Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Semântica Multidimensional de Java

Mirlaine Aparecida Crepalde
Roberto da Silva Bigonha

LLP01/2008

Relatório Final: POCII

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Semântica Multidimensional de Java

por

Mirlaine Aparecida Crepalde
mirlaine@dcc.ufmg.br

Monografia de Projeto Orientado em Computação II

Apresentado como requisito da disciplina de Projeto Orientado em Computação II do Curso de Bacharelado em Ciência da Computação da UFMG

Prof. Dr. Roberto da Silva Bigonha
Orientador

Belo Horizonte
2008/2° semestre

Resumo

O principal objetivo do presente trabalho é validar a metodologia denominada Multidimensional para definição formal de linguagens de programação. Essa metodologia recente trata problemas de extensibilidade encontrados em outras soluções para definição de semântica formal. Para validar a metodologia, será feita uma definição incremental da semântica de Java usando a linguagem *Notus*, a primeira a implementar essa abordagem. Inicialmente será descrita a semântica de um conjunto básico de construções de Java, chamado de Java 0. Em seguida, serão inseridas novas construções de Java na sub-linguagem Java 0, utilizando recursos específicos de *Notus* para escrita incremental. Por fim, é realizada uma verificação do impacto dessas inserções na definição de Java 0, e abordadas as dificuldades envolvidas nessa metodologia.

Abstract

The main objective of this report is to validate the Multidimensional methodology for formal definition of programming languages. This recent methodology is implemented by the *Notus* language and treats extensibility problems that exist in other solutions for formal definition of semantic. To validate the methodology, a Java language definition will be made using *Notus*. Initially, the semantics of a basic set of constructions Java will be made, called Java 0. Thereafter, new Java constructions will be inserted in the sub-language Java 0 using specific *Notus* resources for incremental writing. Finally, it is verified the impact of the new embeded constructions in basic set on the initial definition of Java 0; and issued the difficulties involved in this methodology.

Lista de Figuras

Sumário

1 Introdução

1.1 Visão geral do assunto

A descrição formal da semântica de linguagens de programação se baseia em princípios matemáticos precisos, permitindo uma compreensão clara e não ambígua de seus aspectos, o que não pode ser garantido ao se definir a semântica de maneira informal, por meio da linguagem natural. Existem diversas abordagens para as descrições formais da semântica de linguagens, como Semântica Axiomática, Semântica Operacional, Semântica Denotacional [Gordon 1979] e Semântica Multidimensional [Tirelo, Bigonha e Saraiva 2008].

A especificação da semântica de uma linguagem de programação de grande porte envolve grandes dificuldades inerentes à sua complexidade. É desejável, portanto, que a metodologia de descrição da semântica permita uma definição formal que possa ser feita de forma modular e incremental para que essa complexidade possa ser controlada.

Semântica Multidimensional é uma abordagem nova de formulação de Semântica Denotacional que permite o estruturamento modular e incremental de definição formal de linguagens de programação. O método de Semântica Multidimensional é baseado na linguagem de definição semântica *Notus*, descrito em [Tirelo e Bigonha 2006].

1.2 Objetivos e motivações

O objetivo desse trabalho é validar a metodologia de Semântica Multidimensional e o poder de expressão da linguagem *Notus*.

Essa validação será feita por meio da definição incremental e modular da linguagem Java, utilizando a Semântica Multidimensional, que é uma abordagem bem recente para tratar problemas de extensibilidade e modularidade encontrados em outras soluções para definição de semântica de linguagens.

Java foi escolhida por ser uma linguagem de grande porte, com várias construções interessantes da programação moderna que podem ser definidas incrementalmente, possibilitando uma melhor aplicação do modelo multidimensional. Além disso, Java é uma linguagem bastante difundida atualmente e a definição formal de seus vários conceitos é bastante relevante.

2 Referencial teórico

A Semântica Denotacional usa objetos matemáticos, chamados denotações, para descrever a semântica de uma linguagem de programação. Denotações são entidades matemáticas abstratas que modelam o significado de cada elemento sintático da linguagem. A Semântica Denotacional pode ser utilizada para registrar decisões de projeto realizadas durante a criação de uma linguagem de forma clara, não ambígua e completa. Ela é constituída pelas seguintes seções: domínio sintático, sintaxe abstrata, domínio semântico, funções semânticas e equações semânticas. No domínio sintático são listadas as categorias sintáticas da linguagem especificada. A sintaxe abstrata especifica os vários construtores e seus constituintes imediatos, sem detalhes sintáticos irrelevantes para a semântica. Os domínios semânticos determinam os objetos matemáticos que compõem a semântica da linguagem. Funções semânticas mapeiam objetos do mundo sintático em objetos semânticos. As equações semânticas especificam a denotação de cada construtor sintático da linguagem.

Uma deficiência em relação a modularidade apresentada pela Semântica Denotacional é que a definição de certas funcionalidades pode ocasionar a redefinição de descrições já existentes. Essa fragilidade motiva o estudo de novas abordagens que visam alcançar a modularidade necessária para a descrição de linguagens de programação reais. [Bigonha 1981] propõem uma metodologia de definição semântica que incorpora a modularidade.

Usualmente, quando ensina-se uma linguagem de programação é conveniente abstrair certos conceitos avançados para explicar conceitos básicos. Posteriormente, esses conceitos avançados são introduzidos, podendo redefinir elementos previamente explicados. Essa explicação vaga é desejável porque geralmente requer menor esforço para aprender os conceitos da linguagem.

Nesse contexto, a Semântica Multidimensional visa aperfeiçoar as técnicas de definição de Semântica Denotacional tradicional de linguagens de programação. Ela provê cláusulas específicas para extensão da gramática concreta e domínios da linguagem, além das **transformações** [Tirelo, Bigonha e Saraiva 2008] que permitem alterar declarações das funções semânticas, possibilitando uma escrita modular e incremental sem alteração nos módulos já existentes.

Notus [Tirelo e Bigonha 2006] é uma linguagem puramente funcional que implementa o modelo de Semântica Multidimensional. Ela oferece suporte à divisão modular das construções, especificação léxica e sintática, especificação dos domínios sintáticos e da estrutura da árvore de sintaxe abstrata, especificação separada das construções e definição de regras de composição das especificações.

3 Metodologia

Para validar a metodologia multidimensional e a linguagem Notus, uma definição semântica incremental de Java é proposta. Essa definição será feita usando três subconjuntos de construções de Java, *Java 0*, *Java 1* e *Java 2*. A sub-linguagem *Java 0* conterá um conjunto inicial de construções de Java que será expandido em *Java 1* e, mais ainda, em Java 2.

Além disso, a definição formal de Java será apresentada em duas partes: a primeira se concentra na verificação de tipo de programas Java, enquanto a segunda descreve a semântica dinâmica de programas Java, incluindo a verificação de tipo dinâmica.

Por Java ser uma linguagem fortemente tipada, a verificação de tipo deve ser realizada para assegurar que operações sem sentido não ocorram durante a execução do programa. Assim a semântica de cada sub-linguagem de Java inclui a semântica para efetuar a verificação de tipo durante a compilação, chamada de semântica estática, e a semântica de execução do programa, chamada de semântica dinâmica.

No POC I foi realizada a definição da sub-linguagem Java 0, enquanto no POC II foram definidas a semântica formal de Java 1 e Java 2.

Para definir a semântica formal de Java 1 e Java 2 foram seguidos os seguites passos:

1. incorporação das novas construções da nova sub-linguagem (Java 1 ou Java 2) à gramática da sub-linguagem já definida. Isso foi feito adicionando novas variáveis e tokens à gramática da sub-linguagem utilizando, sobretudo, a cláusula **extend** de *Notus*;

2. definição da semântica estática das novas construções da sub-linguagem sendo definida. O recurso de transformações de *Notus* muitas vezes auxilia nesse processo;

3. descrição da semântica dinâmica das novas construções da sub-linguagem sendo definida. Novamente o recurso de transformações de *Notus* é utilizado;

Além dessas três tarefas, que visam definir a semântica formal das sub-linguagens especificadas, uma avaliação da metologia multidimensional é realizada ao final desse processo.

4 Java 0

Java 0 é uma linguagem composta por um conjunto de construções básicas de Java. Um programa é definido como uma única classe, que encapsula outras construções. Os comandos consistem basicamente em condicional, repetição, impressão, expressão (em Java pode-se, por exemplo, fazer chamadas a uma função que retorna um tipo, sem atribuir o resultado a alguma variável), declaração e comando de retorno de função. As expressões podem ser uma chamada de função, uma constante literal, um identificador ou uma operação lógica ou aritmética envolvendo outras expressões. As declarações podem ser de função ou de variável e devem ser estáticas. Em Java 0 pode haver definições sobrecarregadas de funções, sendo os parâmetros reais, usados na chamada à função, determinantes na escolha da função a ser executada.

Uma definição completa da semântica formal dessa sub-linguagem utilizando *Notus* foi elaborada no **POC I**. Todavia, durante as definições das sub-linguagens Java 1 e Java 2 foi necessário alterar essa definição inicial, resultando em um novo modelo. Essa nova definição de Java 0 criada em *Notus* encontra-se descrita nas seções seguintes.

4.1 Especificação Semântica de Java 0

4.1.1 Organização dos Módulos

Para definir a semântica de Java 0 foram utilizados vários módulos, distribuídos em quatro pacotes. A figura 1 mostra essa organização.

- **Global**: contém os módulos *Domains*, *Lexico* e *Util* que possuem domínios sintáticos, semânticos e funções utilizados pelos módulos dos demais pacotes. No módulo *Lexico* são especificadas expressões regulares para definir os *tokens* que representam identificadores e constantes em Java 0. Esses módulos não importam módulos de outros pacotes. Os módulos desse pacote são apresentados no Anexo **A**.
- **ConcreteGrammar**: contém os módulos *Commands*, *Declarations*, *Expressions*, *Operators* e *Program*. Nesses módulos são feitas a definição da gramática concreta de Java 0 e a especificação de como a correspondente gramática abstrata deve ser gerada.
- **StaticSemantic**: contém os módulos *Commands*, *Declarations*, *Domains*, *Expressions*, *Functions* e *Operators*. Esses módulos, juntos, definem a semântica de verificação de tipo de Java 0.

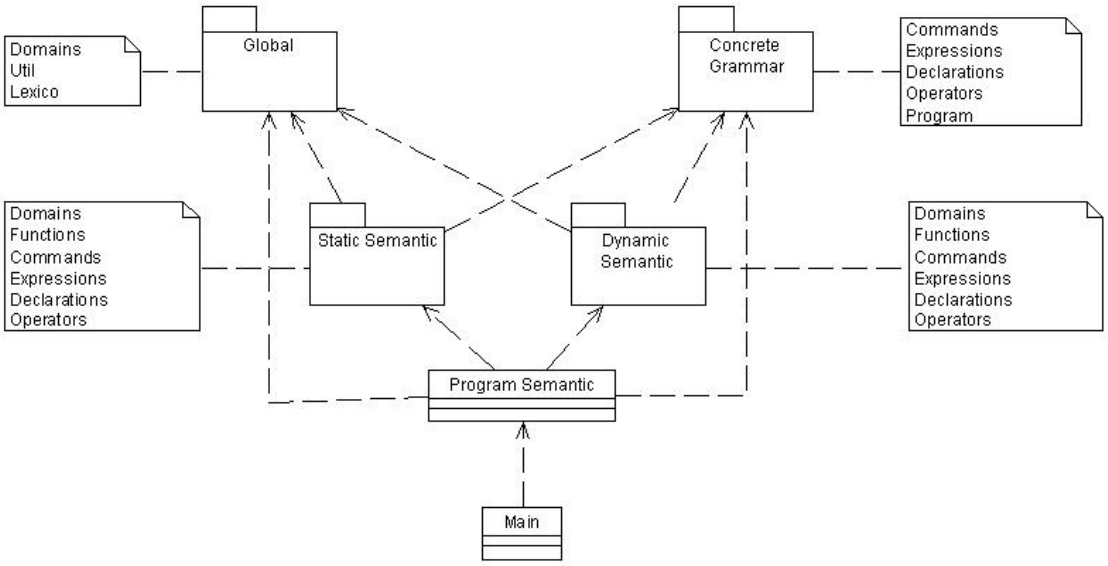

Figura 1: Organização dos Módulos para Definição Semântica de Java 0

- **DynamicSemantic**: contém os módulos *Commands*, *Declarations*, *Domains*, *Expressions*, *Functions* e *Operators*. Esses módulos, juntos, definem a semântica dinâmica de Java 0.

Além dos módulos desses pacotes, existem dois módulos adicionais. O *ProgramSemantic* define a semântica formal de um programa Java 0, que consiste em realizar a verificação de tipo e posteriormente, se não houver erros, executar o programa Java 0. O módulo *Main*, obrigatório na estrutura de módulos especificada por *Notus*, contém funções e declarações para especificar o símbolo inicial da gramática, dado que a definição pode estar espalhada por vários módulos, e o mapeamento de um programa Java 0 no seu significado.

4.1.2 Sintaxe Concreta e Abstrata de Java 0

Pode-se particionar os domínios das construções de Java 0 em **Pro**, **Dec**, **Exp**, **Com**, **Dvar** e **Arg** para programa, declarações de funções e variáveis, expressões, comandos, declaração de uma única variável e argumentos de funções, respectivamente. A cada membro desses domínios serão associadas duas funções semânticas: uma para descrever a verificação de tipo das construções de Java 0 e a outra para definir a semântica dinâmica das construções de Java 0.

Outros domínios sintáticos podem ser identificados na sintaxe concreta de Java 0, como **Id** de identificadores, os domínios **ArrayType**, **BasicType**, **VoidType** e **ReferenceType**, que denotam os tipos arranjo, os tipos básicos (boolean e int), o tipo void e o tipo referência de Java 0, respectivamente. Todos esses domínios de tipo são parte do domínio **Dtype** para tipos que podem ser usados na declaração de variáveis e funções. Os domínios **Int** e **Bool** são os tipos usados para constantes de Java 0.

A sintaxe concreta de Java 0 é mostrada no Anexo **A**. Nela são feitas indicações,

por meio da linguagem *Notus*, para especificar como a sintaxe abstrata correspondente deve ser gerada. Segue a gramática abstrata gerada com base na gramática concreta especificada.

```
Pro ::= ListaClasses

ListaClasses ::= "class" Id Dec
                 "emptyProgram"

Dec ::= Dec Dec
        "emptyDeclaration"
        Modifier Dec
        Dtype Id Com
        Dtype Id Param Com
        Dtype Dvar

Param ::= Param Param
          Dtype Id
          Dtype Id AbreFechaColchetes

Dtype ::= ArrayType
          ReferenceOrBasicType
          VoidType
          "functionType" BasicType

Dvar ::= Dvar Dvar
         Id "=" Exp
         Id AbreFechaColchetes
         Id

ArrayType ::= ReferenceOrBasicType AbreFechaColchetes

ReferenceOrBasicType ::= BasicType
                         ReferenceType

BasicType ::= "int"
              "boolean"

ReferenceType ::= Id

Modifier ::= "public" "static"
             "public"
             "static"
             "emptyModifier"
```

```
Exp ::= Exp "=" Exp
        Exp Opr Exp
        Id "(" Arg ")"
        Id "(" ")"
        Int
        Bool
        Id

Arg ::= Arg Arg
        Exp

Com ::= "emptyCommand"
        Dec
        Com Com
        "if" Exp Com
        "if" Exp Com "else" Com
        "while" Exp Com
        "com" Exp
        "print" Exp
        "emptyCommand"

Opr ::= Logic
        Arithmetic

Arithmetic ::= "+"
               "*"
               "/"
               "-"

Logic ::= ">"
          "<"
          "=="
          "!="
```

4.1.3 Verificação de Tipo de Java 0

4.1.3.1 Visão Geral

A função da verificação de tipo de Java 0 é:

- verificar se todos os identificadores usados são previamente declarados;

- verificar se todos os identificadores são declarados uma única vez dentro do mesmo escopo;
- verificar se são atribuídos às variáveis valores com tipos que concordam com o tipo com o qual foram declarados;
- verificar se as expressões a serem avaliadas em comandos de repetição e condicionais são do tipo *boolean*, conforme a linguagem Java prevê;
- verificar se os parâmetros reais usados na chamada de uma função estão de acordo com os parâmetros formais;
- resolver a sobrecarga de função, permitida em Java;
- verificar se as operações aritméticas e lógicas são aplicadas a operandos que possuem tipos por elas esperados;
- verificar se elementos estáticos de classe, como funções, só acessam elementos estáticos da classe;
- reunir no início de um bloco todas as declarações que são distribuídas por ele no programa fonte;
- colocar informações de controle de acessibilidade e visibilidade coletadas durante a checagem no programa fonte.

A verificação de tipo pode também estender (ou alterar) o texto de um programa Java 0 para simplificar o mapeamento das construções de Java 0 para suas denotações na semântica dinâmica. É possível resolver, em tempo de compilação, chamadas às funções sobrecarregadas. Na verificação de tipo, todas as funções sobrecarregadas recebem um nome distinto, de forma que na segunda fase da definição semântica (semântica dinâmica) não existam funções sobrecarregadas. Para isso, a verificação de tipo de Java 0 pode alterar a gramática abstrata inicial, produzindo uma nova gramática abstrata, sobre a qual a segunda fase da definição irá atuar. De forma semelhante, a semântica estática reuni no início de um bloco todas as declarações espalhadas por ele. Novamente, a semântica estática altera a gramática abstrata de um programa. Esse procedimento facilita a definição da semântica dinâmica, que poderá assumir que todas as declarações de um bloco se encontram no seu início. Por fim, a semântica estática altera o programa fonte modificando os identificadores para que eles carregueminformação para designá-los como estáticos ou não. Essa informação é coletada durante a semântica estática e precisa

ser utilizada durante a semântica dinâmica, sendo necessário trasmití-la de uma etapa a outra. A opção escolhida foi transmití-la através da própria árvore abstrata, dado que a mesma já é modificada na semântica estática para outros fins, como descrito anteriormente. Dessa forma, a verificação de tipo de Java 0 pode ser definida como uma coleção de mapeamentos dos construtores de Java 0 para construtores estendidos de Java 0 ou para um erro de verificação de tipo.

Foi adotada a semântica de continuação na definição da semântica estática de Java 0. Assim, a verificação de tipo ou produz um programa Java 0 estendido, na forma de uma sintaxe abstrata, ou produz uma mensagem de erro. Com isso, as condições de erro podem ser tratadas de uma maneira mais conveniente, evitando poluir a definição com testes por valores inválidos.

Foi modelado um apropriado *environment* de tipo para fazer a verificação de tipo de comandos, declarações e expressões de um programa Java 0. Ele é definido como um mapeamento de identificadores em Java 0 para suas respectivas denotações de tipos.

4.1.3.2 Domínios Semânticos

Seguem os domínios semânticos e denotações definidos para a semântica de verificação de tipo. Eles são definidos no módulo *StaticSemantic.Domains*

```
Env     = Id -> EnvValue;
EnvValue = Unbound | Type;
Type  = SemanticBasicType | SemanticArrayType | Id
                  | FunctionType | TypeError | Tmodifier | AcessEnv;
SemanticBasicType = {void,inteiro,booleano};
SemanticArrayType = Array (Int,Type);
Cc  = Env -> Dec -> Com -> A;
Dc = Env -> Dec -> A;
Ec   = Type -> Env -> Exp -> A;
Ac  = Type* -> Env -> Arg -> A;
A  = Iv | TypeError | DynamicSemantic.Domains.A;
Iv = (Env,Dec);
TypeError = {typeerror};
FunctionType = FunctionValue*;
FunctionValue = Func (FormalParameters,Type,Id);
FormalParameters = Type*;
public Tmodifier = {tpublic, tprivate, tstatic, tprotected, tpackage, tunspecified} ;
AcessEnv  = Id -> Tmodifier*;
```

O domínio **Env** representa uma função que mapeia um identificador de Java 0 para seu tipo semântico, caso o identificador tenha sido declarado, ou para **Unbound**, caso contrário. O **AcessEnv** é também um *environment* que permite recuperar informações sobre acessibilidade dos identificadores. Ele está contido dentro do *environment* principal, **Env**.

O **SemanticBasicType** denota os tipos básicos de Java 0. O **SemanticArrayType** denota um tipo arranjo, que é representado por um par. O primeiro elemento do par denota o número de indireções, ou dimensões do arranjo, e o segundo denota o tipo do array. O tipo **Id** é usado para denotar referências à Classes. Como em Java 0 a noção de objeto ainda não existe, essa modelagem foi bastante simplificada. **TypeError** representa um tipo incorreto, indicando que a verificação de tipo deve ser finalizada. **FunctionType** representa um conjunto de funções. Cada função é descrita pela lista dos tipos de seus parâmetros formais, pelo seu tipo de retorno e pelo novo identificador assumido pela função, caso ela esteje sobrecarregada. **Tmodifier** é um tipo enumerado que descreve os tipos de controle de visibilidade e acessibilidade dos identificadores em Java 0 (private, static, public, package, protected).

Cc, **Dc**, **Ec**, **Ac** denotam, respectivamente, continuação de comando, de declaração, de expressão e de argumentos (ou parâmetros reais). É interessante notar que as funções semânticas desses construtores podem modificar o ambiente de declaração, não sendo essa modificação restrita ao construtor de declaração, como esperado. Isso acontece porque em Java um comando pode ser uma declaração.

O domínio **A** refere-se a resposta final da semântica estática. Devido à uma metodologia utilizada para possibilitar que várias declarações possam ser feitas em um mesmo ambiente e recursivamente, chamada de continuação retrátil, definida em [Bigonha 2003], a resposta final de um programa deve incluir o valor intermediário produzido pela função semântica estática de declaração e que é recebido pela continuação de declaração. A resposta final da semântica estática inclui também, por usar semântica de continuação, a resposta final da semântica dinâmica. Isso porque, após a semântica estática produzir a nova árvore sintática com os tipos verificados, a execução do programa deve iniciar-se. Por fim, o domínio **Typeerror** também faz parte da resposta final e é utilizado se a verificação de tipo encontra algum erro.

4.1.3.3 Funções da Semântica Estática

No módulo *StaticSemantics.Functions* estão definidos dois grupos de funções. O primeiro grupo engloba as funções semânticas que mapeiam objetos do domínio sintático, construções de Java 0, em elementos semânticos. O segundo grupo de funções auxiliam as funções do primeiro grupo no processo de definição semântica.

As funções semânticas são:

```
function dtexp  : Exp -> Env -> Ec -> A;
function dtcom  : Com -> Env -> Cc -> A;
function dtarg  : Arg -> Env -> Ac -> A;
function dtdec  : Dec -> Env -> Dc -> A;
function dtdvar : Dvar -> Type -> Env -> Dc -> A;
function oprOk : Type -> Opr -> Type;
```

A função **dtcom** transforma comandos de Java 0 em novos comandos com tipos verificados, dado o contexto que é definido pelo *environment* de tipo e pela continuação de comando. Em adição pode enriquecer o *environment* de tipo dado.

A função **dtdec**, de forma semelhante, transforma declarações de funções e variáveis de Java 0 em novas declarações com tipos verificados, dado o contexto que é definido pelo *environment* de tipo e pela continuação de declaração. Em adição, enriquece o *environment* de tipo dado.

A função **dtexp** transforma expressões de Java 0 em novas expressões com tipos verificados e obtém seu tipo semântico, dado o contexto que é definido pelo *environment* de tipo e pela continuação de expressão. Em adição, pode enriquecer o *environment* de tipo dado.

A função **dtarg** transforma parâmetros de chamada de Java 0 em novos parâmetros com tipos verificados e obtém os seus tipos semânticos, dado o contexto que é definido pelo *environment* de tipo e pela continuação de argumentos. Em adição, pode enriquecer o *environment* de tipo dado.

A função **dtvar** transforma a declaração de uma variável de Java 0 em uma nova declaração com tipos verificados, dado o contexto que é definido pelo *environment* de tipo, a continuação de declaração e o tipo semântico a que a declaração pertence. Em adição, enriquece o *environment* de tipo dado.

As funções auxiliares são brevemente descritas abaixo, a implementação delas encontra-se no Anexo **A**.

- **cardinality : FunctionType → Int**: retorna o número de funções contidas em uma lista de funções;
- **equalArray : SemanticArrayType → SemanticArrayType → Bool**: verifica se dois tipos semânticos *SemanticArrayType* são iguais;
- **equalType : Type → Type → Bool**: verifica se dois tipos semânticos quaisquer são iguais;
- **isBooleanType : Ec → Type → Env → Exp → A**: verifica se um dado tipo semântico é um tipo básico **booleano**. Se for, continua a verificação de tipo. Caso contrário, retorna um *typeerror*, indicando que a verificação de tipo terminou;
- **isEqualArgs : Type* → Type* → Bool**: verifica se duas listas de tipos semânticos são iguais;
- **isFunctionSetMember : Type* → FunctionType → Bool**: verifica se existe alguma função, em um dado conjunto de funções, que possui os parâmetros formais com tipos correspondentes a uma lista de tipos fornecida;
- **isCompatibleArgs : FormalParameters → Type* → Bool**: verifica se os tipos semânticos dos parâmetros reais são compatíveis com os dos parâmetros formais;
- **newTenv : Env**: cria um novo *environment* de tipo que mapeia, dado qualquer identificador, para o valor semântico *unbound*;
- **subArray : SemanticArrayType → SemanticArrayType → Bool**: verifica se um dado tipo semântica *SemanticArrayType* é subtipo de um segundo tipo semântico *SemanticArrayType*;
- **subType : Type → Type → Bool**: verifica se um dado tipo semântico qualquer é subtipo de outro dado tipo semântico;
- **update : Env → Env → Env**: faz a junção de dois *environments* de tipo, de forma que, para obter o valor associado a um dado identificador no *environment* de tipo resultante, é primeiramente feita um consulta no *environment* de tipo mais interno e, posteriormente, no mais externo;

- **getReturnType : FunctionType → Type* → Type**: essa função implementa um algoritmo para encontrar o tipo semântico de retorno do método correto a ser utilizado em uma chamada a uma função em Java. O algoritmo é melhor descrito na seção seguinte;

- **getInternalName : FunctionType → Type* → Id**: implementa o mesmo algoritmo anterior, entretanto retorna o novo nome interno, que será utilizado na semântica dinâmica do programa Java 0, da função sendo chamada;

- **getCompatibleSet : FunctionType → Type* → FunctionType**: dado um conjunto de funções, retorna um subconjunto dessas que possui parâmetros formais compatíveis com os reais, em tipo e em número. Para verificar a compatibilidade de tipo é utilizada informação de hierarquia de classes (onde um tipo é esperado, seu tipo pode ser aceito);

- **getMoreSpecificSet : FunctionType → Type* → FunctionType**: dado um conjunto de funções, elimina funções menos específicas na hierarquia de classes de acordo com todos os parâmetros. Para isso faz uma chamada à *getMoreSpecificType*;

- **getMoreSpecificType: Type* → Type**: função que, de fato, obtém as funções mais específicas, de acordo com os parâmetros formais, dentre um conjunto de funções e dado o conjunto de parâmetros reais;

- **getFunctions : FunctionType → Type → Int → FunctionType**: retorna um subconjunto de funções cujo tipo semântico do parâmetro formal de posição especificada por *Int* é *Type*;

- **getNesimoTypeforAll : FunctionType → Int → Type***: retorna o tipo semântico do parâmetro formal de posição especificada por *Int*, para todas as funções de um conjunto de funções;

- **getNesimoType : FormalParameters → Int → Type**: retorna o tipo semântico do parâmetro formal de posição especificada por *Int*;

- **intersection : FunctionType → FunctionType → FunctionType**: efetua a interseção de dois conjuntos de funções;

- **typeToString : Type → String**: transforma um tipo semântico básico em uma string que o representa;

- **assignableType : Type → Type → Bool**: verifica se o segundo tipo semântico passado como parâmetro pode ser convertido em Java para o primeiro usando *Widening Primitive Conversions*;

Além dessa funções auxiliares que manipulam domínios semânticos, existem outras funções que auxiliam o processo de definição semântica das construções que manipulam certos domínios sintáticos e, por isso, estão em módulos específicos.

4.1.3.4 Resolução de Sobrecarga de Java 0

Como dito, um dos objetivos da semântica de verificação de tipo de Java 0 é produzir uma nova árvore sintática abstrata em que todas as funções sobrecarregadas (funções que possuem o mesmo nome e que se distinguem pelo número e tipo dos parâmetros formais) tenham nomes distintos. Para isso, foi adicionado ao domínio **Type** o tipo **FunctiontType**, que representa um conjunto de funções. Assim, quando um nome de função é consultado no *environment* de tipo, é retornado o conjunto de funções que possuem o mesmo nome, ou seja, o conjunto de funções sobrecarregadas.

Para tratar o problema, cada função sobrecarregada, ao ser declarada, recebe um novo nome. Quando uma função sobrecarregada é utilizada, o novo nome que ela recebeu é, então, usado para montar um novo nodo da árvore sintática abstrata e passada para a continuação que segue a construção de declaração.

Primeiramente, ao se declarar uma função, segue-se os seguintes passos:

1. Consulta-se o identificador da função no *environment*.
   (a) Se a resposta for o tipo **unbound**, significa que não há outra função com o mesmo nome. Logo, cria-se uma lista de funções, **functionType**, somente com o descritor da função sendo declarada. O seu identificador na árvore sintática abstrata permanece o mesmo;
   (b) Se a resposta for do tipo **functionType**, já existe, pelos menos, uma função com mesmo nome (sobrecarregada). Se a função sendo declarada está no conjunto de funções retornado (tem o mesmo número e tipo de parâmetros formais de alguma função do conjunto), então um erro de verificação de tipo é gerado. Em caso contrário, um novo nome é gerado para aquela função e é armazenado no descritor da função, **functionValue**. Esse novo descritor da função sendo declarada é adicionado ao conjunto de funções sobrecarregadas em relação ao identificador original.

No momento da chamada de uma função, a escolha de qual função sobrecarregada deve ser utilizada envolve um algoritmo mais complexo, descrito em [Arnolde, Gosling e Holmes 2007]. Isso acontece porque em Java pode haver subtipos. Embora Java 0 ainda não suporte essa característica de subtipos (existem apenas tipos inteiro e booleano), o arcabouço foi criado a fim de que a modelagem da semântica se torne menos suceptível a mudanças futuras.

As funções
getReturnType: FunctionType → Type* → Type e **getInternalName: FunctionType → Type* → Id**, com o auxílio de várias outras, se propõem a encontrar o tipo de retorno e o novo nome da função a ser usada na chamada, dado a lista de funções sobrecarregadas e a lista dos tipos dos parâmetros reais.

O algoritmo implementado funciona basicamente da seguinte forma:

1. Primeiramente, é selecionado um conjunto de funções compatíveis, ou seja, funções que possuem argumentos formais compatíveis com os argumentos reais da chamada, em número e tipo. Isso é feito pela função **getCompatibleSet : FunctionType → Type* → FunctionType**, que retorna um subconjunto das funções passadas como parâmetro a ela;

2. Em um segundo momento, desse subconjunto é selecionada a função mais específica segundo todos os parâmetros. Isso é feito pela função **getMoreSpecificSet : FunctionType → Type* → FunctionType**. Essa função procede da seguinte forma:

   (a) Para cada parâmetro real, é selecionada a função (ou conjunto de funções) mais específica, na hierarquia de classes, segundo o dado parâmetro;

   (b) Uma intersecção entre os conjuntos obtidos para cada parâmetro é feita;

   (c) Se existir mais que uma função no conjunto resultante da intersecção, um erro é gerado, pois existe ambiguidade na escolha da função a ser executada. Se o resultado for um conjunto vazio, um erro também é gerado, dado que não existe função declarada compatível com a chamada. Por fim, a verificação de tipo prossegue se somente uma única função restar dessa operação de intersecção. A função, então, a ser executada na chamada é a retornada na interseção. O novo identificador da função, armazenado no seu descritor, é então usado para criar um novo nodo na árvore sintática abstrata.

4.1.3.5 Semântica Estática de Comandos

No módulo *StaticSemantic.Commands* estão as equações semânticas que definem a semântica de verificação de tipo de todas as construções de Java 0 que representam um comando. A função **update: Env → Env → Env** faz a junção de dois *environments* de tipo. A função **subType: Type → Type → Bool** verifica se o primeiro tipo semântico passado como parâmetro é subtipo do segundo tipo semântico passado como parâmetro. A função **joinCom : Com → Com → Com** e **joinDec : Dec → Dec → Dec** agrupam dois comando em um só e duas declarações em uma só, respectivamente, eliminando comandos e declarações vazias. Seguem as equações semânticas.

```
module StaticSemantic.Commands

import ConcreteGrammar.Commands, Global.Domains, Domains,Global.Util,Functions;

dtcom [dec] r c = dtdec dec r (\r1 dec1 -> c r1 dec1 ["emptyCommand"]);

dtcom ["if" exp com] r c =
 evaluate {dtexp exp r; isBooleanType k}
  where {
 k = \t r1 exp1 -> dtcom com r c1
 where {
  c1 = \r2 dec1 com1 -> c newTenv ["emptyDeclaration"] ["if" exp1 com2];
  com2 = [ "{" dec1 com1 "}" ]
 }
 };

dtcom ["if" exp com1 "else" com2] r c = evaluate {dtexp exp r; isBooleanType k}
where {
 k = \t r1 exp1 -> dtcom com1 r c1
  where {
   c1 = \r2 dec11 com11 -> dtcom com2 r c2
    where {
     c2 = \r3 dec21 com21 -> c newTenv ["emptyDeclaration"] ["if" exp1 com12 "else" com22]
     com12 = ["{" dec11 com11 "}"];
     com22 = ["{" dec21 com21 "}"]
    }
  }
};

dtcom ["while" exp com] r c = evaluate {dtexp exp r; isBooleanType k}
where {
 k = \t r1 exp1 -> dtcom com r c1
 where {
  c1 = \r2 dec1 com1 -> c newTenv ["emptyDeclaration"] ["while" exp1 com11];
  com11 = ["{" dec1 com1 "}"]
 }
};

dtcom ["print" exp] r c = dtexp exp r (\t r1 exp1 -> c newTenv ["emptyDeclaration"] ["print" exp1]);
```

```
dtcom ["com" exp] r c = dtexp exp r (\t r1 exp1 -> c newTenv ["emptyDeclaration"] ["com" exp1]);

dtcom [com1 com2] r c = dtcom com1 r c1
where {
 c1 = \r1 dec11 com11 -> dtcom com2 r2 c2
 where {
  r2 = update r r1;
  c2 = \r3 dec21 com21 -> c (update r1 r3) (joinDec dec11 dec21) (joinCom com11 com21) //passa so o que for novo
 }
};

dtcom ["emptyCommand"] r c = c newTenv ["emptyDeclaration"] ["emptyCommand"];

dtcom ["return" exp] r c = dtexp exp r k
where {
 k = \t r1 exp1 ->
    if assignableType t (r -1) is true
    then c newTenv ["emptyDeclaration"] ["return" exp1]
    else typeerror
};

dtcom _ r c = typeerror;

joinCom ["emptyCommand"] dec = dec;
joinCom ["emptyCommand"] [com1 com2] = [com1 com2];
joinCom ["emptyCommand"] ["if" exp com] = ["if" exp com];
joinCom ["emptyCommand"] ["if" exp com "else" com] = ["if" exp com "else" com];
joinCom ["emptyCommand"] ["while" exp com] = ["while" exp com];
joinCom ["emptyCommand"] ["com" exp] = ["com" exp];
joinCom ["emptyCommand"] ["print" exp] = ["print" exp];
joinCom ["emptyCommand"] ["return" exp] = ["return" exp];
joinCom dec ["emptyCommand"] = dec;
joinCom [com1 com2] ["emptyCommand"] = [com1 com2];
joinCom ["if" exp com] ["emptyCommand"] = ["if" exp com];
joinCom ["if" exp com "else" com] ["emptyCommand"] = ["if" exp com "else" com];
joinCom ["while" exp com] ["emptyCommand"] = ["while" exp com];
joinCom ["com" exp] ["emptyCommand"] = ["com" exp];
joinCom ["print" exp] ["emptyCommand"] = ["print" exp];
joinCom ["return" exp] ["emptyCommand"] = ["return" exp];
joinCom ["emptyCommand"] ["emptyCommand"] = ["emptyCommand"];
joinCom com1 com2 = [com1 com2];

end
```

4.1.3.6 Semântica Estática de Declarações

No módulo *StaticSemantic.Declarations* estão as equações semânticas que definem a semântica de verificação de tipo de todas as construções de Java 0 que representam uma declaração de variável ou função. Nesse módulo existe ainda funções que auxiliam no processo de definição, obtendo informações semânticas dos elementos sintáticos. Essas

funções são descritas abaixo:

- **numberOfColchetes : AbreFechaColchetes → Int**: retorna o número de pares de abre e fecha colchetes dado o domínio sintático *AbreFechaColchetes*;
- **getArrayTypeDomain: ArrayType → Type**: retorna o tipo semântico correspondente a um domínio sintático que representa o tipo *ArrayType*;
- **getReferenceTypeDomain : ReferenceType → Type**: retorna o tipo semântico correspondente a um domínio sintático que representa o tipo *ReferenceType*;
- **getBasicTypeDomain : BasicType → Type**: retorna o tipo semântico correspondente a um domínio sintático que representa o tipo *BasicType*;
- **getTypeDomain : ReferenceOrBasicType → Type**: retorna o tipo semântico correspondente um domínio sintático que representa um tipo *ReferenceType* ou *BasicType*;
- **getDeclTypeDomain : Dtype → Type**: retorna o tipo semântico correspondente a um domínio sintático que representa os tipos possíveis que podem ser usados em declaração de funções e variáveis;
- **getTypeParam : Param → Type***: retorna os tipos semânticos dos parâmetros formais de uma função;
- **getModifier : Modifier → Tmodifier**: faz o mapeamento entre o elemento sintático que representa o controle de acesso e visibilidade de um identificador no código fonte em um elemento semântico que descreve esse controle.

Os identificadores reservados **acessEnv** e **visibility** são posições específicas do *environment* de tipo que armazenam informações internas usadas durante a semântica estática. O identificador **acessEnv** é mapeado para o *environment* de controle de acesso que, por sua vez, mapeia identificadores em seus controles de acesso (private, static, private static, public, public static, empty). O identificador **visibility** mapeia para os descritores de controle de visibilidade e acesso do bloco corrente. Seguem as equações semânticas para a verificação de tipo de declarações:

```
module StaticSemantic.Declarations

import Global.Lexico, Global.Domains, Domains, ConcreteGramamar.Declarations,
Global.Util, Functions;
```

```
dtdec ["emptyDeclaration"] r u = u newTenv ["emptyDeclarations"];

dtdec [dec1 dec2] r u =
let {
 (r1,dec11)= case dtdec dec1 (update r r5) (\r2 dec12 -> (r2,dec12)) of {
             iv -> iv;
             };
 (r3,dec21)= case dtdec dec2 (update r r5) (\r4 dec22 -> (r4, dec22)) of {
             iv -> iv;
             };
 r5 = update r1 r3
} in u r5;


//Declaracao de funcao sem parametro
dtdec [dtype id com] r u =
let {
 string = (idToString id) ++ "#";  //necessario para concatenar e depois fazer casting
 id1 = string;
 formalParameters = ();            //lista vazia de parametros
 type = getDeclTypeDomain dtype;
 ar = r "acessEnv";
 m* = r "visibility"
} in case r id1 of {
   unbound -> dtcom com (update r r4) (\r3 dec1 com1 -> u r2 [dtype id com2])
     where {
       com2 = ["{" dec1 com1 "}"];
       functionValue = Func (formalParameters,type,id1);
       ar1 = ar[id<-m*];
       r21 = newTenv[id <- (functionValue)];
       r2 = r21["acessEnv"<-ar1];
       r4 = r2[-1 <- type] //para o return
     };  //Nao tem funcao sobrecarregada. Vai direto para env. Cria lista com unico functionValue
   functionType -> if (isFunctionSetMember () functionType) is true //tem sobrecarga. Vai ter que mudar nome
                   then typeerror //ja tem funcao igual declarada
                   else dtcom com (update r r4) (\r3 dec1 com1 -> u r2 [dtype id1 com2])
                    where {
                     string1 = string ++ (intToString (cardinality functionType));
                     com2 = ["{" dec1 com1 "}"];
                     id2 = string1;
                     functionValue = Func (formalParameters,type,id2);
                     functionType1 = functionValue:functionType; //Adiciona nova funcao ao conjunto de
                     ar1 = ar[id2<-m*];  //funcoes com mesmo nome
                     r21 = newTenv[id1 <- functionType1];
                     r2 = r21["acessEnv"<-ar1];
                     r4 = r2[-1 <- type] //para o return
                    }
  };

dtdec [dtype id param com] r u =
let {
 string = (idToString id) ++ "#";  //necessario para concatenar e depois fazer casting
 id1 = string;
```

```
 formalParameters = getTypeParam param;
 type = getDeclTypeDomain dtype;
 ar = r "acessEnv";
 m* = r "visibility"
} in case r id1 of {
     unbound -> dtcom com (update r r4) (\r3 dec1 com1 -> u r2 [dtype id param com2])
      where {
       com2 = ["{" dec1 com1 "}"];
       functionValue = Func (formalParameters,type,id1);
       ar1 = ar[id1 <- m*];
       r21 = newTenv[id <- (functionValue)];
       r2 = r21["acessEnv"<- ar1];
       r4 = r2[-1 <- type] //para o return
      };  //Nao tem funcao sobrecarregada. Vai direto para env. Cria lista com unico functionValue
     functionType -> if (isFunctionSetMember formalParameters functionType) is true
                     then typeerror //ja tem funcao igual declarada
                     else dtcom com (update r r4) (\r3 com1 dec1 -> u r2 [dtype id1 param com2])
                      where {
                       com2 = ["{" dec1 com1 "}"];
                       string1 = string ++ (intToString (cardinality functionType));
                       id2 = string1;
                       functionValue = Func (formalParameters,type,id2);
                       functionType1 = functionValue:functionType; //Adiciona nova funcao ao conjunto
                       ar1 = ar[id2<-m*];                       //de funcoes com mesmo
                       r21 = newTenv[id1 <- functionType1];
                       r2 = r21["acessEnv"<-ar1];
                       r4 = r2[-1 <- type] //para o return
                      }
};

dtdec [declType dvar] r u = dtdvar dvar (getDeclTypeDomain declType) r (\r1 dvar1 -> u r1 [declType dvar1]);

dtdvar [dvar1 dvar2] t r u = dtdvar dvar1 t r (\r1 dvar11 -> dtdvar dvar2 t (update r r1) (\r2 dvar21 ->
                             u (update r1 r2) [dvar11 dvar12]));

dtdvar [id] t r u =
 case r id of {
  unbound -> u r2 [id]; //passa novo pequeno environment para continuacao
   where {
    ar = r "acessEnv";
    m* = r "visibility";
    ar1 = ar[id <- m*];
    r1 = newTenv[id <- t];
    r2 = r1["acessEnv" <- ar1]
   }
  _  -> typeerror //se variavel ja existe localmente e erro
 };

dtdvar [id "=" exp] t r u =
let {
 r1 = newTenv[id <- t];
 k = \t1 r2 exp1 ->
    if (subType t1 t) is true
    then case r id of {
```

```
          unbound -> u r3 [id "=" exp1]
            where {
              ar = r "acessEnv";
              m* = r "visibility";
              ar1 = ar[id <- m*];
              r3 =  r1["acessEnv" <- ar1]
            };
          _  -> typeerror //se variavel ja existe localmente: erro
         };
    else typeerror
} in dtexp exp r k;

dtdvar [id abreFechaColchetes] t r u =
let {
 int = numberOfColchetes abreFechaColchetes;
 r1 =  case t of {
        Array (int1,t1) -> newTenv[id <- type2]
         where {
          type2 = Array (int1 + int,t1)
         };
        _ -> newTenv[id <- type2]
         where {
          type2 = Array (int,t)
         }
       }
} in case r id of {
      unbound -> u r2 [id abreFechaColchetes]
       where {
        ar = r "acessEnv";
        m* = r "visibility";
        ar1 = ar[id <- m*];
        r2 =  r1["acessEnv" <- ar1]
       }; //passa novo pequeno environment para continuacao
      _  -> typeerror //se variavel ja existe localmente: erro
     };

////////////////////////////////////////////////////////////////////////////////////////
//Obtem o numero de [] (indirecoes)
public function numberOfColchetes : AbreFechaColchetes -> Int;
numberOfColchetes ["[" "]"] = 1;
numberOfColchetes [abreFechaColchetes "[" "]"] = (numberOfColchetes abreFechaColchetes) + 1;


////////////////////////////////////////////////////////////////////////////////////////////////
//Funcao para retornar tipo semantico  de um tipo array    //
//Nao foi testado
function getArrayTypeDomain: ArrayType -> Type;
getArrayTypeDomain [referenceOrBasicType abreFechaColchetes] =
let {
 type = getTypeDomain referenceOrBasicType;
 int  = numberOfColchetes abreFechaColchetes
} in Array(int,type);

////////////////////////////////////////////////////////////////////////////////////////
```

```
//Funcao que obtem dominio semantico, dado o tipo referencia reconhecido na gramatica
//concreta
function getReferenceTypeDomain : ReferenceType -> Type;
getReferenceTypeDomain [id] = [id]; //ERRO: se tirar [], tanto do lado direito, quanto do esquerdo gera erro

////////////////////////////////////////////////////////////////////////////////////////
//Funcao que obtem dominio semantico, dado o tipo basico reconhecido na gramatica concreta
function getBasicTypeDomain : BasicType -> Type;
getBasicTypeDomain ["int"] = inteiro;
getBasicTypeDomain ["boolean"] = booleano;

////////////////////////////////////////////////////////////////////////////////////////
//Funcao que obtem dominio semantico de tipo, sendo ele simples ou referencia
function getTypeDomain : ReferenceOrBasicType -> Type;
getTypeDomain [referenceType] = getReferenceTypeDomain referenceType;
getTypeDomain [basicType] = getBasicTypeDomain basicType;

////////////////////////////////////////////////////////////////////////////////////////
//Funcao que obtem dominio sintatico da declaracao e retorna tipo semantico de declaracao
function getDeclTypeDomain : Dtype -> Type;
getDeclTypeDomain [arrayType] = getArrayTypeDomain arrayType;
getDeclTypeDomain [referenceOrBasicType] = getTypeDomain basicType; //ERRO necessidade do marcador
getDeclTypeDomain [voidType] = void;
getDeclTypeDomain ["functionType" basicType] = getTypeDomain basicType;

////////////////////////////////////////////////////////////////////////////////////////
//Funcao que obtem os tipos dos parametros de uma funcao
function getTypeParam : Param -> Type*;
getTypeParam [param1 param2] = getTypeParam param1 ++ getTypeParam param2;
getTypeParam [dtype id] = (getDeclTypeDomain dtype);
getTypeParam [dtype abreFechaColchetes] =
let {
 type = getDeclTypeDomain dtype;
 int  = numberOfColchetes abreFechaColchetes
}
case type of {
 Array (int1,type1) -> Array (int+int1,type1);
 _         -> Array (int,type);
}

////////////////////////////////////////////////////////////////////////////////////////
//Funcao para eliminar declaracoes vazias geradas na construcao da nova arvore

joinDec ["emptyDeclaration"] [dec1 dec2] = [dec1 dec2];
joinDec ["emptyDeclaration"] [dtype id com] = [dtype id com];
joinDec ["emptyDeclaration"] [dtype id param com] = [dtype id param com];
joinDec ["emptyDeclaration"] [dtype dvar] = [dtype dvar];
joinDec ["emptyDeclaration"] [modifier dec] = [modifier dec];
joinDec [dec1 dec2] ["emptyDeclaration"] = [com1 com2];
joinDec [dtype id com] ["emptyDeclaration"] = ["if" exp com];
joinDec [dtype id param com] ["emptyDeclaration"] = ["if" exp com "else" com];
joinDec [dtype dvar] ["emptyDeclaration"] = ["while" exp com];
joinDec [modifier dec] ["emptyDeclaration"] = ["while" exp com];
joinDec ["emptyDeclaration"] ["emptyDeclaration"] = ["emptyDeclaration"];
```

```
joinDec dec1 dec2 = [dec1 dec2];

function getModifier : Modifier -> Tmodifier;
getModifier ["public"] = tpublic:();
getModifier ["static"] = tstatic:();
getModifier ["public static"] = tstatic:(tpublic:());
getModifier ["emptyModifier"] = tpackage:();

end
```

4.1.3.7 Semântica Estática de Expressões

No módulo *StaticSemantic.Expressions* estão as equações semânticas que definem a semântica de verificação de tipo de todas as construções de Java 0 que representam uma expressão. As funções **getReturnType:FunctionType → Type* → Type** e **getInternalName: FunctionType → Type* → Id**, retornam, respectivamente o tipo de retorno e o novo nome da função sobrecarregada a ser chamada, após ela ser encontrada usando o algoritmo descrito na seção 4.1.3.4.

Como já dito, **acessEnv** e **visibility** são posições reservadas do *environment* de tipo.

Seguem as equações:

```
module StaticSemantic.Expressions

import Global.Domains,Domains,Global.Lexico,ConcreteGrammar.Expressions,
Global.Utils,Functions,ConcreteGrammar.Operators,Operators;

dtexp [int] r k = k inteiro newTenv int;

dtexp [bool] r k = k booleano newTenv bool;

dtexp [id] r k =
let {
 m* = r "visibility";
 ar = r "acessEnv";
 m1* = ar id;
 type = r id;
 id1 = "#" ++ id
} in if type is unbound
     then typeerror
     else if (checkStatic m*) is false
          then k type newTenv id;
          else if (checkStatic m1*) is false
               then k type newTenv id1;
               else typeerror;

dtexp [id "(" ")"] r k =
let {
```

```
 id1 = (idToString id) ++ "#";
 m* = r "visibility";
 ar = r "acessEnv";
 m1* = ar id1
} in case r id1 of {
      functionType -> let {
        t = getReturnType functionType ();
        id2 = getInternalName functionType ();
        id3 = "#" ++ id2
  } in
          if t is typeerror then typeerror
          else if (checkStatic m*) is false
               then k t newTenv [id2 "(" ")"]
               else if (checkStatic m1*) is false
                    then typeerror
                    else k t newTenv [id3 "(" ")"];
      _    -> typeerror
     };

dtexp [id "(" arg ")"] r k =
let {
  p  = \t* r1 arg1 -> let {
id1 = (idToString id) ++ "#";
m* = r "visibility";
ar = r "acessEnv";
m1* = ar id1
  } in case r id1 of {
 functionType ->let {
     t1 = getReturnType functionType t*;
          id2 = getInternalName functionType t*;
          id3 = "#" ++ id2
                      } in if t1 is typeerror then typeerror
                           else if (checkStatic m*) is false
                          then k t1 newTenv [id2 "(" arg1 ")"]
                          else if (checkStatic m1*) is false
             then typeerror
         else k t1 newTenv [id3 "(" arg1 ")"];

               _     -> typeerror
   };
} in dtarg arg r p;

dtexp [exp1 "=" exp2] r k  =  dtexp exp2 r k1
    where {
k1 = \t r1 exp21 -> dtexp exp1 r2 k2
where {
r2 = update r r1;
k2 = \t1 r3 exp11 -> if (assignableType t t1) is true
   then case t of {
void -> typeerror;
_  -> if equalType t t1 is false
        then k t newTenv [exp11 "=" exp22]
           where {
            exp22 = [ id "(" exp21 ")"];
```

```
            id = makeTypeConverterFunctionName t1 t
           }
        else k t newTenv [exp11 "=" exp21]
}
      else typeerror

}
    };

dtexp [exp1 opr exp2] r k = dtexp exp1 r k1
where {
 k1 = \t r1 exp11 -> dtexp exp2 r2 k2
 where {
  r2 = update r r1;
  k2 = \t1 r3 exp21 ->
   if (equalType t t1) is false
   then if assignableType t t1 is true
        then if (oprOk t opr) is void
             then typeerror
             else k t newTenv [exp12 opr exp21]
               where {
                 exp12 = [id "(" exp11 ")"];
                 id = makeTypeConverterFunctionName t1 t
               }
        else if assignableType t1 t is true
             then if (oprOk t1 opr) is void
                  then typeerror
                  else k t1 newTenv [exp11 opr exp22]
                    where {
                     exp22 = [id "(" exp21 ")"];
                     id = makeTypeConverterFunctionName t t1
                    }
             else typeerror
   else -> case t of {
           void -> typeerror;
           _  -> case (oprOk t opr) of {
                  void -> typeerror;
                  t3 -> if equalType t t3 is false
                        then k t1 newTenv [exp12 opr exp22]
                          where {
                            exp22 = [id "(" exp21 ")"];
                            exp12 = [id "(" exp11 ")"];
                            id = makeTypeConverterFunctionName t t3
                          }
                        else k t newTenv [exp11 opr exp21]
                 }
          }
 }
};

dtexp _ r k = typeerror;

dtarg [arg1 arg2] r p =
let {
```

```
 p1 = \t* r1 arg11 -> dtarg arg2 r2 p2
    where {
     r2 = update r r1;
     p2 = \t1* r3 arg21 -> p (t* ++ t1*) newTenv [arg11 arg21]
    }
} in dtarg arg1 r p1;

dtarg [exp] r p = dtexp exp r k
where{
k = \t r1 exp1 -> p (t) newTenv exp1
};
```

4.1.3.8 Funções Para Verificação de Tipo de Operações

O módulo *StaticSemantic.Operators* define uma única função *oprOk*, que verifica se um operador aritmético ou lógico pode ser aplicado a um tipo semântico dado, além de fornecer o tipo semântico resultante da operação, segundo Java.

```
module StaticSemantic.Operators

import ConcreteGramar.Operators, Domains;

oprOk inteiro [arithmetic] = inteiro;
oprOk inteiro [logic] = booleano;
oprOk booleano [logic] = case logic of {
   ["=="] -> booleano;
 ["!="] -> booleano;
_    -> void
 }
oprOk _ _ = void;

end
```

4.1.4 Semântica Dinâmica de Java 0

4.1.4.1 Visão Geral

A semântica de continuação novamente é utilizada para facilitar tratamento de erros durante a execução:

- acesso a variáveis não inicializadas;
- atribuição de objeto de tipo inapropriado à componente de arranjo. Um exemplo desse erro ocorre quando se tem:

```
A a [] = new B [10];
a[0] = new A();
```

sendo B subtipo de A. Esse erro só é possível de ser verificado em tempo de execução.

Java 0 somente trata o primeiro tipo de erro, dado que não há ainda construtores para criação de objeto e nem atribuição de valores à arranjos ou elementos de arrays nessa sub-linguagem de Java.

É importante notar que a semântica de execução de Java 0 atua sobre uma árvore abstrata que estende a árvore abstrata original, fornecida à semântica de verificação de tipo de Java 0. Isso porque, como já foi dito, a semântica de tipo pode modificar a árvore abstrata. Em Java 0, as modificações incluem à resolução de funções sobrecarregadas. Assim, a gramática abstrata resultante da semântica estática, e usada na semântica dinâmica, não contém funções dentro de uma classe Java 0, com nomes iguais. A gramática abstrata resultante da semântica estática de Java 0 inclui também um novo elemento ao domínio sintático **Com**

$$Com ::= "\{" dec\ com "\}".$$

A semântica estática ao reunir todas as declarações de um bloco no seu início cria esse novo elemento que precisa ser tratado adicionalmente na semântica dinâmica.

4.1.4.2 Domínios Semânticos

Seguem os domínios semânticos definidos no módulo *DynamicSemantic.Domains* para a execução de um programa Java 0:

```
Bv         = Bool | Int | Void | String;
Loc        = Int;
Sv         = Rv;
Ev         = Sv | Loc;
Void       = {void};
Dv         = Loc | Fun;
Rv         = Bv;
Env        = Id -> EnvValue;
EnvValue   = Dv | Unbound;
Store      = Loc -> StoreValue;
StoreValue = Sv | Unused;
Cc         = Store -> A;
Ec         = Ev -> Store -> A;
Dc         = Env -> Store -> A;
Aa         = Ev* -> Store -> A;
public A   = Iv | Ans;
Iv         = (Env,Store);
```

```
Fun        = Ec -> Ev* -> Store -> A;
Ans        = AnsValue (Rv,Ans) | {error,stop} ;
File  = String*;
```

Em java 0 existem vários domínios de valores. **Bv**, *Basic Values*, consistem nos dominíos de valores básicos de Java 0, que podem ser **Bool**, **Int**, **String** ou **Void**. **Rv**, *Right Values*, são domínios de valores obtidos ao "derreferenciar"valores que podem resultar de expressões, ou seja, se o valor resultante da expressão for uma posição de memória, o valor passado à continuação é o armazenada na dada posição de memória. **Sv**, *Storable Values*, consistem nos domínios de valores que podem ser armazenados na memória. **Ev**, *Expressible Values*, são valores que podem ser resultados de avaliações de expressões. E, por fim, **Dv**, *Denotable Values*, são os valores que podem estar associados a identificadores no *environment* de execução.

Às construções de Java 0 nomeadas de comandos, expressões, declarações e argumentos são associados os domínios de continuação **Cc**, **Ec**, **Dc** e **Ac**, respectivamente. O *environment* de execução, **Env**, mapeia identificadores de Java 0 e os identificadores estendidos pela definição semântica de checagem de tipo, para resolver o problema de sobrecarga de funções, para **Dv**, caso tenha algo associado ao identificador, ou **Unbound**, em caso contrário. Uma função, denotada por **Fun**, para ser executada, precisa receber uma continuação de expressão, a lista de argumentos (**Ev***) e o estado da memória corrente, **Store**. Um **Store** denota a memória de uma máquina abstrata que mapeia posições de memória em **Sv**, caso à posição esteje associada um valor, ou **Unused**, em caso contrário.

A resposta final do programa, **A**, é uma lista de **Rv**, seguida de **error**, caso haja algum erro durante a execução, ou **stop**, em caso contrário. **A** também pode denotar um valor intermediário, **Iv**, necessário para que as declarações em uma classe possam todas serem feitas em um mesmo ambiente, usando a mesma técnica utilizada para definir a semântica estática de Java 0. **Iv** representa os valores intermediários que podem ser produzidos por uma declaração e que são passados à continuação de declaração.

File consiste de uma lista de strings que pode ser passada como parâmetros de entrada para a execução de um programa Java 0.

4.1.4.3 Funções definidas para a Semântica Dinâmica

No módulo *DynamicSemantic.Functions* estão definidos dois grupos de funções. O primeiro grupo engloba as funções semânticas que mapeiam objetos do domínio sintático,

construções de Java 0 possivelmente estendidas, em elementos semânticos. O segundo grupo de funções auxiliam as funções do primeiro grupo no processo de definição semântica.

As funções semânticas são:

```
function dopr  : Opr -> Rv -> Rv -> Ec -> Store -> A;
function dexp  : Exp -> Env -> Ec -> Store -> A
function dcom  : Com -> Env -> Cc -> Store -> A
function ddec  : Dec -> Env -> Dc -> Store -> A
function darg  : Arg -> Env -> Ac -> Store -> A;
function ddvar  : Dvar -> Env -> Dc -> Store -> A;
function dr    : Exp -> Env -> Ec -> Store -> A;
```

A função **ddec** liga as variáveis declaradas em suas posições de memória, no *environment*. Também liga as funções declaradas em uma classe às suas descrições semânticas. Esse novo pequeno *environment* produzido é passado à continuação de declaração.

A função **dexp** avalia expressões Java 0, na presença de um *environment* e de uma memória, e passa o valor produzido para a continuação.

A função **dr** também avalia expressões Java 0, como **dexp**, entretanto, se o valor resultante da avaliação for um **loc**, o valor que é passado para a continuação é o associado àquela posição na memória.

A função **dcom** modela a execução dos comandos de Java 0. Geralmente atualiza valores na memória. A memória modificada é passada à continuação de comando.

A função **ddvar** liga um variável declarada em sua posição de memória, no *environment*. Esse novo pequeno *environment* produzido é passado à continuação que segue a construção de declaração de variável.

A função **darg** avalia expressões Java 0 que representam os parâmetros reais na chamada a uma função, na presença de um *environment* e de uma memória, e passa a lista de valores resultantes juntamente com a memória, possivelmente modificada, para a continuação.

A função **dopr** aplica as operações aritméticas e lógicas de Java 0 a dois operandos. O valor resultante da operação juntamente com a memória, possivelmente modificada por efeitos colaterais, são passados à continuação de expressão.

As funções auxiliares são descritas abaixo e suas respectivas implementações são mostradas no Anexo **A**.

- **update : Env → Env → Env**: função que junta dois *environments* de execução. O *environment* resultante é formado pela combinação dos dois iniciais, de forma que, se a consulta pelo identificador falha no segundo *environment* passado como parâmetro, uma consulta é feita no primeiro *environment* passado como parâmetro;
- **updateE : Loc → Ec → Ev → Store → A**: função que atualiza uma dada posição de memória com um valor dado;
- **newEnv : Env**: cria um *environment* de execução vazio em que a todo identificador está associado o valor *unbound*;
- **deref : Ec → Ev → Store → A**: função que "derreferencia"um valor do tipo **Ev**, ou seja, se o valor for uma posição de memória, o valor daquela posição de memória é retornado;
- **cont : Ec → Ev → Store → A**: auxilia a função **deref** no processo descrito anteriormente;
- **function isRv : Ec → Ev → A**: se um dado valor é do tipo **Rv** ele é passado à continuação, em caso contrário, **error** é produzido;
- **isBoolEqual : Bool → Bool → Bool**: verifica se dois valores do tipo *Bool* são iguais. Aqui foi implementada uma aritmética booleana para produzir o valor adequado, dado que em *Notus* não existem operadores relacionais para o tipo **Bool**;
- **new : Store → Loc**: encontra uma nova posição de memória que não esteja sendo usada;
- **findNew : Store → Loc → Loc**: auxilia a função **new** no processo descrito anteriormente;
- **isStatic : Id → Bool**: verifica se um identificador é estático. Essa informação está no próprio nome do identificador, que foi alterado na semântica dinâmica para contê-la;
- **get : Id → Id**: retorna identificador retirando o marcador interno que informa se ele é estático ou não.

4.1.4.4 Semântica Dinâmica de Comandos

No módulo *DynamicSemantic.Commands* estão as equações semânticas que definem a semântica dinâmica de todas as construções de Java 0 que representam um comando.

É interessante ressaltar a equação que descreve a semântica de ["{"*dec com*"}"], dado que esse elemento da gramática abstrata foi adicionado pela semântica estática, e, por isso, não há equações que descrevem essa construção nessa primeira etapa da especificação semântica. Seguem as equações:

```
module DynamicSemantic.Commands

import ConcreteGrammar.Commands, Global.Domains, Domains,Util,DynamicSemantic.Functions;

dcom [dec] r c s = ddec dec r (\r1 s1 -> c r1 s1) s;

dcom["if" exp com] r c s=
dr exp r k s
where {
k = \ev s1 ->  if ev
   then dcom com r c s1
   else c s1
};

dcom["if" exp com1 "else" com2] r c s =
dr exp r k s
where {
k = \ev s1 ->  if ev
   then dcom com1 r c s1
   else dcom com2 r c s1
};

dcom["while" exp com] r c s =
dr exp r k s
where {
k = \ev s1 ->  if ev
   then dcom com r c1 s1
    where {
    c1 = \r1 s2 -> dcom ["while" exp com] r c s2
    }
   else c s1
};

dcom["print" exp] r c s = dr exp r k s
where {
k = \ev s1 -> case ev of {rv -> AnsValue(rv,c s1);
  _  -> error})
};

dcom["com" exp] r c s =  dexp exp r k s
where {
k = \ev s1 -> c s1;
};

dcom[com1 com2] r c s = dcom com1 r c1 s
where {
c1 = \s1 -> dcom com2 r c s1
};
```

```
dcom["emptyCommand"] r c s = c s;

dcom["return" exp] r c s = dexp r k s
where {
k = s -1
}

dcom["{" dec com "}"] r c s = ddec [dec] r u s
where{
u = \r1 s1 -> dcom com (update r r1) c s1
}

dcom _ r c = error;
```

4.1.4.5 Semântica Dinâmica de Declarações

No módulo *DynamicSemantic.Declarations* estão as equações semânticas que definem a semântica dinâmica de todas as construções de Java 0 que representam uma declaração de variável ou função. Nesse módulo existem ainda funções que auxiliam no processo de definição, obtendo informações semânticas dos elementos sintáticos. Essas funções são descritas abaixo:

- **buildEnv : Param* → Ev* → Store → (Env,Store)**: função que constrói um pequeno *environment* formado pelos parâmetros formais de uma função, atribuindo a eles os valores passados à função no momento da chamada;
- **getListParam : Param → Param***: transforma a subárvore que descreve os parâmetros formais em uma lista que contém somente as folhas dessa árvore.

Seguem as equações da semântica dinâmica de declarações:

```
module DynamicSemantic.Declarations

import Global.Lexico, Global.Domains, Domains, ConcreteGrammar.Declarations,
Util, Functions;

ddec ["emptyDeclaration"] r u s = u newEnv s;

ddec [dec dec] r u s =
let {
 (r1,s1)= case ddec dec1 r5 (\r2 s2 -> (r2,s2)) of {
          iv -> iv;
          };
 (r3,s3) = case ddec dec2 r5 (\r4 s4 -> (r4, s4)) of {
           iv -> iv;
```

```
          };
 r5 = update r1 r3
} in u r5;

ddec [dtype id com] r u s =
let {
 fun =  \k ev* s1 -> dcom com (update r r1) c s2
  where {
   s2 = s1[-1 <- k];
   string = (idToString id) ++ "#";
   id1 = string;
   r1 = newEnv[id1 <- fun];
   c = k void
  }
}
in u newEnv[id <- fun] s;

ddec [dtype id param com] r u s =
let {
 fun =  \k ev* s1 -> dcom com (update r2 r3) c s3
  where {
   string = (idToString id) ++ "#";
   id1 = string;
   r1 = newEnv[id1 <- fun];
   r2 = update r r1;
   (r3,s2) = buildEnv (getlistParam param) ev* s1;
   s3 = s2[-1 <- k];
   c = k void
 }
}
in dc newEnv[id <- fun] s;

ddec [dtype dvar] r u s = ddvar [dvar] r1 u s;

ddvar [dvar1 dvar2] r u s = ddvar1 r u1 s
where {
 u1 = \r1 s1 -> ddvar dvar2 r2 u2 s1
 where {
  r2 = update r r1;
  u2 = \r3 s2 = u (update r1 r3) s2
 }
};

dvar [id] r u s = u (r1[id <- loc]) (s[loc <- unused])
 where {
  loc = new s;
  r1 = newEnv
 }
;

dvar [id "=" exp] r u s = dexp exp r k s
where {
 k = \ev s1 -> case ev of {
               sv -> u (r1[id <- loc]) (s1[loc <- sv])
```

```
                where {
                 r1 = newEnv
                 loc = new s1
                }
               }
};

dvar [id abreFechaColchetes] r u s = u (r1[id <- loc]) (s[loc <- unused])
where {
 loc = new s;
 r1 = newEnv
};

////////////////////////////////////////////////////////////////////////////////////////////
// Constroi pequeno environment com parametros formais e seus valores, passados na chamada da funcao

function buildEnv : Param* -> Ev* -> Store -> (Env,Store);
buildEnv [dtype id]:param* ev:ev1* s = (r2,s2)
where {
 (r1,s1) = buildEnv param* ev1* s;
 loc = new s1;
 s2 = s1[loc <- ev];
 r2 = r1[id <- loc]
}

buildEnv [dtype id abreFechaColchetes]:param* ev:ev1* s = (r2,s2)
where {
 (r1,s1) = buildEnv param* ev1* s;
 loc = new s1;
 s2 = s1[loc <- ev];
 r2 = r1[id <- loc]
}

buildEnv () () s = (newEnv,s);

////////////////////////////////////////////////////////////////////////////////////////////
//Transforma subarvore do tipo Param em lista de elementos do tipo Param

function getListParam : Param -> Param*;
getListParam [param1 param2] = (getListParam param1) ++ (getListParam param2);
getListParam [dtype id] = ([dtype id]);
getListParam [dtype id abreFechaColchetes] = (dtype id abreFechaColchetes);

end
```

4.1.4.6 Semântica Dinâmica de Expressões

No módulo *DynamicSemantic.Expressions* estão as equações semânticas que definem a semântica dinâmica de todas as construções de Java 0 que representam uma expressão. A função **updateE: Loc → Ec → Ev → Store → A** atualiza uma dada posição de memória com um dado valor. Seguem as equações:

```
module DynamicSemantic.Expressions

import Global.Domains,Domains,Global.Lexico,ConcreteGrammar.Expressions,Utils,
Functions,ConcreteGrammar.Operators,Operators;

dexp [int] r k s = k int s;

dexp [bool] r k s = k bool s;

dexp [id] r k s = case (r id) of {
                   loc -> k loc s;
                   _ -> error
                  };

dexp [id "(" ")"] r k s =  dexp [id] r k1 s
where {
 k1 = \ev s1 -> case ev of {
                 fun -> fun k2 () s1
                 where {
                  k3 = s -1;
                  k2 = \ev1 s2 -> k ev1 (s2[-1 <- k3])
                 };
                 _  -> error
                }
};

dexp [id "(" arg ")"] r k s = dexp [id] r k1 s
where {
 k1 = \ev s1 -> case ev of {
                 fun -> darg arg r p s1
                 where {
                  p = \ev1* s2 -> fun k2 ev1* s2
                  where {
                   k3 = s -1;
                   k2 = \ev2 s3 -> k ev2 (s3[-1 <- k3])
                  }
                 };
       _ -> error
        }
};

dexp [exp1 "=" exp2] r k s = dexp exp1 r k1 s
where {
 k1 = \ev s1 -> case ev of {
                 loc -> dr exp2 r k2 s1
                 where {
                  k2 = \ev1 s2 -> updateE loc k ev1 s2
                 }
       _ -> error
                }
};

dexp [exp1 opr exp2] r k s = dr exp1 r k1 s
where {
```

```
 k1 = \ev1 s1 -> dr exp2 r k2 s2
  where {
   k2 = \ev2 s2 -> case ev1 of {
                     rv1 -> case ev2 of {
                            rv2 -> dopr opr rv1 rv2 k s2
                            }
                   }
  }
};

dexp _ r k s = error;

dr exp r k s = dexp exp r k1 s
where {
k1 = \ev s1 -> deref k2 ev s1
 where {
  k2 = \ev1 s2 -> isRv k ev1
 }
};

darg [arg1 arg2] r p s =
let {
 p1 = \ev* s1 -> darg arg2 r p2 s1
 where {
  p2 = \ev1* s2-> p (ev* ++ ev1*) s2
 }
} in darg arg1 r p1 s

darg [exp] r p s = dr exp r k s
where{
 k = \ev s1 -> p (ev) s1
};

end
```

4.1.4.7 Semântica Dinâmica de Operações

O módulo *DynamicSemantic.Operators* possui a definição de uma equação semântica para a função semântica **dopr**, que forncece a semântica dos operadores de Java 0, quando aplicados aos valores semânticos. Segue a equação semântica:

```
module DynamicSemantic.Operators

import ConcreteGrammar.Operators, Domains, Functions;

dopr ["+"] int1 int2 k s = k (int1 + int2) s;
dopr ["-"] int1 int2 k s = k (int1 - int2) s;
dopr ["*"] int1 int2 k s = k (int1 * int2) s;
dopr ["<"] int1 int2 k s = k (int1 < int2) s;
dopr ["=="] int1 int2 k s = k (int1 = int2) s;
dopr ["!="] int1 int2 k s = k (int1 != int2) s;
```

```
dopr ["/"] int1 int2 k s = k (int1 / int2) s;
dopr ["=="] bool1 bool2 k s = k (isBoolEqual bool1 bool2) s;
dopr ["!="] bool1 bool2 k s = if isBoolEqual bool1 bool2 is true
  then k false s
  else k true s;
```

4.1.5 Semântica de Java 0

A semântica completa de Java 0 é descrita no módulo **ProgramSemantic**. Nele há a declaração da função **dprog : Pro → File → DynamicSemantic.Domains.A** que recebe o elemento sintático **Pro**, um programa Java 0, a lista de parâmetros de entrada e produz a resposta final de um programa Java 0. A resposta da semântica de um programa Java 0 pode ser um erro na verificação de tipos, um erro durante a execução do programa ou um valor (inteiro ou booleano) _ou lista de valores_, produzido durante a execução do programa. Essa função chama **dclass** que, por sua vez, chama as funções de verificação de tipos dos elementos sintáticos que constituem um programa e, posteriormente, passa o resultado produzido nessa primeira etapa, um árvore sintática modificada, para as funções que proveêm a semântica dinâmica dos elementos dessa nova árvore. A função **newTenv** produz um *environment* de tipo em que para toda entrada ou nome é retornado o valor *unbound*. Como um programa Java 0 consiste em uma única classe com suas declarações, a semântica de execução irá, primeiramente, declarar todas as funções e variáveis da classe. Em seguida, verifica-se se a função *main* foi declarada. Caso sim, a execução inicia por essa função. Caso contrário, um erro é produzido.

```
module ProgramSemantic

import ConcreteGrammar.Program, StaticSemantic.Functions, DynamicSemantic.Functions,
ConcreteGrammar.Declarations, Global.Domains, DynamicSemantic.Domains, StaticSemantic.Domains;

  public function dpro : Pro -> Input -> DynamicSemantic.Domains.A;
dpro listaclasses i = dclass listaclasses i;

public function dclass : ListaClasses -> File -> A;

dclasse ["class" id dec] r i = dtdec dec newTenv (\r1 dec1 -> ddec dec1 newEnv u s)
  where {
    s = initialStore;
    u = \r2 s1 -> case r2 "main" of {
                    fun -> fun k i s1
                    where {
                    k = \ev s2 -> stop;
                    };
                    _ -> error
                  }
);
```

```
dclass ["emptyProgram"] = error;

function initialStore : Loc -> StoreValue;
initialStore loc = unused;
```

5 *Java 1*

Java 1 é uma sub-linguagem de Java que contém todas as construções de Java 0, além de novas construções de Java. Java 1 possui todos os tipos primitivos permitidos por Java, **integer, long, float, double, char, byte** e **boolean**, diferente de Java 0 que só possui os tipos **inteiro** e **boolean**. Java 1 também permite a manipulação (declaração, inicialização e acesso) de arranjos de tipos básicos. Os comandos **switch**, **break** e **continue** de Java também foram adicionados a essa nova sub-linguagem.

5.1 Extensões em Notus

Para possibilitar a extensão de Java 0 foram utilizados os seguintes recursos da linguagem *Notus*: transformações e a cláusula **extends**.

O primeiro permite alterar assinaturas de funções definidas em Java 0. Seguem as construções da linguagem *Notus* utilizadas para esse fim:

1. **signature**: permite adicionar novos argumentos ao cabeçalho das funções;
2. **default**: permite especificar valores iniciais para os novos argumentos definidos com a cláusula **signature**;
3. **redefine**: permite redefinir o corpo de uma declaração de função.

A cláusula **extends** permite estender *tokens*, variáveis da gramática e domínios especificados em uma definição anterior.

5.2 Organização dos Módulos

Para definir a semântica de Java 1, além da importação dos módulos utilizados para definir Java 0, foram definidos vários novos módulos, distribuídos em três pacotes. A figura 2 mostra essa organização.

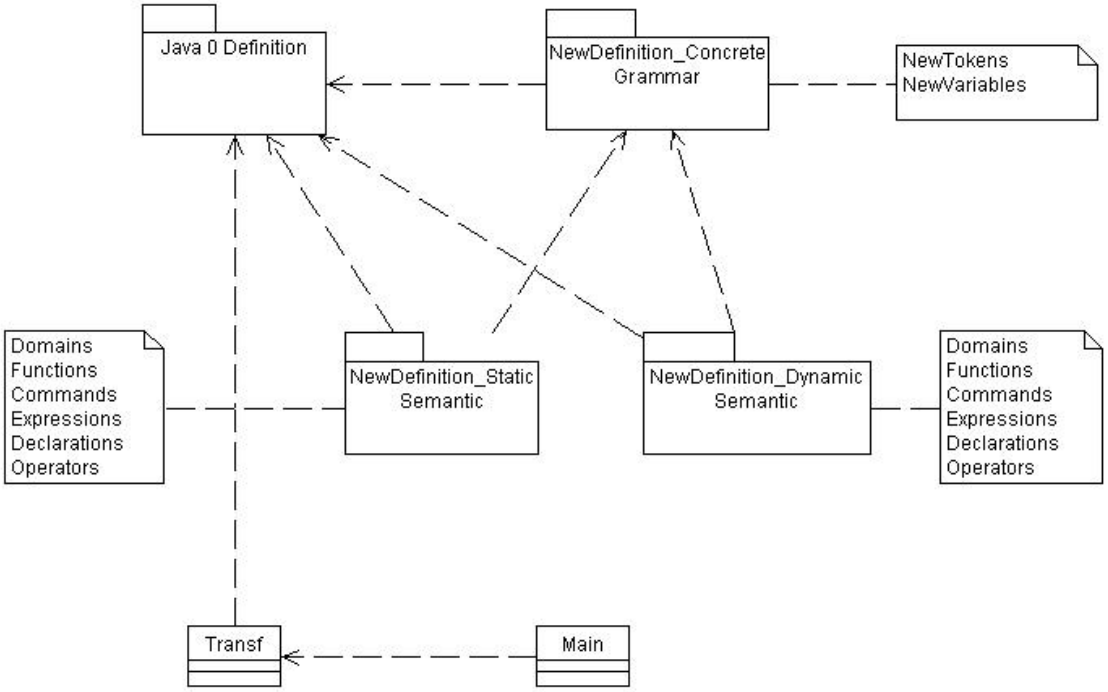

Figura 2: Organização dos Módulos para Definição Semântica de Java 1

- **NewDefinitons_ConcreteGrammar**: contém os módulos *NewTokens* e *NewVariables*. Nesses módulos são feitas novas definições e extensões (a partir de Java 0) de *tokens* e variáveis da gramática concreta para que Java 1 suporte as novas construções descritas no início na seção 5.

- **NewDefinitios_StaticSemantic**: contém os módulos *Commands*, *Declarations*, *Domains*, *Expressions*, *Functions* e *Operators*. Esses módulos, juntos, definem a semântica de verificação das novas construções de Java 1.

- **NewDefinitions_DynamicSemantic**: contém os módulos *Commands*, *Declarations*, *Domains*, *Expressions*, *Functions* e *Operators*. Esses módulos, juntos, definem a semântica dinâmica das novas construções de Java 1.

Além dos módulos desses pacotes, existem dois módulos adicionais. O *Transf* define os transformadores que deverão ser aplicados a definição semântica de Java 0. O módulo *Main*, obrigatório na estrutura de módulos especificada por *Notus*, contém a especificação de qual definição anterior será transformada (no caso, a de Java 0) e qual o nome do módulo onde os transformadores são definidos (Transf).

5.3 Especificação Semântica de Java 1

5.3.1 Sintaxe Concreta e Abstrata de Java 1

Alguns novos elementos foram adicionados aos domínios sintáticos **Dvar**, **BasicType**, **Com** e **Exp**, já existentes em Java 0. A esses novos elementos serão associadas duas equações semânticas em Java 1, uma para descrever a verificação de tipo e a outra para definir a semântica dinâmica. As declarações das funções, entretanto, que descrevem os tipos dos argumentos das equações já foram feitas em Java 0.

Por outro lado, novos domínios sintáticos são definidos em Java 1. Esses novos domínios são **ArrayInit** e **SwitchBlock**, para inicialização de arranjos e para bloco de *switch*, que contém cláusulas *case* e *default*, respectivamente. Novas funções semânticas precisam ser declaradas para tratar esses novos domínios sintáticos.

As adições na sintaxe concreta de Java 0, feitas pela definição de Java 1 podem ser vistas no Anexo **B**. Segue a gramática abstrata gerada com base na gramática concreta especificada em Java 1.

```
Dvar ::= Id AbreFechaColchetes "=" ArrayInit
         Id "=" ArrayInit

BasicType ::= "byte"
              "short"
              "char"
              "long"
              "float"
              "double"

Com ::=  "break" Id
         "break"
         "continue" Id
         "continue"
         "switch" Exp SwitchBlock
         Id ":" Com

Exp ::= Exp "[" Exp "]"
        Long
        Character
        Float
        Double

ArrayInit ::= "{" ArrayInit "}"
              ArrayInit ArrayInit
              Exp

SwitchBlock ::= SwitchBlock SwitchBlock
```

```
"case" Exp ":" Com
"default" ":" Com
```

5.3.2 Verificação de Tipo de Java 1

5.3.2.1 Visão Geral

A verificação de tipo de Java 1 tem os mesmos objetivos da verificação de tipo de Java 0, além de incluir:

- garantir que a inicialização de arranjo seja compatível em dimensões e tipo com a declaração do arranjo;
- garantir que os comandos *break* e *continue* estejam associados a alvos;
- garantir que o tipo da expressão de um comando *switch* seja do tipo esperado por Java;
- garantir que os tipos das expressões nas cláusulas *case* de um bloco *switch* sejam de tipos compatíveis ao da expressão no comando *switch*.

Como em Java 0, a semântica de verificação de tipo de Java 1 pode alterar a gramática abstrata inicial e, portanto, o programa fonte a ela fornecido. A semântica estática de Java 1 também utiliza o conceito de continuação, como em Java 0.

5.3.2.2 Domínios Semânticos

Seguem os domínios semânticos e denotações para definir a semântica estática das construções de Java 1. Eles são definidos no módulo *NewDefinitions_StaticSemantic.Domains*.

```
extend Type with Label;
extend SemanticBasicType with {mybyte, myshort, mylong, myfloat, mydouble, mychar};
Label = {label};
Sc = Env -> SwitchBlock -> TypeCheckAns;
Xc = Int -> Env -> ArrayInit -> TypeCheckAns;
```

A cláusula *extends* de *Notus* foi utilizada para adicionar ao domínio semântico **Type**, que denota os tipos associados aos identificadores e às expressões, o tipo **Label**, que denota um identificador de alvo dos comandos *break id* e *continue id*. O domínio **SemanticBasicType** de Java 0 também é estendido para denotar todos os possíveis tipos básicos de Java.

Os domínios **Sc** e **Xc** denotam, respectivamente, continuação de um bloco de *switch* e continuação de inicializador de *array*.

5.3.2.3 Funções da Semântica Estática

No módulo *NewDefinitions_StaticSemantic.Functions* estão definidos dois grupos de funções. O primeiro grupo engloba as funções semânticas que mapeiam objetos do domínio sintático, construções de Java 1, em elementos semânticos. O segundo grupo de funções auxiliam o primeiro nesse processo. Além disso, nesse módulo existem definições de funções declaradas em Java 0, que fornecem o significado das funções quando casadas com as novas definições de Java 1.

As novas funções semânticas de Java 1 são:

```
function dtsw : SwitchBlock -> Env -> Sc -> Int -> Int -> Type -> TypeCheckAns;
function dtarrayInit : ArrayInit -> Env -> Xc -> Type -> TypeCheckAns;
```

A função **dtsw** transforma blocos de *switch*, que incluem cláusulas *case* e *default*, em novos blocos com os tipos verificados, dado o contexto, que é definido pelo *environment* de tipo e pelos níveis de aninhamento para comandos *break* e *continue*, a continuação de um bloco *switch* e o tipo da expressão no comando *switch*.

A função **dtarrayInit**, por sua vez, transforma inicializações de *arrays* em novas inicializações com os tipos verificados, dado o contexto definido pelo *environment* de tipo, a continuação de inicialização e o tipo do *array* sendo inicializado.

A nova função auxiliar para verificação de tipo é **arrayTypeResult : Type → Type**, que retorna o tipo final de um acesso a elemento de *array*. A implementação dela pode ser vista no Anexo **B**.

5.3.2.4 Transformadores para Verificação de Tipo

No módulo *Transf* são definidos algumas transformações aplicadas às funções e equações semânticas de Java 0 para verificação de tipos. Seguem essas transformações:

```
signature StaticSemantic.Functions.dtcom :
Com -> StaticSemantic.Domains.Env -> StaticSemantic.Domains.Cc
-> (nivel1: Int) -> (nivel2: Int)-> TypeCheckAns;

default nivel1 = 0,nivel2 = 0;

redefine  dtcom ["while" exp com] r c n1 n2 by
```

```
        evaluate {dtexp exp r; isBooleanType k}
        where {
        k = \t r1 exp1 -> dtcom com r2 c1 (n1+1) n2
       where {
      r2 = update r r1;
      c1 = \r3 com1 -> c newTenv ["while" exp1 com1] n1 n2
       }
     };
```

Com a adição dos comandos *break* e *continue*, é preciso que a equação semântica para verificação de tipos de comandos incorpore a acumulação de níveis de *while* e *switch*, o que permite verificar se o comando de mudança de fluxo é interno a pelo menos um *while* ou *switch*, no caso de *break*, ou a pelo menos um *while* no caso de um *continue*. Dessa forma, dois novos parâmetros foram adicionados à função **dtcom**, que verifica tipos de comandos, usando o transformador *signature*. O primeiro, controla o nível de aninhamento de comandos *while* e o segundo de comandos *switch*.

Também foi preciso redefinir a equação semântica do comando *while* para que ele atualize os níveis de aninhamento.

5.3.2.5 Semântica Estática de Comandos

No módulo *NewDefinitions_StaticSemantic.Commands* estão as equações semânticas para verificação de tipo das construções de Java 1 que representam um comando ou uma cláusula de bloco *switch*(*case* ou *default*).

```
dtcom ["break" id] r c n1 n2 = if r id is label
                               then c newTenv ["emptyDeclaration"] ["break" id]
                               else typeerror;

dtcom ["continue" id] r c n1 n2 = if r id is label
                                  then c newTenv ["emptyDeclaration"] ["break" id]
                                  else typeerror;

dtcom ["continue"] r c n1 n2 = if n1 = 0
                               then typeerror
                               else c newTenv ["emptyDeclaration"] ["break" id];

dtcom ["break"] r c n1 n2 = if n1 = 0 and n2 = 0 //sai de switch, for, while, do
                            then typeerror
                            else c newTenv ["emptyDeclaration"] ["break" id];

dtcom ["switch" exp switchBlock] r c n1 n2 = dtexp exp r k1
where {
 k1 = \type r1 exp1 -> case type of {
                         mychar -> dtsw switchBlock r w n1 (n2+1) type
                          where {
```

```
                          w = \r2 switchBlock1 -> c newTenv ["emptyDeclaration"] ["switch" exp1 switchBlock1]
                         };
                        mybyte -> dtsw switchBlock r w n1 (n2+1) type
                         where {
                          w = \r2 switchBlock1 -> c newTenv ["emptyDeclaration"] ["switch" exp1 switchBlock1]
                         };
                        myshort -> dtsw switchBlock r w n1 (n2+1) type
                         where {
                          w = \r2 switchBlock1 -> c newTenv ["emptyDeclaration"] ["switch" exp1 switchBlock1]
                         };
                        inteiro -> dtsw switchBlock r w n1 (n2+1) type
                         where {
                          w = \r2 switchBlock1 -> c newTenv ["emptyDeclaration"] ["switch" exp1 switchBlock1]
                         };
                        _ -> typeerror
                      }
};

dtcom [id ":" com] r c n1 n2 =
let {
 string = (idToString id) ++ "##";  //necessario para concatenar e depois fazer casting
 id1 = string;
 r1 = r[id1 <- label]);
 c1 = \r2 dec1 com1 -> c newTenv ["emptyDeclaration"] [id ":" com11];
 com11 = ["{" dec1 com1 "}"]

} in if r id1 is unbound  //Ja tem outro label com mesmo nome
     then dtcom com r1 c1 n1 n2
     else typeerror

dtsw ["case" exp ":" com] r w n1 n2 type = dtexp exp r k
where {
 k = \type1 r1 exp1 -> if (assignableType type type1) is true
                       then dtcom com r c n1 n2
                             where {
                              id = makeTypeConverterFunctionName type1 type;
                              exp2 = [ id "(" exp1 ")"];
                              c = \r2 dec1 com1 -> w newTenv ["emptyDeclaration"] ["case" exp2 ":" com11];
                              com11 = ["{" dec1 com1 "}"]

                             }
                       else typeerror
};

dtsw ["default" ":" com] r w n1 n2 type =
dtcom com r c n1 n2
 where {
  c = \r1 dec1 com1 -> w newTenv ["default" ":" com11];
  com11 = ["{" dec1 com1 "}"]
 };

dtsw [switchBlock1 switchBlock2] r w n1 n2 type = dtsw switchBlock1 r w1 n1 n2 type
where {
 w1 = \r1 switchBlock11 -> dtsw switchBlock2 r w2 n1 n2 type
```

```
   where {
    w2 =\r2 switchBlock21 -> w newTenv [switchBlock11 switchBlock21] n1 n2
   }
};
```

5.3.2.6 Semântica Estática de Declarações

No módulo *NewDefinitions_StaticSemantic.Declarations* se encontram as equações semânticas para verificação de tipo das novas construções de Java 1 que representam declarações ou estão a elas associadas, como no caso de equações para inicialização de *arrays*. Os identificadores **acessEnv** e **visibility** são posições específicas no *environment* de tipo, como já descrito anteriormente.

```
dtdvar [id abrefechacolchetes "=" arrayInit] Array(int1,type1) r u = dtarrayInit arrayInit r x type1
where {
 x = \int r1 arrayInit1 -> let {
                int2 = int1 + (numberOfColchetes abreFechaColchetes)
                } in if r id is unbound
                     then if int = int2
                          then u r4 [id "=" arrayInit1]
                            where {
                             r2 = newTenv;
                             type2 = Array(int2,type1);
                             ar = r "acessEnv";
                             m* = r "visibility";
                             ar1 = ar[id <- m*];
                             r3 = r2[id<-type2];
                             r4 =  r3["acessEnv" <- ar1]
                            }
                           else typeerror
                     else typeerror
};

dtdvar [id abrefechacolchetes "=" arrayInit] type r u = dtarrayInit arrayInit r x type
where {
 x = \int r1 arrayInit1 -> let {
             int = numberOfColchetes abreFechaColchetes
             } in
                   if r id is unbound
                   then if int = int1
                        then u r4 [id abreFechaColchetes "=" arrayInit1]
                             where {
                              r2 = newTenv;
                              type1 = Array (int1,type);
                              ar = r "acessEnv";
                              m* = r "visibility";
                              ar1 = ar[id <- m*];
                              r3 = r2[id<-type1];
                              r4 =  r3["acessEnv" <- ar1]
                             }
                        else typeerror
                   else typeerror
};
```

```
dtdvar [id "=" arrayInit] Array(int,type) r u = dtarrayInit arrayInit r x type
where {
x = \int r1 arrayInit1 ->
              if r id is unbound
  then if int = int1
       then u r4 [id "=" arrayInit1]
           where {
            type1=Array(int,type);
            r2 = newTenv;
            ar = r "acessEnv";
       m* = r "visibility";
       ar1 = ar[id <- m*];
       r3 = r2[id<-type];
       r4 =  r3["acessEnv" <- ar1]
          }
                     else typeerror
                else typeerror
};

dtarrayInit [ "{" arrayInit "}" ] r x type =  dtarrayInit arrayInit r x1 type
 where {
   x1 = \int r1 arrayInit1 -> x (int+1) newTenv ["{" arrayInit1 "}"]
 }
};

dtarrayInit [ arrayInit1 arrayInit2 ] r x type =  dtarrayInit arrayInit1 r x1 type
 where {
   x1 = \int r1 arrayInit11 -> dtarrayInit arrayInit2 r2 x2 type
    where {
      x2 = \int1 r3 arrayInit21 ->
                      if int = int1
          then x int newTenv [arrayInit11 arrayInit21]
          else typeerror
    }
 }
};

dtarrayInit [exp] r x type = dtexp exp r k
where {
  k = \type1 r1 exp1 -> if assignableType type type1 is true
                        then x 0 newTenv arrayInit
                             where {
                               arrayInit = exp1
                             }
                        else typeerror
};
```

5.3.2.7 Semântica Estática de Expressões

No módulo *NewDefinitions_StaticSemantic.Expressions* estão as equações semânticas para a checagem de tipo das novas expressões de Java 1. A função **arrayTypeResult:**

Type → Type, devolve o tipo resultante de um acesso a um *array*. Seguem as novas equações.

```
dtexp [long] r k = k mylong newTenv long;
dtexp [char] r k = k mychar newTenv char;
dtexp [float] r k = k myfloat newTenv float;
dtexp [double] r k = k mydouble newTenv double;
dtexp [ exp1 "[" exp2 "]" ] r k = dtexp exp1 newTenv k1
 where {
  k1 = \type r1 exp11 -> case type of {
                           semanticArrayType -> dtexp exp2 r2 k2
                            where {
                                r2 = update r r1;
                               k2 = \type1 r3 exp21 ->
                               case type1 of {
                                  myshort -> k (arrayTypeResult semanticArrayType) newTenv [ exp11 "[" exp22 "]" ]
                                    where {
                                     exp22 = [ id "(" exp21 ")"];
                                     id = makeTypeConverterFunctionName myshort inteiro
                                    };
                                  mybyte  -> k (arrayTypeResult semanticArrayType) newTenv [ exp11 "[" exp22 "]" ]
                                    where {
                                     exp22 = [ id "(" exp21 ")"];
                                     id = makeTypeConverterFunctionName mybyte inteiro
                                    };
                                  mychar  -> k (arrayTypeResult semanticArrayType) newTenv [ exp11 "[" exp22 "]" ]
                                    where {
                                     exp22 = [ id "(" exp21 ")"];
                                     id = makeTypeConverterFunctionName mychar inteiro;
                                    };
                                  inteiro -> k (arrayTypeResult semanticArrayType) newTenv [ exp11 "[" exp21 "]" ];
                                  _       -> typeerror
                                }
                            };
                           -   -> typeerror
                         }

};
```

É importante destacar as conversões *Widening Primitive* feitas. Essas conversões necessárias são adicionadas à árvore sintática, através da chamada de funções *built-in*, que devem ser inicialmente carregadas no *environment* de execução.

5.3.2.8 Funções para Verificação de Tipos de Operações

O módulo *NewDefinitions_StaticSemantic.Operators* define novos casos da função **oprOk**, declarada em Java 0, para verificar se um dado tipo semântico pode ser aplicado ao operador especificado. A função também retorna o tipo resultante ao aplicar a operação.

```
oprOk myshort [arithmetic] = myshort;
oprOk mylong [arithmetic] = mylong;
oprOk mychar [arithmetic] = inteiro;
oprOk mydouble [arithmetic] = mydouble;
oprOk myfloat [arithmetic] = myfloat;
oprOk mybyte [arithmetic] = mybyte;
oprOk mybyte [logic] = booleano;
oprOk myfloat [logic] = booleano;
oprOk mylong [logic] = booleano;
oprOk myshort [logic] = booleano;
oprOk mydouble [logic] = booleano;
oprOk mychar [logic] = booleano;
```

Ressalta-se que antes de aplicar a operação aos operando, uma conversão *Widening Primitive* é realizada, de forma que o operador é aplicado à expressões de mesmo tipo.

5.3.3 Semântica Dinâmica de Java 1

A semântica de execução de Java 1 estende e modifica algumas definições feitas na semântica de execução das construções de Java 0, além de adicionar novas definições. Como em Java 0, a semântica de execução atua sobre uma árvore abstrata que estende a gramática abstrata original, fornecida à semântica de verificação de tipo de Java 1.

A semântica dinâmica de Java 1 também tem a função de detectar alguns erros de execução, como por exemplo, a indexação de arranjos fora do limite.

5.3.3.1 Domínios Semânticos

Seguem os domínios semânticos e denotações para definir a semântica de execução das construções de Java 1. Eles são definidos no módulo *NewDefinitions_DynamicSemantic.Domains*.

```
extend Rv with ArrayDescriptor;
extend Bv with Char | Double | Float | Long | Byte;
extend Dv with Cc;
Byte = Int;
ArrayDescriptor = (Loc,Int,Loc*);
Xc = Loc* -> Int -> Store -> A;
```

Uma entidade que denota um *array*, **ArrayDescriptor**, foi adicionada a **Rv**, domínio de valores obtidos ao "derreferenciar"valores que podem resultar de uma expressão. Um **ArrayDescriptor** é representado por uma posição na memória de execução onde se encontra o descritor da classe dinâmica do *array*, pelo número de dimensões do *array* e pelas localizações de cada elemento do *array* na memória.

Novas denotações foram adicionadas ao domínio **Bv** para incorporar novos tipos básicos de Java à Java 1. Além disso, uma continuação de comando foi incluída ao **Dv**, valores que podem estar associados a identificadores no *environment* de execução. Essa necessidade é decorrente da adição dos comandos *break id* e *continue id* à Java 1. Assim, ao encontrar essas construções, é preciso mudar o fluxo de execução para um alvo específico.

O domínio semântico **Xc** denota a continuação de execução para inicialização de arranjo.

5.3.3.2 Funções Definidas para a Semântica Dinâmica

No módulo *NewDefinitions_DynamicSemantic.Functions* estão definidas as novas funções semânticas e funções auxiliares que descrevem o significado das construções de Java 1.

As funções semânticas são:

```
function dsw : SwitchBlock -> Env -> DynamicSemantic.Domains.Cc ->  DynamicSemantic.Domains.Cc
               -> DynamicSemantic.Domains.Cc -> Ev -> Store -> A;
function darrayInit : ArrayInit -> Env -> Xc -> Store -> A;
function j : Com -> Env -> Cc -> Cc -> Cc -> Env;
```

A função **dsw** dá o significado de executar um bloco *switch* que contém cláusulas *case* e *default*. Para isso, recebe o *environment* de execução, a continuação do fluxo normal, as continuações para *break* e *continue*, respectivamente, o valor da expressão do comando *switch* que precede o bloco e a memória de execução. A memória possivelmente modificada é passada à continuação de comando a ser seguida.

A função **darrayInit** produz o significado de executar uma inicialização de *array* na presença do *environment* de execução, da continuação de uma inicialização de *array* e da memória de execução. A memória modificada é passada à continuação.

Por fim, a função **j** coleta todos os alvos de comandos *break id* e *continue id* dentro de um dado bloco de comando, armazenando-os no *environment* de execução e associando-os às respectivas continuações a serem seguidas.

Seguem breves descrições das funções auxiliares utilizadas:

- **getLoc : Loc* → Int → Loc**: obtém o i-ésimo *loc* de uma lista de *loc's*, em que *i* é o primeiro inteiro passado como parâmetro;

- **function getLoc2 : Loc* → Int → Int → Loc**: auxilia função anterior no processo de obter o i-ésimo *loc* de uma lista de *loc's*;
- **function evToInt : Ev → Int**: faz conversão de um valor *Expressible Value* para um valor inteiro;
- **startDynamicEnv : Env**: cria um *environment* de execução inicial com funções padrões para conversões de tipo *Widening Primitive* e que foram adicionadas ao programa fonte original pela semântica estática, quando a necessidade de tais conversões foi identificada. Parte da especificação dessa função é deixada em aberto.

5.3.3.3 Transformadores para Semântica Dinâmica

No módulo *Transf* são definidas algumas transformações aplicadas às funções e equações semânticas de Java 0 para execução. Seguem essas transformações:

```
signature DynamicSemantic.Functions.dcom : Com -> DynamicSemantic.Domains.Env -> DynamicSemantic.Domains.Cc
    -> (contBreak : DynamicSemantic.Domains.Cc) -> (contContinue : DynamicSemantic.Domains.Cc) -> Store -> A;

default contBreak = (\s -> error), contContinue = (\s -> error);

redefine

dcom ["while" exp com] r c c1 c2 s by
dr exp r k s
where {
k = \ev s1 ->  if ev
                  then dcom com r c3 c4 c3 s1 //muda continuacao de break e switch
                  where {
                   c3 = \s2 -> dcom ["while" exp com] r c c1 c2 s2;
                   c4 = \s2 -> c s2; //se for break sai fora
                  }
                  else c s1
},

dcom["{" dec com "}"] r c c1 c2 s by
 ddec [dec] r u s
  where{
   u = \r1 s1 -> dcom com (update r2 r3) c c1 c2 s1;
    where {
    r2 = update r r1;
    r3 = j com (update r2 r3) c c1 c2
    }
 },

dclasse ["class" id dec] r i by
dtdec dec newTenv (\r1 dec1 -> ddec dec1 r0 u s)
 where {
  s = initialStore;
```

```
  u = \r2 s1 -> case r2 "main" of {
                   fun -> fun k i s1
                   where {
                    k = \ev s2 -> stop;
                   };
                _ -> error
                }
  r0 = startDynamicEnv
);
```

O transformador *signature* adiciona dois novos parâmetros à função **dcom**, que descreve o significado dos comandos. Os dois novos parâmetros são a continuação dos comandos **break** e **continue**, respectivamente.

Além disso, o transformador **redefine** também é utilizado para redefinir equações da semântica dinâmica especificadas em Java 0. A semântica de execução do comando *while* é alterada para que os alvos dos comandos *break* e *continue* sejam especificados. A semântica da construção de um bloco, "{" *dec com* "}" introduzida pela semântica de verificação de tipos de Java 0, também é alterada para que o comando que segue a declaração seja executado em um *environment* que já incorpora os alvos para os comandos *break id* e *continue id*, e que devem ser coletados pela função **j**, definida em Java 1. Por fim, a equação que descreve o significado de uma classe é também alterada para que a semântica de execução da classe tenha um *environment* inicial, diferente de vazio, com todas as funções internas necessárias para conversão de tipos *Widening Primitive* e que foram adicionadas ao programa fonte original pela semântica estática, quando a necessidade de tais conversões foi identificada.

5.3.3.4 Semântica Dinâmica de Comandos

No módulo *NewDefinitions_DynamicSemantic.Commands* estão as equações semânticas que definem a semântica de execução das novas construções de Java 1 que são comandos. Além disso, estão também nesse módulo as equações semânticas para a coleta de alvos para os comandos *break id* e *continue id* e as equações semânticas para blocos *switch*, compostos por cláusulas *case* e *default*. Seguem as equações:

```
j [com1 com2] r c c1 c2 = update r1 r2
where {
 r1 = j com1 r c3 c1 c2;
  where {
   c3 = \s -> dcom com2 r c c1 c2
  }
 r2 = j com2 r c c1 c2 (update r r1)
```

```
}

j [id ":" com] r c c1 c2 = r1[id1<-c]
where{
 r1 = j [com] r c c1 c2;
 string = idToString(id) ++ "##";
 id1 = string;
 c = \s -> dcom com r c c1 c2 s
}

j _ r = newEnv;

dcom ["break" id] r c c1 c2 s = if r id is c3
                                then c3 s
                                else error;

dcom ["continue" id] r c c1 c2 s = if r id is c3
                                   then c3 s
                                   else error;

dcom ["continue"] r c c1 c2 s = c2 s;

dcom ["break"] r c c1 c2 s  = c1 s;

dcom ["switch" exp switchBlock] r c c1 c2 s = dr exp r k1 s
where {
 k1 = \ev s1 -> dsw switchBlock r c c c2 ev s1 //muda so continuacao de break
};

dcom [id ":" com] r c c1 c2 s = dcom com r c c1 c2 s;

dsw ["case" exp ":" com] r c c1 c2 ev s = dr exp r k s
where {
 k = \ev1 s1 -> if ((evToInt ev)=(evToInt ev1))
                then dcom com r c c1 c2 s1
                else c s1
};

dsw ["default" ":" com] r c c1 c2 ev s = dcom com r c c1 c2 s;

dsw [switchBlock1 switchBlock2] r c c1 c2 ev s = dsw switchBlock1 r c3 c1 c2 ev s
where {
  c3 = \s1 -> dsw switchBlock2 r c c1 c2 ev s1
}
```

5.3.3.5 Semântica Dinâmica de Declarações

No módulo *NewDefinitions_DynamicSemantic.Declarations* estão as equações semânticas para descrever o significado das construções novas de Java 1 que representam declarações ou inicialização de *arrays*. **locationClassDescriptor** é uma posição específica do *en-*

vironment que mapeia para a posição do descritor da classe do arranjo que está sendo criado.

Seguem as equações:

```
ddvar [id abrefechacolchetes "=" arrayInit] r u s = darrayInit arrayInit r x s
where {
x = \loc* n s1 -> u r2[id<-loc2] s1
                  where {
   loc2 = case loc* of {
           loc3:loc4* -> loc3;
           _          -> error
          };
   r2 = newEnv
     }
};

ddvar [id "=" arrayInit] r u s = darrayInit arrayInit r x loc1 s
where {
x = \loc* n s1 -> u r2[id<-loc2] s1
                 where {
   loc2 = case loc* of {
           loc3:loc4* -> loc3;
           _          -> error
          };
   r2 = newEnv
     };
};


darrayInit [ "{" arrayInit "}" ] r x loc1 s =  darrayInit arrayInit r x1 loc1 s
 where {
   x1 = \loc* n s1 -> x loc2:() 1 s2
   where {
    sv = ArrayDescriptor (loc1,n,loc*);
  loc2 = new s1;
  s2 = s1[loc2<-sv];
  loc1 = r "locationClassDescriptor"
   }
};

darrayInit [ arrayInit1 arrayInit2 ] r x loc2 s =  darrayInit arrayInit1 r x1 loc2 s
 where {
   x1 = \loc* n s1 -> darrayInit arrayInit2 r x2 loc2 s1
    where {
          x2 = \loc1* n1 s2 -> x (loc* ++ loc1*) (n+n1) s2
    }
 }
};

darrayInit [exp] r x loc2 s = dr exp r k s
where {
  k = \ev s1 -> x loc* 1 s2
  loc* = loc1:();
```

```
  loc1 = new s1;
  s2 = s1[loc1<-ev]
};
```

5.3.3.6 Semântica Dinâmica de Expressões

O módulo *NewDefinitions_DynamicSemantic.Expressions* contém as equações de execução para as novas construções de Java 1 que são expressões. A função **getLoc: Loc* → Int → Int** retorna o i-ésimo *loc* da lista, onde o *i* é o primeiro parâmetro inteiro.

```
dexp [long] r k s = k long s;
dexp [char] r k s = k char s;
dexp [float] r k s = k float s;
dexp [double] r k s = k double s;
dexp [ exp1 "[" exp2 "]" ] r k s = dexp exp1 r k1 s
where {
k1 = \ev s1 -> case ev of {
       loc -> case s1 loc of {
             ArrayDescritor (loc1,int,loc2*) -> dr exp2 r k2 s1
                                 where {
                                 k2 = \ev1 s2 -> case ev of {
                                         int1 -> if int1 < 0 or int1 >= int
                                                 then error
                                                 else k (s2 (getLoc loc2* int1)) s2
                                         _ -> error;
                                         }
                                 }
             _ -> error
             };
        _ -> error
};
```

5.3.3.7 Semântica Dinâmica de Operações

No módulo *NewDefinitions_DynamicSemantic.Operators* estão novas definições para a função **dopr: Opr → Rv → Rv → Ec → Store → A** declarada em Java 1, que fornece a semântica dinâmica de operações.

```
dopr ["+"] float1 float2 k s = k (float1 + float2) s;
dopr ["-"] float1 float2 k s = k (float1 - float2) s;
dopr ["*"] float1 float2 k s = k (float1 * float2) s;
dopr ["<"] float1 float2 k s = k (float1 < float2) s;
dopr [">"] float1 float2 k s = k (float1 > float2) s;
dopr ["=="] float1 float2 k s = k (float1 = float2) s;
dopr ["!="] float1 float2 k s = k (float1 != float2) s;
dopr ["/"] float1 float2 k s = k (float1 / float2) s;
```

```
dopr ["+"] double1 double2 k s = k (double1 + double2) s;
dopr ["-"] double1 double2 k s = k (double1 - double2) s;
dopr ["*"] double1 double2 k s = k (double1 * double2) s;
dopr ["<"] double1 double2 k s = k (double1 < double2) s;
dopr [">"] double1 double2 k s = k (double1 > double2) s;
dopr ["=="] double1 double2 k s = k (double1 = double2) s;
dopr ["!="] double1 double2 k s = k (double1 != double2) s;
dopr ["/"] double1 double2 k s = k (double1 / double2) s;

dopr ["+"] long1 long2 k s = k (long1 + long2) s;
dopr ["-"] long1 long2 k s = k (long1 - long2) s;
dopr ["*"] long1 long2 k s = k (long1 * long2) s;
dopr ["<"] long1 long2 k s = k (long1 < long2) s;
dopr [">"] long1 long2 k s = k (long1 > long2) s;
dopr ["=="] long1 long2 k s = k (long1 = long2) s;
dopr ["!="] long1 long2 k s = k (long1 != long2) s;
dopr ["/"] long1 long2 k s = k (long1 / long2) s;

dopr ["+"] byte1 byte2 k s = k (addByte byte1 byte2) s; //especificacao deixada em aberto s;
dopr ["-"] byte1 byte2 k s = k (subByte byte1 byte2) s; //especificacao deixada em aberto s;
dopr ["*"] byte1 byte2 k s = k (multByte byte1 byte2) s;
dopr ["<"] byte1 byte2 k s = k (lessThanByte byte1 byte2) s;
dopr [">"] byte1 byte2 k s = k (greaterThenByte byte1 byte2) s;
dopr ["=="] byte1 byte2 k s = k (equalByte byte1 byte2) s;
dopr ["!="] byte1 byte2 k s = k (difByte byte1 byte2) s;
dopr ["/"] byte1 byte2 k s = k (divByte byte1 byte2) s;
```

6 Java 2

Java 2 é uma sub-linguagem de Java que contém todas as construções de Java 1, além de novas construções de Java. Em Java 1 e Java 2 uma classe é apenas um elemento para encapsular um programa. Já em Java 2 a noção de classe é bem mais próxima da de Java: podem existir várias classes e pode-se criar instâncias dessas classes através do construtor **new**, incorporando, assim, o conceito de objeto. Novas expressões para acesso de campos e funções dos objetos são definidas. Java 2 permite também a manipulação (criação, inicialização e acesso) de arranjos cujos tipos são classes.

6.1 Organização dos Módulos

Para definir a semântica de Java 2, além da importação dos módulos utilizados para definir Java 0 e Java 1, foram definidos vários novos módulos, distribuídos em três pacotes. A figura 3 mostra essa organização.

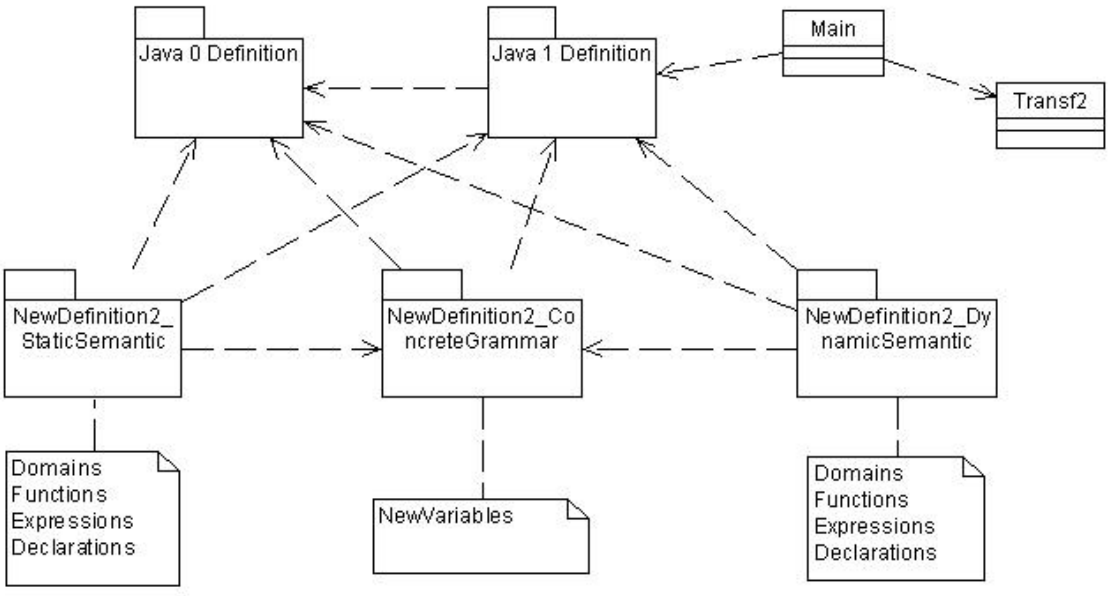

Figura 3: Organização dos Módulos para Definição Semântica de Java 2

- **NewDefinitions2_ConcreteGrammar**: contém o módulo *NewVariables* que faz novas definições e extensões de variáveis da gramática concreta para que Java 2 suporte as novas construções descritas no início dessa seção.

- **NewDefinitions2_StaticSemantic**: contém os módulos *Declarations*, *Domains*, *Expressions* e *Functions*. Esses módulos, juntos, definem a semântica de verificação das novas construções de Java 2.

- **NewDefinitions2_DynamicSemantic**: contém os módulos *Declarations*, *Domains*, *Expressions* e *Functions*. Esses módulos, juntos, definem a semântica dinâmica das novas construções de Java 2.

Além dos módulos desses pacotes, existem três módulos adicionais. O *Transf2* define os transformadores que deverão ser aplicados a definição semântica de Java 1. O módulo *Main*, obrigatório na estrutura de módulos especificada por *Notus*, contém a especificação de qual definição anterior será transformada (no caso, a de Java 1) e qual o nome do módulo onde os transformadores são definidos (Transf2). Por fim, o módulo *NewProgramSemantic* que define novas equações e funções semânticas para descrever o significado de um programa Java 2.

6.2 Especificação Semântica de Java 2

6.2.1 Sintaxe Concreta e Abstrata de Java 2

Alguns novos elementos foram adicionados aos domínios sintáticos **ListaClasses**, **Dvar**, **Exp**, **ReferenceType** e **Modifier** já existentes em Java 1. Além disso, foram criados os domínios sintáticos **Dimentions** para as dimensões quando um arranjo é inicializado com o operador *new*, **SpecifiedDimentions** para as dimensões que possuem um tamanho espeficificado por uma expressão e **Reference** para a criação de objetos ou arranjo de objetos.

As adições na gramática concreta de Java 1 feitas na definição de Java 2 podem ser vistas no Anexo **C**. Segue a gramática abstrata gerada com base na gramática concreta especificada em Java 2:

```
ListaClasses ::= ListaClasses Classe

ReferenceType ::= "this"
```

```
Modifier ::= "private"
             "private static"

Exp ::= "new" Id Dimentions
        "new" BasicType Dimentions
        "new" Id "(" Arg ")"
        "new" Id "(" ")"
        ReferenceType "." Id
        ReferenceType "." Id "(" Arg ")"
        ReferenceType "." Id "(" ")"

Dimentions ::= SpecifiedDimentions AbreFechaColchetes
               SpecifiedDimentions
               AbreFechaColchetes ArrayInit

SpecifiedDimentions ::= SpecifiedDimentions "[" Exp "]"
                        "[" Exp "]"
```

A semântica estática de Java 2 altera essa gramática abstrata original, para anotar informações necessárias a segunda fase da definição semântica, como a resolução de sobrecarga de construtores de classe. Segue a gramática abstrata resultante da verificação de tipos:

```
ListaClasses ::= ListaClasses Classe

ReferenceType ::= "this"

Modifier ::= "private"
             "private static"

Exp ::= "new" Id Id Dimentions
        "new" BasicType Dimentions
        "new" Id "(" Arg ")"
        "new" Id "(" ")"
        ReferenceType "." Id
        ReferenceType "." Id "(" Arg ")"
        ReferenceType "." Id "(" ")"

Dimentions ::= SpecifiedDimentions AbreFechaColchetes
               SpecifiedDimentions
               AbreFechaColchetes ArrayInit

SpecifiedDimentions ::= SpecifiedDimentions "[" Exp "]"
                        "[" Exp "]"
```

6.2.2 Verificação de Tipo de Java 2

6.2.2.1 Visão Geral

A verificação de tipo de Java 2 objetiva-se principalmente a:

- vericar se funções estáticas de classes acessam somente elementos estáticos da respectiva classe;
- anotar na árvore sintático se os identificadores são estáticos ou não. Essa informação é necessária durante a semântica dinâmica.

A semântica estática de Java 2 altera a gramática abstrata inicial, como discutido na seção 6.2.1. Ela também usa o conceito de continuação, como nas sub-linguagens anteriores.

6.2.2.2 Domínios Semânticos

Seguem os domínios semânticos e denotações para definir a semântica estática das construções de Java 2. Eles são definidos nos módulos *NewDefinitions2.StaticDomains*.

```
extend Type with ClassType;
extend Tmodifier with tprivate;
Tc = Int -> Env -> DimentionsOrSpecifiedDimentions -> TypeCheckAns; //y
ClassType = (Env,AcessEnv);
DimentionsOrSpecifiedDimentions = Dimentions | SpecifiedDimentions;
```

O domínio **ClassType**, para denotar classes, foi adicionado ao domínio **Type**, que denotam tipos na semântica estática. O elemento **tprivate** também foi adicionado ao domínio **Tmodifier**, que denota visibilidade e acessibilidade de identificadores. O domínio **ClassType** definido possui um par a ele associado: um *environment* que mapeia identificadores em tipos e um *environment* que mapeia identificadores em controles de acesso e visibilidade. O domínio semântico **DimentionsOrSpecifiedDimentions** denota o domínio sintático **Dimentions** ou o domínio sintático **SpecifiedDimentions**. O domínio **Tc** denota continuação de uma construção para especificar as dimensões de um array sendo inicializado com o operador *new*.

6.2.2.3 Funções da Semântica Estática

No módulo *NewDefinitions2_StaticSemantic.Functions* estão definidas as funções semânticas que mapeiam objetos sintáticos em elementos semânticos, além de outras funções que auxiliam nesse processo.

A função semântica **dtdim : DimentionsOrSpecifiedDimentions → Env → Tc → Type → TypeCheckAns** transforma dimensões de arranjos em novas dimensões

com os tipos verificados, dado o contexto (que é definido pelo *environment* e pelo tipo do arranjo), e a continuação que segue essa construção.

As funções que auxiliam são:

- **getTypeToId:Type → Id**: faz conversão de elemento no super-domínio **Type** para o sub-domínio **Id**;
- **getClassName:Id → Id**: obtém a representação interna do nome de uma classe. Essa representação é obtida adicionando ao final do identificador os marcadores "###";
- **getConstructorName:Env → ReferenceType → Id**: obtém a representação interna de nome de função. Essa representação é obtida adicionando ao final do identificador o marcador "#";
- **getIdName:Env → ReferenceType → Id**: retorna o nome do identificador associado ao tipo **ReferenceType**;
- **checkPrivate:Tmodifier* → Bool**: verifica se na lista de descritores de visibilidade e controle de acesso existe o descritor **tprivate**.

6.2.2.4 Semântica Estática de Declarações

A semântica das novas construções para declarações são definidas no módulo *NewDefinitions2_StaticSemantic.Declarations*. Os identificadores **acessEnv** e **visibility** são posições reservadas do *environment* de tipos que mapeiam, respectivamente, para um *environment* de modificadores e para uma lista dos modificadores correntes.

```
dtdvar [id abreFechaColchete "=" exp] Array(int,type) r u = dtexp exp r k
where{
  k = \type1 r1 exp1 =
    let {
      type2 = Array(int+int2,type);
      int2 = numberOfColchetes abreFechaColchetes
    } in if subType type1 type2 is true
         then if r id is unbound
              then u r4 [id abreFechaColchete "=" exp1]
              where {
                 ar = r "acessEnv";
                 m* = r "visibility";
                 ar1 = ar[id <- m*];
                 r3 = newTenv[id<-type2];
                 r4 =  r3["acessEnv" <- ar1]
```

```
             }
             else typeerror
        else typeerror
};

dtdvar [id abreFechaColchete "=" exp] type r u = dtexp exp r k
where{
  k = \type1 r1 exp1 =
    let {
      type2 = Array(int,type);
      int2 = numberOfColchetes abreFechaColchetes
    } in if subType type1 type2 is true
         then if r id is unbound
              then u r4 [id abreFechaColchete "=" exp1]
                where {
                  ar = r "acessEnv";
                  m* = r "visibility";
                  ar1 = ar[id <- m*];
                  r3 = newTenv[id<-type2];
                  r4 =  r3["acessEnv" <- ar1]
                }
              else typeerror
        else typeerror
};
```

6.2.2.5 Semântica Estática de Expressões

No módulo *NewDefinitions2_StaticSemantic.Expressions* se encontram as equações semânticas para definir a semântica estática das novas expressões e das construções de dimensões para inicializar arranjos com operador *new* de Java 2. O identificador **scope** é uma posição reservada do *environment* de tipo que mapeia para o nome da classe corrente. A função **checkStatic** verifica se a denotação **tstatic** se encontra em uma lista de denotações de modificadores. A função **checkAcess** verifica se um dado componente de uma classe pode ser acessado dentro de uma classe especificada.

```
dtexp ["this" "." id ] r k = if r id is unbound
       then typeerror
                            else k (r id) [id1]
                            where{
                            id2 = r "scope";
                            ClassType (r1,acessEnv) = r id2;
                            id1 = if checkStatic (acessEnv id) is true
                                  then "#" ++ (idToString id) //marca estatico
                                  else id
                            };

dtexp ["this" "." id "(" ")"] r k = let {
id1 = (idToString id) ++ "#"; //para diferencia funcao de variavel
m* = r "visibility";
```

```
ar = r "acessEnv";
m1* = ar id1
} in case r id1 of {
functionType -> let {
                     t = getReturnType functionType ();
                     id2 = getInternalName functionType ();
                     id3 = "#" ++ id2
                     } in
                     if t is typeerror then typeerror
                     else if (checkStatic m*) is false
                          then k t newTenv ["this" "." id2 "(" ")"]
                          else if (checkStatic m1*) is false
                               then typeerror
                               else k t newTenv ["this" "." id3 "(" ")"];
       _    -> typeerror
 };


dtexp ["this" "." id "(" arg ")"] r k =
let {
  p  = \t* r1 arg1 -> let {
            id1 = (idToString id) ++ "#";
            m* = r "visibility";
            ar = r "acessEnv";
            m1* = ar id1
            } in case r id1 of {
                          functionType ->let {
                                   t1 = getReturnType functionType t*;
                                   id2 = getInternalName functionType t*;
                                   id3 = "#" ++ id2
                                   } in if t1 is typeerror then typeerror
                                        else if (checkStatic m*) is false
                                             then k t1 newTenv ["this" "." id2 "(" arg1 ")"]
                                             else if (checkStatic m1*) is false
                                                  then typeerror
                                                  else k t1 newTenv ["this" "." id3 "(" arg1 ")"];
                              _     -> typeerror
                };
} in dtarg arg r p;


dtexp["new" id dimentions] r k =
let {
x = \int r1 dimentions2 -> k type (update r r1) ["new" id dimentions2];
id = getClassName r referenceType;
type = Array(int,id)
} in if r id is unbound
   then typeerror
   else dtdim dimentions r x id ;


dtexp["new" basicType dimentions] r k =
let {
x = \int r1 dimentions2 -> k type (update r r1) ["new" basicType dimentions2];
```

```
type = Array(int,type1);
type1 = getBasicTypeDomain basicType
} in if r id is unbound
   then typeerror
   else dtdim dimentions r x type1;

dtexp["new" id0 "(" arg ")"] r k =
let {
id = getClassName r id0;
id1 = getConstructorName r id0;
id2 = r "scope"
p  = \t* r1 arg1 -> case r id of {
     (env,acessEnv) -> case r id1 of {
     functionType ->
      let {
           t1 = getReturnType functionType t*;
          id3 = getInternalName functionType t*
           } in if t1 is typeerror
             then typeerror
             else if checkAcess acessEnv id3 (r id2) id 1 is true
                then k id newTenv ["new" id0 id3 "(" arg1 ")"];
                else typeerror;
         _ -> typeerror
        };
           _ -> typeerror
    }
} in dtarg arg r p;


dtexp["new" id0 "(" ")"] r k =
let {
id = getClassName r id0;
id1 = getConstructorName r id0;
id2 = r "scope"
} in case r id of {
     (r1,acessEnv) ->  case r id1 of {
     functionType ->
      let {
           t1 = getReturnType functionType ();
          id3 = getInternalName functionType ()
           } in if t1 is typeerror
           then typeerror
           else if checkAcess acessEnv id3 (r id2) id 1 is true
              then k id newTenv ["new" id0 id3 "(" ")"];
              else typeerror;
          _ -> typeerror
        };
           _ -> typeerror
    };

dtexp[id "." id0] r k =
let {
  id1 = getIdName r id;
id2 = getClassName r id
```

```
} in
case r id1 of {
    //E objeto
(r1,acessEnv) -> if checkAcess acessEnv id0 (r "scope") id2 0 is true
  then dtexp [id0] r1 k1
where {
 k1 = \type r2 exp1 -> k type newTenv [id "." exp1]
}
  else typeerror;
       //E acesso estatico de classe
 _  -> case r id2 of {
   (r1,acessEnv) -> if checkAcess acessEnv id0 (r "scope") id2 0 is true
              then if checkStatic (acessEnv id) is true
                  then dtexp [id0] r1 k1
       where {
        k1 = \type r2 exp1 -> k type newTenv [id3 "." exp1];
        id3 = "#" ++ (idToString id1)
       };
       else typeerror
              else typeerror;
    _ -> typeerror
      }
};

dtexp[id0 "." id "(" arg ")"] r k =
let {
  id1 = getIdName r id0;
id2 = getClassName r id0;
type = r id1
} in
case type of {
     //E referencia a objeto
     (r1,acessEnv) ->
      let {
        p  = \t* r2 arg1 -> case r1 id3 of {
                  functionType ->
                     let {
                         t2 = getReturnType functionType t*;
                         id4 = getInternalName functionType t*;
                         id5 = "#" ++ id4
                     }  in if t2 is typeerror then typeerror
                           else if (checkAcess acessEnv id4 (r "scope") id2 0) is true
                                then if (checkStatic (acessEnv id4) is true)
                                     then k t2 newTenv [id0 "." id5 "(" arg1 ")"]
                                     else k t2 newTenv [id0 "." id4 "(" arg1 ")"]
                                else typeerror;
                  _  -> typeerror
                 }
      } in dtarg arg r p
     //E acesso estatico de classe
    _  -> case r id2 of {
         (r1,acessEnv) -> let {  //acesso a classe estatica
                          p  = \t* r2 arg1 ->
                            case r1 id3 of {
```

```
                          functionType ->
                         let {
                             t1 = getReturnType functionType t*;
                             id4 = getInternalName functionType t*;
                             id5 = "#" ++ id4;
                             id8 = "#" ++ (idToString id0)
                          } in
                          if t1 is typeerror then typeerror
                          else if (checkAcess acessEnv id4 (r "scope") id2 0) is true
                               then if (checkStatic (acessEnv id4)) is true
                                    then k t1 newTenv [id6 "." id5 "(" arg1 ")"]
                                    else typeerror
                               else typeerror
                         _     -> typeerror
                        }
                  } in dtarg arg r p;
          _ -> typeerror
       }
};

//acesso de metodo
dtexp [id0 "." id "(" ")"] r k =
let {
  id1 = getIdName r id0;
id2 = getClassName r id0;
type = r id1
} in
case type of {
     //E objeto
    (r1,acessEnv) -> case r1 id3 of {
                functionType ->
                  let {
                       t1 = getReturnType functionType ();
                       id4 = getInternalName functionType ();
                       id5 = "#" ++ id4
                  } in if t1 is typeerror then typeerror
                       else if (checkAcess acessEnv id4 (r "scope") id4 0) is true
                            then if (checkStatic (acessEnv id4) is true)
                                 then k t2 newTenv [id0 "." id5 "("  ")"]
                                 else k t2 newTenv [id0 "." id4 "(" ")"]
                            else typeerror;
                 _   -> typeerror
               }
     //E acesso estatico de classe
     _  -> case r id2 of {
          (r1,acessEnv) -> case r1 id3 of {
                functionType ->
                    let {
                         t1 = getReturnType functionType ();
                         id4 = getInternalName functionType ();
                         id5 = "#" ++ id4;
                         id6 = "#" ++ (idToString id0)
                    } in if t1 is typeerror then typeerror
                         else if (checkAcess acessEnv id4 (r "scope") id2 0) is true
```

```
                              then if (checkStatic (acessEnv id4)) is true
                                   then k t1 newTenv [id6 "." id5 "(" ")"]
                                   else typeerror
                              else typeerror
               _  -> typeerror
        };
         -              -> typeerror
        }
  };

dtdim [specifiedDimentions abreFechaColchetes] r y type = dtdim [specifiedDimentions] r y1 type
where {
y1 = \int r1 specifiedDimentions1 -> y int2 newTenv [specifiedDimentions1 abreFechaColchetes];
int2 = int + (getNumberOfColchetes abreFechaColchetes)
};

/*A expressao deve ser de um tipo promovido para inteiro */
dtdim [specifiedDimentions "[" exp "]"] r y type = dtdim specifiedDimentions r y1 type
where {
  y1 = \int r1 specifiedDimentions1 -> dtexp exp r k
   where {
     k = \type r2 exp1 -> if assignableType inteiro type is true
                  then inteiro -> y (int+1) (update r3 r2) [specifiedDimentions1 "[" exp2 "]"]
                  else typeerror
                 where {
                   r3 = update r r1;
                   exp2 = [ id "(" exp1 ")"];
                   id = makeTypeConverterFunctionName type inteiro
                 }
  }
};

dtdim ["[" exp "]"] r y type = dtexp exp r k
where {
k = \type r1 exp1 -> if assignableType inteiro type is true
                  then inteiro -> y 1 (update r r1) [ "[" exp2 "]"]
                  else typeerror
     where {
       exp2 = [ id "(" exp1 ")"];
   id = makeTypeConverterFunctionName type inteiro
      }
};

//Xc = Int -> Env -> ArrayInit -> TypeCheckAns; //x
dtdim [abreFechaColchetes arrayInit] r y type = dtarrayInit arrayInit r x type
where {
x = \int r1 arrayInit1 -> if int = int1
                then y (int) (update r r1) [abreFechaColchetes arrayInit1];
                else typeerror
where {
int1 = numberOfColchetes abreFechaColchetes
   }
}';
```

6.2.3 Semântica Dinâmica de Java 2

A semântica de execução de Java 2 estende e modifica a definição semântica dinâmica de Java 1 e adiciona novos elementos ao modelo já definido. Como nas definições das sub-linguagens anteriores, a semântica dinâmica atua sobre uma árvore abstrata modificada pela semântica estática e também utiliza o conceito de continuação ao incrementar as definições anteriores.

6.2.3.1 Domínios Semânticos

Seguem os domínios semânticos e denotações para definir a semântica dinâmica das construções de Java 2. Eles são definidos nos módulos *NewDefinitions2.DynamicDomains*.

```
extend Sv with ClassDescriptor
 | ObjectDescriptor;
ClassDescriptor = (SuperLoc,ClassVariables,ObjectVariables,ObjectFunctions);
ObjectDescriptor = (ClassLoc,ObjectFields);
SuperLoc = Loc;
ClassVariables = Env;
ObjectVariables = (Id,Loc)*;
ObjectFunctions = (Id,Fun)*;
ClassLoc = Loc;
ObjectFields = Env;
Tc = Int -> Loc* -> Store -> Ans;
```

O domínio **Sv**, valores que podem ser armazenados na memória, foi estendido com os domínios **ClassDescriptor** e **ObjectDescriptor**. O primeiro denota um descritor de classe, que possue uma lista de elementos: a posição de memória do descritor da super classe em questão, um *environment* para obter valores dos elementos estáticos da classe, uma lista para obter os valores iniciais das variáveis não estáticos da classe e, por fim, uma outra lista para obter as denotações das funções não estáticas da classe. O segundo denota um objeto, que possui dois campos: a posição de memória do descritor da classe que representa o tipo dinâmico do objeto em questão. Um *environment* que contém o mapeamento dos identificadores do objeto para seus valores.

O domínio **Tc** denota continuação da semântica dinâmica de uma construção para especificar as dimensões de um array sendo inicializado com o operador *new*.

6.2.3.2 Funções da Semântica Dinâmica

No módulo *NewDefinitions2_DynamicSemantic.Functions* está declarada a função semântica **ddim : DimentionsOrSpecifiedDimentions → Env → Tc → Loc → Store →**

Ans que fornece o significado de inicialização dos elementos de um arranjo usando o operador *new*. O número de dimensões do arranjo, as posições de memória dos elementos do arranjo e a memória modificada são passadas a continuação da construção.

Nesse módulo estão definidas algumas funções que auxiliam a definição semântica das novas construções de Java 2:

- **startLocs : Loc* → Store → Store**: devolve uma memória de execução com as posições fornecidas na entrada marcadas como usadas;
- **news : Int → Store → Loc***: devolve uma lista de posições de memória que não estão sendo usadas;
- **createEnvObject : Store → ObjectVariables → ObjectFunctions → (Env, Store)**: aloca espaço de memória para conter as variáveis e funções de um objeto. Um novo *environment* de execução com os mapeamentos específicos dos componentes do objeto é também retornado, junto com a memória modificada.

6.2.3.3 Semântica Dinâmica de Declarações

No módulo *NewDefinitions2_DynamicSemantic.Declarations* se encontra a equação semântica para descrever o significado da construção para declaração que Java 2 adiciona. **newEnv** cria um *environment* vazio, que mapeia qualquer identificador para *unbound*.

```
ddvar [id abreFechaColchete "=" exp] r u s = dexp exp r k s
where {
    r2 = newEnv;
  k = \ev s1 -> case ev of {
               loc -> u r2[id<-loc];
               _ -> error
}
}
```

6.2.3.4 Semântica Dinâmica de Expressões

No módulo *NewDefinitions2_DynamicSemantic.Expressions* se encontram as equações semânticas que descrevem o significado das novas construções de Java 2 que denotam expressões e inicialização de arranjos através do operador *new*. A função **createEnvObject** aloca espaço de memória para os elementos de um array com base nos elementos do descritor da classe que define o tipo do objeto. A função **getClassName** retorna o identificar com o marcador interno de classe: adição de "###"ao final do identificador.

this é uma posição reservada do *environment* que contém o descritor de um objeto. Essa posição é normalmente preenchida quando uma função de objeto é chamada.

```
dexp ["new" id dimentions] r k s =
let {
    loc = r id;
    y = \int loc1* s1 -> k loc2 s2
     where {
loc = r id;
arrayDescriptor = (loc,int,loc1*);
loc2 = new s1;
s2 = s1[loc2<-arrayDescriptor]
     }
} in ddim dimentions r y loc s;

dexp ["new" basicType dimentions] r k s =
let {
    loc = -2;
    y = \int loc1* s1 -> k loc2 s2
     where {
loc = r id;
arrayDescriptor = (loc,int,loc1*);
loc2 = new s1;
s2 = s1[loc2<-arrayDescriptor]
     }
} in ddim dimentions r y loc s;

dexp ["new" id id0 "(" arg ")"] r k s =
let {
  id1 = getClassName id r;
  loc = r id
} in case s loc of {
classDescriptor (loc, r1, objectVariables, objectFunctions) ->
let {
loc1 = new s;
s1 = s[loc1<-ObjectDescriptor];
        ObjectDescriptor = (loc,r4);
        (r4,s2) = createEnvObject s1 objectVariables objectFunctions;
        k1 = \ev s3 -> case ev of {
    fun -> darg arg r p s3
  where {
  p = \ev1* s4 -> fun k2 ev1* s4;
   k3 = s -1;
  k2 = \ev1 s5 -> k loc1 (s5[-1 <- k3])
    };
  _  -> error
  }
} in dexp id0 r1 k1 s2  //assume que construtor sempre e armazenado como variavel de classe
_ -> error
 };

dexp ["new" id id0 "(" ")"] r k s
let {
  id1 = getClassName id r;
```

```
  loc = r id
} in case s loc of {
classDescriptor (loc, r1, objectVariables, objectFunctions) ->
let {
loc1 = new s;
s1 = s[loc<-ObjectDescriptor];
      ObjectDescriptor = (loc,r4);
        (r4,s2) = createEnvObject s1 r2 r3;
        k1 = \ev s3 -> case ev of {
    fun -> fun k2 () s3
  where {
   k3 = s -1;
  k2 = \ev1 s4 -> k loc1 (s3[-1 <- k3])
    };
  _  -> error
}
} in dexp id0 r1 s2 //assume que construtor esta nas variaveis de classe
_ -> error
   };

dexp [id1 "." id2] r k s =
let {
id3 = getId id1;
id4 = getClassName id3;  //classe do objeto
id5 = getId id2          //nome do campo
} in if (isStatic id1) is true //e acesso estatico atraves de nome classe
     then case s (r id3) of {
         classDescriptor(loc,r1,objectVariables,objectFunctions) ->  k (r1 id5) s;
         _  -> error
        }
     else if (isStatic id2) is true //e objeto acessando componente estatico
          then case s (r id1) of {
          objectDescriptor(loc,r1) ->  case s loc of {
           classDescriptor(loc1,r2,objectVariables,objectFunctions) -> k (r2 id5) s;
           _ -> error
             };
          _  -> error
            }
          else case s (r id3) of { //e objeto acessando componente de objeto
          objectDescriptor(loc,r1) -> k (r1 id5) s;
          _ -> error
             };


dexp [id1 "." id2 "(" arg ")"] r k s =
let {
id3 = getId id1;
id4 = getClassName id3; //classe escritora de id
id5 = getId id2         //metodo id2
p = \ev1* s2 ->
       if (isStatic id1) is true
   then case s2 (r id4) of {
  classDescriptor (loc,r1,objectVariables,objectFunctions) ->
 case r1 id5 of {
```

```
        fun -> fun k2 ev1* s2
  where {
   k3 = s -1;
  k2 = \ev1 s3 -> k ev1 (s3[-1 <- k3])
    };
          _ -> error
   };
     _ -> error

    }
   else if (isStatic id2) is true //objeto acessando metodo estatico
      then case s2 (r id4) of {
      objectDescriptor(loc,r1) -> case s2 loc of {
              classDescriptor (loc,r2,objectVariables,objectFunctions) ->
case r2 id5 of {
   fun -> fun k2 ev1* s2
  where {
      k3 = s -1;
     k2 = \ev1 s3 -> k ev1 (s3[-1 <- k3])
    };
     _ -> error
  };
_ -> error
     };
       _ -> error
      }
      else case s2 (r id) { //objeto acessando metodo do objeto
         objectDescriptor(loc,r1) -> case r1 id5 of {
   fun -> fun k2 ev2* s
  where {
      k3 = s -1;
    ev2*=objectDescriptor(loc,r1):ev1*; //passa objeto para metodo
       k2 = \ev1 s3 -> k ev1 (s3[-1 <- k3])
    };
     _ -> error
  };
_ -> error
     };
  _ -> error
}
} in darg arg r p s;

dexp [id1 "." id2 "(" ")"] r k s =
let {
id3 = getId id1;
id4 = getClassName id3; //classe escritora de id
id5 = getId id2         //metodo id2
} in if (isStatic id1) is true
   then case s (r id4) of {
  classDescriptor (loc,r1,objectVariables,objectFunctions) -> case r1 id5 of {
  fun -> fun k2 () s
  where {
   k3 = s -1;
  k2 = \ev1 s2 -> k ev1 (s2[-1 <- k3])
```

```
    };
    _ -> error
    };
     _ -> error

    }
   else if (isStatic id2) is true //objeto acessando metodo estatico
      then case s (r id4) of {
      objectDescriptor(loc,r1) -> case s loc of {
                classDescriptor (loc,r2,objectVariables,objectFunctions) ->
case r2 id5 of {
   fun -> fun k2 () s
  where {
      k3 = s -1;
     k2 = \ev1 s2 -> k ev1 (s2[-1 <- k3])
    };
     _ -> error
  };
  _ -> error
     };
      _ -> error
     }
      else case s (r id) { //objeto acessando metodo do objeto
     objectDescriptor(loc,r1) -> case r1 id5 of {
   fun -> fun k2 ev* s
  where {
      k3 = s -1;
ev*=objectDescriptor(loc,r1):(); //passa objeto para metodo
     k2 = \ev1 s2 -> k ev1 (s2[-1 <- k3])
    };
     _ -> error
    };
       _ -> error
         };
_ -> error
};

dexp ["this" "." id] r k s =
let {
id1 = getId id;
ObjectDescriptor (loc,r1) = r "this"
} in dexp [id1] r1 k s;

dexp ["this" "." id "(" ")"] r k s =
let {
id1 = getId id;
ObjectDescriptor (loc,r1) = r "this"
} in if (isStatic id) is false
     then  dexp [id1] r k1 s
where {
  k1 = \ev s1 ->
   case ev of {
  fun -> fun k2 ev1* s1
  where {
```

```
ev1* = ObjectDescriptor (loc,r1):();
   k3 = s -1;
  k2 = \ev1 s2 -> k ev1 (s2[-1 <- k3])
    };
  _  -> error
}
}
     else dexp [id "(" ")"] r k s;

dexp ["this" "." id "(" arg ")"] r k s =
let {
id1 = getId id;
ObjectDescriptor (loc,r1) = r "this"
} in if (isStatic id) is false
     then  dexp [id] r k1 s
   where {
   k1 = \ev s1 -> case ev of {
fun -> darg arg r p s1
where {
p = \ev1* s2 -> fun k2 ev2* s2
where {
ev2*=ObjectDescriptor(loc,r1):ev1*;
k3 = s -1;
k2 = \ev2 s3 -> k ev2 (s3[-1 <- k3])
}
};
_  -> error
   }
   }
     else dexp [id "(" arg ")"] r k s;

ddim [specifiedDimentions abrefechaColchetes] r y loc2 s = ddim [specifiedDimentions] r y1 loc2 s
where {
int = numberOfColchetes abrefechaColchetes;
y1 = \int1 loc* s1 -> y (int+int1) (loc* ++ loc1*) s2
where {
loc1* = news int s1;
s2 = startLocs loc1* s1
}
};

ddim [specifiedDimentions "[" exp "]"] r y loc3 s = ddim [specifiedDimentions] r y1 loc3 s
where {
y1 = \int loc* s1 -> dexp exp r k s1
where {
k = \ev s2 -> case ev of {
int1 -> y (int+1) (loc* ++ loc1*) s3
where {
loc1* = loc2:();
loc2 = new s2;
s3 = s4[loc<-arrayDescriptor];
arrayDescriptor = (loc3,int1,loc2*);
loc2* = news int s2;
s4 = startLocs loc2* s2
```

```
};
_ -> error
  }
}
};

ddim [ "[" exp "]" ] r y loc3 s = dexp exp r k s
where {
k = \ev s2 -> case ev of {
int1 -> y 1 loc:() s3
where {
loc = new s2;
s3 = s4[loc<-arrayDescriptor];
arrayDescriptor = (loc3,int1,loc2*);
loc2* = news int s2;
s4 = startLocs loc2* s2
};
_ -> error
  }
};

ddim [ abreFechaColchetes arrayInit ] r y loc3 s = darrayInit arrayInit r x loc3 s
where {
x = \loc* int s1 -> y int loc* s1
};
```

6.2.4 Transformadores Utilizados em Java 2

O módulo *Transf2* define transformadores *redefine* para modificar as equações semântica de Java 0 e Java 1 que descrevem declaração de métodos em Java 1 e que descrevem a semântica de executar um programa.

É preciso redefinir as equações semânticas de declaração de método para que quando um método não estático em Java 2 for chamado, ele consiga lidar com a passagem de um parâmetro adicional, o própio objeto que iniciou a sua chamada. Dessa forma, as variáveis atualizadas pelo método serão sempre as do objeto. O objeto passado como parâmetro é associado no *environment* de execução à posição reservada denominada "this". Essas redefinições são apresentadas a seguir:

```
redefine ddec [dtype id com] r u s by
let {
    id1 = getId id;
    fun =  \k ev* s1 ->
    if (isStatic id) is false //funcao de classe
    then case ev* of {
     ObjectDescriptor(loc,r2):ev2* -> dcom com r3 c s2
                where {
                   s2 = s1[-1 <- k];
                   string = (idToString id1) ++ "#";
```

```
                  id1 = string;
                  r1 = newEnv[id1 <- fun];
                  c = k void;
                  r3 = (update r r1);
                  r4 = r3["this"<-ObjectDescriptor(loc,r2)]
               }
     _ -> error
        }
    else dcom com (update r r1) c s2
                where {
                   s2 = s1[-1 <- k];
                   string = (idToString id1) ++ "#";
                   id1 = string;
                   r1 = newEnv[id1 <- fun];
                   c = k void
                }


}
in u newEnv[id <- fun] s,

ddec [dtype id param com] r u s by
let {
  id1 = getId id;
  fun =  \k ev* s1 ->
    if (isStatic id) is false //funcao de classe
    then case ev* of {
      ObjectDescriptor(loc,r6):ev2* ->
        dcom com r5 c s3
         where {
          string = (idToString id1) ++ "#";  //necessario para concatenar e depois fazer casting
          id1 = string;
          r1 = newEnv[id1 <- fun];
          r2 = update r r1;
          (r3,s2) = buildEnv (getlistParam param) ev* s1;
          s3 = s2[-1 <- k];
          c = k void;
          r4 = (update r2 r3);
          r5 = r4["this"<-ObjectDescriptor(loc,r6)]
         };
      _ -> error
    else dcom com r4 c s3
         where {
          string = (idToString id1) ++ "#";  //necessario para concatenar e depois fazer casting
          id1 = string;
          r1 = newEnv[id1 <- fun];
          r2 = update r r1;
          (r3,s2) = buildEnv (getlistParam param) ev* s1;
          s3 = s2[-1 <- k];
          c = k void;
          r4 = (update r2 r3)
         }
} in dc newEnv[id <- fun] s;
```

Outras transformações são utilizadas para redefinir a semântica de um programa Java 2 e são apresentadas na seção seguinte, que melhor contextualiza a necessidade das redefinições.

6.2.5 Semântica de um Programa Java 2

Para definir a semântica de um programa em Java 2, composto por várias classes, foi preciso adicionar novas definições ao modelo de Java 1. Elas encontram-se no módulo *NewProgramSemantic*. A semântica de um programa com várias classes consiste em primeiramente efetuar a verificação de tipos de todas as classe e então iniciar a execução pela função **main**.

A função **dtclass : ListaClasses → StaticSemantic.Domains.Env → Dtclass → TypeCheckAns** foi criada para transformar uma lista de classes em uma nova lista com os tipos verificados, na presença de um *environment* de tipos e de uma continuação de classes. A função **updateStore : Store → Store → Store** junta duas memórias de execução. **escope** e **acessEnv** são posições reservadas no *environment* de tipo que armazenam, respectivamente, o nome da classe corrente e um *environment*. A semântica de verificação de tipos de uma classe consiste em fazer a verificação de tipos dos seus componentes e passar à continuação um *environment* que contém informações da classe.

Também foi feita uma nova definição da função **dclass**, declarada em Java 0, para dar o significado dinâmico quando a lista de classes contiver mais de uma classe (em Java 0 só existe uma classe, logo a função só é definida para esse caso). Nesse caso a semântica dinâmica da lista de classes é definida como a semântica de execução de cada classe em um mesmo *environment* e memória de execução que contém todas as declarações realizadas.

Seguem as novas definições para fazer checagem de tipo de um programa Java 2:

```
Dtclass = Env -> ListaClasses -> TypeCheckAns; //z
public dtclass : ListaClasses -> StaticSemantic.Domains.Env -> Dtclass -> TypeCheckAns;

  dtclass ["class" id dec] r z = dtdec dec r11 u
     where {
      u = \r2 dec1 -> z r3 ["class" id dec1];
              where {
                acessEnv1 = r2 "acessEnv";
                r21 = r2["acessEnv"<-(\id -> tunspecified)];
                classType = (r21,acessEnv1);
                r3 = newTenv[id1<-classType];
              }
      r1 = r["escopo"<-id];
      acessEnv = \id -> tunspecified;
```

```
      r11 = r1["acessEnv"<-acessEnv]
     };

dtclass [listaClasses class] r z =
  let {
        (r1,class12)= case dtclass class (update r r5) (\r2 class1 -> (r2,class12)) of {
         ivc -> ivc;
        };
        (r3,listaClasses12) = case dtclass listaClasses (update r r5)
                              (\r4 listaClasses1 -> (r4, listaClasses1)) of {
         ivc -> ivc;
        };
        r5 = update r1 r3
  } in z r5 [listaClasses12 class12];


dclass [listaClasses class] r i u s = let {
     (r1,s11)= case dclass class (update r r5) (\r2 s1 -> (r2,s1)) s3 of {
         ivd -> ivd;
         }
;
     (r2,s21) = case dclass listaClasses (update r r5) (\r2 s1 -> (r2, s1)) s3 of {
         ivd -> id;
        };
    r5 = update r1 r2;
    s3 = updateStore s11 s21
  } in u r5 s3;

function updateStore : Store -> Store -> Store;
updateStore s1 s2 = s3
where {
   s3 = \loc -> if s2 loc is unused
                then s1 loc
                else s2 loc
}
```

A semântica dinâmica de cada classe consiste, na verdade, em declarar os componentes da classe e armazenar no *environment* de execução o descritor da classe declarada. Para definir a semântica de execução de uma classe, por isso, foi preciso redefinir a função **dclass** definida em Java 0, no módulo *Transf2*:

```
redefine  dclass ["class" id dec] r i u s by
  ddec dec r1 s u1
    where {
      r1 = r["classDescriptor"<-classDescriptor];
      classDescriptor=(-2,newEnv,((),()),((),()));
      u1 = \r2 s1 -> case r2 "main" of {
           fun -> u r3 s1
               where {
                r4 = newEnv[id<-(r2 "classDescriptor")];
                r3 = r4["main"<-fun]
               };
```

```
          _   -> u r3 s1
              where {
               r4 = newEnv[id<-(r2 "classDescriptor")]
              }
         }

};
```

Nota-se que se a função *main* é encontrada no *environment* de execução da classe, ela é transferida para o environment de execução global. Nesse mesmo módulo, a função **dprog**, que fornece a semântica de um programa, é redefinida, para primeiramente fazer a verificação de tipos das classes e executar a função *main*, em um contexto que contém a denotação de cada uma das classes.

```
redefine  dclass ["class" id dec] r i u s by
dpro listaclasses i by
    dtclasse listaclasses newTenv z
    where {
     z = \r1 listaclasses1 -> dclass listaclasses1 newEnv u initialStore
      where {
        u = \r2 s1 -> case r2 "main" of {
                fun -> fun k i s1
                where {
                  k = \ev s2 -> stop;
                };
                _ -> error
                }
     }
   };
```

7 Discussões

Durante a definição de Java 1 foi necessário realizar alterações na definição de Java 0, bem como foi necessário também realizar alterações na especificação de Java 0 e Java 1 ao definir Java 2. Muitas dessas mudanças foram bem resolvidas com o uso dos transformadores de *Notus*, por exemplo para o tratamento dos comandos *break* e *continue* em Java 1, e com as cláusulas **extends**, para a inserção de novos tipos básicos. Entretanto, algumas alterações precisaram ser efetuadas diretamente em definições anteriores, não sendo os recursos da linguagem *Notus* convenientes para tratar a inserção de alguns conceitos da linguagem, principalmente por fazer uso de um grande número da claúsula *redefine*.

Com a inserção dos novos tipos básicos em Java 1, foi preciso refazer a definição de Java 0 para, principalmente, permitir uma definição de Java 1 mais legível e menos verbosa. Foi preciso, por exemplo, alterar a definição das funções para checagem de tipo de expressões **dtexp [exp1 "="exp2] r k** e **dtexp [exp1 opr exp2] r k** para prever conversões implícitas de tipos básicos, o que não havia sido feito em Java 0. A função da semântica estática **oprOk : Type → Opr → Type**, que verifica se um dado operador pode ser aplicado a um dado tipo, retornando o tipo resultante da operação, precisou ser definida por casos em Java 0, o que facilitou a definição de Java 1 ao inserir novos tipos primitivos. Dessa forma, posteriormente apenas foi necessário ter novas definições da função por casos. Da mesma forma, a função da semântica dinâmica **dopr : Opr → Rv → Rv → Ec → Store → A** foi definida por casos em Java 0, facilitando a definição de Java 1. Caso essas reformulações no modelo de Java 0 não fossem feitas, seria necessário usar demasiadamente os tranformadores do tipo **redefine** em Java 1 para redefinir o corpo dessas funções de Java 0, além de permanecer com um modelo pouco estensível em termos de tipos.

A adição de várias classes em Java 2 também influenciou bastante as definições das sub-linguagens anteriores. Com a noção de várias classes e a possibilidade de declarações não-estáticas, foi preciso criar um arcabouço para suportar a verificação do uso dos elementos de acordo com seu controle de visibilidade e acessibilidade (é preciso, por exemplo,

verificar se em uma função estática só se faz uso de elementos estáticos da classe). Dessa forma, foi necessário inserir novos domínios em Java 0, como o **AcessEnv**, que é um *environment* que mapeia identificadores em modificadores (*static*, *public*, *private*). Esse domínio não existia em Java 0 antes da definição de Java 2, dado que só existia uma classe e todas as declarações eram estáticas. Além da necessidade dessa inserção, foi preciso alterar as equações das construções de declaração para registrar os modificadores de cada declaração nesse *environment* tanto de Java 0 quanto de Java 1. Esse *environment* de modificadores foi armazenado em uma posição específica do próprio *environment* de tipo. Outras soluções mais elegantes foram cogitadas, como o uso de um único mapeamento que retornaria dois componentes, o tipo e os modificadores do identificador. Entretanto, como os modelos das sub-linguagens Java 0 e Java 1 já se encontravam definidos, essa solução afetaria uma grande quantidade de equações de Java 0 e Java 1. A solução adotada, embora não tão elegante, minimizou a quantidade de mudanças a serem efetuadas nas definições das sub-linguagens Java 0 e Java 1.

A inclusão da noção de objeto em Java 2 também provou vários impactos nas definições das sub-linguagens de Java 0 e Java 1. O modelo de classe criado **ClassDescriptor = (SuperLoc,ClassVariables,ObjectVariables,ObjectFunctions)** necessita que as declarações de classe sejam identificadas. Antes, em Java 0 e Java 1, todas as declarações eram potencialmente iguais (dentro da classe ou dentro de uma função), sendo o *environment* de execução encarregado de acumular declarações e desempilhá-las definindo os contextos. Entretanto, com a noção de objeto é preciso identificar as declarações de classe e separá-las em estáticas e dinâmicas. As informações **ObjectVariables** e **ObjectFunctions** são usadas quando um objeto é criado, sendo cruciais para alocar e inicializar o espaço na memória para eles. A informação **ClassVariables** é usadas quando campos estáticos, tanto por instâncias de objetos como por nome da classe, são acessados. O maior impacto se deve pela necessidade de todas declarações necessitarem guardar informações nesse descritor. Uma solução possível é alterar manualmente todas as equações semânticas de declaração em Java 0 e Java 1, deixando a definição de Java 2 menos verbosa. Em contrapartida, ao adotar essa solução, Java 0 e Java 1 passam a tratar um conceito introduzido em Java 2, o que contradiz, de certa forma, o princípio da "vaguesa", que sugere a apresentação incremental dos conceitos de uma linguagem. Outra solução é a utilização do transformador *redefine* para redefinir todas as equações de declaração de Java 0 e Java 1, tornando a definição de Java 2 verbosa e pouco legível. A implementação de alguma dessas soluções ou de uma solução alternativa para tratar esse problema, encontrado durante o desenvolvimento do trabalho, é deixada para trabalhos futuros.

Nota-se, portanto, com as discussões anteriores, que eventuais mudanças na semântica de definições anteriores são necessárias, principalmente para melhorar e adequar o modelo já definido. Quanto mais se conhecem os conceitos da linguagem de programação, cuja semântica é definida, mais robusto pode ser o arcabouço de uma sub-linguagem para suportar inclusões de conceitos futuros. É interessente ressaltar que a definição semântica é feita com dois propósitos: construção e apresentação. Na construção, a necessidade da incrementabilidade é decorrente da inserção das construções futuras. Nesse caso, a estensibilidade é desejável para que as novas definições não interfiram diretamente no modelo já definido, embora não se saiba exatamente a direção que essas extensões ocorrerão. A semântica multidimensional contribui muito para o processo de construção incremental, entretanto, há dificuldades, pois certas decisões não podem ser facilmente estendidas pelos transformadores, o que pode ocasionar no grande uso da cláusula *redefine*. Por outro lado, na apresentação todos os aspectos da linguagem são conhecidos, o que permite criar um arcabouço mais adequado para suportar novas definições. A incrementabilidade é utilizada para melhorar a legibilidade e apresentar os conceitos da linguagem passo a passo, via o conceito da "vaguesa". Assim, a semântica multidimensional parece ser bem mais adequada para a apresentação incremental que para construção incremental.

Também é interessante citar algumas necessidades encontradas durante o desenvolvimento do trabalho que resultaram na inclusão de um novo transformador na linguagem *Notus*: **within**. A seguinte discussão exemplifica a nova necessidade.

Os seguintes transformadores foram aplicados à definição semântica dinâmica de Java 0 para inserir as construções de *break* e *continue*, conforme visto na seção 5.3.2.4:

```
signature DynamicSemantic.Functions.dcom : Com -> DynamicSemantic.Domains.Env -> DynamicSemantic.Domains.Cc
    -> (contBreak : DynamicSemantic.Domains.Cc) -> (contContinue : DynamicSemantic.Domains.Cc) -> Store -> A;

default contBreak = (\s -> error), contContinue = (\s -> error);

redefine dcom ["while" exp com] r c c1 c2 s by
  dr exp r k s
where {
k = \ev s1 ->  if ev
   then dcom com r c3 c4 c3 s1 //muda continuacao de break e switch
   where {
    c3 = \s2 -> dcom ["while" exp com] r c c1 c2 s2;
    c4 = \s2 -> c s2;       //se for break sai fora
   }
       else c s1
};
```

Os valores *default* são usados, por exemplo, na situação abaixo, em que as duas novas

continuações são inseridas automaticamente na chamada do comando **dcom**.

```
ddec [dtype id com] r u s =
let {
 fun =  \k ev* s1 -> dcom com (update r r1) c s2
  where {
   . . .
  }
}
in u newEnv[id <- fun] s;
```

Após o compilador aplicar as transformações, a equação fica da seguinte forma:

```
ddec [dtype id com] r u s =
let {
 fun =  \k ev* s1 -> dcom com (update r r1) c (\s -> error) (\s -> error) s2
  where {
   . . .
  }
}
in u newEnv[id <- fun] s;
```

Considerando uma situação diferente, para outra linguagem, em que fosse necessário que esses valores *default* dependessem de informações no contexto da chamada da função, cujos parâmetros estão sendo alterados. Por exemplo, supondo que as continuações dos comandos *break* e *continue* fossem a própria continuação que segue o comando *while*. Logo, como mostra as equações seguintes, a continuação de fluxo de execução normal que o comando *while* recebe é passada como continuação de *break* e *continue* na chamada recursiva de **dcom** dentro do corpo da equação semântica que descreve a semântica do *while*.

```
dcom ["while" exp com] r c c1 c2 s by
    dr exp r k s
  where {
 k = \ev s1 ->  if ev
          then dcom com r c c c s1
          . . .
  }
```

Para permitir esse tipo de inserção de valores *default*, foi adicionado o transformador *within*. O uso desse transformador para produzir o resultado anterior, seria:

```
signature DynamicSemantic.Functions.dcom : Com -> DynamicSemantic.Domains.Env -> DynamicSemantic.Domains.Cc
    -> (contBreak : DynamicSemantic.Domains.Cc) -> (contContinue : DynamicSemantic.Domains.Cc) -> Store -> A;

default contBreak = (\s -> error), contContinue = (\s -> error),
        contBreak within dcom ["while" exp com] r c c1 c2 s = c,
        contContinue within dcom ["while" exp com] r c c1 c2 s = c;
```

Portanto, os valores *defaults* usados quando **dcom** é chamado dentro de uma função **dcom** [”**while**”**exp com**] **r c c1 c2 s** são especificados de forma distinta. Para maiores detalhes sobre o transformador *within* ver [Tirelo, Bigonha e Saraiva 2008].

8 Conclusões e Trabalhos Futuros

Foi realizada uma definição incremental de Java utilizando a linguagem *Notus*, que implementa a metodologia de definição semântica multidimensional. Alguns conceitos de Java, como arranjos, comandos de mudança de fluxo de controle e objetos foram inseridos incrementalmente usando as três sub-linguagens: Java 0, Java 1, Java 2.

A definição de Java 0 demandou um maior tempo porque toda as construções básicas da linguagem precisaram ser modeladas. As definições de Java 1 e Java 2, por sua vez, tiveram objetivo de usar os recursos de transformações e claúsula *extends* para inserir novas construções e conceitos de Java à linguagem base. Impactos nas definições anteriores foram inevitáveis para tornar o modelo mais estensível e menos verboso.

O exercício de definir as construções de Java incrementalmente permitiu concluir que a metodologia multidimensional é mais adequada para apresentação de uma linguagem de programação. Ela é bastante interessante porque permite modelar os conceitos de uma linguagem de forma focada, embora não isolada, o que facilita a descrição e apresentação da linguagem. Além disso novos transformadores foram identificados no processo e adicionados à linguagem *Notus*.

Trabalhos futuros poderiam incluir novos conceitos de Java à definição atual, como exceções, herança e tipos genéricos. Um modelo mais simplificado, principalmente para incluir os conceitos de múltiplas classes e objeto poderia ser estudado, bem como os detalhes envolvidos, que não foram completamente contemplados nesse trabalho, dada a complexidade do modelo resultante.

ANEXO A – Especificação de Java 0

A.1 Gramática Concreta

A gramática concreta de Java 0 e a especificação da correspondente gramática abstrata a ser gerada estão contidas em vários módulos, no pacote *ConcreteGrammar*.

A.1.1 Comandos

```
module ConcreteGrammar.Commands
import Global.Lexico, Global.Domains, Declarations, Expressions;

public methodBody: Com ::= block:block;

public block: Com ::= "{" blockStatements "}": blockStatements | "{" "}":["emptyCommand"];

public blockStatements: Com ::= localVariableDeclarationStatement
| blockStatements statement
                            | statement:statement
                            ;

public statement: Com ::= block:block
                      | "if" parExpression statement
                      | "if" parExpression statement "else" statement
                      | "while" parExpression statement
                      | expression ";": ["com" expression]
                      | "System.out.println" "(" expression ")": ["print" expression]
                      | ";":["emptyCommand"]
                      | "return" expression
                      ;

end
```

A.1.2 Declarações

```
module ConcreteGrammar.Declarations

import Global.Lexico, Global.Domains, Expressions, Commands;
```

```
public classBodyDeclarations : Dec ::= classBodyDeclarations classBodyDeclaration
                                    | empty:["emptyDeclaration"];

classBodyDeclaration: Dec ::= modifier memberDec :[modifier memberDec];

memberDec: Dec ::= functionType id "(" ")" methodBody: [functionType id methodBody]
| functionType id "(" formalParameterDecls ")" methodBody:[functionType id formalParameterDecls methodBody]
;

formalParameterDecls: Param ::= typeFormalParameter:typeFormalParameter
| typeFormalParameter "," formalParameterDecls:[typeFormalParameter formalParameterDecls]
;

typeFormalParameter: Param ::= declType id
                            | declType id abreFechaColchetes;

declType: Dtype ::= arrayType
                | referenceOrBasicType;

functionType:Dtype ::= voidType
                   | basicType:["functionType" basicType];

public localVariableDeclarationStatement: Dec ::= declType variableDeclarators;

variableDeclarators: Dvar ::= variableDeclarators "," variableDeclarator :
      [variableDeclarators variableDeclarator]
                           | variableDeclarator:variableDeclarator;

variableDeclarator: Dvar ::= id "=" expression
                          | id:id
                          | id abreFechaColchetes;

//Regras auxiliares

arrayType ::= referenceOrBasicType abreFechaColchetes;

referenceOrBasicType ::= basicType
                      | referenceType;

basicType ::= "int" | "boolean";

voidType ::= "void";

referenceType ::= id:id;

abreFechaColchetes ::= abreFechaColchetes "[" "]" | "[" "]";

modifier ::= elementModifier modifier | empty;

public modifier ::= "public" "static" | "public" | "static" | empty:["emptyModifier"];

end
```

A.1.3 Expressões

```
module ConcreteGrammar.Expressions
import Global.Lexico, Global.Domains, Operators;

public parExpression: Exp ::= "(" expression ")" : expression;

public expression: Exp ::= expressionA "=" expressionA
                       | expressionA:expressionA;

expressionA: Exp ::= expressionA infixOp primary
                 | primary:primary;

callMethod: Exp ::= id "(" arguments ")"
                | id "(" ")";

arguments: Arg ::= expressionArg "," arguments: [expressionArg arguments]
               | expressionArg :expressionArg;

expressionArg: Arg ::= expression;

primary: Exp ::= literal:literal
             | id:id
             | callMethod:callMethod;

literal: Exp ::= integerLiteral:integerLiteral
             | booleanLiteral:booleanLiteral;

end
```

A.1.4 Operadores

```
module ConcreteGrammar.Operators

import Global.Lexico, Global.Domains;

public infixOp  : Opr ::= logic | arithmetic;
arithmetic    ::= "+" | "*" | "/" | "-";
logic    ::= ">" | "<" | "==" | "!=";

end
```

A.1.5 Programa

```
public pacote : Pro  ::= listaClasses;

public listaClasses ::= cUnit:cUnit;

cUnit : Classe ::= "class" id "{" classBodyDeclarations "}":["class" id classBodyDeclarations]
          | pontoVirgula:["emptyProgram"]
               ;
```

```
pontoVirgula  ::= ";" pontoVirgula | ";";
```

A.2 Funções Auxiliares

A.2.1 Funções da Semântica Estática

Funções que auxiliam a definição semântica de verificação de tipos de Java 0. Elas se encontram no módulo *StaticSemantic.Functions*.

```
// AUXILIARY FUNCTIONS

////////////////////////////////////////////////////////////////////////////////////////
//Constroi nomes de funcoes de conversao de tipo, dado os tipos
public function makeTypeConverterFunctionName : Type -> Type -> String;
makeTypeConverterFunctionName type1 type2 = (typeToString type1) ++ "To" ++ (typeToString type2) ++ "#";

////////////////////////////////////////////////////////////////////////////////////////
//Obtem string que representa tipo
public function typeToString : Type -> String;
typeToString inteiro = "int";
typeToString booelano = "bool";

////////////////////////////////////////////////////////////////////////////////////////
//Conversao do segundo tipo pode ser feita para o primeiro tipo - widening conversions
public function assignableType : Type -> Type -> Bool;
assignableType bool1 bool2 = true;
assignableType inteiro1 inteiro2 = true;

////////////////////////////////////////////////////////////////////////////////////////
//Retorna numero de funcoes em uma lista de funcoes
function cardinality : FunctionType -> Int;
cardinality functionValue:functionValue1* = 1 + (cardinality functionValue1*);
cardinality () = 0;

////////////////////////////////////////////////////////////////////////////////////////
function equalArray : SemanticArrayType -> SemanticArrayType -> Bool;
equalArray Array (int,type) Array (int1,type1) =
if int = int1 is true
then equalType type type1;
else false;
equalArray _ _ = false;

////////////////////////////////////////////////////////////////////////////////////////
//Funcao que, junto com equalArray, verifica se dois tipos sao iguais
public function equalType : Type -> Type -> Bool;
equalType inteiro1 inteiro2 = true;
equalType booleano1 booleano2 = true;
equalType void1 void2 = true;
equalType semanticArrayType1 semanticArrayType2 = equalArray semanticArrayType1 semanticArrayType2;
```

```
equalType id1 id2 = if strCompare (idToString id1) (idToString id2) is true
then true
else false;

////////////////////////////////////////////////////////////////////////////////////
// Funcao usada na semantica de checagem de tipo e que verifica se o tipo de uma expressao
// e booleana. Caso seja a checagem prossegue normalmente, caso contrario, a checagem termina
// com erro
function isBooleanType : Ec -> Type -> Env -> Exp -> A;
isBooleanoType ec t r exp = case t of {
booleano -> ec t r exp;
_  -> typeerror
};

////////////////////////////////////////////////////////////////////////////////////
//Verifica se duas listas de tipos sao iguais.

function isEqualArgs : Type* -> Type* -> Bool;

isEqualArgs type1:type11*  type2:type22* =
if (equalType type1 type2) is true
then isEqualArgs type11* type22*
else false;

//Funcao sem argumento
isEqualArgs () () = true;

isEqualArgs _ _ = false;


////////////////////////////////////////////////////////////////////////////////////
//Verifica se existe alguma funcao em um dado grupo de funcoes que tem parametros igual ao
//estabelecido
function isFunctionSetMember : Type* -> FunctionType -> Bool;

isFunctionSetMember type* functionValue:functionValue1* =
case functionValue of {
Func (formalParameters,t,id) -> if isEqualArgs type* formalParameters is true
then true
else isFunctionSetMember type* functionValue1*
};

isFunctionSetMember type* () = false;

////////////////////////////////////////////////////////////////////////////////////
//Verifica se os argumentos reais sao compativeis com os formais

function isCompatibleArgs : FormalParameters -> Type* -> Bool;

isCompatibleArgs type1:type11*  type2:type22* =
if (subType type2 type1) is true //Se type2 e subtipo de type1
then isCompatibleArgs type11* type22*
else false;
```

```
//Funcao sem argumento
isCompatibleArgs () () = true;

isCompatibleArgs _ _ = false;

////////////////////////////////////////////////////////////////////////////////////////
//Retorna um novo environment de tipo
function newTenv : Env;
newTenv = \id -> unbound;

////////////////////////////////////////////////////////////////////////////////////////
function subArray : SemanticArrayType -> SemancitArrayType -> Bool;
subArray Array (int,type) Array (int1,type1) =
if int = int1 is true
then subType type type1;
else false;

subArray _ _ = false;

////////////////////////////////////////////////////////////////////////////////////////
//Funcao que, junto com subArray, verifica se um tipo (primeiro tipo) e subtipo de outro
//(segundo subtipo)
function subType : Type -> Type -> Bool;
subType inteiro1 inteiro2 = true;
subType booleano1 booleano2 = true;
subType void1 void2 = true;
subType semanticArrayType1 semanticArrayType2 = subArray semanticArrayType1 semanticArrayType2;
subType id1 id2 = if strCompare (idToString id1) (idToString id2) is true
                  then true
  else false;

////////////////////////////////////////////////////////////////////////////////////////
//funcao que junta dois environments em um unico env1[env2]
function update : Env -> Env -> Env;
update r r1 = let {
r2 =  \id ->  if r1 id is unbound
  then r id
  else r1 id;
ar1 = r1 "acessEnv";
ar  = r  "acessEnv";
ar2 = \id -> if ar1 id is unbound
 then ar id
 else ar1 id;
r3 = r2 ["acessEnv"<-ar2]
} in r3;

////////////////////////////////////////////////////////////////////////////////////////
//Algoritmo que avalia conjunto de funcoes e retorna  informacoes da que melhor casa com o tipo dos
//parametros de chamada

function getReturnType : FunctionType -> Type* -> Type;

getReturnType functionType type* =
let {
```

```
functionType1=getCompatibleSet functionType type*;      //obtem conjunto inicial, dado por hieraquia dos parametro
functionType2= get MoreSpecificSet functionType1 type*  //filtra funcao mais especifica.

} in case cardinality functionType2 of {
1 -> case functionType2 of {
Func (formalParameters,type1,id) -> type1
};
_ -> typeerror //ambiguidade ou nao possui declaracao que casa. Retorna tipo Id
}

function getInternalName : FunctionType -> Type* -> Id;

getInternalName functionType type* =
let {
functionType1=getCompatibleSet functionType type*;
functionType2= getMoreSpecificSet functionType1 type*
} in case cardinality functionType2 of {
1 -> case functionType2 of {
Func (formalParameters,type1,id) -> id
};
_ -> typeerror //ambiguidade ou nao possui declaracao que casa. Retorna Id
}

////////////////////////////////////////////////////////////////////////////////////
//Primeiro considera todas as possibilidades de funcoes dada a hierarquia
//Dado conjunto de funcoes, as analisa e retorna novo conjunto de funcoes
function getCompatibleSet : FunctionType -> Type* -> FunctionType;

getCompatibleSet functionValue:functionValue1* type* =
case functionValue of {
Func (formalParameters,type1,id) -> if (isCompatibleArgs formalParameters type*) is true
     then functionValue:(getCompatibleSet functionValue1* type*);
     else getCompatibleSet functionValue1* type*;

}
getCompatibleSet () type* = ();

////////////////////////////////////////////////////////////////////////////////////
//Em um segundo momento, elimina funções menos específicas
//Aqui, todas as funcoes tem, pelo menos, o mesmo numero de parametros
function getMoreSpecificSet : FunctionType -> Type* -> FunctionType;

getMoreSpecificSet functionType type* = if type* is empty  //Se funcao e sem parametro, nao tem especificidade
    then functionType
    else MoreSpecificSet functionType type* 1; //filtra funcao mais especifica.

function MoreSpecificSet : FunctionType -> Type* -> Int -> FunctionType;
MoreSpecificSet functionType type:type1* int =
let {
type2* = getNesimoTypeforAll functionType int;
type3  = getMoreSpecificType type2*
} in intersection (getFunctions functionType type3 int) (MoreSpecificSet functionType type1* (int+1));

MoreSpecificSet functionType () int = ();
```

```
//Nao ha funcoes resultantes do primeiro passo
MoreSpecificSet () _ _ = ();

////////////////////////////////////////////////////////////////////////////////////////
//Retorna o tipo mais especifico de um conjunto de tipos
function getMoreSpecificType: Type* -> Type;

getMoreSpecificType type:type1* =
if empty type1* is true
then type
else let {
type1=getMoreSpecificType type1*
} in if subType type type1  is true
then type
else type1;

////////////////////////////////////////////////////////////////////////////////////////
//Retorna funcoes que tem type no parametro de numero Int
function getFunctions : FunctionType -> Type -> Int -> FunctionType;
getFunctions functionValue:functionValue1* type int =
case functionValue of {
Func (formalParameters,type,id) -> if equalType (getNesimoType formalParameters int) type is true
   then functionValue:(getFunctions functionValue1* type int)
   else getFunctions functionValue1* type int
};

function () type int = ();

////////////////////////////////////////////////////////////////////////////////////////
// Retorna os e-nesimos tipos dos parametros de um conjunto de funcoes

function getNesimoTypeforAll : FunctionType -> Int -> Type*;
getNesimoTypeforAll functionValue:functionValue* int =
case functionValue of {
Func (formalParameters,type,id) -> (getNesimoType formalParameters int): (getNesimoTypeforAll functionValue* int)
};

getNesimoTypeforAll () int = ();

////////////////////////////////////////////////////////////////////////////////////////
// Retorna o e-nesimo tipo do parametro de uma funcao

function getNesimoType : formalParameters -> Int -> Type;
getNesimoType type:type1* 1 = type;
getNesimoType type:type1* int = getNesimoType type1* (int - 1);

////////////////////////////////////////////////////////////////////////////////////////
//Interseccao entre dois conjuntos de funcao
//Cada conjunto, por construcao, e disjunto
function intersection FunctionType -> FunctionType -> FunctionType;

intersection functionValue:functionValue1+ functionType =
(intersection functionValue functionType) ++ (intersection functionValue1* functionType);
```

```
intersection (functionValue) functionValue1:functionValue2* =
case functionValue of {
Func (formalParameters,type,id) -> case functionValue1 of {
Func (formalParameters1,type1,id1) -> if (isEqualArgs formalParameters formalParameters1) is true
      then (functionValue)
      else (intersection functionValue functionValue2*)
           }
};

intersection () functionType = ();
intersection functionType () = ();
intersection _ _ = ();

////////////////////////////////////////////////////////////////////////////////////
//Funcao que verifica se uma lista de modificadores tem modificador estatico
function checkStatic : Tmodifier* -> Bool;
checkStatic tmodifier:tmodifier* = if tmodifier is tstatic
   then true
   else checkStatic tmodifier*;

checkStatic () = false;
end
```

A.2.2 Funções da Semântica Dinâmica

Funções que auxiliam a definição semântica de execução de Java 0. Elas se encontram no módulo *DynamicSemantic.Functions*.

```
////////////////////////////////////////////////////////////////////////////////////
//funcao que junta dois environments em um unico env1[env2]
public function update : Env -> Env -> Env;
update r r1 = let {
r2 =  \id ->  if r1 id is unbound
  then r id
  else r1 id
 in r2;

////////////////////////////////////////////////////////////////////////////////////
//Para colocar valor em Loc, no environment
function updateE : Loc -> Ec -> Ev -> Store -> A;
update loc k ev s = case ev of {
rv -> k rv s[ref <- rv];
_  -> error
    };

////////////////////////////////////////////////////////////////////////////////////
//Retorna um novo environment
function newEnv : Env;
newEnv = \id -> unbound;

function cont : Ec -> Ev -> Store -> A;
```

```
cont ec ev store = case ev of {
         loc -> case (s loc) of {
          rv -> k rv s;
          unused -> error;
          _ -> error};
         _   -> error};

////////////////////////////////////////////////////////////////////////////////////////
//Segue apontador ate obter valor de memoria
function deref : Ec -> Ev -> Store -> A;
    deref k ev s = case ev of {
      loc -> cont k loc s;
      _   -> k ev s
        };

function isRv : Ec -> Ev -> A;
    isRv k ev = if ev is rv then k ev else error;


function isBoolEqual : Bool -> Bool -> Bool;
isBoolEqual bool1 bool2 =
let {
bool3 = bool1 and bool2;
bool4 = (not bool1) and (not bool2);
} in (bool3 or bool4);


function new : Store -> Loc;
new store = findNew store 0;

function findNew : Store -> Loc -> Loc;
findNew store int = if (store int) is unused then int else findNew store newLoc
where {newLoc = int+1};

////////////////////////////////////////////////////////////////////////////////////////
//Remove marcador interno do tipo estatico de um dado identificador
public function getId : Id -> Id;
getId id = case id of {
"#":string* -> id1  where { id1 = string*};
       _  -> id
   };

////////////////////////////////////////////////////////////////////////////////////////
//Verifica se identificador possui marcador interno para dizer se e estatico ou nao
public function isStatic : Id -> Bool;
isStatic id = case id of {
 "#":string* -> true;
       _  -> false
      };
end
```

ANEXO B – Especificação de Java 1

B.1 Gramática Concreta

A gramática concreta de Java 1 e a especificação da correspondente gramática abstrata a ser gerada estão contidas em vários módulos, no pacote *NewDefinitons_ConcreteGrammar* de Java 1.

B.1.1 Novas Construções

Seguem as especificação das novas construções e a extensão das variáveis de gramática de Java 0 para obter as novas construções de Java 1.

```
module NewDefinitions_ConcreteGrammar.NewVariables

import ConcreteGrammar.Commands, ConcreteGrammar.Declarations, ConcreteGrammar.Expressions,
       Global.Lexico, Global.Domains;

switchBlockStmtGroups : SwitchBlock ::= switchBlockStmtGroups switchBlockStmtGroup
                                      | empty
                                      ;

switchBlockStmtGroup : SwitchBlock ::= "case" expression ":" blockStatements
                                           | "default" ":" blockStatements
                                           ;
extend statement with "break" id
                      | "break"
                      | "continue" id
                      | "continue"
                      | "switch" parExpression "{" switchBlockStmtGroups "}"
                       : ["switch" parExpression switchBlockStmtGroups ]
                      | id ":" statement
                      ;

extend basicType with "byte"
                      | "short"
                      | "char"
                      | "long"
```

```
                    | "float"
                    | "double"
                    ;

extend literal with longIntLiteral
                  | characterLiteral
                  | floatLiteral
                  | doubleLiteral
                  ;

extend variableDeclarator with id abreFechaColchetes "=" arrayInitializer
                              | id "=" arrayInicializer;

arrayInitializer : ArrayInit ::= "{" virgulaVariableInitializer "}" ;
virgulaVariableInitializer : ArrayInit ::= virgulaVariableInitializer "," variableInitializer
: [virgulaVariableInitializer variableInitializer] | variableInitializer:variableInitializer;
variableInitializer : ArrayInit ::= arrayInitializer:arrayInitializer | expression;

extend primary with primary "[" expression "]"
                  | parExpression
                  ;

end
```

B.1.2 Novos Tokens

Segue a especificação de novos *tokens* para que Java 1 incorpore literais de todos os tipos básicos.

```
module NewDefinitions_ConcreteGrammar.NewTokens

public token longIntLiteral : Long = [0-9]+ [lL] is asLong;
public token floatLiteral : Float = ([0-9]+ "." [0-9]* exponentPart? [fF]?)
|  ("." [0-9]+ exponentPart? [fF]?) | ([0-9]+ exponentPart [fF]?) | ([0-9]+ exponentPart? [fF]) is asFloat;
public token doubleLiteral : Double = ([0-9]+ "." [0-9]* exponentPart? [dD])
|  ("." [0-9]+ exponentPart? [dD]) | ([0-9]+ exponentPart? [dD]) is asDouble;
element exponentPart = [eE] ("+" | "-")? [0-9]+;
public token characterLiteral : Char = "\'" ( singleCharacter | escapeCharacter ) "\'" is asCharacter;
element escapeCharacter = [\b\n\t\f\r\'\"];
element singleCharacter = [^\'\\];

end
```

B.2 Funções Auxiliares

B.2.1 Funções da Semântica Estática

Funções que auxiliam a definição semântica de verificação de tipos de Java 1. Elas se encontram no módulo *NewDefinition_StaticSemantic.Functions*.

```
assignableType mychar1 mychar2 = true;
assignableType myshort1 myshort2 = true;
assignableType myfloat1 myfloat2 = true;
assignableType mydouble1 mydouble2 = true;
assignableType mylong1 mylong2 = true;
assignableType myshort mybyte = true;
assignableType inteiro mybyte = true;
assignableType mylong mybyte = true;
assignableType myfloat mybyte = true;
assignableType mydouble mybyte = true;
assignableType inteiro myshort = true;
assignableType mylong myshort = true;
assignableType myfloat myshort = true;
assignableType mydouble myshort = true;
assignableType mylong inteiro = true;
assignableType myfloat inteiro = true;
assignableType mydouble inteiro = true;
assignableType myfloat mylong = true;
assignableType mydouble mylong = true;
assignableType mydouble myfloat = true;
assignableType mylong mychar = true;
assignableType inteiro mychar = true;
assignableType myfloat mychar = true;
assignableType mydouble mychar = true;
assignableType Array(int1,type1) Array(int2,type2) =
  if equalType type1 type2 then true else false;
assignableType type1 type2 = subType type1 type2;
assignableType _ _ = false;

//New definitions to typeToString
typeToString myshort = "short";
typeToString mybyte = "byte";
typeToString mychar = "char";
typeToString mylong = "long";
typeToString myfloat = "float";
typeToString mydouble = "double";

////////////////////////////////////////////////////////////////////////////////////////////
// retorna tipo final de acesso a array
function arrayTypeResult : Type -> Type;
arrayTypeResult array (int,type) = if int-1 = 0
                                   then type //ja e valor final do array
                                   else array (int-1, type);

////////////////////////////////////////////////////////////////////////////////////////////
```

```
//new definitions of public function equalType : Type -> Type -> Bool;
equalType myshort1 myshort2 = true;
equalType mybyte1 mybyte2 = true;
equalType mylong1 mylong2 = true;
equalType myfloat1 myfloat2 = true;
equalType mydouble1 mydouble2 = true;
equalType mychar1 mychar2 = true;
```

B.2.2 Funções da Semântica Dinâmica

Funções que auxiliam a definição semântica de execução Java 1. Elas se encontram no módulo *NewDefinition_DynamicSemantic.Functions*.

```
////////////////////////////////////////////////////////////////////////////////
//Funcoes que obtem o i loc de um dada lista de loc
function getLoc : Loc* -> Int -> Loc;
getLoc loc* int = getLoc2 loc* int 0;

function getLoc2 : Loc* -> Int -> Int -> Loc;
getLoc2 loc:loc* int1 int2 = if int1=int2
 then loc
 else getLoc2 int1 (int2+1) ;
getLoc2 () int1 int2 = loc
where {
loc = -1
};

////////////////////////////////////////////////////////////////////////////////
//Funcao que faz transformacoes de tipo
function evToInt : Ev -> Int;
evToInt int -> int;

function startDynamicEnv : Env;
startDynamicEnv = r
where{
 r0 = newEnv;
 r1 = r0[id1<-fun1];
 r2 = r1[id2<-fun2];
 r3 = r2[id3<-fun3];
 r4 = r3[id4<-fun4];
 r5 = r4[id5<-fun5];
 r6 = r5[id6<-fun6];
 r7 = r6[id7<-fun7];
 r8 = r7[id8<-fun8];
 r9 = r8[id9<-fun9];
 r10 = r9[id10<-fun10];
 r11 = r10[id11<-fun11];
 r12 = r11[id12<-fun12];
 r13 = r12[id13<-fun13];
 r14 = r13[id14<-fun14];
 r15 = r14[id15<-fun15];
 r16 = r15[id16<-fun16];
 r17 = r16[id17<-fun17];
```

```
 r18 = r17[id18<-fun18];
 r19 = r18[id19<-fun19];
 id1 = makeTypeConverterFunctionName mybyte myshort;
 id2 = makeTypeConverterFunctionName mybyte inteiro;
 id3 = makeTypeConverterFunctionName mybyte mylong;
 id4 = makeTypeConverterFunctionName mybyte myfloat;
 id5 = makeTypeConverterFunctionName mybyte mydouble;
 id6 = makeTypeConverterFunctionName myshort inteiro;
 id7 = makeTypeConverterFunctionName myshort mylong;
 id8 = makeTypeConverterFunctionName myshort myfloat;
 id9 = makeTypeConverterFunctionName myshort mydouble;
 id10 = makeTypeConverterFunctionName inteiro mylong;
 id11 = makeTypeConverterFunctionName inteiro myfloat;
 id12 = makeTypeConverterFunctionName inteiro mydouble;
 id13 = makeTypeConverterFunctionName mylong myfloat;
 id14 = makeTypeConverterFunctionName mylong mydouble;
 id15 = makeTypeConverterFunctionName myfloat mydouble;
 id16 = makeTypeConverterFunctionName mychar mylong;
 id17 = makeTypeConverterFunctionName mychar inteiro;
 id18 = makeTypeConverterFunctionName mychar myfloat;
 id19 = makeTypeConverterFunctionName mychar mydouble;
 fun1 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun2 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun3 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun4 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun5 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun6 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun7 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun8 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun9 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun10 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun11 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun12 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun13 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun14 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun15 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun16 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun17 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun18 = \ec ev* s -> /*Especificação deixada em aberto*/
 fun19 = \ec ev* s -> /*Especificação deixada em aberto*/
};
```

ANEXO C – Especificação de Java 2

C.1 Gramática Concreta de Java 2

```
module NewDefinitions2_ConcreteGrammar.NewVariables

import NewDefinitions_ConcreteGrammar.Commands, NewDefinitions_ConcreteGrammar.Declarations,
NewDefinitions_ConcreteGrammar.Expressions, Global.Domains, NewDefinitios_NewTokens,
ConcreteGrammar.Declarations;

extend variableDeclarator with id abreFechaColchetes "=" reference
|  id "=" reference;

extend listaClasses with listaClasses cUnit;

extend modifier with "private static" | "private";

dimentions  ::= specifiedDimentions abreFechaColchetes
      |  specifiedDimentions
      |  abreFechaColchetes arrayInitializer
      ;

specifiedDimentions  ::= specifiedDimentions "[" expression "]"
     |  "[" expression "]"
     ;

reference: Exp ::=  "new" id dimentions
        | "new" basicType dimentions
                | "new" id "(" arguments ")"
                | "new" id "(" ")"
                | referenceType "." id
                | referenceType "." id "(" arg ")"
                | referenceType "." id "(" ")"
                ;
end
```

C.2 Funções Auxiliares

C.2.1 Funções da Semântica Estática

```
module NewDefinitions2_StaticSemantic.Functions
```

```
import StaticSemantic.Functions, NewDefinitions2_ConcreteGrammar.NewVariables,
NewDefinitions2_StaticSemantic.Domains, StaticSemantic.Domains;

public function dtdim : DimentionsOrSpecifiedDimentions -> Env -> Tc -> Type -> TypeCheckAns;

////////////////////////////////////////////////////////////////////////////////
//Novas definicoes da funcao ja declarada em sub-linguagens anteriores
getModifier ["private static"] = tstatic:(tprivate:());
getModifier ["private"] = tprivate:();

////////////////////////////////////////////////////////////////////////////////
//Funca que checa se acesso a membro de classe e permitido
/*AcessEnv: Environment que determina acesso para */
/*Id1: nome do campo*/
/*Id2: nome da classe atual*/
/*Id3: nome da classe do metodo sendo chamado*/
/*Int: indica se campo e construtor ou nao. 0 -> nao; 1 -> sim */
function checkAcess : AcessEnv -> Id -> Id -> Id -> Int -> Bool;
checkAcess ar id1 id2 id3 0 = if id2 = id3
  then true
  else if checkPrivate (ar id1) is true
       then false //aqui vai ter que verificar se e subclasse
       else true
checkAcess ar id1 id2 id3 1 = if checkStatic (ar id1) is true
  then false
  else checkAcess ar id1 id2 0;

////////////////////////////////////////////////////////////////////////////////
//Verifica se lista tem elemento que representa modificador private
function checkPrivate : Tmodifier* -> Bool;
checkPrivate tmodifier:tmodifier* = if  tmodifier is tprivate
   then true
   else checkPrivate tmodifier*;

checkPrivate tModifier = if  tmodifier is tprivate
then true
else false;

function getTypeToId : Type -> Id;
getTypeToId id = id;

public function getClassName : Id -> Id;
getClassName id = id1
where {
string1= (idToString id) ++ "###"; //para dizer que e classe
id1 = string1
};

public function getConstructorName : Env -> ReferenceType -> Id;
getClassName r id = id1
where {
string1= (idToString id) ++ "#";
id1 = string1
```

```
};

public function getIdName : Env -> ReferenceType -> Id;
getClassName r id = id;

end
```

C.2.2 Funções da Semântica Dinâmica

```
module NewDefinitions2_DynamicSemantic.Functions

import NewDefinitions2_ConcreteGrammar.NewVariables, NewDefinitions2_DynamicSemantic.Domains,
DynamicSemantic.Domains;

public function ddim : DimentionsOrSpecifiedDimentions -> Env -> Tc -> Loc -> Store -> Ans;

////////////////////////////////////////////////////////////////////////////////////////
//Inicializa posicoes de memoria
function startLocs : Loc* -> Store -> Store;
startLocs loc:loc1* s = s1
where{
s1 = s2[loc<-used];
s2 = startLocs loc1* s
};
startLocs () s = s;

////////////////////////////////////////////////////////////////////////////////////////
//Aloca lista de posicoes de memoria
function news : Int -> Store -> Loc*;
news int s = if int = 0 then ()
 else loc:loc*
   where {
   loc = new s;
   loc* = news (int-1) s1;
   s1 = s[loc<-used];
   }

////////////////////////////////////////////////////////////////////////////////////////
//Funcao que cria environment de execucao para objeto a partir do descritor de classes
//do objeto
//Env1: environment de variáveis  nao estaticas
//Env2: environment de funcoes nao estaticas
function createEnvObject : Store -> ObjectVariables -> ObjectFunctions -> (Env, Store);
createEnvObject s (id,loc):(id1,loc1)* (id2,fun)* = (r3,s1)
where{
(r4,s2) = createEnvObject s (id1,loc1)* (id2,fun)*;
loc2 = new s2;
      r3 = r4[id<-loc2];
s1 = s2[loc1<-(s2 loc)]
}

createEnvObject s () (id,fun):(id1,fun1)* = (r3,s2)
where{
```

```
(r4,s2) = createEnvObject s () (id1,fun1)*;
r3 = r4[id<-fun1]
}

createEnvObject s () () = (newEnv,s);
end
```

Referências

[Arnolde, Gosling e Holmes 2007]ARNOLDE, K.; GOSLING, J.; HOLMES, D. *A linguagem de Programação Java*. [S.l.]: Bookman, 2007.

[Bigonha 1981]BIGONHA, R. da S. *A Denotational Semantics Implementation System*. Tese (Doutorado) — UNIVERSITY OF CALIFORNIA, 1981.

[Bigonha 2003]BIGONHA, R. da S. *Retractil Continuations*. [S.l.], 2003.

[Gordon 1979]GORDON, M. J. *The Denotational Description of Programming Languages - An Introduction*. [S.l.]: Springer-Verlag, 1979.

[Tirelo e Bigonha 2006]TIRELO, F.; BIGONHA, R. S. *Notus*. [S.l.], 2006. Laboratório de Linguagens de Programação, UFMG.

[Tirelo, Bigonha e Saraiva 2008]TIRELO, F.; BIGONHA, R. S.; SARAIVA, J. A. B. Disentangling denotational semantics definitions. *Journal of Universal Computer Science*, 2008.

[Tirelo, Bigonha e Saraiva 2008]TIRELO, F.; BIGONHA, R. S.; SARAIVA, J. A. B. *Semântica Multidimensional de Linguagens de Programação*. Tese (Doutorado) — Universidade Federal de Minas Gerais, 2008.

Belo Horizonte, 02 de Dezembro de 2008.

_____

Mirlaine Aparecida Crepalde

_____

Roberto da Silva Bigonha