UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

SAMUEL GRIMM MATSCHULAT

Ambiente didático para experimentação com a definição formal de linguagens orientadas a objetos

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Álvaro Freitas Moreira
Co-orientador: Prof. Dr. Rodrigo Machado

Porto Alegre
2016

“The doorstep to the temple of wisdom is a knowledge of our own ignorance. He cannot learn aright who has not first been taught that he knows nothing.”

— CHARLES SPURGEON

AGRADECIMENTOS

Agradeço à sociedade brasileira e à Universidade Federal do Rio Grande do Sul, incluindo seu corpo docente, direção e administração, por possibilitarem a minha formação no curso de Ciência da Computação, e às pessoas que fizeram parte do meu desenvolvimento acadêmico durante essa etapa da minha vida – em especial, os meus orientadores, Prof. Dr. Álvaro Freitas Moreira e Prof. Dr. Rodrigo Machado, pelo direcionamento, pelo incentivo e pela paciência durante o desenvolvimento deste trabalho.

Agradeço também aos meus pais e meus irmãos pelo constante e incondicional apoio, que muitas vezes exigiu a abdicação das suas vontades e aspirações em favor do meu crescimento, e pela convicção, muito acima da minha própria, na minha capacidade de superação de obstáculos e dificuldades.

Por fim, agradeço a todas as pessoas que trabalham e lutam para fazer do mundo um lugar melhor, com mais igualdade e oportunidades a todos.

RESUMO

Neste trabalho é descrita a implementação de um *software* aplicativo gráfico, denominado Cool Inspector, cujo objetivo é oferecer um ambiente que permite a visualização detalhada, tendo a área de semântica formal como contexto, dos passos de avaliação de programas escritos na linguagem Cool 2016. Esta linguagem foi desenvolvida por John Tang Boyland com o objetivo de ser implementada por estudantes de cursos de compiladores no período de um semestre. Os passos de execução de programas são descritos em termos de aplicações de regras da semântica operacional da linguagem, que descrevem de modo formal o comportamento de um programa e compõem uma árvore de prova que mostra a derivação do resultado proveniente de sua execução.

Palavras-chave: Semântica formal. sistema de tipos. linguagens orientadas a objetos.

Didactic environment for experimentation with the formal definition of object-oriented languages

ABSTRACT

This thesis describes the implementation of a graphical software application, named Cool Inspector, whose objective is to allow for a detailed visualization, with the area of formal semantics as context, of the evaluation steps of programs written in the Cool 2016 language. This language was developed by John Tang Boyland with the goal of being implemented by students of compiler courses during a semester. These execution steps are described in terms of applications of the operational semantics rules of the language, which formally describe the behavior of a program and form a proof tree that shows the derivation of the result obtained from the program's execution.

Keywords: formal semantics. type systems. object-oriented languages.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

AST	*Abstract Syntax Tree* (Árvore de Sintaxe Abstrata)
Caml	*Categorical Abstract Machine Language*
Camlp4	*Caml Preprocessor and Pretty-Printer*
Cool	*Classroom Object-Oriented Language*
GTK	*GIMP Toolkit*
INRIA	*Institut National de Recherche en Informatique et en Automatique* (Instituto Nacional de Pesquisa em Informática e Automação)
OCaml	*Objective Caml*

SUMÁRIO

1 INTRODUÇÃO

Em ciência da computação, semântica formal é o estudo matemático do significado de linguagens de programação. Segundo Nielson e Nielson, a semântica formal procura "especificar rigorosamente o significado, ou o comportamento, de programas, peças de hardware, etc", e que a necessidade desse rigor surge porque "ele pode revelar ambiguidades e complexidades sutis em documentos de especificações aparentemente claros, e pode formar a base para a implementação, análise e verificação" (1992, p. 1). Slonneger e Kurtz procuram esclarecer, de forma mais acessível, o significado da semântica formal aplicada às linguagens de programação, traçando um paralelo com a semântica de linguagens naturais:

> A semântica revela o significado de expressões sintaticamente válidas em uma linguagem. Para as linguagens naturais, isso significa correlacionar sentenças e frases com os objetos, pensamentos e sentimentos de nossas experiências. Para as linguagens de programação, a semântica descreve o comportamento que um computador toma quando executa um programa em tal linguagem. Podemos mostrar esse comportamento descrevendo a relação entre a entrada e a saída de um programa ou por uma explicação passo-a-passo de como um programa vai ser executado em uma máquina real ou abstrata. (SLONNEGER; KURTZ, 1995, p. 1)

O estudo de semântica formal por vezes se mostra complexo para os estudantes, pois muitos conceitos são ensinados de forma abstrata e possuem notações carregadas. O objetivo deste trabalho é o desenvolvimento de um aplicativo de auxílio na visualização e experimentação desses conceitos.

O aplicativo desenvolvido explora em particular a definição de linguagens orientadas a objetos, visto que programas de tais linguagens apresentam comportamentos complexos e são de difícil análise, em razão de que "os estados de execução desses programas contém objetos inter-relacionados com estruturas e propriedades complexas" (KE et al., 2009, p. 1). Mais especificamente, foi implementado um interpretador para a linguagem Cool 2016, proporcionando um ambiente onde a definição da semântica de uma linguagem orientada a objetos pode ser estudada e analisada do forma concreta.

Cool (*Classroom Object-Oriented Language*) é uma linguagem de programação projetada para ser implementada em um semestre como projeto em cursos de compiladores. A primeira versão foi desenvolvida por Alex Aiken em 1995 com base na linguagem Sather164, e continuou sendo atualizada por Aiken e, desde 2000, por John Tang

Boyland. Cool 2013 é uma nova linguagem projetada por Boyland e difere substancialmente de versões anteriores, sendo ela um subconjunto da linguagem Scala. Cool 2016 (BOYLAND, 2016) é a versão utilizada neste trabalho. Todas as menções à Cool que se seguem referem-se à Cool 2016.

O manual de referência de Cool descreve a semântica da linguagem de maneira formal, utilizando-se de *regras de avaliação*. Uma dessas regras, que mostra distintamente a notação carregada mencionada previamente, é reproduzida abaixo:

Figura 1.1: Regra de avaliação DISPATCH da linguagem Cool

DISPATCH

$$\frac{\begin{array}{c} C, so, S_0, E \vdash e_0 : v_0, S_1 \qquad C, so, S_1, E \vdash e_1 : v_1, S_2 \qquad \vdots \qquad C, so, S_n, E \vdash e_n : v_n, S_{n+1} \\ v_0 = X(a_1 = l_{a_1}, \ldots, a_m = l_{a_m}) \qquad implementation(X, f) = (X', x_1, \ldots, x_n, e_{n+1}) \\ l_{x_i} = newloc(S_{n+1}), \text{ for } i = 1 \ldots n \text{ and each } l_{x_i} \text{ is distinct} \qquad S_{n+2} = S_{n+1}[v_1/l_{x_1}, \ldots, v_n/l_{x_n}] \\ X', v_0, S_{n+2}, [a_1 : l_{a_1}, \ldots, a_m : l_{a_m}, x_1 : l_{x_1}, \ldots, x_n : l_{x_n}] \vdash e_{n+1} : v_{n+1}, S_{n+3} \end{array}}{C, so, S_0, E \vdash v = e_0.f(e_1,\ldots,e_n) : v_{n+1}, S_{n+3}}$$

Fonte: Boyland (2016, p. 26)

O aplicativo desenvolvido permite a visualização dessas regras de forma contextualizada, ou seja, sendo utilizadas por um interpretador da linguagem no processo de avaliação de programas.

O texto está organizado da seguinte maneira: o capítulo 2 se dedica à revisão da literatura existente sobre semântica formal e a linguagem Cool, o capitulo 3 apresenta o aplicativo desenvolvido e expõe decisões tomadas ao longo de sua implementação, e o capítulo 4 apresenta as conclusões obtidas e possibilidades de extensão do trabalho.

2 REVISÃO TEÓRICA

Neste capítulo é feita a revisão de literatura relacionada à semântica formal e à linguagem Cool.

2.1 Semântica formal

A semântica formal de uma linguagem de programação pode ser definida em dois níveis: *Semântica estática*, ou *tipagem*, e *Semântica dinâmica*, ou simplesmente semântica.

A semântica estática de uma linguagem descreve propriedades de um programa que podem ser determinadas sem que se considere os resultados provenientes de sua execução. Programas que apresentam essas propriedades são programas *bem tipados*, pois elas garantem que operações não serão aplicadas a tipos de dados que elas não suportam. Por exemplo, diversas linguagens de programação, como Java e C#, não permitem a expressão `1 + true` em programas bem tipados devido à operação + suportar apenas argumentos de tipos numéricos e `true` ser uma expressão de tipo *booleano*.

A semântica dinâmica de uma linguagem descreve o comportamento de um programa durante a sua execução. Manuais e especificações de linguagens de programação geralmente utilizam linguagem natural para expressar esse comportamento, o que pode resultar em ambiguidades e omissões que não são aparentes à primeira vista. A fim de se evitar tais erros, é possível definir formalmente a semântica de uma linguagem ou parte dela.

Diferentes estilos podem ser utilizados para a definição formal da semântica de uma linguagem. Os estilos mais frequentemente mencionados na literatura são *semântica denotacional*, *semântica axiomática* e *semântica operacional*, sendo este último o estilo empregado neste trabalho.

A seguir são apresentados alguns conceitos utilizados na definição de semânticas operacionais de linguagens de programação. Para um detalhamento mais profundo tais conceitos pode-se consultar a literatura sobre o assunto, incluindo Fernández (2014), Nielson e Nielson (1992), Slonneger e Kurtz (1995) e Winskel (1993).

2.1.1 Árvores de sintaxe abstrata

Usuários de uma linguagem de programação usual escrevem programas através de sequências de caracteres que constituem o texto do código do programa. Esse texto deve seguir as regras sintáticas da linguagem para ser aceito como código executável. Essas regras sintáticas são chamadas de *sintaxe concreta* da linguagem.

Sequências de caracteres, entretanto, não são estruturas de dados adequadas para serem interpretadas como um programa, pois não correspondem à estrutura formada pelos elementos que constituem o programa. Por esse motivo, antes da interpretação do programa o texto do código é transformado em uma estrutura mais apropriada. Para isso, é necessária a identificação dos elementos previamente mencionados.

Na primeira fase da transformação, os elementos identificados são as "palavras" consideradas válidas pela linguagem, chamadas de *tokens*, através de um processo chamado *análise léxica* ou *lexing*.

Na segunda fase os *tokens* são agrupados e organizados em uma árvore n-ária. Cada nó dessa árvore corresponde a um elemento do programa ao qual pode-se atribuir um significado. Tal processo é chamado de *análise sintática* ou *parsing*, e os elementos são chamados de *expressões* ou, de forma mais geral, *construções* da linguagem.

A razão pela qual são utilizadas árvores é o fato de as construções de uma linguagem serem formadas pela combinação de zero ou mais subconstruções, de forma recursiva. Essas árvores correspondem à *sintaxe abstrata* da linguagem e, portanto, são chamadas de *árvores de sintaxe abstrata* (ou ASTs, do inglês *abstract syntax trees*).

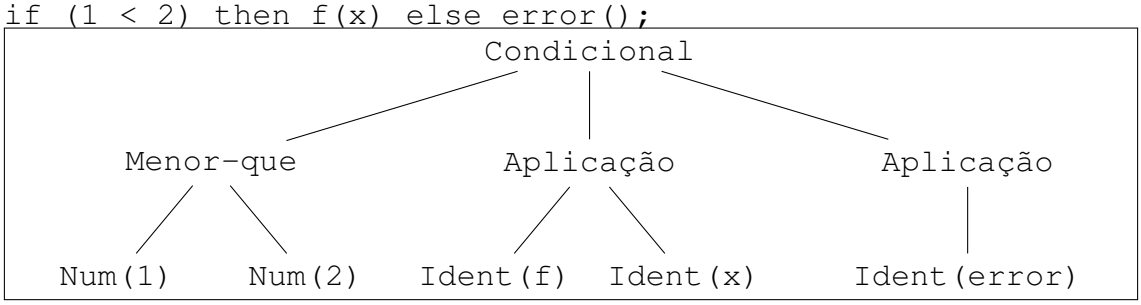
Por exemplo, o código `if (1 < 2) then f(x) else error();`, ao ser usado como entrada por um interpretador hipotético, passará pelas duas fases de identificação de elementos descritas acima, produzindo:

1. A lista de *tokens* | `if` | `(` | `1` | `<` | `2` | `)` | `then` | `f` | `(` | `x` | `)` | `else` | `error` | `(` | `)` | `;` |;
2. A árvore de sintaxe abstrata apresentada na Figura 2.1.

Pode-se observar na Figura 2.1 que certos elementos do texto do programa, como parênteses e pontos e vírgulas, são omitidos na árvore de sintaxe abstrata. Isto pode ser feito porque a relação entre os elementos são expressados diretamente na estrutura da árvore, tornando símbolos de agrupamento e separação desnecessários.

A semântica formal de uma linguagem é definida sobre a sua sintaxe abstrata. Entretanto, a representação dessa sintaxe em forma de árvore não é prática devido ao

Figura 2.1: Exemplo de árvore de sintaxe abstrata que pode ser gerada a partir do código `if (1 < 2) then f(x) else error();`



Fonte: o autor

espaço ocupado e à complexidade de reprodução. Por esse motivo é comum essas árvores serem representadas através dos textos em sintaxe concreta a partir dos quais elas podem ser geradas.

2.1.2 Regras de inferência

Regras de inferência são formas argumentativas que permitem a justificação de uma conclusão que se segue de determinadas premissas.

Uma possível definição formal matemática de regras de inferência é: dado um conjunto T de objetos (conjunto universo), uma regra de inferência é um par ordenado (H, c) onde $H \subseteq T$ é um conjunto de *hipóteses* ou *premissas* e $c \in T$ é uma *conclusão*. Caso o conjunto de hipóteses seja vazio a regra é um *axioma*.

A notação usual para a descrição de regras de inferência é:

$$\frac{h_1 \qquad h_2 \qquad \ldots \qquad h_n}{c} \qquad \text{(NOME-DA-REGRA)}$$

Essas regras são úteis para a definição indutiva de subconjuntos do conjunto universo T. Um subconjunto $I \subseteq T$ é formado pelas conclusões que fazem parte de regras nas quais todas as hipóteses também fazem parte do conjunto I ou que são axiomas. Por exemplo, a seguir são descritas duas regras que definem o conjunto dos números pares:

$$\frac{}{0}\ \text{(ZERO)} \qquad\qquad \frac{n}{n+2}\ \text{(MAIS-2)}$$

A regra MAIS-2 é, na realidade, um *esquema* ou *modelo de regra de inferência*, devido à presença da metavariável n. As regras reais que são representadas por um esquema podem ser formadas através da substituição das metavariáveis por elementos do

conjunto universo, possivelmente resultando em um número infinito de regras. O esquema exemplo MAIS-2 representa, considerando os números naturais como conjunto universo, as seguintes regras:

$$\frac{0}{0+2 \text{ (ou } 2)} \qquad \frac{1}{1+2 \text{ (ou } 3)} \qquad \frac{2}{2+2 \text{ (ou } 4)} \qquad \frac{3}{3+2 \text{ (ou } 5)} \qquad \ldots$$

Note que as regras que concluem números ímpares não resultam na inclusão destes no conjunto, visto que as suas hipóteses não podem ser inferidas através do sistema formado pelo conjunto completo das regras.

É também possível restringir a substituição de metavariáveis através da introdução de condições junto ao esquema. Por exemplo, para limitar o conjunto acima a números pares menores ou iguais a 10, pode-se substituir o esquema de regra MAIS-2 por:

$$\frac{n}{n+2}, \text{ se } n \leqslant 8 \quad (\text{MAIS-2}_{\leq 10}) \qquad \text{ou} \qquad \frac{n \qquad n \leqslant 8}{n+2} \ (\text{MAIS-2}_{\leq 10})$$

A segunda versão do esquema MAIS-$2_{\leq 10}$ apresentado acima mostra que pode-se tomar a liberdade de estender a notação de descrição de regras de inferência para permitir que restrições sejam declaradas junto às hipóteses.

A utilização de uma regra no processo de prova ou justificação de uma conclusão é chamada de *aplicação* da regra. A aplicação sucessiva das regras de um sistema forma uma *árvore de prova*, onde a raiz é a conclusão provada, as folhas da árvore são axiomas e os nodos são regras cujas hipóteses são as conclusões das sub-árvores. O formato geral de uma árvore de prova é:

$$\dfrac{\dfrac{\dfrac{\vdots}{t_{1\,1}} \quad \ldots \quad \dfrac{\vdots}{t_{1\,m_1}}}{t_1} \quad \ldots \quad \dfrac{\dfrac{\vdots}{t_{n\,1}} \quad \ldots \quad \dfrac{\vdots}{t_{n\,m_1}}}{t_n}}{t}$$

Convencionalmente, esquemas de regras são tratados como se fossem as próprias regras de inferência; essa convenção será adotada também no restante deste texto.

2.1.3 Relações binárias

Uma relação binária entre dois conjuntos A e B é um conjunto R de pares ordenados onde $R \subseteq A \times B$. Para expressar que o par ordenado (a, b) pertence a uma relação R, é comum o uso da notação infixa $a\ R\ b$. A declaração oposta, de que o par ordenado (a, b) **não** pertence à relação R, pode ser expressa como $a \not R\ b$.

Um exemplo de uma relação binária é a relação $\leq$ (menor ou igual a) entre números naturais. Nesse contexto, $\leq$ é um subconjunto de $\mathbb{N} \times \mathbb{N}$ e contém todos os pares ordenados (a, b) onde a e b são números naturais e a é menor ou igual a b; ou seja, $\leq = \{(0,0), (0,1), (1,1), (0,2), (1,2), (2,2), (0,3), \dots\}$. Utilizando-se da notação infixa temos, por exemplo, que $1 \leq 2$ e $2 \not\leq 1$.

2.1.4 Sistemas de transição

Um sistema de transição é especificado por uma relação $\rightarrow\ \subseteq\ \mathrm{Conf} \times \mathrm{Conf}$, chamada de *relação de transição*, onde Conf é um conjunto de *configurações* ou *estados*. A expressão $c \rightarrow c'$ denota uma transição de c para c', que pode ser entendida como uma mudança de estado produzida por uma computação.

Um sistema de transição é *determinístico* se, para qualquer configuração c, $c \rightarrow c_1$ e $c \rightarrow c_2$ implica em $c_1 = c_2$. Em termos de semântica de linguagens de programação, um sistema é determinístico se a qualquer ponto da execução de um programa o próximo passo de computação é unicamente definido. Sistemas *não determinísticos* podem ser utilizados para modelar linguagens concorrentes. A análise de tais linguagens não está no escopo deste trabalho.

Uma configuração c é *final* ou *terminal* quando não existe uma configuração c' tal que $c \rightarrow c'$. A definição da semântica operacional de uma linguagem inclui a especificação de quais configurações descrevem resultados de uma computação. Se uma configuração final não condiz com um desses resultados ela é dita *bloqueada* e, em geral, indica um estado de erro na execução de um programa. A ocorrência desses erros pode ser evitada, ou ao menos minimizada, através da definição de uma semântica estática, como descrito no início deste capítulo.

2.1.5 Semântica operacional

A semântica operacional de uma linguagem é definida por um sistema de transição entre estados de programas. Existem dois estilos que podem ser seguidos ao se decidir quais estados estão diretamente relacionados, definindo duas categorias de semânticas operacionais:

- Semântica operacional *small-step* ou estrutural — cada transição representa um passo individual de computação. A avaliação de um programa se dá pela justificação sucessiva das configurações não finais, ou seja, se $c \rightarrow c'$ e c' não é uma configuração final, a execução do programa se segue pela justificação de $c' \rightarrow c''$, e assim sucessivamente.

- Semântica operacional *big-step* ou natural — cada transição representa uma computação completa, ou seja, para todo $c \Downarrow c'$, sendo $\Downarrow$ a relação que define o sistema, c' é uma configuração final.

Figura 2.2: Exemplos de regras de redução referentes à construção *if* em semântica operacional *small-step*

$$\frac{\langle B, s_1\rangle \rightarrow \langle B', s_2\rangle}{\langle \textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, s_1\rangle \rightarrow \langle \textsf{if } B' \textsf{ then } C_1 \textsf{ else } C_2, s_2\rangle} \quad \text{(IF)}$$

$$\frac{}{\langle \textsf{if true then } C_1 \textsf{ else } C_2, s\rangle \rightarrow \langle C_1, s\rangle} \quad \text{(IF-TRUE)}$$

$$\frac{}{\langle \textsf{if false then } C_1 \textsf{ else } C_2, s\rangle \rightarrow \langle C_2, s\rangle} \quad \text{(IF-FALSE)}$$

Fonte: adaptado de Fernández (2014, p. 78)

Figura 2.3: Exemplos de regras de avaliação referentes à construção *if* em semântica operacional *big-step*

$$\frac{\langle B, s_1\rangle \Downarrow \langle \textsf{true}, s_2\rangle \qquad \langle C_1, s_2\rangle \Downarrow \langle C_1', s_3\rangle}{\langle \textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, s_1\rangle \Downarrow \langle C_1', s_3\rangle} \quad \text{(IF-TRUE)}$$

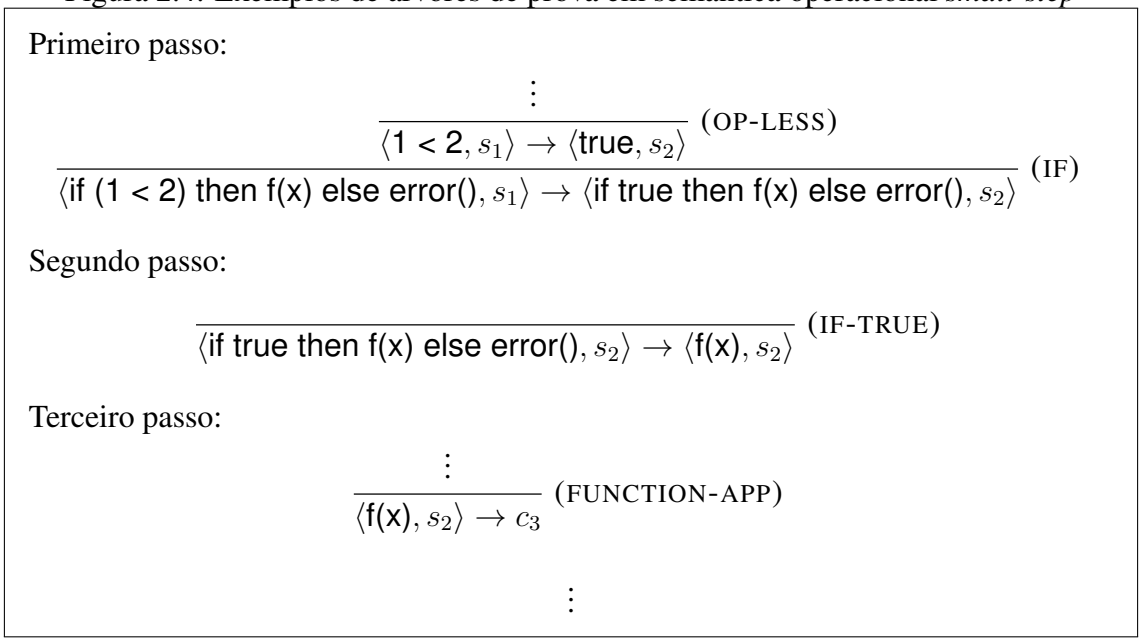
$$\frac{\langle B, s_1\rangle \Downarrow \langle \textsf{false}, s_2\rangle \qquad \langle C_2, s_2\rangle \Downarrow \langle C_2', s_3\rangle}{\langle \textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, s_1\rangle \Downarrow \langle C_2', s_3\rangle} \quad \text{(IF-FALSE)}$$

Fonte: adaptado de Fernández (2014, p. 82)

Sistemas de transições são comumente definidos através de regras de inferência. Essas regras são chamadas *regras de redução* em semânticas *small-step*, e *regras de avaliação* em semânticas *big-step*. As figuras 2.2 e 2.3 mostram como as regras para a avaliação da construção *if* de uma linguagem de programação tradicional poderiam ser expressadas utilizando-se de cada uma dessas duas categorias de semânticas operacionais.

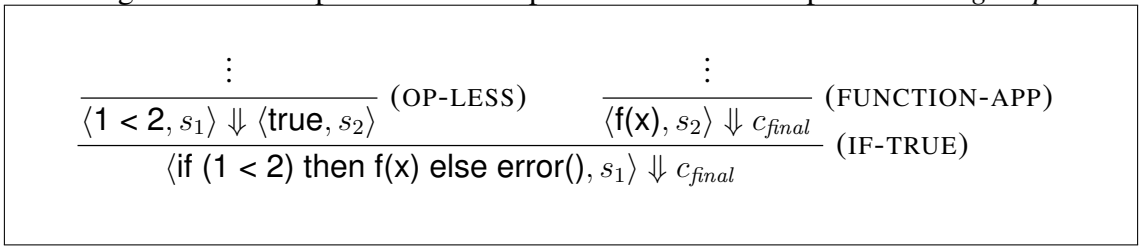
Nesses exemplos acima, uma configuração é representada pelo par $\langle e, s \rangle$, onde e é a expressão sendo avaliada e s é o estado da máquina onde o programa está sendo executado. As figuras 2.4 e 2.5 exemplificam como regras de inferência são utilizadas na avaliação da expressão if (1 < 2) then f(x) else error(), formando uma (semântica *big-step*) ou mais (semântica *small-step*) árvores de prova.

Figura 2.4: Exemplos de árvores de prova em semântica operacional *small-step*



Fonte: o autor

Figura 2.5: Exemplo de árvore de prova em semântica operacional *big-step*



Fonte: o autor

A inclusão do estado da máquina nas regras de transição permite a obtenção de resultados diferentes de acordo com o conteúdo da memória da máquina, possibilitando à linguagem oferecer recursos de manipulação dessa memória, como definição e atribuição de variáveis e estruturas de dados mutáveis.

2.2 Linguagem Cool

Esta seção apresenta de maneira informal a linguagem Cool. Ela é descrita de forma mais extensiva em *CoolAid: The Cool 2016 Reference Manual* (BOYLAND, 2016), o manual de referência da linguagem.

A linguagem de programação Cool apresenta várias características que existem em linguagens modernas, como objetos, tipagem estática e gerenciamento automático de memória. Um programa em Cool consiste de um conjunto de *classes*. Cada classe define um novo tipo de dado, encapsulando as variáveis e os procedimentos aplicáveis a ele. Uma instância de uma classe é um *objeto*.

A maior parte das construções em Cool são *expressões*, e cada expressão possui um valor e um tipo. Cool é *type safe*, ou seja, é garantido que um procedimento será aplicado a dados do tipo correto, evitando assim erros de tipo em tempo de execução.

Comentários em Cool podem ser escritos de duas formas: iniciados por `//` e finalizados ao fim da linha ou entre `/*` e `*/`, podendo se estender por múltiplas linhas. Sequências de comentários, espaços e quebras de linha não possuem significado na linguagem além de servir como separadores entre *tokens*.

Na descrição da linguagem que se segue, as seguintes metavariáveis são utilizadas em certos trechos de código:

- id — um identificador iniciado em letra minúscula e seguido de zero ou mais letras, números e caracteres _;
- $type$ — um identificador de tipo iniciado em letra maiúscula e seguido de zero ou mais letras, números e caracteres _;
- $expr$ — uma expressão em Cool, como descrito na Seção 2.2.4;
- $block$ — um bloco implícito, como descrito na Seção 2.2.4.5.

Adicionalmente, as seguintes notações também são utilizadas:

- $[\ldots]$ — denota uma construção opcional;
- $\ldots \mid \ldots$ — significa que apenas uma de duas construções deve ser utilizada;
- $(\ldots)^+$ — significa que a construção entre parênteses deve ocorrer uma ou mais vezes.

2.2.1 Classes

Uma classe em Cool possui a seguinte forma:

`class` *type*`(`*var-formals*`)` [`extends` *type*`(`*actuals*`)`] `{`
    feature-list
`}`

var-formals é a lista dos parâmetros formais da classe, contendo zero ou mais elementos separados por '`,`', onde cada parâmetro possui a forma `var` *id* `:` *type*.

O corpo da classe define uma lista de *features*, que podem ser *atributos* ou *métodos*. Todos os atributos possuem escopo local à classe e todos os métodos possuem escopo global, ou seja, o único meio de acesso ao estado de um objeto é através dos métodos. Parâmetros formais de classes (*var-formals*) também são considerados atributos.

A linguagem Cool também suporta herança simples através da construção `extends` *type*`(`*actuals*`)`, onde *type* é a classe pai ou superclasse e *actuals* é a lista de argumentos na forma *expr*`,...,`*expr* a serem passados para a classe pai. Classes definidas sem especificar uma classe pai herdam automaticamente da classe `Any`, que é a classe base em Cool.

Além de `Any`, Cool possui outras sete *classes básicas*: `Unit`, `Int`, `String`, `Boolean`, `ArrayAny`, `IO` e `Symbol`. Dessas classes adicionais, apenas `IO` pode ser usada como superclasse. `Unit`, `Int` e `Boolean` são *classes valores* pois variáveis desses tipos não podem ter o valor `null` atribuído a elas. Essas classes básicas são implementadas no arquivo `basic.cool` cujo conteúdo é adicionado ao programa do usuário.

Um objeto de um determinado tipo é construído usando-se a expressão `new` *type*`(`*expr*`,...,`*expr*`)`. Todo programa em Cool deve possuir uma classe chamada `Main` com zero parâmetros (usualmente sendo uma subclasse de `IO`). Um programa é executado a partir da avaliação da expressão `new Main()`.

2.2.2 Tipos

Cool possui apenas dois tipos que não são classes: `Null` e `Nothing`. O único valor do tipo `Null` é `null`. Não há valores que possuem o tipo `Nothing`.

Se em certo ponto de um programa um valor do tipo A é esperado, pode-se usar também um valor do tipo B desde que A seja um antecessor de B na hierarquia de tipos.

Nesse caso, dizemos B está em *conformidade* com A ou $B \leq A$. Dessa definição, segue-se que $A \leq$ `Any` para qualquer tipo A, visto que todas as classes herdam da classe `Any`.

A operação de conformidade é adicionalmente definida pelas regras: $A \leq A$ e `Nothing` $\leq A$ para qualquer tipo A, e `Null` $\leq A$ para qualquer tipo A exceto as três classes valores (`Unit`, `Int` e `Boolean`).

Outra operação definida sobre tipos em Cool é a operação de *combinação*, representada pelo operador $\vee$. Dizemos que $A \vee B = C$ se $A \leq C$, $B \leq C$ e, para todo C' onde $A \leq C'$ e $B \leq C'$, $C \leq C'$.

O verificador de tipos de Cool infere, usando as declarações de tipos fornecidas pelo programador, o tipo de todas as expressões de um programa. Esses tipos inferidos são chamados de *tipos estáticos*. Durante a execução, é possível que uma expressão seja avaliada a um valor de um tipo diferente de seu tipo estático; esse tipo é chamado de *tipo dinâmico*.

Para evitar erros de incompatibilidade de tipos durante a execução de um programa, é necessário que o verificador de tipos seja *correto*, ou seja, para qualquer expressão e com tipo dinâmico D_e e tipo estático S_e, $D_e \leq S_e$.

2.2.3 *Features*

Definições de atributos possuem a seguinte forma:

`var` *id* `:` *type* `=` *expr*`;`

O tipo de um atributo não pode ser `Nothing`.

Definições de métodos possuem a seguinte forma:

`[ override ] def` *id*`(`*id* `:` *type*`,...,`*id* `:` *type*`):` *type* `=` *expr*`;`

A lista de parâmetros pode possuir zero ou mais elementos, e os identificadores utilizados na lista devem ser distintos. O tipo do corpo do método deve estar em conformidade com tipo de retorno declarado.

O modificador `override` deve ser usado se e somente se uma classe ancestral define um método com o mesmo nome. Nesse caso, o número de parâmetros e os seus tipos devem ser os mesmos, e o tipo de retorno do método redefinido deve estar em conformidade com o tipo de retorno do método original.

Construtores são métodos definidos de forma implícita para todas as classes. Os construtores inicializam os parâmetros da classe, chamam o construtor da classe pai,

inicializam os atributos, executam os *inicializadores*, que são definidos usando-se `{` *block* `};` no corpo da classe (i.e. na lista de *features*), e, por fim, retornam o novo objeto. Os inicializadores e as inicializações dos atributos são executados na ordem em que eles aparecem no texto do programa.

2.2.4 Expressões

A seguir são apresentadas as expressões de Cool, assim como os tipos e valores que elas produzem de acordo com as semânticas estática e operacional da linguagem.

2.2.4.1 Literais

São expressados em código Cool em uma das formas a seguir:

- `()`, o único valor do tipo `Unit`;
- `true` e `false`, os valores do tipo `Boolean`;
- `0` ou uma sequência de dígitos não iniciada em `0`, que são valores do tipo `Int`;
- Sequência de caracteres entre aspas duplas, como `"exemplo"`, ou três aspas duplas, como `"""exemplo"""`, que são valores do tipo `String`;
- `null`, que pode assumir qualquer tipo exceto os definidos pelas três classes valores (`Unit`, `Int` e `Boolean`).

Literais *string* da primeira forma não podem conter quebras de linha ou o caractere '`\`', a não ser que o caractere que segue forme uma das seguintes sequências:

`\0`	caractere nulo
`\b`	*backspace*
`\t`	tabulação horizontal
`\n`	nova linha
`\r`	retorno de carro
`\f`	*form feed*
`\"`	aspas duplas
`\\`	barra invertida

Literais *string* da segunda forma (entre três aspas duplas) podem conter qualquer sequência de caracteres exceto três aspas duplas.

2.2.4.2 Identificadores

Se referem a nomes de variáveis locais, parâmetros formais, atributos da classe e `this`. Os identificadores devem iniciar em letra minúscula e podem conter letras, números e o caractere _.

Variáveis locais e parâmetros formais possuem escopo léxico. Atributos são visíveis nas classes às quais eles pertencem, exceto se eles forem ocultados por parâmetros de métodos. Variáveis são visíveis a partir do ponto em que são introduzidas até o fechamento do bloco de sua declaração. Parâmetros formais de métodos são visíveis dentro dos métodos. Qualquer identificador pode ser ocultado por identificadores introduzidos em um casamento de padrões, mas não por variáveis.

O valor de um identificador é o valor contido no local de memória referente ao identificador. O tipo de um identificador é o tipo que foi conferido a ele no momento de sua introdução.

O identificador `this` é vinculado implicitamente em todas as classes. Ele pode ser referenciado, mas é um erro atribuir a `this` ou vinculá-lo a uma variável ou um parâmetro.

O valor de `this` é o objeto sobre o qual o método sendo executado foi chamado. O tipo de `this` é a classe na qual o método que contém a expressão é definido.

2.2.4.3 Atribuições

Possuem a forma id `=` $expr$, onde id referencia um atributo ou uma variável. O tipo estático da expressão deve estar em conformidade com o tipo declarado do identificador. O valor e o tipo de uma atribuição são, respectivamente, `()` e `Unit`. As regras da semântica formal de Cool que estabelecem esses resultados são vistas na Seção 2.2.6, ao final deste capítulo.

2.2.4.4 Chamadas de métodos

Podem ser expressadas de uma das seguintes três formas:

1. $expr$ `.` id `(` $expr$ `,` ... `,` $expr$ `)`

   A avaliação da expressão e_0 `.` f `(` e_1 `,` ... `,` e_n `)` incorre primeiramente na avaliação das subexpressões e_0 a e_n, da esquerda para a direita. Se o valor de e_0 for `null`, um erro de execução é gerado; caso contrário, o método f da classe do tipo dinâmico de

e_0 é invocado, com os seus parâmetros vinculados aos argumentos e_1 a e_n e `this` vinculado a e_0;

2. *id* `(`*expr*`,...,`*expr*`)`
   Essa forma é uma abreviação de `this.`*id* `(`*expr*`,...,`*expr*`)`;

3. `super.`*id* `(`*expr*`,...,`*expr*`)`
   Essa forma provê acesso a métodos de classes ancestrais que foram sobrescritos (através de `override`).

O valor de uma chamada de método é o valor da expressão dada como corpo do método, com os parâmetros devidamente vinculados aos argumentos passados. O tipo de uma chamada de método é o tipo de retorno declarado na definição do método.

2.2.4.5 Blocos

Podem ser explícitos ou implícitos. Blocos explícitos devem estar entre chaves (`{ ... }`), enquanto blocos implícitos são usados quando a expressão esperada deve ser um bloco. Blocos possuem a forma *expr*`;...;` *expr* (a última expressão não é seguida de `;`). O valor e o tipo de um bloco são o valor e o tipo da última expressão do bloco, ou `()` e `Unit` caso o bloco seja vazio.

2.2.4.6 Declarações de variáveis

Possuem a forma `var` *id* `:` *type* `=` *expr*. Devem ocorrer diretamente dentro de blocos (ou seja, não podem ser subexpressões de expressões que não sejam blocos) e devem ser seguidas de ao menos uma expressão antes do fechamento do bloco. Não possuem valor ou tipo, visto que o valor e o tipo de uma expressão não final em um bloco são ignorados.

2.2.4.7 Condicionais

Possuem a forma `if (`*expr*`)` *expr* `else` *expr*. A avaliação de um condicional é iniciada pela avaliação do predicado, cujo tipo deve ser `Boolean`. Se o valor for `true`, a segunda expressão é avaliada. Se o valor for `false`, a terceira expressão é avaliada.

O valor de um condicional é o valor da última expressão avaliada. O tipo de um condicional é a combinação dos tipos da segunda e da terceira expressão.

2.2.4.8 Laços

Possuem a forma `while (`*expr*`)` *expr*. Primeiramente o predicado, cujo tipo deve ser `Boolean`, é avaliado. Se o valor do predicado for `false`, o laço é finalizado e o valor `()` é retornado. Se o valor do predicado for `true`, o corpo do laço é avaliado e o processo repetido. O valor e o tipo de um laço são, respectivamente, `()` e `Unit`.

2.2.4.9 Casamento de padrões

São expressados da seguinte forma:

expr `match {`
`(` `case` *id* `:` *type* `=>` *block*
`|` `case null =>` *block* `)`$^{+}$
`}`

Cada construção `case ... =>` *block* é chamada de *ramo*. Nos ramos do tipo `case` *id* `:` *type*, *type* deve ser uma classe (não `Null` ou `Nothing`).

Casamentos de padrões proveem uma maneira de testar o tipo dinâmico de um objeto. Primeiramente, *expr* é avaliado. Caso o resultado seja `null`, o ramo `case null` é escolhido. Caso contrário, sendo C o tipo dinâmico de *expr*, o primeiro ramo `case` *id* `:` *type* onde $C \leq$ *type* é escolhido e o identificador *id* é temporariamente vinculado ao resultado de *expr*. Por fim, o bloco associado ao ramo escolhido é avaliado. Se nenhum ramo puder ser escolhido um erro de execução é gerado.

O valor de um casamento de padrões é o valor do bloco avaliado, e o seu tipo é a combinação dos tipos dos blocos de todos os ramos.

2.2.4.10 Criação de objetos

Possuem a forma `new` *type*`(`*expr*`,...,`*expr*`)`, onde *type* deve ser uma classe qualquer exceto uma classe valor, `Any` ou `Symbol`. A avaliação inicia-se pela criação de um novo objeto do tipo *type* e avaliação do construtor de *type* com os valores dos argumentos vinculados aos parâmetros formais de *type* e o objeto criado a `this`.

Na sintaxe abstrata de Cool uma expressão `new` é expandida em uma chamada de método na forma (new *type*).*type*(*expr*,...,*expr*), onde o método invocado é o construtor da classe *type*.

O valor de uma expressão `new` é o novo objeto criado, e o seu tipo é $type$.

2.2.4.11 Operações binárias

Possuem a forma $expr \;\; op \;\; expr$, onde op deve ser +, −, $\star$ ou / para operações aritméticas, ou <, <= ou == para operações comparativas.

Exceto quando o operador é ==, a avaliação de uma operação binária se dá pela avaliação das duas subexpressões na ordem em que aparecem, seguida pela avaliação da operação. As subexpressões devem ser do tipo `Int`.

O valor da expressão é o valor resultante da avaliação da operação, e o tipo é `Int` para operações aritméticas e `Boolean` para operações comparativas.

Expressões na forma e_1 `==` e_2 são transformadas em chamadas de método e_1`.equals(`e_2`)`. Assim sendo, as regras de chamadas de método se aplicam, o que significa que o operador == não pode ser utilizado para comparar `null`. Para tal, pode-se utilizar $expr$ `match { case null => ... }`.

2.2.4.12 Operações unárias

Possuem a forma $op \;\; expr$, onde op deve ser − ou `!` (negação lógica). O valor da expressão é o valor resultante da avaliação da operação, e o tipo é o mesmo de $expr$, que deve ser `Int` para a operação − e `Boolean` para a operação `!`.

2.2.5 Exemplos

O Código 2.1 apresenta o programa "*Hello World*" na linguagem Cool.

Código 2.1 – Programa exemplo `hello_word.cool`

```
1. class Main() extends IO() {
2.     {
3.         out_string("Hello world!")
4.     };
5. }
```

O Código 2.2 apresenta um programa que define classes que implementam listas, baseando-se no exemplo presente no manual de referência da linguagem Cool, e um procedimento que ordena essas listas. O inicializador da classe `Main` constrói uma lista com

valores não ordenados e chama a rotina de ordenamento, imprimindo os valores da lista antes e depois da operação.

Código 2.2 – Programa exemplo `sort_list.cool`

```
1.  class List() {
2.      def isNil() : Boolean = abort();
3.      def head() : Int = abort();
4.      def tail() : List = abort();
5.  }
6.
7.  class Nil() extends List() {
8.      override def isNil() : Boolean = true;
9.  }
10.
11. class Cons(var car : Int,
12.             var cdr : List) extends List() {
13.     override def isNil() : Boolean = false;
14.     override def head() : Int = car;
15.     override def tail() : List = cdr;
16. }
17.
18. class Main() extends IO() {
19.     def insert(lst : List, i : Int) : List =
20.         if (lst.isNil())
21.             new Cons(i, lst)
22.         else if (i <= lst.head())
23.             new Cons(i, lst)
24.         else
25.             new Cons(lst.head(), insert(lst.tail(), i));
26.
27.     def sort(lst : List) : List =
28.         if (lst.isNil())
29.             lst
30.         else
31.             insert(sort(lst.tail()), lst.head());
```

```
32.
33.     def out_list(lst : List) : Unit =
34.         if (lst.isNil())
35.             out_nl()
36.         else {
37.             out_int(lst.head());
38.             out_string(" ");
39.             out_list(lst.tail())
40.         };
41.
42.     {
43.         var lst : List =
44.             new Cons(30, new Cons(20, new Cons(50,
45.             new Cons(40, new Cons(10, new Nil())))));
46.         out_list(lst);
47.         out_list(sort(lst))
48.     };
49. }
```

O Código 2.3 apresenta um programa que implementa o algoritmo *quicksort*. O inicializador da classe `Main` executa a mesma operação que o exemplo anterior, utilizando porém um *array* ao invés de uma lista.

Código 2.3 – Programa exemplo `quicksort.cool`

```
1.  class Main() extends IO() {
2.      def quicksort(array : ArrayAny,
3.                    lo : Int, hi : Int) : Unit = {
4.          if (lo < hi) {
5.              var p : Int = partition(array, lo, hi);
6.              quicksort(array, lo, p - 1);
7.              quicksort(array, p + 1, hi)
8.          } else ()
9.      };
10.
11.     def partition(array : ArrayAny,
```

```
12.                   lo : Int, hi : Int) : Int = {
13.         var pivot : Int =
14.             array.get(lo) match { case i : Int => i };
15.         var p : Int = lo;
16.         var i : Int = lo + 1;
17.         while (i <= hi) {
18.             if (array.get(i) match { case i : Int => i }
19.                     <= pivot)
20.                 array_swap(array, i, { p = p + 1; p })
21.             else
22.                 ();
23.             i = i + 1
24.         };
25.         array_swap(array, p, lo);
26.         p
27.     };
28.
29.     def array_swap(array : ArrayAny,
30.                    p : Int, q : Int) : Unit = {
31.         var tmp : Any = array.get(p);
32.         array.set(p, array.get(q));
33.         array.set(q, tmp)
34.     };
35.
36.     def out_array(array : ArrayAny) : Unit = {
37.         var i : Int = 0;
38.         while (i < array.length()) {
39.             array.get(i) match {
40.                 case i : Int =>
41.                     out_int(i);
42.                     out_string(" ")
43.             };
44.             i = i + 1
45.         };
```

```
46.             out_nl()
47.         };
48.
49.         {
50.             var array : ArrayAny = new ArrayAny(5);
51.             array.set(0, 30);
52.             array.set(1, 20);
53.             array.set(2, 50);
54.             array.set(3, 40);
55.             array.set(4, 10);
56.             out_array(array);
57.             quicksort(array, 0, array.length() - 1);
58.             out_array(array)
59.         };
60.     }
```

2.2.6 Semântica formal da linguagem Cool

O manual de referência da linguagem Cool apresenta a definição formal do sistema de tipos e da semântica operacional da linguagem.

Figura 2.6: Regra de verificação de tipos ASSIGN da linguagem Cool

$$\frac{O(v) = T \qquad O, M \vdash e_1 : T' \qquad T' \leq T}{O, M \vdash v = e_1 : \texttt{Unit}} \quad \text{(ASSIGN)}$$

Fonte: Boyland (2016, p. 18)

A Figura 2.6 reproduz uma das regras de verificação de tipos descritas no manual. Essa regra faz parte da definição da relação binária : (dois pontos) entre expressões e tipos. O e M formam o ambiente de tipos e contém os tipos de variáveis, atributos, métodos e parâmetros previamente declarados. $O(v) = T$ significa que o identificador v foi previamente declarado como tendo o tipo T. $T \leq T'$ significa que o tipo T está em conformidade com o tipo T', como descrito na Seção 2.2.2.

O símbolo $\vdash$ indica que é possível derivar a expressão à direita a partir do contexto especificado à esquerda em um sistema formal. A expressão $O, M \vdash v = e_1 : \texttt{Unit}$ sig-

nifica, portanto, que podemos concluir que v = e_1 possui o tipo `Unit` dado o ambiente de tipos formado por O e M. Como essa expressão é a conclusão de uma regra de inferência, a afirmação anterior é verdadeira apenas se as hipóteses podem ser provadas e as restrições satisfeitas.

Figura 2.7: Regra de avaliação *big-step* ASSIGN da linguagem Cool

$$\frac{C, so, S_1, E \vdash e_1 : v_1, S_2 \qquad E(v) = l_1 \qquad S_3 = S_2[v_1/l_1]}{C, so, S_1, E \vdash v = e_1 : \texttt{Unit}\,(), S_3} \quad \text{(ASSIGN)}$$

Fonte: Boyland (2016, p. 25)

A Figura 2.7 reproduz uma das regras da semântica operacional *big-step* descritas no manual. Ela faz parte da definição do sistema de transição : (dois pontos, o mesmo nome dado à relação que define o sistema de tipos). so, S_n e E formam o *contexto* de avaliação na classe C, onde C é o tipo estático do qual o método sendo avaliado faz parte, so é o objeto que é referenciado pelo identificador `this`, S_n é uma *memória* e E é um *ambiente*.

Ambientes e memórias em Cool são mapeamentos. Ambientes mapeiam identificadores para *localizações* e memórias mapeiam localizações para valores. A expressão $E(v) = l_1$ é uma consulta ao ambiente para obter a localização l_1 relacionada ao identificador v. A expressão $S_3 = S_2[v_1/l_1]$ é uma modificação da memória S_2, substituindo o valor previamente associado à localização l_1 pelo valor v_1, sendo que essa memória modificada é chamada de S_3.

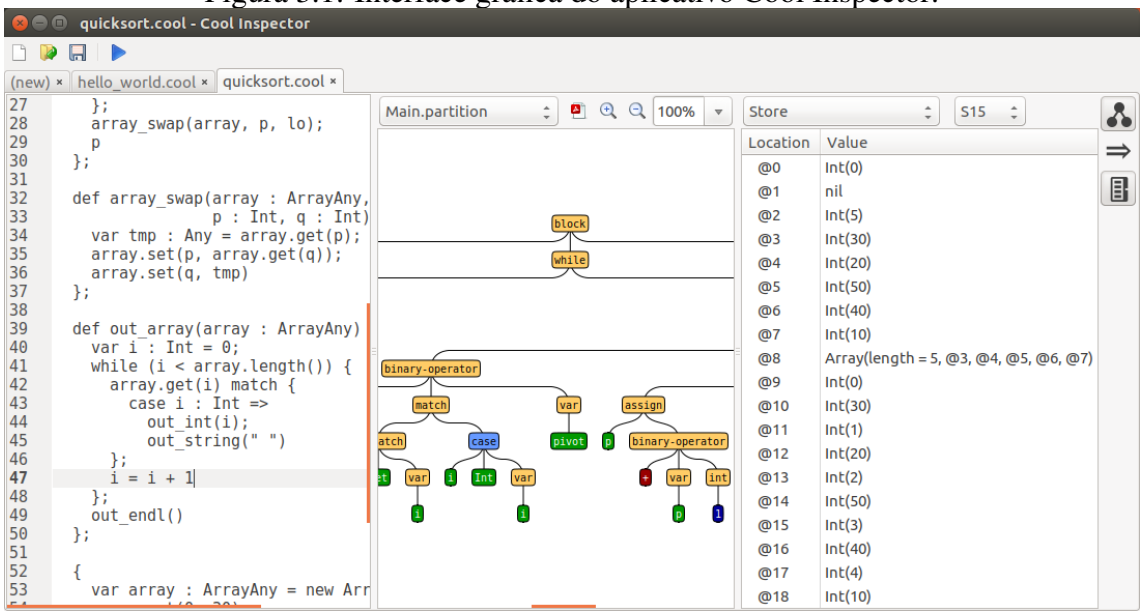
As configurações finais da linguagem Cool são pares ordenados (v, S) onde S é uma memória e v é um valor. Valores são descritos, em geral, pela sintaxe $X\,(a_1 = l_1, \ldots, a_n, l_n)$ onde X é uma classe, a_n são os identificadores referentes aos atributos da classe X e l_n são as localizações a serem associadas aos identificadores a_1 a a_n durante a avaliação de um método da classe X.

Assim sendo, a regra de avaliação ASSIGN estabelece que os resultados de uma atribuição são o valor `Unit()` e uma memória com o valor da variável atribuída atualizado para o resultado da expressão à direita. Se essa atribuição fizer parte de um bloco (ver Seção 2.2.4.5) a regra BLOCK-EXPR, descrita no manual, utiliza a memória atualizada para avaliar a expressão que se segue.

As demais regras de verificação de tipos e avaliação *big-step*, como a regra BLOCK-EXPR mencionada acima, podem ser consultadas no manual de referência da linguagem Cool (BOYLAND, 2016).

3 APLICATIVO GRÁFICO PARA ANÁLISE DA SEMÂNTICA OPERACIONAL DE COOL

Figura 3.1: Interface gráfica do aplicativo Cool Inspector.



Fonte: o autor

A proposta deste trabalho é a implementação de um aplicativo gráfico que permite a visualização dos passos intermediários da avaliação de um programa da linguagem Cool. A Figura 3.1 apresenta a interface principal do aplicativo, o qual denominamos Cool Inspector e cujo código pode ser encontrado através do link <https://github.com/sgm/cool-inspector>.

O Cool Inspector foi desenvolvido utilizando a linguagem de programação OCaml, fazendo uso das bibliotecas Camlp4 para a geração de parsers, GTK+ para a interface gráfica e Cairo para visualização das árvores de derivação e de sintaxe abstrata. Mais detalhes sobre estas ferramentas são apresentados na Seção 3.3.

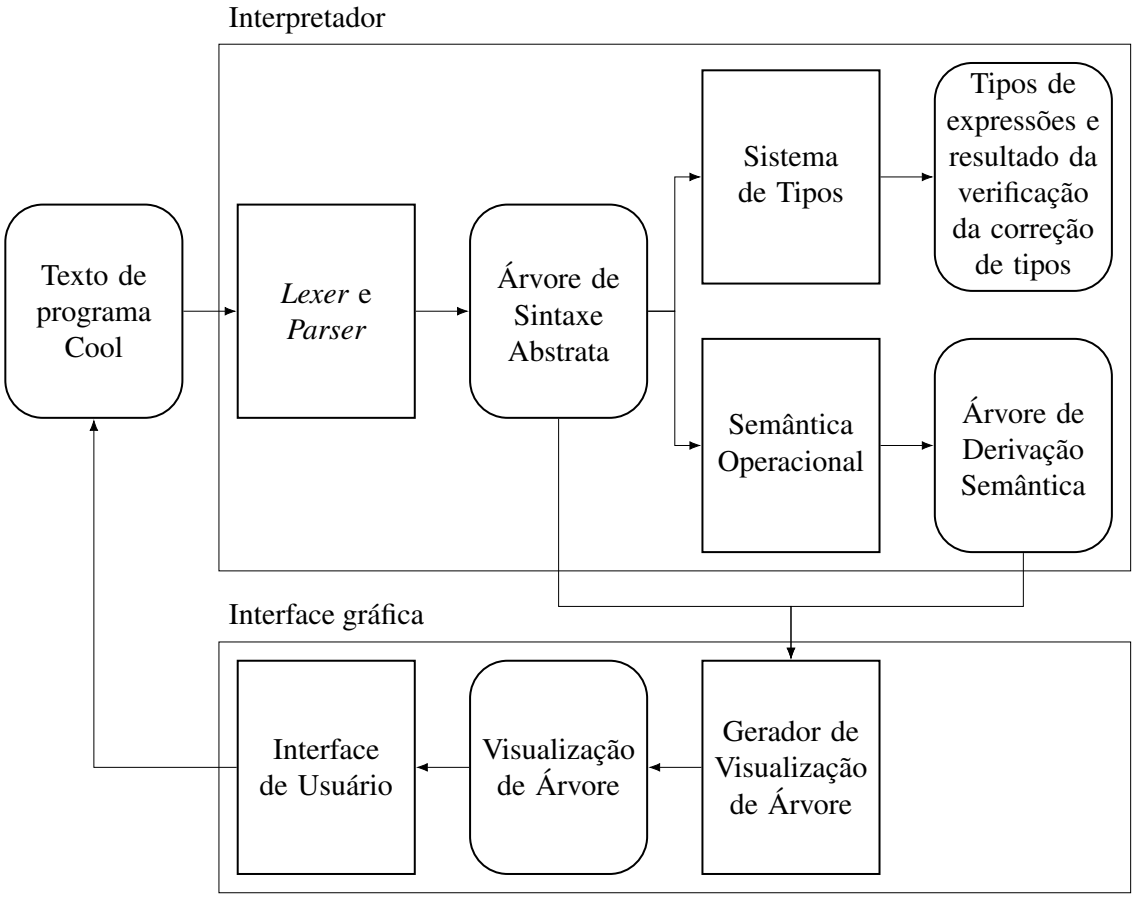
Os blocos principais do aplicativo podem ser vistos na Figura 3.2. Retângulos normais representam processos e retângulos com cantos arredondados representam dados.

3.1 Arquitetura

Cool Inspector possui dois componentes principais: o interpretador e a interface gráfica. O interpretador implementa os seguintes módulos:

- Analizadores léxico e sintático — responsáveis pela transformação de um código

Figura 3.2: Diagrama de blocos do aplicativo Cool Inspector.



Fonte: o autor

em Cool em árvores de sintaxe abstrata, conforme visto na Seção 2.1.1;

- Avaliador das regras de derivação de tipos — avalia tipos de expressões em Cool e verifica se um programa Cool é bem tipado;

- Avaliador das regras da semântica operacional — avalia o resultado da execução de um programa Cool e registra os passos intermediários, construindo uma árvore de derivação semântica;

- *Pretty-printer* — transforma árvores de sintaxe abstratas de programas Cool em código Cool em sintaxe concreta, formatado de forma a facilitar a leitura. Esta funcionalidade não está atualmente exposta ao usuário do Cool Inspector, mas é utilizada pela interface gráfica para gerar as visualizações dos passos intermediários registrados pelo avaliador da semântica operacional.

O código OCaml do interpretador e da interface gráfica do Cool Inspector possuem, respectivamente, cerca de 3229 e 2150 linhas de código.

3.1.1 Analizador léxico (*lexer*)

O *lexer* do Cool Inspector é implementado com o ocamllex, um gerador de *lexers* que é instalado por padrão em distribuições do OCaml. Ele gera os seguintes tipos de *tokens*:

- `KEYWORD` — uma palavra chave da linguagem Cool que não pode ser usada como identificador (nome de variável, método, tipo, etc.) pois possui um significado especial na linguagem;

- `ILLEGAL` — uma palavra chave ilegal que, assim como um token do tipo `KEYWORD`, não pode ser usada como um identificador. Estas palavras chaves, porém, não possuem significado especial em Cool, e são considerados ilegais apenas para que possíveis versões futuras da linguagem possam atribuir algum significado a elas sem que programas escritos para versões anteriores se tornem inválidos;

- `SYMBOL` — caracteres não alfanuméricos, como operadores matemáticos e parênteses;

- `TYPE` — um identificador de tipo, formado por uma sequência de caracteres alfanuméricos iniciada em letra maiúscula;

- `ID` — outro identificador, usado, por exemplo, como nome de variável ou método, formado por uma sequência de caracteres alfanuméricos iniciada em letra minúscula;

- `INTEGER` — um literal inteiro não negativo, formado por uma sequência de caracteres numéricos como descrito na Seção 2.2.4.1;

- `STRING` — um literal *string*, formado por uma sequência de caracteres entre aspas duplas como descrito na Seção 2.2.4.1;

- `EOI` — *end of input* – fim do stream de entrada.

As palavras chaves que resultam em um *token* de tipo `KEYWORD` são: `case`, `class`, `def`, `else`, `extends`, `false`, `if`, `match`, `native`, `new`, `null`, `override`, `super`, `this`, `true`, `var` e `while`.

As palavras chaves ilegais que resultam em um *token* de tipo `ILLEGAL` são: `abstract`, `catch`, `do`, `final`, `finally`, `for`, `forSome`, `implicit`, `import`, `lazy`, `object`, `package`, `private`, `protected`, `requires`, `return`, `sealed`, `throw`, `trait`, `try`, `type`, `val`, `with` e `yield`.

As sequências de caracteres que resultam em um *token* de tipo `SYMBOL` são: `==`, `<=`, `=>`, `(`, `)`, `{`, `}`, `,`, `.`, `:`, `;`, `=`, `<`, `!`, `+`, `-`, `*` e `/`.

Além dos tipos já descritos, os seguintes tipos *tokens* são gerados porém descartados pelo parser: `COMMENT` (comentários, como descrito na Seção 2.2), `BLANKS` (sequência de espaços e tabulações horizontais) e `NEWLINE` (quebra de linha).

3.1.2 Analizador sintático (*parser*)

A Seção 2.1.1 descreve o processo de análise sintática e estabelece que o resultado de tal processo é uma árvore de sintaxe abstrata. O *parser* do Cool Inspector, no entanto, não gera apenas uma árvore, mas uma para cada método de cada classe do programa. Um conjunto de estruturas de dados do tipo árvore é chamado de *floresta*. Todavia, neste trabalho é tomada a liberdade de, por vezes, referenciar esta floresta como a árvore de sintaxe abstrata resultante da execução do *parser* do Cool Inspector.

O sistema Camlp4, a ser apresentado na Seção 3.3.2, foi utilizado para a implementação do *parser* do Cool Inspector.

3.1.3 Avaliador das regras de derivação de tipos

O avaliador das regras de derivação de tipos, ou *verificador de tipos*, implementa as regras de derivação de tipos descritas no manual de referência da linguagem Cool.

Este componente é implementado no módulo `Type` do código OCaml do Cool Inspector. As principais funções implementadas neste módulo são:

- `Type.of_expr` — infere o tipo de uma expressão;
- `Type.check_feature` — verifica erros de tipos em uma *feature*;
- `Type.check_class` — verifica erros de tipos em uma classe;
- `Type.check_program` — verifica erros de tipos em um programa.

Dentre estas funções, apenas a `of_expr` retorna um resultado significativo. Para indicar um erro durante a derivação de tipos, estas funções geram exceções que descrevem exatamente qual o erro encontrado e a localização da construção inválida no código do programa.

3.1.4 Avaliador das regras da semântica operacional

O avaliador das regras da semântica operacional implementa as regras de derivação *big-step* descritas no manual de referência da linguagem Cool.

Este componente é implementado no módulo `Eval` do código OCaml do Cool Inspector. As principais funções implementadas neste módulo são:

- `Eval.expr` — retorna um valor e uma memória que são o resultado da avaliação de uma expressão sobre uma memória inicial;
- `Eval.program` — chama a função `Eval.expr` sobre a expressão `new Main()`, conforme especificado na Seção 2.2.1.

Estas funções podem ser customizadas através do módulo funtorial `Eval.Make` que aceita como argumento um outro módulo que implementa várias funções que agem como pontos de customização nas funções descritas acima.

Utilizando-se deste módulo funtorial, as funções `expr` e `program` são customizadas para registrar os vários passos intermediários da interpretação de um programa, gerando uma árvore de derivação que permite a análise da execução do programa após o seu término. Estas funções customizadas podem ser acessadas pelo sub-módulo `Rec` do módulo `Eval`.

Da mesma forma que ocorre na avaliação das regras de derivação de tipos, erros são indicados através da geração de exceções que contém informações sobre o erro. As exceções que podem ser geradas são:

- `Null_dispatch` — chamada de método sobre um objeto `null`, conforme descrito na Seção 2.2.4.4;
- `No_match` — execução de uma expressão `match` que não contém um ramo que do tipo esperado, conforme descrito na Seção 2.2.4.9;
- `Division_by_zero` — divisão por zero ou `MIN_INT`$/-1$, onde `MIN_INT` é o menor número inteiro que pode ser representado pelo tipo `Int` na implementação da linguagem Cool;
- `Out_of_range` — índice ou tamanho inválido em operações sobre *strings* ou *arrays*;
- `Heap_overflow` — memória insuficiente para o prosseguimento da execução do programa;
- `Abort` — código chamou o método `abort` da classe `Any`;
- `No_rule_applies` — nenhuma das regras da semântica operacional se aplica.

A exceção `No_rule_applies` é gerada apenas se o programa possui erros de tipo (salvo eventuais falhas de implementação). Assim sendo, mais informações sobre a causa do erro podem ser adquiridas através da execução do verificador de tipos sobre o programa.

3.1.5 Interface Gráfica

Os dois principais módulos da interface gráfica são:

- Gerenciador de janelas e *widgets* — implementado usando o GTK+, apresentado na Seção 3.3.3, é responsável por apresentar a interface gráfica do aplicativo e responder aos comandos do usuário;

- Gerador de visualizações — implementado usando a biblioteca Cairo, apresentada na Seção 3.3.4, é responsável por gerar as visualizações gráficas das árvores de sintaxe abstrata e de derivação semântica.

3.2 Funcionalidades

Nesta seção serão descritas as funcionalidades presentes no aplicativo Cool Inspector do ponto de vista do usuário.

3.2.1 Edição de programas Cool em sintaxe concreta

O Cool Inspector oferece uma janela de edição na qual o usuário pode inserir o programa Cool. Esse editor possui alguns dos comandos comuns de um editor de código, como *novo*, *abrir* e *salvar*.

O aplicativo também permite a execução do código, que se dá a partir da execução dos seguintes passos no momento em que usuário clica no botão *executar*:

1. Execução do *lexer* e do *parser*, gerando as árvores de sintaxe abstrata;

2. Avaliação das regras de derivação de tipos;

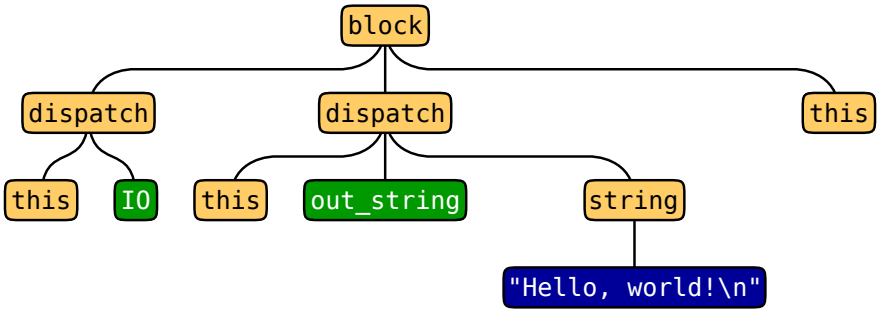
3. Avaliação das regras da semântica operacional.

Em caso de erro na execução de qualquer um dos passos, os passos subsequentes não são executados e uma mensagem de erro é exibida ao usuário.

3.2.2 Visualização das árvores de sintaxe abstrata

Após a geração das árvores de sintaxe abstrata o usuário pode visualizá-las através da barra de inspeção lateral, clicando no botão e selecionando o método desejado na caixa de combinação situada no topo da área de visualização.

Figura 3.3: Árvore de sintaxe abstrata do construtor da classe `Main` apresentado no Código 2.1, como exibida pelo Cool Inspector.



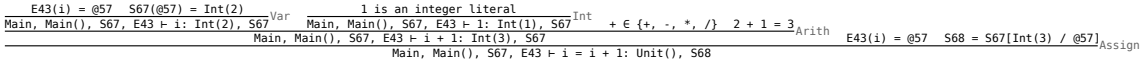
Fonte: o autor

A Figura 3.3 mostra um exemplo de como a árvore de sintaxe abstrata é exibida pelo Cool Inspector (outro exemplo pode ser parcialmente visto na Figura 3.1). As cores dos nós das árvores indicam o tipo de construção da linguagem Cool que o nó representa. Os 5 tipos que são evidenciados pela visualização, e suas respetivas cores, são:

- Laranja: expressão, como descrito na Seção 2.2.4;
- Azul claro: ramo de uma expressão `match`;
- Verde: tipo ou identificador definido pelo usuário;
- Azul: literal (número, string ou booleano);
- Vermelho: operador.

3.2.3 Visualização da árvore de derivação da semântica operacional

Figura 3.4: Seção da árvore de derivação semântica do programa apresentado no Código 2.3, como exibida pelo Cool Inspector.



Fonte: o autor

Após a interpretação do programa Cool, a árvore de derivação semântica do programa pode ser visualizada pelo usuário ao clicar no botão ⇒. A árvore apresenta todas as regras que foram utilizadas na avaliação do programa, fazendo referência às memórias e ambientes intermediários.

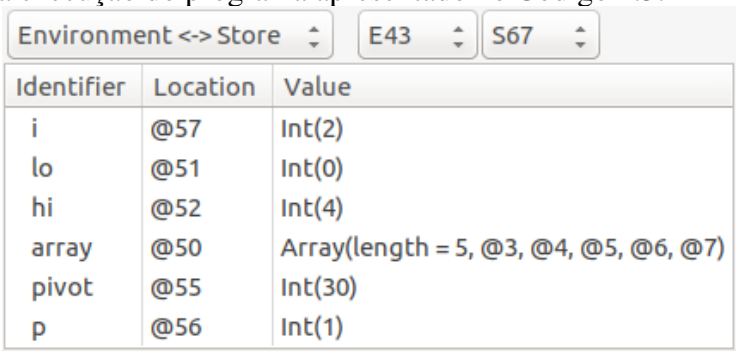
As memórias e os ambientes são referenciados através de nomes dados a cada

memória e a cada ambiente de acordo com os seus conteúdos. Assim, se duas ou mais memórias ou ambientes possuem o mesmo conteúdo, eles são considerados iguais e lhes é atribuído o mesmo nome.

Às memórias são dados nomes da forma $\texttt{S}n$ e aos ambientes, $\texttt{E}n$, onde n é um número. Localizações em memória são indicados da forma $\texttt{@}n$. Outras informações presentes na árvore de derivação são apresentadas seguindo o estilo usado no manual de referência da linguagem Cool.

3.2.4 Visualização de memórias e ambientes

Figura 3.5: Visualizador de memórias e ambientes do Cool Inspector, exibindo um estado intermediário da execução do programa apresentado no Código 2.3.



Fonte: o autor

As memórias e ambientes intermediários podem ser visualizados através do botão . O Cool Inspector permite a visualização de cada memória e ambiente de forma independente ou de um mapeamento entre um ambiente e uma memória escolhidas pelo usuário, como demonstrado na Figura 3.5.

3.3 Recursos utilizados

Esta seção apresenta os recursos de software que foram utilizados para o desenvolvimento do aplicativo Cool Inspector.

3.3.1 Linguagem OCaml

OCaml é uma linguagem de programação de propósito geral com ênfase em expressividade e segurança, em desenvolvimento por mais de 20 pesquisadores no INRIA

(*Institut National de Recherche en Informatique et en Automatique*, que é o instituto nacional francês de ciência da computação e matemática aplicada).[1]

A linguagem OCaml foi escolhida pelo autor de *Types and Programming Languages* (PIERCE, 2002) para a implementação dos algoritmos descritos no livro devido à presença de gerenciamento automático de memória (*garbage collection*) e facilidade de implementação de funções recursivas através de casamento de padrões (*pattern matching*) sobre tipos de dados estruturados. Inspirado nesta escolha, o aplicativo Cool Inspector também foi implementado utilizando a linguagem OCaml.

3.3.2 Camlp4

Camlp4 é um sistema de software para escrita de *parsers* extensíveis. Ele provê um conjunto de bibliotecas OCaml que são usadas para definir gramáticas e extensões de sintaxe carregáveis para tais gramáticas. Camlp4 significa *Caml Preprocessor and Pretty-Printer* e uma das suas mais importantes aplicações é a especificação de extensões de domínio específico da sintaxe de OCaml.[2] O Cool Inspector, contudo, faz uso do Camlp4 para gerar o parser da linguagem Cool.

Foi dada a preferência para o Camlp4 sobre outras ferramentas de geração de *parsers*, como ocamlyacc, devido a simplicidade com a qual as regras da gramática podem ser descritas, a produção de mensagens de erro mais significativas e a sua melhor integração com a linguagem OCaml.

3.3.3 GTK+

GTK+, ou *GIMP Toolkit*, é um conjunto de ferramentas multiplataforma para a criação de interfaces gráficas. O GTK+ é escrito na linguagem C mas é desenvolvido para suportar várias lingugagens de programação.[3]

A escolha de usar o GTK+ foi feita devido a existência da biblioteca lablgtk, que oferece uma interface ao GTK+ em OCaml, facilitando a integração entre o interpretador Cool e a interface gráfica do Cool Inspector.

---

[1] https://ocaml.org/learn/description.html
[2] https://github.com/ocaml/camlp4/blob/f2d62524324d903d5716d3be5c787d9dc5c43013/README.md
[3] https://www.gtk.org/

3.3.4 Cairo

Cairo é uma biblioteca gráfica 2D que oferece uma API para criação e exibição de gráficos vetoriais.[4]

Como dito na Seção 3.1.5, o Cool Inspector utiliza Cairo para desenhar as visualizações das árvores de sintaxe abstrata e de derivação da semântica operacional, além de permitir a exportação dessas visualizações para arquivos PDF.

3.4 Divergências entre a implementação e a especificação da linguagem Cool

O Cool Inspector procura implementar a linguagem Cool tal qual especificada na documentação disponibilizada por Boyland, excetuando-se os seguintes pontos:

- *Garbage collection* — Cool depende de um mecanismo de gerenciamento automático de memória para desalocar objetos que não serão mais utilizados em um programa. O Cool Inspector, porém, não implementa este mecanismo, visto que o aplicativo deve permitir a visualização de todos os estados intermediários da memória, incluindo o conteúdo dos objetos nela contidos, após o término da execução do programa;

- Largura dos números inteiros — o manual de referência especifica que valores do tipo `Int` devem ser inteiros de 32 bits. O Cool Inspector, entretanto, utiliza o tipo inteiro padrão fornecido pela linguagem OCaml, que possui 31 ou 63 bits de acordo com a arquitetura da máquina. Esta decisão de implementação foi tomada devido ao favorecimento da simplicidade do código OCaml sobre a conformidade com a especificação;

- Implementação da classe `Symbol` — a classe `Symbol` da linguagem Cool permite a análise de árvores de sintaxe abstrata de programas. Esta funcionalidade não foi implementada e não há como instanciar objetos desta classe no Cool Inspector;

- Arquivo `basic.cool` — o conteúdo deste arquivo não é descrito nos documentos disponíveis e, portanto, as funcionalidades oferecidas pelas classes básicas difere entre a versão disponibilizada pelo Cool Inspector e a versão desenvolvida pelo autor da linguagem Cool.

[4] https://www.cairographics.org/

4 CONCLUSÃO

Neste trabalho foi apresentado o aplicativo Cool Inspector, que permite a visualização da aplicação das regras da semântica operacional da linguagem Cool sobre um programa fornecido pelo usuário, evidenciando de maneira concreta o uso desta regras na formação de uma árvore de prova que descreve completamente o comportamento do programa.

O aplicativo oferece um interpretador da linguagem Cool e uma interface gráfica para a edição de programas e visualização dos resultados de vários passos da execução destes programas, como árvores de sintaxe abstrata, árvores de avaliação e memórias intermediárias.

Para a implementação do aplicativo foi utilizada a linguagem OCaml devido à disponibilidade de gerenciamento automático de memória e casamento de padrões sobre tipos de dados estruturados. Também foram utilizados o sistema Camlp4 para a geração do *parser*, a biblioteca GTK+ para o desenvolvimento da interface gráfica e a biblioteca Cairo para a geração das visualizações.

4.1 Possibilidades de uso do aplicativo

O aplicativo pode servir como recurso didático, junto ao manual de referência da linguagem Cool, a ser utilizado por professores de disciplinas de semântica formal. Espera-se que as visualizações oferecidas e a experimentação possibilitada assistam no entendimento da notação comumente empregada na área de semântica formal e deixem mais claro como os conceitos abstratos estudados podem ser empregados na definição de semânticas formais de linguagens de programação.

O código fonte do interpretador também pode ser utilizado como recurso didático. Cool, sendo uma linguagem designada a ser implementada no curso de um semestre, é relativamente simples e limitada. Assim sendo, o código do interpretador não deve ser de difícil compreensão e extensões à linguagem de diversos níveis de complexidade podem ser sugeridas como exercício. Exemplos de extensões são: remoção da obrigatoriedade do ramo *else* da construção *if*, adição de novos operadores *booleanos*, de novas estruturas de controle de fluxo, como laços *for*, e de números em ponto flutuante.

4.2 Trabalhos futuros

O aplicativo Cool Inspector ainda pode ser melhorado em vários aspectos, sendo o mais aparente deles a construção e visualização de árvores de derivação de tipos. Esta funcionalidade pode ser implementada de maneira bastante similar à visualização de árvores de derivação da semântica operacional.

Possíveis melhorias no editor de código incluem proporcionar realce de sintaxe para a linguagem Cool, auxiliar na formatação de código, incluir ferramentas de pesquisa e substituição de trechos do texto e permitir o usuário configurar a aparência do texto e do aplicativo.

Seria também possível oferecer mais dinamismo e interatividade nos painéis de inspeção, permitindo, por exemplo, que o usuário encontre nós específicos nas visualizações de árvores a partir de posições no texto do programa e vice-versa.

Uma outra funcionalidade que pode ser considerada é a execução passo-a-passo de programas, permitindo que o usuário visualize como a construção da árvore de derivação da semântica operacional se prossegue a cada expressão que é avaliada.

Por fim, pode-se avaliar a eficácia do aplicativo no ensino de semântica formal através do seu uso no curso de um semestre e a realização de uma comparação com alunos que não utilizaram este recurso, verificando se houve uma maior facilidade por parte dos alunos em compreender de maneira intuitiva como as regras de uma semântica formal descrevem a avaliação de um programa.

REFERÊNCIAS

BOYLAND, J. T. **CoolAid: The Cool 2016 Reference Manual**. 2016. Disponível em: <http://www.cs.uwm.edu/~cs654/handouts/cool-manual.pdf>. Acesso em: 13 nov. 2016.

FERNÁNDEZ, M. **Programming Languages and Operational Semantics: A Concise Overview**. [S.l.]: Springer Publishing Company, Incorporated, 2014. ISBN 978-1-447-16367-1.

KE, W. et al. A graph-based operational semantics of OO programs. In: BREITMAN, K.; CAVALCANTI, A. (Ed.). **Formal Methods and Software Engineering: 11th International Conference on Formal Engineering Methods ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 347–366. ISBN 978-3-642-10373-5. Disponível em: <http://dx.doi.org/10.1007/978-3-642-10373-5_18>.

NIELSON, H. R.; NIELSON, F. **Semantics with Applications: A Formal Introduction**. New York, NY, USA: John Wiley & Sons, Inc., 1992. ISBN 978-0-471-92980-2.

PIERCE, B. C. **Types and Programming Languages**. 1st. ed. Cambridge, MA, USA: The MIT Press, 2002. ISBN 978-0-262-16209-8.

SLONNEGER, K.; KURTZ, B. **Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach**. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 978-0-201-65697-8.

WINSKEL, G. **The Formal Semantics of Programming Languages: An Introduction**. Cambridge, MA, USA: MIT Press, 1993. ISBN 978-0-262-23169-5.