente depende de alguma indireção, que efetivamente permita o uso de algum nome antes de sua definição estar completa, ou que a definição em sí seja imcompleta, de maneira que ela possa se receber como parâmetro. Por exemplo uma função que pode chamar a sí mesma para computar seu resultado. Ou mesmo um tipo que possua a sí mesmo, como em uma lista ligada, ou uma string formatada de C, que pode receber vários tipos de valores, inclusive a si mesma.

Sem algum tipo de recursão, seria necessário que todo programa terminasse dentro de um número máximo de passos, determinado em tempo de compilação. Mesmo um laço como um "WHILE" é na verdade um sistema recursivo, pois o mesmo trecho de código é executado de forma que ele seja (ou pelo menos possa ser) sua própria sequência.

2.8 Encapsulamento

Encapsulamento é a capacidade de uma linguagem de restringir o acesso ou a certas variáveis e funções, de forma a tornar mais claro o modo como estes devem ser usados, e evitar erros pelo mau uso.

Um exemplo comum de encapsulamento é a criação de "singletons". Em Java, é possível usar o método "new" para criar uma nova instância de alguma classe. Mas nem sempre é desejável que o usuário a crie desta maneira. Quando uma classe representa, por exemplo, uma configuração provida em um arquivo externo, não queremos que o arquivo seja relido toda vez que o programador precisar acessar estas.

Então, o que podemos fazer é tornar o construtor privado. Isso quer dizer que apenas métodos da própria classe podem acessá-lo. Então, podemos criar um método pelo qual o usuário acessa o arquivo de configuração. Se for a primeira vez que este método está sendo acessado na execução atual, lemos o arquivo e criamos o objeto que o representa. Porém, acessos futuros vão sempre retornar a mesma instância.

Um dos tipos de encapsulamento mais comuns é o de classes, frequentemente usado em orientação a objetos e popularizado por C++ e Java. Neste caso, métodos e atributos podem ter seu acesso restrito a própria classe, sua família (isto é, as classes que derivam da mesma) ou seu pacote. Existem também outras formas de encapsulamento, no entanto. No próprio Racket, há a possibilidade de se criar contratos de pacote, onde o uso de funções e variáveis do pacote é restrito por regras arbitrárias criadas pelo programador.

2.9 Memória

A manipulação da memória do computador está longe de ser algo simples. Na verdade, alguns dos problemas e bugs mais comuns nos programas estão relacionados a memória.

Normalmente, a memória computacional associa um ou mais bytes a um "endereço", um número (ou se preferir, uma sequência de bytes) único àqueles bytes específicos. Valores e variáveis ambos dependem da memória do computador para existirem. De fato, eles são conceitos de um nível semântico mais alto, e por isso pode parecer estranho que trabalhemos diretamente com a memória do computador.

De fato, muitas linguagens tentam eliminar qualquer possibilidade de manipulação direta da memória, sendo Java talvez uma das mais famosas por isso. Em Java, quando um objeto é "criado", então alguma posição de memória é reservada (ou, no jargão de programação, alocada) para o mesmo. Esta posição é liberada para ser usada novamente quando o objeto não possui mais nenhuma referência, isto é, quando nenhuma variável no sistema referência o objeto em questão.

Isto é a chamada coleta de lixo. Com a coleta de lixo automatizada pelo sistema, é possível retirar a memória dos conceitos linguísticos. Claro, o computador ainda vai usar a memória que tiver, mas o programador não possui controle direto sobre a mesma. Por um lado, isso evita os problemas mencionados anteriormente. Por outro lado, é possível que qualquer política de coleta de lixo seja muito lenta para o programa em questão. De fato, em qualquer sistema onde o tempo é algo que precisa ser cuidadosamente manejado, como o software para controlar os movimentos de algum maquinário, o uso de manejamento de coleta de lixo é praticamente tido como impossível.

Em outros sistemas, o manejamento de memória é mais ou menos direto. Por exemplo, C permite que o programador aloque e desaloque pedaços de memória de maneira como bem entender. Em C++, existe também o conceito de criação e deleção, onde o programador pode controlar não apenas como um tipo é inicializado ou finalizado, mas como a memória é manejada para permitir isto. Sistemas operacionais modernos, no entanto, não dão acesso a toda e qualquer posição de memória a um programa. Acessar uma posição de memória qualquer em um programa normal pode resultar em um erro de access violation. Porém, isto é um aspecto do sistema operacional em si, não da linguagem.

Capítulo 3

Implementação

Neste capítulo discutimos o trabalho que foi realizado para a implementação do SWNE.

3.1 Árvores Semânticas

Para o SWNE, não nos preocupamos em otimizar o uso dos recursos computacionais, dado que ele é ainda uma mera prova de conceito. Por isso, programas SWNE são compilados para uma árvore semântica de objetos na linguagem Racket, que é uma variante de Lisp/Scheme. A linguagem Racket é em si interpretada, o que quer dizer que programas do SWNE são processados por uma máquina virtual sobre outra máquina virtual. No futuro, seria interessante mudar isso, talvez até criando um compilador de máquina do SWNE.

Aqui usamos uma árvore de objetos semânticos para representar um programa compilado. Por árvore, aqui queremos dizer que a estrutura de objetos (chamados de *CompilationElement*) é recursiva, ou seja, um CompilationElement pode possuir um ou mais CompilationElements, que por sua vez podem possuir outros elementos e assim por diante. Porém, esta estrutura recursiva é uma árvore e portanto não há ciclos nesta estrutura. Um elemento de compilação não pode possuir a si mesmo, direta ou indiretamente.

Cada um destes elementos representa um aspecto semântico da linguagem. Durante a compilação, cada elemento sintático do código (como o uso de um "loop" ou a declaração de uma variável) é traduzido para um destes elementos. Todos os elementos implementam um método chamado "evaluate", que os chama para que realizem a computação que representam.

Por exemplo, uma cláusula condicional "if-else" é implementada por um elemento de compilação que possui três outros. O primeiro elemento é o teste, que é qualquer trecho de código que retorne um resultado booleano (poderia ser uma função ou mesmo outra cláusula condicional). Dependendo se o resultado retornado é "true" ou "false", o segundo ou o terceiro elemento é rodado, ou "avaliado".

Além de ser uma árvore, um programa SWNE compilado sempre começa num vértice inicial. Usando o método evaluate deste vértice, é possível executar o programa todo.

O processo de compilação no SWNE é bem direto. Como no racket, o abrir e fechar de parenteses representa alguma estrutura sintática básica ou a chamada de alguma função. Estes são traduzidos diretamente para um CompilationElement que os representa. Também existem valores que podem ser usados diretamente no código, ou acesso a variáveis, que também são traduzidos para estes elementos de compilação. A linguagem, por ser extremamente simples, não usa quaisquer recursos de açúcar sintático.

3.2 Ambiente

No SWNE, usamos uma lista chamada de "environment", ou ambiente, para implementar o conceito de contexto. A lista é varrida do começo ao fim sempre que um nome é usado, e termina por retornar um índice de um vetor chamado "store", que faz as vezes de memória física para a execução do programa.

O environment é então passado ao método evaluate quando este é executado. A execução do programa começa com um ambiente vazio. Porém, certas instruções podem adicionar coisas ao ambiente. A instrução "let" adiciona uma nova variável, cujo nome ela recebe. Já uma chamada de função adiciona como variáveis todos os argumentos que esta recebe. Porém, o ambiente modificado só é repassado para os vértices dentro do elemento que modificou o ambiente.

Isto garante que o contexto criado pelo ambiente é estático e pode ser checado sem que o programa seja rodado. O escopo de qualquer variável é sempre interno ao elemento que a cria. Quando um lambda é declarado, ele guarda o environment atual, ao qual retorna quando é chamado. Porém, o ambiente não pode "vazar". O escopo de cada ambiente está sempre preso aos elementos da árvore, e como não há ciclos na árvore semântica, o contexto de qualquer elemento é conhecido em tempo de compilação, e qualquer irregularidade pode ser detectada sem a necessidade de executar o programa.

3.3 Variáveis

Cada nome no ambiente é associado a uma variável. Além da posição de memória correspondente, cada variável também possui uma lista de predicados que se aplicam a esta. Estes predicados de certa forma funcionam como o "tipo" da variável. Assim como tipos, alguns predicados podem ser checados de maneira estática para averiguarmos que a lógica do programa está sendo respeitada.

O compilador averigua através das relações entre os predicados se alguma operação é insegura ou errônea. Uma operação é considerada errônea se uma variável recebe um valor que possui um predicado incompatível com os predicados da variável. Por exemplo, se a variável a é par, e ela recebe uma variável que é impar, o sistema detecta uma inconsistência e não permite a compilação. Por outro lado, se o valor sendo colocado em a é um inteiro, que não sabemos se é par ou impar, o programa avisa durante a compilação que é possível que haja um erro, mas permite que o usuário prossiga.

Durante a execução do programa, quando uma variável é alterada, a máquina virtual também realiza uma verificação para ter certeza que todos os seus predicados se aplicam ao novo valor associado. Assim, os predicados também funcionam como um tipo de "assert", checando a consistência do programa conforme ele roda. Este tipo de verificação, porém, pode ser custosa e é, estritamente falando, desnecessária. Ela foi incluída na linguagem apenas para podermos mostrar a capacidade de usar os predicados para tal fim, mas fora de uma prova de conceito, seria interessante que o comportamento de verificar ou não cada predicado fosse configurado pelo programador.

3.4 Predicados

Os predicados no SWNE tomam o lugar de classes e tipos em outras linguagens. Como mencionamos na seção anterior, os predicados são responsáveis pelas checagens estáticas. Sempre que um valor é passado, checamos se é uma tautologia que o que sabemos sobre o

valor implica nos predicados esperados. Se este não for o caso, um warning é emitido. Se a negação da implicação é uma tautologia, no entanto, o processo de compilação para com um erro.

Os predicados se dividem em dois tipos: que chamamos de predicados estáticos e predicados dinâmicos. Os predicados estáticos funcionam como uma lista de objetos. Quando um objeto é adicionado a um predicado estático, o predicado passa a valer para ele. Se ele for removido, o predicado não mais vale. Predicados estáticos permitem que sejam adicionados campos aos objetos membros de forma similar a como classes permitem definir a estrutura de um objeto.

Predicados dinâmicos, no entanto, usam uma função para determinar se um objeto pertence ou não ao predicado. Em particular, este lambda é sempre uma função *booleana*, isto é, que retorna verdadeiro ou falso. O predicado dinâmico se aplica então a qualquer tupla de argumentos para os quais seja verdadeiro.

Os predicados não são armazenados na árvore semântica do programa. Ao invés disso, são armazenados em um container próprio para serem repassados para o programa sempre que necessário. Isso porque todo predicado tem escopo global.

3.5 Despacho por Predicados

Quando uma função é invocada, o SWNE precisa associar aos argumentos alguma assinatura válida. Cada função possui uma ou mais assinaturas e para cada assinatura, possui o que chamamos de lambda.

A assinatura define os argumentos (número e nomes) que o lambda associado possui. Além disto, também define uma expressão lógica de predicados que se aplicam a estes argumentos. Quando uma função é criada ou alterada, estas assinaturas são armazenadas na mesma ordem que são declaradas, com apenas uma exceção: se uma das assinaturas implica em outra, então a assinatura mais específica é movida na lista para cima da mais genérica. Este é um processo um pouco lento, pois a averiguação dos predicados depende da comparação de cada assinatura com cada outra.

Quando a chamada em si é feita para a função, cada assinatura é testada em ordem. O teste de uma assinatura é um processo simples, mas que pode ser bem lento. Toda assinatura possui uma expressão lógica construída em cima de predicados. Esta expressão é checada, sendo as funções associadas aos predicados chamadas conforme o necessário. Se a expressão total é avaliada como verdadeira, o processo termina; a assinatura atual é a escolhida e o lambda associado a ela é chamado para dar continuidade ao programa. Caso contrário, a próxima expressão é testada, até que alguma sirva ou até que não tenhamos mais assinaturas para testar.

3.5.1 Sistema Lógico

O SWNE, por lidar com predicados, eventualmente precisa lidar com testes sobre expressões lógicas envolvendo estes predicados. Para tanto, usamos um algoritmo de tabela da verdade. Este algoritmo simples basicamente testa todas as alternativas lógicas das variáveis para tentar averiguar se uma expressão dada é uma tautologia.

Por exemplo, para sabermos se uma assinatura implica em uma outra, damos ao algoritmo a expressão de todas as relações que o usuário nos deus sobre os algoritmos implicando que a assinatura a implica a assinatura b. Se esta expressão for uma tautologia, então a primeira assinatura é mais específica do que a segunda (ou pelo menos equivalente).

3.6 Gramática

As instruções do SWNE são todas delimitadas pelo uso de parêntesis. Cada abrir e fechar de parêntesis indica uma instrução específica. Isto foi usado para que o processo de compilação fosse mais fácil, mais próxima do próprio Racket. Enquanto o resultado é um pouco difícil de ler, essa decisão foi tomada para reduzir o escopo deste trabalho.

A primeira palavra de cada parêntesis determina qual a instrução específica a ser usada. Isso cria uma correspondência quase direta entre o código escrito e o compilado, tornando o processo de compilação mais fácil. Quaisquer outros elementos dentro do parêntesis são repassados como argumentos à instrução.

As instruções principais são:

3.6.1 Declare

A instrução declare recebe um nome e uma sequência arbitrária de outras instruções. Ela cria uma variável com o nome desejado, e a coloca dentro do escopo para todas as instruções que estiverem dentro do mesmo parêntesis.

Fora o nome da variável, uma lista de predicados também pode ser passada, como segundo parâmetro, uma sequência de predicados que se aplicam ao valor da variável. Esta lista deve estar delimitada pelo sinal de dois pontos ':'. Por exemplo:

```
1 (define a :number:
2   (set a 5)
3   (a) )
```

O código acima declara a variável a que só pode receber valores que sejam números. Logo depois, o valor 5 é guardado em a, e o código termina com a leitura de a (e portanto retorna 5).

Como SWNE usa clausuras, também é possível guardá-las em variáveis. Como demonstrado no código abaixo:

```
1 (define divide :[x, y -> z] ^ (natural x) ^ (natural y) ^ (natural z):
2   (...))
```

O código acima é mais complicado; a estrutura entre chaves "[x, y -> z]" indica que divide vai guardar uma função de duas variáveis, e que ela retorna uma. Os outros predicados indicam que estes três valores são números naturais. Esse tipo de estrutura permite que definamos predicados que devem ser aplicados a uma função. O resto do código é omitido por não ser relevante a esta seção.

3.6.2 Predicate

Esta instrução permite a criação de um novo predicado. Ela sempre recebe três argumentos. O primeiro é o nome do predicado. O conjunto de nomes de predicados é separado do conjunto de variáveis, então não há problemas de conflitos entre estes. O segundo argumento é a aridade do predicado e o terceiro argumento é o nome da variável que contém a função que verifica o predicado.

```
1 (predicate divides 2 is_divisible)
```

A código acima determina que o predicado divides tem aridade 2. Ele usa a função is_divisible para determinar se ele se aplica ou não aos valores questionados. A instrução predicate se aplica ao programa inteiro e normalmente os predicados vão estar no começo do programa.

3.6.3 Relation

Relation permite criar uma relação entre predicados. Estas relações são checadas pelo SWNE para a verificação da compilação e para o processo de despacho. Um comando de relation usa o nome de predicados e sinais lógicos. Por exemplo:

```
1 (relation (divides a b) -> (natural a) ^ (natural b))
```

3.6.4 Set

Set é a instrução que permite guardar em uma variável um valor. O valor pode ser passado como uma expressão, e nesse caso o que é armazenado é o resultado da expressão. Um exemplo disto é:

```
1 (define a
2   (define b
3     (define c
4       (set c 5)
5       (set b 4)
6       (set a (sum c b))
7     )
8   )
9 )
```

O código acima guarda em a o valor de $b + c$, ou seja, 9.

3.6.5 Lambda

Normalmente, lambdas e clausuras são equivalentes. Porém, no SWNE, existe uma distinção. Na nossa linguagem, o que chamamos de lambda não possui o despacho por predicados. Ao invés disso, ele funciona como um lambda normal, recebendo parâmetros e rodando suas instruções sobre estes.

Porém, lambdas não são usados diretamente. Ao invés disso, eles formam os pedaços a partir dos quais as funções em si, também chamadas de clausuras, são formadas. Um exemplo disto é dado na explicação da cláusula Function.

Os argumentos que lambda recebe são uma lista de instruções a serem processadas sequencialmente. A aridade do lambda não é declarada por ele. Ao invés disso, ele depende dum parâmetro de Function para saber quais variáveis estão associadas. Isso porque lambdas não devem ser usadas em nenhum outro contexto.

3.6.6 Function

Function é uma cláusula que forma uma clausura. Function recebe pares de assinaturas e lambdas, e a partir destes cria uma função que realmente usa o despacho por predicados.

```
1 (function
2   (:[x] ^ (natural x) ^ (leq x 1):
3     (lambda ...))
4   (:[x] ^ (natural x) ^ (geq x 2):
5     (lambda ...))
6 )
```

No exemplo acima, declaramos uma nova função com duas assinaturas. As duas assinaturas recebem um argumento, mas são separadas por se este argumento (chamado x) é menor

ou igual a 1 ou maior que este número. Note que o primeiro predicado nos diz não apenas que a assinatura se aplica a um argumento, mas também dá o nome deste argumento. Essa sintaxe é similar a usada no define, mas aqui o resultado não deve ser nomeado porque é impossível criar uma assinatura em cima do resultado.

3.6.7 Call

Para chamar uma função, basta colocá-la como uma instrução dentro do próprio parêntesis. O programa interpreta isto como uma chamada, e assume que os próximos elementos no parêntesis são os argumentos. Se a função for uma variável, há uma checagem em tempo de compilação para verificar se os predicados dos parâmetros causariam algum erro lógico com os predicados da função.

Por exemplo, se uma função recebe um número natural, mas colocamos como parâmetro a mesma uma variável que sabemos ser uma string, o programa avisará sobre o erro. Porém, o programa não impede usos que podem ou não estar corretos. Por exemplo, se o usuário colocar um número inteiro como parâmetro, é possível que este número inteiro também seja natural e portanto o programa não acusa um erro. Seria possível avisar o usuário sobre isso num modo de compilação pedântico, mas isto pode causar certa lentidão na compilação, pois averiguar isto requer novos testes de tautologia.

Capítulo 4

Resultados

Neste capítulo, discutiremos alguns resultados do projeto do SWNE. No momento apenas alguns poucos dados são suportados e o sistema de desenvolvimento de algoritmos se foca em predicados de uma variável. O SWNE ainda precisa de muito trabalho para realmente ser útil. Porém, conseguimos resultados bem interessantes:

Apesar de rudimentar, é possível ver algumas das vantagens do SWNE através de exemplos bem específicos. Enquanto ele não se mostra melhor do que uma implementação mais completa de despacho por predicados, o SWNE se mostra viável. Mas talvez mais interessante, obtivemos também uma boa noção de como o SWNE poderá ser desenvolvido. Quais elementos precisam ser melhorados e que tipo de capacidades poderão ser implementadas com ele.

4.1 Exemplos

4.1.1 Fibonacci

O seguinte programa em SWNE calcula um número de Fibonacci arbitrário. O algoritmo depende da "biblioteca" embutida no programa, onde os predicados de *leq* e *geq* são definidos, e funções como o "+" e "-" são implementadas.

```
1 (define fibonacci :[x -> z] ^ (natural x) ^ (natural y):
2   (function
3     :[x] ^ (leq 0 x) ^ (leq x 1): (lambda (1))
4     :[x] ^ (geq x 2): (lambda (+ (fibonacci (- x 1)) (fibonacci (- x 2))))
5   )
6 )
```

Compare o código com o código do mesmo método em C:

```
1 int fibonacci (int n) {
2   if (0 <= n <=1) {
3     return (1);
4   }
5   else if (n >= 2) {
6     return (fibonacci(n - 1) + fibonacci(n - 2));
7   }
8   else {
9     /* Error code ommited. */
10  }
11 }
```

Este exemplo simples não mostra muito bem como os SWNE pode ajudar a melhorar a legibilidade do código. De fato, acreditamos que o código em C seja mais fácil de ser lido por causa de como sua notação funciona. Este aspecto é algo que poderia ser revisto em uma futura mudança, de permitir que o programador use uma notação além da pré-fixa comum nas linguagens funcionais.

Porém, este exemplo mostra algumas qualidades do SWNE. Primeiro, apesar da notação, cremos que o exemplo acima do SWNE é mais parecido com a definição algébrica dada anteriormente quando discutíamos recursão.

Além disso, o código do SWNE não precisa de um caso para erro. Tentar usar um valor negativo com o fibonacci do SWNE já gerará um erro pois a função fibonacci lá é definida sobre o predicado "natural", que já exclui números negativos.

4.1.2 Asteroides

Este exemplo um pouco mais extenso mostra melhor as vantagens do uso de despacho por predicados. Aqui vamos comparar SWNE com Java, tentando implementar a lógica de um jogo de naves espaciais, similar ao clássico de arcade, *Asteroids*. Para simplificar a situação, os trechos de código aqui se focam apenas com a colisão de objetos, sendo que uma string é impressa para cada caso. Isso não é o que realmente aconteceria num jogo assim, mas o nosso objetivo é mostrar a capacidade do programa em lidar com diferentes aspectos.

```
1 abstract class SpaceObject {
2
3   public abstract String collide (SpaceObject o);
4
5   protected abstract String asteroidCollide (Asteroid other);
6
7   protected abstract String shipCollide (Ship other);
8
9   protected abstract String shotCollide (Shot other);
10 }
11
12 class Asteroid extends SpaceObject {
13
14   private int mass;
15
16   public Asteroid(int mass) {
17     this.mass = mass;
18   }
19
20   public String collide(SpaceObject o) {
21     return(o.asteroidCollide(this));
22   }
23
24   protected String asteroidCollide (Asteroid other) {
25     if (this.mass > 2 * other.mass || other.mass > 2 * this.mass) {
26       return "Two asteroids collide, with the bigger one absorbing the
             smaller.";
27     }
28     else {
29       return "Two asteroids collide, detroying each other.";
30     }
31   }
32
33   protected String shipCollide (Ship other) {
34     return "The ship collides with an asteroid and is destroyed.";
```

```
35   }
36
37   protected String shotCollide (Shot other) {
38     if (this.mass > 1) {
39       return("The asteroid splits in two.");
40     }
41     else {
42       return("The asteroid is destroyed.");
43     }
44   }
45
46 }
47
48 class Ship extends SpaceObject {
49
50   public String collide (SpaceObject o) {
51     return o.shipCollide(this);
52   }
53
54   protected String asteroidCollide (Asteroid other) {
55     return other.shipCollide(this);
56   }
57
58   protected String shipCollide (Ship other) {
59     return ("Error!"); //There is only one ship.
60   }
61
62   protected String shotCollide (Shot other) {
63     return ("The shot is dissipated by the ship's shields.");
64   }
65 }
66
67 class Shot extends SpaceObject {
68
69   public String collide(SpaceObject o) {
70     return o.shotCollide(this);
71   }
72
73   protected String asteroidCollide(Asteroid other) {
74     return other.shotCollide(this);
75   }
76
77   protected String shipCollide(Ship other) {
78     return other.shotCollide(this);
79   }
80
81   protected String shotCollide(Shot other) {
82     return ("Two shots meet each other, dissipating hamlessly.");
83   }
84
85 }
```

No código acima, usamos um padrão chamado *Despacho Duplo* para evitar a limitação de java de apenas despacho sobre um argumento. Como a colisão é comutativa, nós acabamos tendo de implementar mais métodos do que realmente precisamos. Quando isso acontece, um dos métodos sempre repassa ao outro. Isso nos ajuda a evitar problemas de código duplicado, que são ruins para a manutenção do código-fonte.

```
1 (static_predicate space_object)
```

```
2 (static_predicate asteroid
3   (define mass :(natural mass):))
4 (static_predicate ship)
5 (static_predicate shot)
6
7 (relation :(asteroid x) -> (space_object x):)
8 (relation :(ship x) -> (space_object x):)
9 (relation :(shot x) -> (space_object x):)
10
11 (predicate near_mass 2 near_mass)
12 (define near_mass :[x, y -> z] ^ (asteroid x) ^ (asteroid y) ^ (boolean z)
     :
13   (function
14     :[x, y]: (or (geq (* 2 (x.mass)) (y.mass)) (geq (* 2 (y.mass)) (x.mass
         )))
15   )
16 )
17 (relation :(near_mass x y) -> (asteroid x) ^ (asteroid y))
18
19 (predicate pebble 1 is_pebble)
20 (define is_pebble :[x -> y] ^ (asteroid x) ^ (boolean y):
21   (function
22     :[x]: (eq 1 (x.mass))
23   )
24 )
25 (relation :(pebble x) -> (asteroid x))
26
27 (define collision :[x, y -> z] ^ (spaceObject x) ^ (space_object y) ^ (
     String z):
28   (function
29     :[x, y -> z] ^ (near_mass x y):
30       ("Two asteroids collide, detroying each other.")
31     :[x, y -> z] ^ (asteroid x) ^ (asteroid y):
32       ("Two asteroids collide, with the bigger one absorbing the smaller."
           )
33     :[x, y -> z] ^ ((asteroid x) ^ (ship y)) | ((ship x) ^ (asteroid y)):
34       ("The ship collides with an asteroid and is destroyed.")
35     :[x, y -> z] ^  ((pebble x) ^ (shot y)) | ((shot x) ^ (pebble y)):
36       ("The asteroid is destroyed.")
37     :[x, y -> z] ^  ((asteroid x) ^ (shot y)) | ((shot x) ^ (asteroid y)):
38       ("The asteroid splits in two.")
39     :[x, y -> z] ^ ((shot x) ^ (ship y)) | ((ship x) ^ (shot y)):
40       ("The shot is dissipated by the ship's shields.")
41     :[x, y -> z] ^ (shot x) ^ (shot y):
42       ("Two shots meet each other, dissipating hamlessly.")
43   )
44 )
```

Não só no código do SWNE as informações estão muito mais acessíveis (apesar de alguns problemas de legibilidade), mas o programa também não precisa de certas informações óbvias. Por exemplo, como o programa sabe que o predicado *pebble* implica em *asteroid*, não há a necessidade de dizer que este caso deve ser checado primeiro. Também não cuidamos do caso que duas naves se chocam porque não há necessidade.

4.2 Melhorias

Como mencionamos, o SWNE ainda está longe de uma linguagem de verdade. Parte dos resultados deste trabalho é mencionar o que, e como, ainda precisa ser feito para que o SWNE se torne útil. Então, nesta seção, vamos mencionar algumas destas falhas:

4.2.1 Verificação de Predicados

O sistema de verificação de predicados pode ser muito lento. O problema em sí é NP-completo. O que quer dizer que não deve existir nenhum algoritmo bom para ele. Enquanto isso é factível em tempo de compilação (apesar de um pouco perigoso), isso é muito ruim em tempo de execução. Porém, montar funções em tempo de execução requer que o verificador de predicados seja chamado sobre cada par de lambdas dentro da nova clausura.

É preciso achar maneiras de repassar este custo para o tempo de compilação. No momento, as assinaturas estão todas montadas em tempo de compilação, então seria factível criar um grafo de implicação entre elas. Porém isso não necessariamente vai ser verdade para sempre. Especialmente se quisermos predicados paramétricos.

4.2.2 Memória

O SWNE no momento não possui nenhum tipo de manejamento de memória. Efetivamente, quando uma variável é criada, ela ocupa uma posição no *store*, que é representativo da memória física. Porém, não existe nenhuma maneira de liberar esta memória uma vez alocada.

Seria interessante se houvesse no SWNE a opção, mas não a necessidade, de se usar algum tipo de coleta automática de lixo. Isso permitiria que o sistema fosse usado sem uma preocupação excessiva com detalhes de baixo nível, mas não impediria que programas no SWNE fossem otimizados neste aspecto quando necessário.

4.2.3 Predicados Estáticos

Os predicados estáticos no SWNE no momento são obrigados a terem aridade unitária. Além disso, estes predicados não funcionam com o relacionamento ainda, pelo menos no aspecto estrutural. Isto é, se um predicado estático define um atributo, este atributo não vai ser repassado para predicados que impliquem este primeiro. Além disso, o processo de compilação não checa os atributos no momento. O que quer dizer que tentar acessar o atributo de um objeto que não o tem só nos dará um erro em tempo de execução.

Para resolver este problema, é necessário criar mais testes lógicos na hora da compilação, de maneira que para cada expressão lógica, nós saibamos exatamente quais nomes de atributos se aplicam a esta.

4.2.4 Valores Intrínsecos

No momento, valores da linguagem e valores criados pelo usuário são tratados de formas completamente diferentes pelo sistema. Valores da linguagem, ou valores intrínsecos, são representados usando objetos do próprio Racket. Enquanto objetos do usuário são representados usando hash-maps.

O problema porém vai mais fundo. Os predicados e funções para os tipos intrínsecos estão todos definidos fora do escopo de programação. Isso torna impossível, por exemplo,

que o usuário crie um novo modelo que representa números, mas que seja reconhecido pelo sistema como equivalente ao atual.

Uma maneira de resolver isso é se houvesse uma função centralizada de igualdade, que pudesse ser estendida. Se essa função fosse usada internamente para verificar a igualdade entre objetos, mas pudesse ser alterada externamente para poder lidar com novos tipos, não haveria diferenças reais entre os valores definidos pelo usuário e os que acompanham o SWNE.

4.2.5 Compilação

O processo de compilação, além de lento, deixa vários possíveis problemas de lado. Em particular, é possível que uma função só perceba que existe um caso que suas assinaturas não cobrem durante a execução. Este problema é particularmente difícil de se resolver, mas seria interessante estudar melhor como o compilador pode ser melhorado.

4.2.6 Assegurar

No momento, existe um sistema de detecção de erros baseados nos predicados. Por exemplo, o sistema não deixará que o usuário guarde um valor ímpar em uma variável que é declarada como par. Porém, a declaração não é bem o melhor lugar para colocar estas restrições. Em particular porque desta forma, só podemos assegurar predicados de uma variável.

4.2.7 Pacotes

Outro problema ruim é que o SWNE no momento não possui nenhuma maneira de criar bibliotecas, pacotes ou o que quer que seja. De fato, não temos nem mesmo uma instrução de import.

Seria interessante criar alguma maneira de se importar funções e variáveis de outros programas. Poderíamos usar algum sistema de contratos similar ao do Racket para garantir invariantes sobre o sistema.

Capítulo 5

Conclusões

O nosso trabalho mostrou, não muito surpreendentemente, que o SWNE é viável. É possível que a dependência de instruções lógicas faça que o SWNE nunca chegue a ser usado comercialmente. Mas ele certamente seria viável, mesmo que apenas como ferramenta exploratória.

O SWNE foi idealizado como uma maneira de organizar o programa de maneira que sua manutenção seja mais simples. Ainda estamos muito longe, porém, de provar isso. Para tanto, será necessário implementar muitas outras ferramentas e aspectos na linguagem.

Porém, o projeto do SWNE mostrou várias possibilidades do seu uso. Durante todo o tempo de pesquisa, apareceram uma variedade de idéias interessantes de como usar os conceitos por trás da linguagem.

5.1 Trabalhos Futuros

Uma das principais idéias do uso do SWNE seria a criação de um tipo de IDL, que permitisse a organização do código de acordo com expressões lógicas. Por exemplo, poderíamos querer ver quaisquer funções que são afetadas pelo predicado *leq*, ou filtrar uma função por apenas uma família de predicados.

Mas fora a esta, as seguintes idéias todas surgiram como resultado do trabalho desenvolvido:

5.1.1 Processamento de Erros em Tempo de Compilação

Um aspecto interessante de trabalhar com predicados é que eles nos trazem mais perto da especificação formal do programa. Toda invariante pode ser pensada também como uma série de predicados que se aplicam ao programa em certos pontos.

Isso quer dizer que é, a princípio, possível colocar mais "força" na checagem de erros da compilação. Assim como um compilador de C checa se o tipo de uma variável é condizente com o seu uso, podemos checar se uma variável obedece este ou aquele predicado.

Por exemplo, se sabemos que em certo ponto o programa assegura que uma variável é positiva, podemos verificar se a maneira como ela é derivada condiz com isso. Se x é par, e seu valor é dado por $y + z$, então y e z devem ter a mesma paridade. A princípio, seria possível propagar estes testes pelo programa inteiro, verificando se há alguma irregularidade.

5.1.2 Despacho Pré-Clausural

Outra ideia interessante seria aplicar a filosofia por trás da abstração do despacho por predicados em um nível ainda mais baixo. Ou seja, se efetivamente tomássemos também a própria função como um parâmetro da chamada e fizéssemos o despacho sobre ela.

Por exemplo, suponhamos que exista um predicado sobre as próprias clausuras, chamado de *log*. Este indica que quando a função for chamada, deve ser logado no arquivo de log o nome da função e os parâmetros que ela recebeu.

O SWNE poderia permitir que fosse criada uma assinatura não para uma função específica, mas que apenas dissesse que se a clausura sendo invocada tem este predicado, então o sistema deve realizar o processo de log antes de devolver o controle a implementação específica. Efetivamente, isso permitiria que usássemos os predicados como forma de implementar aspectos.

Apêndice A

Parte Subjetiva

Este trabalho representa algo que o autor muitas vezes pensou ser impossível. Depois de quinze anos, sem nunca ter chegado perto de entregar um trabalho escrito completo para a disciplina MAC0499, parecia mais do que claro que esta não era uma boa área para ele.

O trabalho foi desenvolvido usando Racket. Este foi o primeiro trabalho de grande porte em que o autor usou uma linguagem funcional. Enquanto aprender uma nova linguagem de computação não é algo muito difícil, ainda mais hoje quando existem tantos recursos online, aprender um novo paradigma, porém, é mais complicado; por isso foi fortuito que o professor Marco Dimas Gubitoso tenha dado um curso onde este paradigma foi explorado em parte.

O SWNE ainda está muito aquém do que gostaríamos, e seria interessante continuar a trabalhar nele, possivelmente o transformando em uma linguagem "de verdade".

A.1 Disciplinas

A disciplina que afetou este trabalho mais diretamente com certeza é MAC0316 - Conceitos de Programação. Esta disciplina foi cursada pelo autor como ouvinte, e deu as ferramentas diretas usadas para montar o SWNE.

Importante também foram as disciplinas MAC0211 e MAC0242, os laboratórios de programação, que são um dos primeiros contatos dos alunos com projetos de um porte maior. Fora a isso, também são o primeiro contato com linguagens orientadas a objeto.

Muitas outras disciplinas tiveram impacto neste projeto também, apesar de maneira menos óbvia. Em particular, as disciplinas matemáticas, como álgebra e cálculo, tiveram um grande impacto na forma do autor pensar. Sem estas, seria muito mais difícil conseguir abstrair as idéias necessárias para montar o SWNE.

A.2 Agradecimentos

Este trabalho teria sido completamente impossível sem o apoio de meu pai, Maurício Biral, e de meu irmão, André Abate Biral. Também foi crucial o apoio e paciência dos professores Marco Dimas Gubitoso e Nina S. T. Hirata.

Também gostaria de agradecer o apoio e amizade dos professore Alair Pereira do Lago, Alan Durham, José Coelho de Pina e Fábio Kon. Dos meus amigos Natacha Lorido, Eduardo Menezes de Morais e Luís Carlos Saito.

Finalmente, quero agradecer ao autor inglês Gilbert Keith Chesterton, o qual nunca conheci pessoalmente e cuja obra não poderia ser mais distante do tema deste. Mas sem sua

obra, esta monografia não poderia existir. Finalmente e acima de tudo agradeço ao Autor da vida, a quem não se pode dar crédito demais ou louvar o suficiente.

Referências Bibliográficas

Ernst *et al.***(1998)** Michael Ernst, Craig Kaplan e Craig Chambers. Predicate dispatching: A unified theory of dispatch. Em *ECOOP'98—Object-Oriented Programming*, páginas 186–211. Springer. Citado na pág. 5

Frost e Millstein(2006) Christopher Frost e Todd Millstein. Modularly typesafe interface dispatch in jpred. *FOOL/WOOD*, 6. Citado na pág. 6

Millstein(2004) Todd Millstein. Practical predicate dispatch. Em *ACM SIGPLAN Notices*, volume 39, páginas 345–364. ACM. Citado na pág. 6

Millstein *et al.***(2009)** Todd Millstein, Christopher Frost, Jason Ryder e Alessandro Warth. Expressive and modular predicate dispatch for java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):7. Citado na pág. 6