UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

BERNARDO SARAIVA OLIVEIRA DUARTE

ANÁLISE ESTÁTICA PARA LIMITES DE ARRANJOS

Ouro Preto, MG
2025

BERNARDO SARAIVA OLIVEIRA DUARTE

ANÁLISE ESTÁTICA PARA LIMITES DE ARRANJOS

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Rodrigo Geraldo Ribeiro

Ouro Preto, MG
2025

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Bernardo Saraiva Oliveira Duarte

Análise estática para limites de arranjos

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 29 de Agosto de 2025.

Membros da banca

Rodrigo Geraldo Ribeiro (Orientador) - Doutor - Universidade Federal de Ouro Preto
Leonardo Vieira dos Santos Reis (Examinador) - Doutor - Universidade Federal de Juiz de Fora
Elton Máximo Cardoso (Examinador) - Doutor - Universidade Federal de Ouro Preto

Rodrigo Geraldo Ribeiro, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 29/08/2025.

Documento assinado eletronicamente por **Rodrigo Geraldo Ribeiro**, **PROFESSOR 3 GRAU**, em 03/09/2025, às 16:00, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0965551** e o código CRC **04C859DD**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Agradecimentos

Gostaria de agradecer ao professor Rodrigo Geraldo Ribeiro por me ajudar orientando durante a escrita deste trabalho.

Resumo

Arranjos são uma estrutura recorrente em programas de computador, porém seu acesso através de índices indevidos é motivo principal para inúmeras invulnerabilidades e erros. Por isso, este trabalho propõe uma solução, auxiliadora ao desenvolvedor, para tratar este problema através de um algoritmo que roda em tempo de compilação de uma linguagem simples e concisa chamada *FonC*. Este trabalho teve como objetivo principal a especificação deste algoritmo além de ter se descrito esta linguagem e utilizado-se de um interpretador da mesma para validar o algoritmo verificador de quebra de limites de arranjo. Mostrou-se que nossa abordagem detecta acessos inválidos, sem falsos positivos e sem falsos negativos, sobre parte do código executável em tempo de compilação. Além disso, mostrou-se que a solução apresenta outras características superiores aos trabalhos relacionados.

Palavras-chave: Acesso de arranjo. Análise estática. Compilação.

Abstract

Arrays are a recurring structure in computer programs, but accessing them through inappropriate indexes is the main reason for numerous invulnerabilities and errors. Therefore, this work proposes a developer-friendly solution to address this problem through an algorithm that runs at compile time in a simple and concise language called *FonC*. The main objective of this work was to specify this algorithm, describe the language, and use an interpreter of the language to validate the array bounds analyser algorithm. Our approach was shown to detect invalid accesses, without false positives or false negatives, over part of the executable code at compile time. Furthermore, it was shown that the solution presents other characteristics superior to related works.

Keywords: Array access. Static analysis. Compilation.

Lista de Abreviaturas e Siglas

EBNF	Forma de Backus-Naur Extendida

Lista de Símbolos

$\in$	Pertence
$\mathbb{N}$	Conjunto dos Números Naturais

Sumário

1 Introdução

Estruturas de dados são formas nas quais organizamos dados durante o desenvolvimento de programas. Elas são parte essencial para algoritmos não triviais em computação, em especial os arranjos, que são recorrentemente usados como base para várias outras estruturas mais complexas. É conveniente então um suporte nativo de compiladores para evitar problemas de quebra de limite de arranjo pois essa violação frequentemente leva a *bugs*, *crashes* e é o principal motivo para vulnerabilidades de segurança (Özkan, 2023).

Computadores, a princípio, processam código de máquina, porém escrever programas de baixo nível manualmente é uma tarefa árdua e propensa a erros. Então, utiliza-se de *Compiladores* que traduzem código escrito em linguagens de alto nível para código de máquina. Durante o processo de tradução, para auxiliar o desenvolvedor, compiladores realizam diversas análises no código. Usualmente, compiladores dividem a etapa de análise em três fases: análise léxica, análise sintática e análise semântica.

Na primeira etapa, chamada de análise léxica, converte-se um texto em uma sequência de unidades chamadas *tokens*. Um token é uma sequência de um ou mais caracteres com algum significado atribuído. Outra tarefa da análise léxica é a de remover partes do texto que não são importantes como espaços em branco e comentários. A segunda etapa é denominada de análise sintática e destina-se a verificar se a sequência de *tokens* produzida pela análise léxica segue a estrutura descrita pela gramática da linguagem. Finalmente, a terceira e última etapa, denominada de análise semântica, lida com verificações que não podem ser expressas utilizando uma gramática livre de contexto como, por exemplo, se chamadas de funções possuem a quantidade e tipos de argumentos corretos, se todo uso possui uma declaração correspondente. Por exemplo, considere a seguinte função escrita em C que soma dois números:

```
1    int soma(int x, int y){
2        return x + y;
3    }
```

Listing 1.1 – Definição de uma função *soma* em C

Identificadores como *int*, *soma* e *return* são *tokens*, assim como os símbolos '+', ';' e '{'. Veja que a linguagem segue um padrão como, por exemplo, parênteses abertos devem ser fechados e denotam uma área específica do código, e o mesmo acontece com as chaves, identificadores de tipos precedem identificadores de nomes de variáveis e ponto e virgula aparecem ao final de comandos. Observe também que seguindo a semântica da linguagem, este programa seria inválido:

```
1    int main(){
2        soma(3, "Um texto aqui");
```

```
3     }
```

Listing 1.2 – Uso inválido da função *soma* definida anteriormente

O trecho de código anterior possui um erro: a chamada da função *soma* é feita com um argumento de tipo inválido. De acordo com a especificação da linguagem C (ISO, 2011), funções devem ser sempre chamadas com argumentos de tipos compatíveis. Ao submetermos o programa anterior ao compilador, este retornará uma mensagem de erro indicando a violação semântica encontrada.

```
1     #include <malloc.h>
2     int main(){
3         int* array = alloca(3);
4         array[4] = 34;
5     }
```

Listing 1.3 – Exemplo de acesso de arranjo inválido em C

No entanto, há um problema comum em linguagens onde não existe uma regra semântica sobre os arranjos. Pode-se, como mostrado no código acima, selecionar um índice inválido de um arranjo e o compilador não avisar sobre a violação, podendo originar *bugs*, *crashes* e vulnerabilidades sobre códigos.

Desta maneira, neste trabalho, será proposto uma solução baseada em análise estática de propriedades sobre o código em tempo de compilação. Será feito uma expansão da capacidade de análise semântica de um compilador para se detectar quando um erro de quebra de limite de arranjo pode ocorrer ou irá ocorrer. Para isto, será apresentado uma linguagem de programação simples e concisa e um compilador que reconhece o código que está violando limites de arranjos, ou que pode violar, e, assim, avisar ao programador que mudanças são necessárias para o programa funcionar corretamente. Será apresentado também o interpretador desta linguagem como protótipo da técnica proposta neste trabalho.

1.1 Justificativa

Visto que é recorrente o uso de arranjos em códigos escritos nas mais diversas linguagens de programação é importante buscar desenvolver um conjunto de ferramentas que garanta seu uso correto.

1.2 Objetivos

Esse trabalho tem por objetivo principal especificar e implementar um compilador para uma linguagem com suporte a verificação de limites de acesso a arranjos. Os objetivos específicos são:

- Desenvolver uma linguagem de programação simples e concisa;
- Projetar um compilador com uma análise semântica que entenda propriedades do código que devem ser satisfeitas;
- Utilizar o compilador desenvolvido para implementar algoritmos clássicos envolvendo arranjos. Essa etapa visa validar a estratégia proposta neste trabalho.

1.3 Organização do Trabalho

O presente trabalho é dividido em 5 capítulos:

Capítulo 1: Introdução.

Capítulo 2: Revisão Bibliográfica

Capítulo 3: Desenvolvimento

Capítulo 4: Resultados

Capítulo 5: Conclusão

2 Revisão Bibliográfica

Neste capítulo apresenta-se a teoria necessária para a compreensão deste trabalho. A Seção 2.1 discorre sobre arranjos, acesso a arranjos, quebra de limite de arranjos e suas consequências. Adicionalmente, discutimos conceitos envolvidos em compilação e projeto de linguagens de programação. A Seção 2.2 apresenta os trabalhos relacionados presentes na literatura.

2.1 Fundamentação Teórica

2.1.1 Arranjos

Um arranjo é uma estrutura de dados que armazena elementos de forma que cada elemento possa ser identificado por um índice. Normalmente, estes elementos são do mesmo tipo, e/ou do mesmo tamanho, e armazenados sequencialmente na memória. Seja $k \in \mathbb{N}$. A operação de acesso a um arranjo $a[k]$ é dita válida se $0 \leq k < tamanho(a)$, sendo $tamanho$ a função que retorna o número de elementos de um arranjo.

Estritamente falando, esta condição de k não é respeitada em várias linguagens. Em C, por exemplo, você pode usar $k < 0$ ou $k \geq tamanho(a)$. O motivo para isso é desempenho: garantir estas condições em tempo de execução pode afetar a eficiência de aplicações críticas. Este fato, por sua vez, é por causa que os operadores de colchete ([ ]), sendo C uma linguagem que preza seu baixo-nível, são um *syntatic sugar* apenas: uma sintaxe que simplifica uma operação que ocorre sobre um ponteiro que representa o arranjo (Ritchie; Kernighan, 1989, p.97).

A manipulação de arranjos pode ser feita utilizando a chamada *aritmética de ponteiro*. De maneira simples, a operação de acesso a um arranjo pode ser representada como se segue: seja $pointer$, o valor de um endereço de memória, $index$ o índice do elemento que se deseja acessar e $dataSize$ o tamanho, em bytes, de um dos elementos. Podemos acessar o elemento na posição $index$ por $newPointer = pointer + index * dataSize$, em que $newPointer$ é o ponteiro para o endereço de memória do elemento na posição $index$ do arranjo. Portanto, em C, quando se escreve:

```
1    int *arranjo;
2    int elemento = arranjo[5];
```

Listing 2.1 – Exemplo de acesso de arranjo em C

A operação *arranjo[5]* soma o endereço de *arranjo* com $5 * sizeof(int)$ e retorna o inteiro deste endereço, sendo *sizeof* uma função que retorna o tamanho, em bytes, de um tipo.

2.1.2 Quebra de limite de arranjo

O problema surge quando, na maioria dos casos em que se utiliza de arranjos, desenvolvedores não obedecem às condições de k mostradas anteriormente. Esse equívoco ocorre porque é muito mais fácil supor que um arranjo corresponde a uma sequência de elementos na memória ao invés de ser apenas um ponteiro que é utilizado com o mesmo intuito. Isso leva a índices que podem acessar elementos anteriores ao início do arranjo ou posteriores ao último elemento.

Este problema é o principal motivo que leva a vulnerabilidades de segurança, nomeadamente *Buffer Overflow* e *Memory Corruption* (Özkan, 2023). Esses tipos de *bugs* são difíceis de detectar e podem causar terminações súbitas de programas. Até mesmo em linguagens que lidam em tempo de execução com este problema, como Java ou C#, a solução padrão que oferecem é terminar a execução do programa quando um índice fora dos limites é encontrado. Exemplo de código em Java que quebra limite de arranjo:

```
1    public class Programa {
2       public static void main(String[] args) {
3            int[] arranjo = new int[4];
4            for (int i = 0; i < 5; i++)
5                System.out.print(arranjo[i]);
6            System.out.print("Isso nao sera printado");
7       }
8    }
```

Listing 2.2 – Quebra de limite de arranjo em Java

Executando este código se depara com uma exceção em tempo de execução:

```
0000Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
    Index 4 out of bounds for length 4 at Programa.main(Programa.java:5)
```

Listing 2.3 – Exceção de quebra de limite de arranjo em Java

2.1.3 Compilação

Programas podem ser entendidos como uma sequência de comandos dados a uma máquina. Porém, cada máquina possui diferentes processadores, que irão rodar os programas, e que possuem diferentes conjuntos de comandos de diferentes nomes e formatos. Escrever programas em baixo-nível, em que se tem de levar em conta todos detalhes técnicos do *hardware* que vai rodar o programa, é difícil, leva tempo e é propenso a erros.

Além disso, a máquina pode ter também um sistema operacional com seu próprio padrão e protocolo de comandos dificultando ainda mais a portabilidade de um programa escrito em uma linguagem de baixo-nível que precise rodar em máquina diferentes com sistema operacionais diferentes. Usa-se então um programa chamado *Compilador* para traduzir um programa escrito em uma linguagem de alto-nível, esta que se abstrai das especificidades da máquina e sistema operacional, para o *código de máquina*.

Adicionalmente, compiladores foram estendidos para verificar diversas propriedades de programas e, com isso, garantir que certos erros não ocorrerão. Estritamente falando, neste trabalho será construído um Compilador que terá como saída uma linguagem intermediária que será lida por um interpretador, que nada mais é que um programa que executa esta representação ao invés de interpretar o código de máquina diretamente.

A compilação, por sua vez, ocorre em quatro fases: análise léxica, análise sintática, análise semântica e geração de código. A primeira simplesmente agrupa o texto em símbolos, chamados *tokens*, estes são passados para a próxima fase para verificar se a sequência destes fazem parte da linguagem respectiva definida por uma gramática. Então é verificado se tal sequência representa um programa válido de acordo com regras predefinidas para a linguagem. Finalmente, a geração de código é responsável por produzir o código executável final.

2.1.4 Projeto de Linguagens de Programação

Após a identificação de *tokens*, para cada linguagem, é necessário definir sua sintaxe, sua aparência, e sua semântica, seu significado (Aho Ravi Sethi, 1995).

2.1.4.1 Estrutura Sintática

Aqui apresentaremos a sintaxe de linguagens usando-se de Gramáticas Livre de Contexto. Usaremos a *Forma de Backus-Naur Extendida* que foi modificada várias vezes desde que foi sugerida e o que o leitor verá abaixo é ela com alguns detalhes a mais, porém não importa tanto pois será explicado o que cada item significa logo abaixo:

1. *programa* → *declaracaoVariavel* eCondicional+*
2. *declaracaoVariavel* → **Identificador =** *eBooleana***;**
3. *eCondicional* → **if** *eBooleana* **then** *BoolOrIf* **else** *BoolOrIf*
4. *BoolOrIf* → *eBooleana* | *eCondicional*
5. *eBooleana* → **true** | **false** | *eBooleana* **||** *eBooleana* | *eBooleana* **&&** *eBooleana* | **Identificador**

Cada uma dessas linhas chama-se *produção*. No lado esquerdo de cada uma temos um símbolo não-terminal, que é um elemento da linguagem que se "transforma" em outro. A seta significa "pode ter a forma de" que, na sua direita, mostra exatamente os elementos que este não-terminal da esquerda pode ser. A barra **|** significa "ou", ou seja, o não-terminal pode ser **ou** o da esquerda desta barra **ou** o que tem na direita. Todo não-terminal aqui está escrito em *itálico* enquanto os terminais, os símbolos em si (*tokens*), aqueles que não se transformam em outro, estão em **negrito**. Os símbolos ***** e **+** adjacentes aos não-terminais significam respectivamente:

"pode ter zero ou mais não-terminais deste em sequência"e "pode ter um ou mais". Os outros símbolos como & e = também são terminais. Finalmente, os colchetes ao redor de um não-terminal significa que este é opcional, ou seja, pode ou não aparecer. No caso da gramática acima não temos não-terminais opcionais. A primeira produção chama-se *símbolo de partida*, é o símbolo que começa o processo de reconhecer a linguagem. Um exemplo de programa que esta gramática reconhece:

```
1   var1 = true;
2   var2 = false && false;
3   var3 = false;
4   var4 = false || var1 && false;
5   if false || false then
6       if var1 then
7           var3
8       else
9           true
10  else
11      if true && true then
12          false || true
13      else
14          var2
```

Listing 2.4 – Exemplo de programa sobre a gramática apresentada anteriormente

Linhas 1 a 4 são produzidas através da produção 2 (*declaracaoVariavel*). Linha 5 a 14 é uma produção de *eCondicional* com uma *eBooleana* que se abriu com *eBooleana* || *eBooleana* e dois *BoolOfIf* que se transformaram em outras duas *eCondicional*.

2.1.4.2 Estrutura semântica

Um texto representando um programa em uma linguagem que passa pela sua checagem gramatical ainda se vê necessário checar se obedece suas regras semânticas. A semântica de uma linguagem descreve um conjunto de regras que determina se um certo programa é válido ou não, regras próximas do que é implementado de fato em um compilador desta linguagem. Além disso, temos a semântica operacional junto de suas regras de redução que mostram processos que o próprio interpretador, ou computador, seguem quando executam certo programa (que será apresentado na próxima subseção).

Antes de apresentar as regras, mostra-se necessário definir, através de uma gramática, os elementos que farão parte destas regras. Numa linguagem temos essa sintaxe para expressões:

$$\textit{expressoes} \rightarrow \textbf{Inteiro} \mid \textbf{Real} \mid \textbf{Caractere} \mid \textit{Identificador}$$

Inteiro denomina uma constante inteira qualquer, assim como **Real** uma constante de ponto flutuante, **Caractere** um caractere e **Identificador** é qualquer palavra alfanumérica

começando com uma letra alfabética.

Durante uma análise semântica podemos utilizar de dados contidos em um *contexto* que pode ser entendido como um conjunto de informações que vai sendo incrementado pelo decorrer do processamento do texto pelo compilador.

Dado um contexto Γ contendo definições de variáveis de um texto, nesta linguagem, formado por pares (x, τ) em que x é um identificador e τ seu tipo, temos a notação $\Gamma(x) = \tau$ denota que (x, τ) $\in \Gamma$ e Γ, x : τ denota $\Gamma \cup (x, \tau)$ (Tais operações também se aplicam a contextos que serão apresentados futuramente). Definindo uma regra de expressões na forma $\Gamma \vdash_{\mathrm{e}} e : \tau$ que representa que a expressão *e* possui um tipo τ sobre o contexto Γ e tendo uma linguagem com tipos **Int**, **Flt**, **Char** podemos afirmar as seguintes regras sobre expressões:

1. $$\frac{}{\Gamma \vdash_{\mathrm{e}} \textbf{Inteiro} : \textbf{Int}}$$

2. $$\frac{}{\Gamma \vdash_{\mathrm{e}} \textbf{Real} : \textbf{Flt}}$$

3. $$\frac{}{\Gamma \vdash_{\mathrm{e}} \textbf{Caractere} : \textbf{Char}}$$

4. $$\frac{\Gamma(x) = \tau}{\Gamma \vdash_{\mathrm{e}} \textbf{Identificador} : \tau}$$

Regra 1, 2 e 3 afirmam o tipo de constantes literais enquanto Regra 4 mostra que variáveis, vulgo identificadores, possuem o tipo que lhes é atribuído pelo contexto de tipos, logo um programa escrito nesta linguagem deve respeitar essas regras de tipo para que este programa seja válido. Mais regras serão apresentadas quando falarmos sobre a linguagem FonC, proposta por este artigo.

2.1.4.3 Semântica Operacional

A semântica operacional é o conjunto de regras que definem como um programa é interpretado, uma vez que é compilado. Utilizando-se da linguagem apresentada em 2.1.4.1 na qual se espera que um programa ao ser interpretado completamente retorne um valor **true** ou **false**, podemos representar sua semântica operacional de expressões com regras do tipo $\sigma; e \Rightarrow_{\mathrm{e}} val$ em que σ é a associação do nome de variáveis ao seu valor, representando assim a memória de execução do programa, e é a expressão a ser executada e val o valor retornado. De maneira simples, um valor representa o resultado final da computação de uma expressão. A notação $\sigma(x) = val$ denota que o valor val está associado ao nome x em σ. Temos assim alguns exemplos de regras para esta linguagem:

$$\sigma; val \Rightarrow_{\mathrm{e}} val$$

$$\sigma; x \Rightarrow_{\mathrm{e}} \sigma(x)$$

Definem que a execução de valores são imediatos e de variáveis apenas se obtém o valor da memória. Para expressões contendo operadores binários, temos:

$$\frac{\sigma; e_1 \Rightarrow_{\mathrm{e}} val_1 \qquad \sigma; e_2 \Rightarrow_{\mathrm{e}} val_2}{\sigma; e_1 \circ e_2 \Rightarrow_{\mathrm{e}} val_1 \oplus val_2}$$

$\circ$ denota um símbolo de operador binário e $\oplus$ representa uma função correspondente a este operador sobre valores. Finalmente, definimos a semântica sobre expressões condicionais:

$$\frac{\sigma; e_1 \Rightarrow_{\mathrm{e}} val_1}{\sigma; \textbf{if true then } e_1 \textbf{ else } e_2 \ \Rightarrow_{\mathrm{e}} val_1}$$

$$\frac{\sigma; e_2 \Rightarrow_{\mathrm{e}} val_2}{\sigma; \textbf{if false then } e_1 \textbf{ else } e_2 \ \Rightarrow_{\mathrm{e}} val_2}$$

Definido a semântica operacional sobre expressões temos ainda a semântica sobre declarações que apenas modificam a memória de execução σ. Usando $\bullet$ para sinalizar que não há mais declarações restantes:

$$\frac{\sigma; c \Rightarrow_{\mathrm{c}} \sigma_1 \qquad \sigma_1; cs \Rightarrow_{\mathrm{cs}} \sigma_2}{\sigma; c\ cs \Rightarrow_{\mathrm{cs}} \sigma_2}$$

$$\sigma; \bullet \Rightarrow_{\mathrm{cs}} \sigma$$

2.2 Trabalhos Relacionados

Várias soluções foram sugeridas ao longo do tempo. Desde verificação dinâmica em tempo de execução, mostrada anteriormente em *Java*, que pode fechar um programa ao detectar acesso inválido, evitando apenas que uma operação indesejada ocorra mas que não previne de fato o problema. Ferramentas de análise de código em C, como *Valgrind* (Nethercote; Seward, 2007) ou *AddressSanitizer* (Serebryany *et al.*, 2012), ajudam porém geram falsos positivos, deixam o código lento, necessitam de um trabalho a mais para serem configurados e não detectam todos os casos de erro.

Outras ferramentas como *Cppcheck* (Marjamaki, 2024), que apresenta bons resultados apenas em casos simples gerando vários falsos negativos em todo o resto, *Checkmarx SAST* (Checkmarx…, 2024), que leva muito tempo para detectar a quebra de limite de arranjos em projetos relativamente grandes mesmo que desativemos todas as outras checagens, gera vários falsos positivos e falsos negativos e não consegue lidar com violação de acesso de índices dentro de *loops*. *Fortify Static Code Analyzer* (Fortify…, 2024) também gera vários falsos positivos e falsos negativos e também não lida com o problema dentro de *loops*.

Ferramentas que encontram erros baseadas em heurísticas e casamento de padrão como *FindBugs* (Ayewah *et al.*, 2008) não garantem encontrar todos os erros. Assistentes de provas necessitam de muita escrita para serem utilizados e são difíceis de entender, além de necessitar a reimplementação de um programa em outro formato.

Trabalhos com soluções semelhantes à análise semântica foram implementados como (Kellogg *et al.*, 2018), em *Java*, através de um sistema de tipos complementar. Porém, este gera muitos falsos positivos, necessita de muitas anotações de tipo explicitamente escritas pelo código, foi feito apenas para casos específicos do problema além de detectar apenas quando o tamanho do arranjo era imutável.

Em (Chimdyalwar, 2012) o autor avalia quatro ferramentas de análise estático de código para o problema da quebra de limite em arranjos: Polyspace, Coverity, UNO e CBMC. Polyspace não gerou falsos negativos porém só pôde rodar em programas com menos de 15 mil linhas pelo grande uso de memória; Coverity gera falsos positivos demais; UNO também gera falsos positivos e negativos além de não rodar em programas grandes e CBMC não possui uma precisão razoável em programas grandes.

Em (Nguyen; Irigoin, 2005) foi proposto um método de análise estática para o problema de quebra de limite dos índices em Fortran. Em (Venet; Brat, 2004) também foi desenvolvido um método de análise estática para este problema em programas embutidos com 80% de precisão, porém testado apenas em programas específicos da NASA.

Em (Gao *et al.*, 2021) é proposto um método baseado em análise estática baseado em análise de contaminação (*Taint Analysis*), análise de ponteiro, análise de fluxo de dados e resolução de restrições com o provador de teoremas Z3 (The…, 2024). Também sofre com a geração de falsos negativos e falsos positivos, com uma taxa de 16.3%.

3 Desenvolvimento

Neste capítulo apresenta-se a especificação da linguagem de programação FonC e o algoritmo verificador de acesso a arranjos. A Seção 3.1 discorre sobre a estrutura léxica, sintaxe, semântica estática e operacional da linguagem FonC. A Seção 3.2 discorre sobre cada etapa do algoritmo de verificação estática.

3.1 Especificação de FonC

3.1.1 Estrutura Léxica

A linguagem usa o conjunto de caracteres da tabela ASCII. Cada uma das possíveis categorias léxicas da linguagem são descritas a seguir:

- **Identificador**: sequência de caracteres alfanuméricos, sempre começando com uma letra. Exemplos de variáveis: *var*, *fonCIsGreat*. Exemplo de tipos: *Vector3*, *IntList*.
- **Inteiro**: um literal inteiro, sequência de um ou mais dígitos. Exemplos: *123*, *42*.
- **Real**: um literal de ponto flutuante, sequência de um ou mais dígitos, seguido por um ponto e uma sequência de um ou mais dígitos. Exemplos: *3.141526535*, *1.618*.
- **Caractere**: um único caractere delimitado por aspas simples. Exemplos: *'a'*, *'{'*. Caracteres especiais como quebra-de-linha, tabulação, *backspace* e *carriage return* são definidos usando o caracter de escape `\n`, `\t`, `\b` e `\r`, respectivamente.
- **CadeiaDeCaracteres**: uma sequência de caracteres delimitada por aspas duplas. Pode também possuir os caracteres especiais. Exemplos: *"A. Isto nao eh um exemplo* `\n`*"*, *"B. E isto nao eh uma cadeia de caracteres* `\n`*"*, *"C. Ambas alternativas anteriores estao incorretas* `\n`*"*.

Dois tipos, variáveis ou funções distintas não podem ter o mesmo nome. Texto escrito após \\ é ignorado pelo compilador, podendo ser utilizado como comentário sobre o código.

3.1.2 Estrutura sintática

1. *programa* → *processo**
2. *processo* → *processoData* | *processoFonc* | *processoUnion* | *processoType* | *processoMain*

3. *processoInterno* → *processoNew* | *processoSet* | *processoReturn* | *processoBreak* | *processoContinue* | *processoIf* | *processoElif* | *processoWhile* | *funcao*
4. *processoData* → **data (Tipo IdentificadorNovo** *constanteLiteral* **)**
5. *processoNew* → **new (Tipo IdentificadorNovo** *valor+* **)**
6. *processoSet* → **set (Tipo Identificador** *valor+* **)**
7. *TipoOuVoid* → **Tipo** | **void**
8. *processoFonc* → **fonc (** *TipoOuVoid* **IdentificadorNovo** *TipoEIdentificador** **) {** *processoInterno** **}**
9. *processoUnion* → **union (IdentificadorNovo** *TipoEIdentificador+* **)**
10. *processoType* → **type (IdentificadorNovo** *TipoEIdentificador+* **)**
11. *processoMain* → **main ( ) {** *processoInterno** **}**
12. *processoReturn* → **ret (** *valor** **)**
13. *processoBreak* → **break ( )**
14. *processoContinue* → **continue ( )**
15. *processoIf* → **if (** *valor1* **) {** *processoInterno** **}**
16. *processoElif* → **elif (** *valor1* **) {** *processoInterno** **}**
17. *processoWhile* → **while (** *valor1* **) {** *processoInterno** **}**
18. *funcao* → **Identificador (** *valor** **)**
19. *TipoEIdentificador* → **Tipo IdentificadorNovo**
20. *variavel* → **Identificador** *variavel2*
21. *variavel2* → | **. Identificador** *variavel2* | **[** *valor1* **]** *variavel2*
22. *valor* → *valor1* | **(** *valor* **)**
23. *valor1* → *constanteLiteral* | *variavel* | *funcao*
24. *constanteLiteral* → **Inteiro** | **Real** | **Caractere** | **CadeiaDeCaracteres**

Como exemplificado em 2.1.4.1, acima é a especificação da gramática da linguagem que utilizaremos durante o desenvolvimento, chamada **FonC**. É importante notar que em nenhuma produção temos operadores como **+**, **-**, ***** e **/** apesar de que normalmente encontra-se operadores aritméticos em outras linguagens de programação. O motivo para isso é tanto simplicidade,

durante a fase de semântica não temos que tratar com precedência de operadores porque isto estará explícito no código naturalmente, quanto expressividade, usamos funções para denotar operações aritméticas porque em computação temos uma preocupação maior de denotar os tipos tanto de retorno quanto de parâmetros de funções para melhor leitura do código e sem operadores estes detalhes ficam "óbvios" ao se olhar o código.

Nota-se que nossa linguagem aceita programas vazios ou uma lista de processos, que nada mais são que funções que não retornam valor. Já adianto que a diferenciação é importante pois não se pode chamar processos em lugares que se espera um valor de retorno nem chamar funções para um lugar que não se espera retorno. Como estamos tratando da aparência da linguagem poderia se juntar a produção 2 (*processo*) com 18 (*função*) porém vale a pena mostrar aqui para que não se pense que pode chamar um *processoData*, por exemplo, num escopo interno, mesmo que o correto seria tratar disso quando formos falar da semântica da linguagem.

Outro detalhe é que diferenciou-se **Identificador** de **IdentificadorNovo** e também de **Tipo**, apesar de todos serem sequências alfanuméricas lexicalmente, isso foi feito para melhor entendimento quando estivermos detalhando a semântica da linguagem. Exemplo de um programa:

```
type(Arr u8* arr u32 count u32 size)
type(Circulo f32 raio f32 area f32 perimetro)
data(f32 PI 3.14)
fonc(f32 areaCirculo f32 r){
    new(f32 area f32mul(PI f32mul(r r))
    ret(area)
}
fonc(f32 perimetroCirculo f32 r){
    ret(f32mul(2 f32mul(PI r)))
}
data(u8* arrBuffer array(60))
main(){
    new(Arr stringSaida (arrBuffer 0 60))
    new(f32 num3 3)
    new(Circulo (num3 areaCirculo(num3) perimetroCirculo(num3)))
    f32Concat(&stringSaida circ.raio)
    c8Concat(&stringSaida '\n')
    f32Concat(&stringSaida circ.area)
    c8Concat(&stringSaida '\n')
    f32Concat(&stringSaida circ.perimetro)
    c8Concat(&stringSaida '\n')
    print(&stringSaida)
}
```

Listing 3.1 – Exemplo de programa em FonC

```
3.000000
28.260000
```

```
3     18.840000
```

Listing 3.2 – Saída do programa acima executado

Vemos acima a utilização do *processoType* (produção 9), *processoData* e *processoFonc* (vamos falar sobre o que cada um faz na próxima seção). Observa-se também o uso de funções, que retornam, nas linhas 5 e 9 com *f32mul*, 11 com *array* e 15 com *areaCirculo* e *perimetroCirculo*. Assim, peço ao leitor que reveja a lista de produções e compare com o código acima durante um tempo para ver como cada um dos não-terminais se abrem até chegar num terminal e confirmar que a gramática apresentada aceita o programa acima.

3.1.3 Contexto inicial da linguagem

No escopo externo de FonC, temos meta-comandos, ou definições, que são códigos processados pelo compilador para criar novas funções (processoFonc), tipos (processoType) ou constantes (processoData) e temos comandos internos, nos escopos internos, que serão interpretados ao rodar o programa. Estes podem retornar um valor ou não, dependendo da sua definição, e possuir zero ou mais tipos como parâmetros. Enquanto isso, as definições têm efeito apenas nos contextos, explicados em 2.1.4.2, e no começo de todo programa todos contextos apresentados aqui já estão populados com definições padronizadas da linguagem. A linguagem possui, por padrão, estes tipos primitivos:

- **u8**, representa um dado com tamanho de 1 byte. Pode ser usado para representar um valor numérico de 0 a 255, um booleano $b\ != \ 0$ (true) ou $b\ == \ 0$ (false) ou qualquer outro tipo de dado conceitual que o usuário deseje representar que caiba nele. É importante notar que o que diferencia os dados primitivos apresentados aqui é apenas o tamanho máximo que suportam, não há um intervalo válido ou regras a mais que devem ser satisfeitas para um valor ser **u8** com exceção de que precisa caber em 1 byte.
- **i32**, representa um dado com tamanho de 4 bytes. Normalmente utilizado em operações aritméticas de números inteiros que precisam levar em conta o sinal positivo ou negativo.
- **u32**, representa também um dado com tamanho de 4 bytes. Novamente, a razão para este existir, visto que i32 possui o mesmo tamanho, é para operações aritméticas de números inteiros quando o sinal é sempre positivo.
- **i64**, representa um dado com tamanho de 8 bytes. Mesma utilidade que **i32** porém maior.
- **u64**, representa um dado com tamanho de 8 bytes. Mesma utilidade que **u32** porém maior.
- **f32**, representa um dado com tamanho de 4 bytes. Normalmente utilizado em operações aritméticas de ponto flutuante.
- **f64**, representa um dado com tamanho de 8 bytes. Mesmo que **f32** porém maior.

Ademais, a linguagem possui suporte a ponteiros que nada mais são que endereços de memória de 4 bytes que apontam para um tipo qualquer. Exemplos: **i32***, **u8***. Estes ponteiros possuem esse tamanho, diferentemente da maioria das linguagens que possuem ponteiro de 8 bytes (em sistemas 64 bits), porque um programa FonC é limitado para ter 4GB de memória no máximo internamente, uma quantidade de memória que o autor considera o suficiente para a grande maioria dos programas.

Além disso, possui funções que representam operações aritméticas que normalmente se apresentam em outras linguagens através dos operadores matemáticos, como *f32add* que adiciona dois argumentos do tipo **f32** (tendo variações para os outros tipos como *u32add*, *u8add*, *i64add*, etc.), como *u32div* que divide dois argumentos do tipo **u32**, *u64mul* para multiplicação, entre outros.

Tem-se também processos, que serão explicados semanticamente na próxima seção, que fazem o básico esperado de uma linguagem de programação. Explicado informalmente são estes:

- *processoData* e *processoNew*: o primeiro pode rodar no escopo global para criar constantes. O último serve para criar variáveis locais e não pode rodar no escopo global;
- *processoSet*: altera uma variável, campo ou endereço de memória qualquer contanto que não seja uma constante;
- *processoFonc*: serve para criar um novo processo ou função;
- *processoType* e *processoUnion*: criam tipos que são a junção de tipos primitivos da linguagem. *processoUnion* cria tipos que terão o tamanho do maior tipo passado, podendo assim armazenar qualquer um dos tipos passados;
- *processoMain*: é um processo onde se define comandos que serão rodados quando o programa for executado. Programas sem *processoMain* também são válidos;
- *processoReturn*: utilizado durante a definição de uma função em *processoFonc* para retornar um valor e voltar o fluxo de controle ao comando que chamou esta função;
- *processoWhile*: utilizado para repetir um bloco de comandos até que uma condição seja falsa;
- *processoBreak*: usado dentro de um *processoWhile* para sair precocemente da repetição de comandos.
- *processoContinue*: chamado dentro de um *processoWhile* para testar a condição de parada novamente, pulando comandos que seriam rodados na iteração atual;
- *processoIf*: caso a condição passada seja verdadeira roda o bloco passado;

- *processoElif*: caso a condição anterior de um *processoIf* ou *processoElif* tenha sido verdadeira este e todos os próximos *processoElif*s consecutivos são pulados. Caso tenha falhado este *processoElif* testa sua condição. Caso esta seja verdadeira seu bloco é rodado, caso contrário o controle de fluxo é passado para os comandos imediatamente depois de seu bloco;

3.1.4 Máquina virtual

Antes de especificarmos a semântica estática da linguagem é importante descrever a máquina virtual que interpreta o código gerado pelo compilador e suas especificidades. Como já foi dito, ponteiros possuem tamanho de 4 bytes, contrário aos de 8 bytes das outras linguagens, e por isso a memória interna de um programa em FonC é limitada a 4GB.

No fim da compilação, é gerado um arquivo objeto binário que possui duas partes importantes: memória de dados, onde é guardado as constantes do programa e a memória pré-alocada durante compilação, e as instruções, um vetor de elementos de 4 bytes que indicam as instruções, e seu conteúdo, que serão interpretadas. Não há alocação dinâmica na linguagem e esta memória pré-alocada é um espaço alocado apenas durante a compilação e este é gerado pela função **array** nativa da linguagem que pode ser usado como valor inicializador de um *processoData*.

A máquina então recebe este arquivo, aloca a memória de dados, preenchendo ela com as constantes e a memória pré-alocada, aloca um espaço predefinido para o *Stack*, e começa a interpretar a partir da instrução de *entry point* que também foi escrita pelo compilador no arquivo. No caso do *entry point* não ter sido definido, ou seja, o programa não definiu um processo *main*, o interpretador apenas fecha pois não há instruções para serem rodadas.

As instruções aceitas pela máquina são:

- MovSS <Endereco1> <Endereco2> <Tamanho>: esta instrução move um dado de *Tamanho* bytes do *Endereco2*, no *Stack*, ao *Endereco1*, também localizado no *Stack*;
- MovSD <Endereco1> <Endereco2> <Tamanho>: semelhante ao anterior, move um dado de *Tamanho* bytes do *Endereco2*, desta vez localizado no espaço de memória constante, ao *Endereco1*, localizado no *Stack*;
- MovSV <Endereco1> <Valor> <Tamanho>: último da família de "mov"s simples, este move ao *Endereco1*, no *Stack*, um *Valor* de *Tamanho* bytes. É importante mencionar que cada parte do arranjo de instruções que a máquina roda possui 4 bytes, ou seja, o "cabeçalho"de comando "MovSS", "MovSD"ou "MovSV", por si só são 4 bytes, o <Endereco>, <Valor> e <Tamanho> também. Logo, o <Valor> nesse caso sempre será entre $1 <= Tamanho <= 4$ bytes;
- MovAddS <Endereco1> <Endereco2>: move o valor de endereço de *Endereco2* para *Endereco1*;

- MovAccessingAddSS <Endereco1> <Endereco2> <Tamanho>: Acessa o *Endereco2*, no *Stack*, interpretando-o como um ponteiro e copia os dados apontados por este de *Tamanho* bytes para o *Endereco1*, no *Stack*;
- AccessingAddMovSS <Endereco1> <Endereco2> <Tamanho>: acessa o *Endereco1*, no *Stack*, interpretando-o como um ponteiro e copia os dados de *Endereco2*, no *Stack*, para o endereco que este ponteiro aponta;
- JmpT <Valor>: pula o fluxo de controle de um programa para a instrução de índice *Valor*;
- JmpS <Endereco> <Valor>: se o byte neste *Endereco* do *Stack* é diferente de zero pula para a instrução *Valor*;
- Jmp2S <Endereco> <Valor>: mesmo que o de cima porém pula quando o valor é igual a zero;
- Ret <Endereco>: pula para a instrução que está armazenada neste endereço do *Stack*;
- Call <FuncaoID> <Endereco>: chama a função interna identificada por *FuncaoID* passando o *Endereco* de onde começará o *workspace* da função, podendo retornar um valor no mesmo;
- UpdateStkF <Valor>: incrementa o *Stack Frame* atual pelo *Valor* passado, este é interpretado como um **i32** por conta dos casos em que se deseja decrementar o *Stack Frame*. Importante mencionar que para isso funcionar a máquina virtual guarda um valor **u32** que representa o *Stack Frame* atual.

É importante ressaltar que quando se diz de endereços no *Stack* ou na memória de dados ambos são números de 4 bytes que representam um valor de deslocamento em relação a algum ponto de referência. Quando se diz "no *Stack*" esta referência é sobre o valor do *Stack Frame* atual e quando se diz "na memória de dados" é referencial ao ponto inicial da memória do programa interno. Ex: Endereço 4 no *Stack*, quer dizer que dado o valor do *Stack Frame* se somarmos 4 bytes a mais chegaremos neste valor; Endereço 4 na memória de dados, quer dizer que somarmos 4 bytes a mais no ponteiro que indica o começo da memória chegamos neste valor.

Sempre que se chama uma função, seja interna através do comando *Call* ou externa pela junção de um *MovSV*, *UpdateStkF* e *Jmpt*, esta possui um *"workspace"*. Isso nada mais é que um termo que define um espaço local onde a função operará. Todo *workspace* começa com num endereço de *Stack* igual a 0. Isso é feito para garantir que cada função pode ser chamada a qualquer momento do código sem que o programador se preocupe em preparar o contexto para que uma função seja executada. Restritamente falando não é necessário saber disso, consiste em um detalhe de implementação, mas é interessante pensar que cada função, ao chamada, possui um espaço "reservado" para si mesma sem afetar o estado alheio, sendo impossível que ela acesse dados fora de seu *workspace*.

Ademais, as funções internas mencionadas anteriormente são aquelas que representam as operações aritméticas e diversas outras que foram definidas por motivos de otimização. Estas existem graças ao código que é escrito em C, que é a linguagem base para a máquina virtual. O código escrito nela pode ser otimizado pelo compilador *Clang* usado durante o processo de construção do código. Estas funções fazem parte do interpretador e já ficam inclusas em qualquer programa escrito em FonC.

3.1.5 Semântica Estática

Além do contexto Γ, mencionado em 2.1.4.2, contendo associações de variáveis com seus tipos e, nesse caso, com mais informações como por exemplo tamanho de arranjo, caso seja uma variável de arranjo criada com tamanho fixo, e se a variável está localizada no *Stack* ou não, temos mais dois contextos:

- Δ: contem informações sobre campos de tipos definidos pelo usuário. Este contexto é um conjunto de pares na forma (x, F) em que x é o nome do tipo e F é uma lista formada por pares (v, τ) em que v é o nome do campo de tipo τ.
- Θ: contem informações sobre os tipos de funções definidas pelo usuário. É um conjunto de pares na forma (f, (P, r)) em que f é o nome da função, P é uma lista formada por pares (v, τ) em que v é o nome do parâmetro de tipo τ e r é o tipo de retorno ou $\emptyset$ quando a função não retorna um valor, ou seja, quando é um processo.

Novamente, como dito em 2.1.4.2, a notação Γ(x) = τ denota que (x, τ) $\in \Gamma$ e Γ, x : τ denota $\Gamma \cup (x, \tau)$. Estas operações aplicam-se aos outros contextos também com seus respectivos pares. A sintaxe dos comandos para chamar (executar) uma função depende de seu par contido em Θ. Uma função $(funcao, ([(arg0, u32), (arg1, u32)], \emptyset))$, tendo 2 parâmetros na sua lista, poderia ser chamada com a *funcao* com dois argumentos em sua chamada. Ex: $funcao(var0\ var1)$.

Além disso, certas funções e processos pedem argumentos de tipos e/ou identificadores novos (produções 4 a 10, por exemplo). Nestes casos teríamos um par com argumentos de tipo τ para denotar um tipo arbitrário e σ para denotar um identificador novo. *processoSet* é o único comando que precisa de um identificador já definido, neste formalismo isso é identificado por ρ. Sendo $P(z)$ uma função que retorna o conjunto de parâmetros de um par $z = (P, r)$ e $F'(t)$ a função que inicializa um tipo t, temos estes exemplos sobre o *processoNew* e *processoSet*:

$$\Theta(processoNew) = ([(tipo, \tau), (nomeVariavel, \sigma), P(\Theta(F'(\tau)))], \emptyset)$$
$$\Theta(processoSet) = ([(tipo, \tau), (nomeVariavel, \rho), P(\Theta(F'(\tau)))], \emptyset)$$

Ademais, certos comandos necessitam da abertura de um bloco de comandos internos logo depois de sua chamada (produções 8, 11, 15, 16, 17). Em nosso formalismo isso fica denotado com a letra β:

$$\Theta(if) = ([(condicao, u8), (bloco, \beta)], \emptyset)$$

Outra sintaxe que vale a pena mencionar são as produções 22 e 23 (os valores). Por ter literais constantes e variáveis ambos não sendo acompanhados de parênteses como os comandos e, além disso, variáveis podendo ter acessos a campos com pontos (.) e acesso a arranjos com colchetes ([]), podemos pensar nas variáveis e literais como funções com lista vazia de parâmetros. Por isso não há necessidade de dividir esses casos em outra categoria semântica da linguagem.

Apesar de estarmos utilizando um contexto apenas para as variáveis (o Γ) este poderia ser substituído pela utilização de uma função $r(z)$ que retorna o tipo de retorno de um par $z = (P, r)$. Sendo n o nome da variável: $\Gamma(n) = r(\Theta(n))$. Dito isto, vamos para as semânticas estáticas de comandos:

Um literal caractere sempre retorna **u8** e cadeias de caracteres sempre retornam **u8***:

$$\frac{}{\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} \textbf{Caractere} : \textbf{u8}}$$

$$\frac{}{\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} \textbf{CadeiaDeCaracteres} : \textbf{u8*}}$$

Porém, literais inteiros retornam o tipo com o qual eles são compatíveis:

$$\frac{\begin{array}{c} P(\Theta(f)) = [..., (v_i, \tau_i), ...] \\ \tau_i = \textbf{u8} \vee \tau_i = \textbf{u32} \vee \tau_i = \textbf{u64} \vee \tau_i = \textbf{i32} \vee \tau_i = \textbf{i64} \vee \tau_i = \textbf{f32} \vee \tau_i = \textbf{f64} \end{array}}{\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} \textbf{Inteiro} : \tau_i}$$

1. Primeiro, pega-se os parâmetros da função f, $P(\Theta(f)) = [..., (v_i, \tau_i), ...]$, visto que os literais só podem ser usados como parâmetros de comandos;

2. Depois verifica-se se o parâmetro do qual o inteiro faz parte possui um tipo compatível, $\tau_i = \textbf{u8} \vee \tau_i = \textbf{u32} \vee \tau_i = \textbf{u64} \vee \tau_i = \textbf{i32} \vee \tau_i = \textbf{i64}$.

O mesmo para literais de ponto flutuante:

$$\frac{\begin{array}{c} P(\Theta(f)) = [..., (v_i, \tau_i), ...] \\ \tau_i = \textbf{f32} \vee \tau_i = \textbf{f64} \end{array}}{\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} \textbf{Real} : \tau_i}$$

Ademais, certas funções possuem argumentos de tipos compostos e podemos usar valores de tipos contidos neste maior para construí-lo abrindo um parênteses:

$$\frac{\begin{array}{c}P(\Theta(f)) = [..., (v, \tau), ...] \\ \Delta(\tau) = [(v_1, \tau_1), ..., (v_m, \tau_m)] \\ \Gamma(valor_t) = \tau_t \vee r(\Theta(valor_t)) = \tau_t \quad 1 \leq t \leq m\end{array}}{\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} \textbf{(}valor_i \ ... \ valor_m\textbf{)} : \tau}$$

1. Se em uma função f temos um par com tipo τ, $P(\Theta(f)) = [..., (v, \tau), ...]$;
2. E esse tipo τ possui tipos internos τ_1 a τ_m, $\Delta(\tau) = [(v_1, \tau_1), ..., (v_m, \tau_m)]$;
3. Sendo cada um dos tipos o retorno de *valor*$_t$, caso seja uma função, ou o tipo de *valor*$_t$, caso seja uma variável, $\Gamma(valor_t) = \tau_t \vee r(\Theta(valor_t)) = \tau_t \ 1 \leq t \leq m$, logo essa operação também é válida.

Uma variável sempre possui o tipo que lhes é atribuído pelo contexto de tipos Γ:

$$\frac{\Gamma(x) = \tau}{\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} x : \tau}$$

O acesso a campos é verificado da seguinte forma:

$$\frac{\begin{array}{c}\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} x_1 : \tau_1 \\ \Delta(\tau_1) = S_1 \\ (x_2, \tau) \in S_1\end{array}}{\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} x_1.x_2 : \tau}$$

1. Primeiro, obtemos o tipo da variável x_1, $\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} x_1 : \tau_1$;
2. Em seguida, obtemos o conjunto de campos do tipo τ_1, $\Delta(\tau_1) = S_1$;
3. Por último, obtemos o tipo de x_2 no conjunto de campos de τ_1, $(x_2, \tau) \in S_1$.

Considerando $\tau*$ um tipo que define um valor que aponta para o tipo τ, assim é definido o acesso a arranjos:

$$\frac{\begin{array}{c}\Gamma(x_1) = \textbf{u32} \vee r(\Theta(x_1)) = \textbf{u32} \\ \Gamma(x) = \tau*\end{array}}{\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} x[x_1] : \tau}$$

1. Primeiro, checa-se x_1 retorna um valor de tipo **u32**, sendo uma variável ou função, $\Gamma(x_1) = \textbf{u32} \vee r(\Theta(x_1)) = \textbf{u32}$;

2. Verifica-se se x é um ponteiro, $\Gamma(x) = \tau*$.

Verificação de chamadas de comandos:

$$\frac{P(\Theta(f)) = [(x_1, t_1), ..., (x_m, t_m)] \quad \Theta; \Delta; \Gamma \vdash_{\mathrm{c}} \textit{valor}_i : t_i \quad 1 \leq i \leq m}{\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} \textbf{Identificador}(\textit{valor}_1, ..., \textit{valor}_m)}$$

1. Primeiro, pega-se os parâmetros da função f, $P(\Theta(f)) = [(x_1, t_1), ..., (x_m, t_m)]$;

2. E verifica-se se cada argumento obedece a ordem dos tipos, $\Theta; \Delta; \Gamma \vdash_{\mathrm{c}} \textit{valor}_i : t_i$.

Como apresentado anteriormente com o *processoNew*, certos comandos mudam seus parâmetros de acordo com um tipo que é passado como parâmetro. Outro comando muda seus parâmetros com um tipo que é o retorno da função no qual bloco o comando está, que é o caso do *processoReturn*. Nestes casos a linguagem gera uma função dinâmica que pode inicializar qualquer tipo. Ou seja, ela espera parâmetros "a mais" de acordo com dados do tipo passado. Exemplo:

```
1    type(V2 f32 x f32 y)
2    type(V3 f32 x f32 y f32 z)
3    main(){
4        new(V2 posicaoObjeto (4 4))
5        new(V3 posicaoObjeto3D (4 4 0))
6    }
```

Listing 3.3 – Exemplo de diferentes tipos inicializados com *processoData*

Percebe-se no código acima que para o mesmo processo, *processoNew*, temos parâmetros em quantidades diferentes. Esses argumentos à direita dos novos identificadores, *posicaoObjeto* e *posicaoObjeto3D*, inicializam o tipo de acordo com seus campos. A função $F'(\tau)$ que inicializa um tipo qualquer τ pode ser definida como:

$$\frac{\Delta(\tau) = f_s}{\Theta(F'(\tau)) = (f_s, \emptyset)}$$

A ordem e quantidade de seus argumentos é idêntica ao que é definido no contexto Δ sobre o tipo composto.

Para as próximas regras precisamos introduzir mais um contexto V que denomina os identificadores de variáveis incluídas anteriormente pelo escopo atual de um julgamento. O

julgamento que usaremos tem a forma $\Theta; \Delta; \Gamma; V \vdash_{\mathrm{d}} d \rightsquigarrow V'; \Gamma'$ que denota que o comando d é bem formado nestes contextos e gera um novo contexto Γ' e V'. A primeira regra se refere a quando um bloco é fechado e todas as variáveis que foram incluídas deste bloco saem do contexto Γ:

$$\frac{\Gamma' = \{(x, \tau) |\ x \notin V\}}{\Theta; \Delta; \Gamma; V \vdash_{\mathrm{d}} \mathbf{\}} \rightsquigarrow \emptyset; \Gamma'}$$

Veja que retira-se os identificadores locais e mantém apenas aqueles globais que foram definidos até o momento do julgamento. A próxima regra lida com blocos não vazios quando um comando pode ter alterado o contexto e este é repassado para os próximos comandos.

$$\frac{\begin{array}{c}\Theta; \Delta; \Gamma; V \vdash_{\mathrm{d}} c \rightsquigarrow V_1; \Gamma_1 \\ \Theta; \Delta; \Gamma_1; V_1 \vdash_{\mathrm{ds}} cs \rightsquigarrow V_2; \Gamma_2\end{array}}{\Theta; \Delta; \Gamma; V \vdash_{\mathrm{ds}} c\ cs \rightsquigarrow V_2; \Gamma_2}$$

Para comandos que tem como parâmetro identificadores novos (σ) que vem depois de um argumento de tipo, como *processoEnum* e *processoData*, incluímos os identificadores nos contextos Γ. Sendo $nAv(n)$ a função que verifica que um nome n não está em nenhum contexto, ou seja, $\Gamma(n) = \emptyset$, $\Theta(n) = \emptyset$, $\Delta(n) = \emptyset$ e $n \notin V$, temos:

$$\frac{\begin{array}{c}P(\Theta(f)) = [..., (x_k, \tau_k), ..., (x_i, \sigma_i), ...] \\ nAv(\sigma_i) \\ \Gamma' = \Gamma \cup (\sigma_i, \tau_k)\end{array}}{\Theta; \Delta; \Gamma; V \vdash_{\mathrm{d}} d \rightsquigarrow \Gamma'}$$

Para o *processoNew* incluímos também em V:

$$\frac{\begin{array}{c}P(\Theta(f)) = [..., (x_k, \tau_k), ..., (x_i, \sigma_i), ...] \\ nAv(\sigma_i) \\ \Gamma' = \Gamma \cup (\sigma_i, \tau_k) \\ V' = V \cup \sigma_i\end{array}}{\Theta; \Delta; \Gamma; V \vdash_{\mathrm{d}} d \rightsquigarrow \Gamma'; V'}$$

Uma exceção que usa identificadores novos é *processoFonc* que atualiza Θ com o nome do novo comando. Considere $a[z]$ a operação que pega o elemento na posição z da lista a. Considere também m sendo o tamanho de uma lista e $v(z)$ a função que retorna o nome de um par (vn, ty), vn o nome do parâmetro e ty seu tipo. Por fim, considere $t(z)$ a função que retorna o tipo de um par (vn, ty):

$$\frac{\begin{array}{c} P' = P(\Theta(processoFonc)) \\ P' = [..., (x_2, \sigma_2), ..., (x_m, \beta_m)] \\ nAv(\sigma_2) \\ P'' = P' \setminus P[1] \setminus P[2] \setminus P[m] \\ retType = t(P[1]) \\ \Theta' = \Theta \cup (\sigma_2, P'', retType) \end{array}}{\Theta; \Delta; \Gamma; V \vdash_{\mathrm{d}} d \rightsquigarrow \Theta'}$$

1. Dado os parâmetros de *processoFonc*;
2. Verifica-se se o nome do comando não existe em algum contexto, $nAv(\sigma_2)$;
3. Caso o comando não retorne preenchemos o novo comando apenas com os argumentos, note que removemos o par que contém o argumento de bloco. Caso contrário colocamos no retorno o tipo correspondente.

Assim *processoFonc* pode ter quantidade arbitrária de parâmetros contanto que venham em pares de τ σ, depois do nome para o comando, e inclui-se estes identificadores novos no contexto V quando não está em nenhum contexto:

$$\frac{\begin{array}{c} P' = P(\Theta(processoFonc)) \\ P' = [..., (x_2, \sigma_2), ..., (x_k, \beta_k)] \\ P'' = P' \setminus P[1] \setminus P[2] \setminus P[k] \\ retType = t(P[1]) \\ V = \sigma_2 \\ P'' = [(x_i, \tau_i), (x_{i+1}, \sigma_{i+1}), ..., (x_{m-1}, \tau_{m-1}), (x_m, \sigma_m)] \\ nAv(\sigma_{i+1}) \quad i \in [1, 3, 5, ..., m] \\ V = V \cup v(P''[i+1]) \quad i \in [1, 3, 5, ..., m] \\ \Theta' = \Theta \cup (\sigma_2, P'', retType) \\ \Gamma' = \Gamma \cup P'' \\ i = 2 \Rightarrow \Gamma' = \Gamma' \cup (\sigma_t, retType) \end{array}}{\Theta; \Delta; \Gamma; \vdash_{\mathrm{d}} d \rightsquigarrow V; \Gamma'; \Theta'}$$

1. Dado os argumentos de *processoFonc*, criamos uma lista P'' que contem todos argumentos do novo comando, ignorando-se o argumento de tipo de retorno $P[1]$, de nome do comando $P[2]$ e também o argumento de bloco $P[k]$;
2. Inicializa-se um novo contexto V de variáveis locais;
3. Verifica-se se as novas variáveis não se encontram nos contextos existentes e atualiza-se o V com estas;

4. Atualiza-se o Θ com o novo comando, Γ com as novas variáveis e com o retorno do proprio comando caso exista.

No caso de *processoFonc* criar uma função é preciso verificar se no corpo ela retorna o tipo definido. Regras $(\tau_1, ..., \tau_n) \vdash_{\text{ret}} cs$ validam que todos os caminhos do bloco cs terminam com um **ret** apropriado. A próxima regra verifica que em um bloco contendo apenas um comando, este deve ser um **ret**:

$$\frac{\Theta; \Delta; \Gamma \vdash_{\text{ret}} \textit{valor*} : \tau}{(\tau_1, ..., \tau_n) \vdash_{\text{ret}} \textbf{ret(}\textit{valor*}\textbf{)}\}}$$

Caso o bloco seja formado por uma sequência de dois ou mais comandos, devemos ignorar o primeiro comando e verificar a cauda do bloco:

$$\frac{(\tau_1, ..., \tau_n) \vdash_{\text{ret}} cs}{(\tau_1, ..., \tau_n) \vdash_{\text{ret}} c\ cs}$$

Outras exceções que usam identificadores novos são *processoType* e *processoUnion* que atualizam Δ com o primeiro identificador novo, que se refere ao nome do novo tipo. Estes também podem ter argumentos ilimitados, em pares $\tau\ \sigma$ sem repetir nome, que são colocados em par incluído no Δ:

$$\frac{\begin{gathered} P' = P(\Theta(f)) \\ P' = [(v_1, \sigma_1), ...] \\ nAv(\sigma_1) \\ P'' = P' \setminus P'[1] \\ P'' = [(v_i, \tau_i), (v_{i+1}, \sigma_{i+1}), ..., (v_{m-1}, \tau_{m-1}), (v_m, \sigma_m)] \\ (v_{i+1}, \sigma_{i+1}) \notin P'' \quad i \in [1, 3, ..., m] \\ \Delta' = \Delta \cup (\sigma_1, P'') \end{gathered}}{\Theta; \Delta; \Gamma; \vdash_{\text{d}} d \rightsquigarrow \Delta'}$$

Uma menção importante, que dessa vez usa identificadores não-novos assim como *processoSet* é o operador '&' que é uma função:

$\Theta('\&') = ([(nomeVariavel, \rho)], \textbf{u8*})$

Ou seja, pega-se uma variável qualquer já conhecida e retorna um ponteiro que é o endereço de memória para esta variável passada.

Regras no formato $CS \vdash_{\text{cs}} cs_1\ cs\ cs_2$ afirmam que um comando cs está numa posição válida entre cs_1 e cs_2. Com isto, temos processos que só podem ser chamados dentro de outros, como é o caso de *processoBreak* e *processoContinue* que podem apenas ser chamados dentro de um *processoWhile*:

$$CS \vdash_{\text{cs}} \textbf{while(}\textit{valor}\textbf{)\{ } cs_1 \textbf{ break() } cs_2 \textbf{ \}}$$

$$CS \vdash_{\text{cs}} \textbf{while(}\textit{valor}\textbf{)\{ } cs_1 \textbf{ continue() } cs_2 \textbf{ \}}$$

Ademais, *processoElif* só pode ser chamado após um *processoIf* ou outro *processoElif*.

$$CS \vdash_{\text{cs}} \textbf{if(}\textit{valor}\textbf{)\{ } cs_1 \textbf{ \} elif(}\textit{valor}\textbf{)\{ } cs_2 \textbf{ \}}$$

$$CS \vdash_{\text{cs}} \textbf{elif(}\textit{valor}\textbf{)\{ } cs_1 \textbf{ \} elif(}\textit{valor}\textbf{)\{ } cs_2 \textbf{ \}}$$

3.1.6 Semântica Operacional

A semântica operacional será descrita com regras do tipo $\Delta; \theta; \sigma; c \Rightarrow_{\text{c}} val; \sigma'$ em que:

- θ é a associação entre nome de funções e sua definição;
- σ é a associação de variáveis e seu valor, representando a memória de uma execução;
- Δ a associação de nomes de campos e seu tipo.
- c é o comando a ser executado
- val o valor retornado, este sendo o resultado final da computação de um comando. Este pode ser $\emptyset$ quando rodamos um processo e σ' quando a memória de execução é alterada;

A notação $\sigma(x) = val$ denota que o valor val está associado ao nome x em σ. E representamos por $\sigma[x \mapsto val]$ o ambiente σ' tal que $\sigma'(i) = \sigma(i)$, para todo $i \neq x$ e $\sigma'(x) = val$. As primeiras regras lidam com a avaliação de valores e variáveis, podendo ser interpretadas como um comando que não possui parâmetros porém retorna um valor:

$$\Delta; \theta; \sigma; \textit{constanteLiteral} \Rightarrow_{\text{c}} val$$

$$\Delta; \theta; \sigma; \textbf{Identificador} \Rightarrow_{\text{c}} \sigma(\textbf{Identificador})$$

A execução de valores é imediata e a de variáveis apenas se obtém o valor associado a ela na memória de execução. Sendo A um valor de ponteiro, n um valor de tipo **u32** e D um valor de tipo, a execução de acesso a elementos de arranjos e a campos de registros são definidas pelas regras a seguir:

$$\frac{\Delta;\theta;\sigma;\textbf{Identificador} \Rightarrow_{\mathrm{c}} A_1 \quad \Delta;\theta;\sigma;\textit{valor} \Rightarrow_{\mathrm{c}} n_1 \quad A_1[n_1] = val}{\Delta;\theta;\sigma;\textbf{Identificador[}\textit{valor}\textbf{]} \Rightarrow_{\mathrm{c}} val}$$

$$\frac{\Delta;\theta;\sigma;\textbf{Identificador}_1 \Rightarrow_{\mathrm{c}} D_1 \quad D_1(\textbf{Identificador}_2) = val}{\Delta;\theta;\sigma;\textbf{Identificador}_1\textbf{.Identificador}_2 \Rightarrow_{\mathrm{c}} val}$$

Para o acesso a arranjos, primeiro avaliamos o **Identificador** que retorna um valor de arranjo, A_1. Em seguida, avaliamos *valor* pegando o índice para a posição desejada resultando em um valor n_1. Finalmente, o valor val é obtido pelo valor presente na posição n_1 do arranjo A_1. Para avaliar o acesso a campo de tipos avalia-se o **Identificador**$_1$ e obtemos um valor de tipo D_1. Em seguida, obtemos o valor val usando como chave o nome do campo acessado **Identificador**$_2$. Observe que tipos são representados como tabelas formadas por pares chave/valor. Para realizarmos a execução de alguns comandos, devemos definir como realizar a execução de sua lista de argumentos. Sendo m o número de parâmetros de uma chamada de comando:

$$\Delta;\theta;\sigma;\textbf{Identificador()} \Rightarrow_{\mathrm{c}} \textbf{Identificador()}$$

$$\frac{\Delta;\theta;\sigma;\textit{valor}_i \Rightarrow_{\mathrm{c}} val_i \qquad 1 \leq i \leq m}{\Delta;\theta;\sigma;\textbf{Identificador(}\textit{valor}_1, ..., \textit{valor}_m\textbf{)} \Rightarrow_{\mathrm{c}} \textbf{Identificador(}val_1, ..., val_m\textbf{)}}$$

A primeira das regras anteriores mostra que uma sequência vazia de argumentos reduz para o próprio comando. A segunda regra mostra que sequências não vazias de argumentos são reduzidas por executarmos da esquerda para a direita e, em seguida, executamos a função "de fora". Utilizando-se desta regra para avaliar uma lista de argumentos, podemos definir a execução de comandos:

$$\frac{\begin{array}{c}\Delta;\theta;\sigma;valores \Rightarrow_{\mathrm{cs}} vals \\ \theta(\textbf{Identificador}) = (\textbf{Identificador}_{ret}, val_{ret})\ \textbf{Identificador}(params)\ def \\ \sigma' = \sigma \cup (\textbf{Identificador}_{ret}, val_{ret}) \cup \{(x_i, val_i) | 1 \leq i \leq |vals|\} \\ \Delta;\theta;\sigma';def \Rightarrow_{\mathrm{cs}} \sigma''\end{array}}{\Delta;\theta;\sigma;\textbf{Identificador(}valores\textbf{)} \Rightarrow_{\mathrm{c}} val_{ret}\sigma_2}$$

1. Primeiro, executamos os argumentos $valores$ da chamada retornando os valores $vals$;

2. Em seguida, obtemos a definição da função **Identificador** no ambiente θ;

3. Criamos o ambiente σ' que estende σ com os valores obtidos pela avaliação dos argumentos e com o valor de retorno do comando, caso exista. Note-se que aqui estamos definindo que o *Stack* onde roda o ambiente da linguagem está alocando um espaço para o tipo de retorno, para os tipos dos argumentos e preenchendo os espaços dos parâmetros;

4. O próximo passo envolve executar o corpo da função **Identificador**, def, usando o ambiente σ';

5. No fim da computação da função é retornado o valor de retorno para σ, podendo não ter sido retornado nada caso não haja um tipo de retorno na sua definição. Além disso, a memória σ é alterada para σ_2 caso seja um comando como *processoData*, *processoNew*, *processoSet* ou *processoReturn* que atribuem valores à memória.

Tendo sido a execução de algum dos comandos de atribuição mencionados acima, com exceção do *processoReturn*, sua estrutura pode ser descrita como:

$$\frac{\Delta; \theta; \sigma; \textit{valor} \Rightarrow_{\text{cs}} val}{\Delta; \theta; \sigma; \textbf{Identificador(Tipo IdentificadorNovo } \textit{valor}\textbf{)} \Rightarrow_{\text{cs}} \sigma[\textbf{IdentificadorNovo} \mapsto val]}$$

A execução de vários comandos em seguida é definido como a execução de um comando que pode ter alterado a memória de execução e esta é usada pelos próximos comandos que podem modificar novamente a memória:

$$\frac{\begin{array}{c}\Delta; \theta; \sigma; c \Rightarrow_{\text{cs}} \sigma_1 \\ \Delta; \theta; \sigma_1; cs \Rightarrow_{\text{cs}} \sigma'\end{array}}{\Delta; \theta; \sigma; c\ cs \Rightarrow_{\text{cs}} \sigma'}$$

A avaliação de comandos condicionais se dá por executar *valor* e, em seguida, executa-se o bloco correspondente, dependendo do resultado da avaliação. $isTrue(x)$ é uma função que retorna verdadeiro quando um valor é $! = 0$ e $isFalse(x)$ quando este é $== 0$. As regras abaixo definem a computação tanto de *processoIf* quanto de *processoElif*, por isso chamaremos de *processoCond* para se referir aos dois:

$$\frac{\begin{array}{c}\Delta; \theta; \sigma; \textit{valor} \Rightarrow_{\text{c}} val \\ isTrue(val) \\ \Delta; \theta; \sigma; cs_1 \Rightarrow_{\text{c}} \sigma'\end{array}}{\Delta; \theta; \sigma; \textit{processoCond}\textbf{(}\textit{valor}\textbf{)\{}cs_1\textbf{\}}cs_2 \Rightarrow_{\text{c}} \sigma'}$$

No caso de computar *valor* e este retornar um valor que seja verdadeiro o bloco correspondente do *processoIf* ou *processoElif* é interpretado. Caso contrário, pula-se o bloco para rodar os comandos imediatamente depois dele:

$$\frac{\Delta;\theta;\sigma;\textit{valor} \Rightarrow_{\mathrm{c}} val \quad isFalse(val) \quad \Delta;\theta;\sigma;cs_2 \Rightarrow_{\mathrm{c}} \sigma''}{\Delta;\theta;\sigma;\textit{processoCond}(\textit{valor})\{cs_1\}cs_2 \Rightarrow_{\mathrm{c}} \sigma''}$$

No caso em que tenha-se *processoElif*s alinhados e *valor* retornou verdadeiro, depois de se executar o bloco deve-se pular todos *processoElif*s depois deste bloco:

$$\frac{\Delta;\theta;\sigma;\textit{valor}_1 \Rightarrow_{\mathrm{c}} val \quad isTrue(val) \quad \Delta;\theta;\sigma;cs_1 \Rightarrow_{\mathrm{c}} \sigma' \quad \Delta;\theta;\sigma';cs_3 \Rightarrow_{\mathrm{c}} \sigma''}{\Delta;\theta;\sigma;\textit{processoCond}(\textit{valor}_1)\{cs_1\}\textbf{elif}(\textit{valor}_2)\{cs_2\}cs_3 \Rightarrow_{\mathrm{c}} \sigma''}$$

Ademais, tem-se dois comandos "especiais" na linguagem, **or** e **and**. Estes podem ser usados no lugar de *valor* em comandos condicionais e servem como portas lógicas AND e OR porém possuem capacidade de curto-circuitar a expressão. Por exemplo, numa expressão como $A\ AND\ B$ se A for falso sabe-se que a expressão vai retornar falso independentemente do resultado de B. E, usando-se numa expressão $A\ OR\ B$ se A for verdadeiro sabe-se que retornará verdadeiro independentemente do resultado de B também. Sendo assim o código abaixo mostra a utilidade deste mecanismo:

```
1    type(Vector2 f32 x f32 y)
2    main(){
3        new(Vector2* v NULL)
4        if(and(u32neq(addtou32(v) 0) f32gr(v[0].x 3))){
5            Vector2SetValue(v 5)
6        }
7    }
```

Listing 3.4 – Exemplo de uso do operador **and**

A condição para que o bloco do **if** rode é que o endereço de *v* seja diferente de *NULL* (esta checagem foi feita transformando o valor de um ponteiro em **u32** e checando se este é diferente de zero) e que o campo *x* do *Vector2* apontado por *v* seja maior que 3. Veja que a segunda checagem apenas faz sentido caso o valor de *v* não seja *NULL*, pois acessando este ponteiro quando o endereço for nulo se lerá uma memória incoerente. Por isso, a maneira "correta" de escrever seria:

```
1    if(u32neq(addtou32(v) 0)){
2        if(f32gr(v[0].x 3)){
3            Vector2SetValue(v 5)
4        }
```

```
5     }
```

Listing 3.5 – Forma extensa com a mesma lógica do código anterior

Porém, temos o inconveniente de abrir dois blocos de **if** para representar uma expressão booleana, o que amplia o código sem necessidade. Por isso tem-se o curto-circuito que fará com que no *if(and(u32neq(addtou32(v) 0) f32gr(v[0].x 3)))* apenas rode a segunda parte *f32gr(v[0].x 3)* quando o primeiro argumento for verdadeiro. Assim, mantém-se o comportamento esperado de uma expressão booleana, deixa o código compacto e melhora o desempenho de um programa que precisará interpretar menos sub-expressões para dar o mesmo resultado esperado. Suas regras são definidas a seguir:

$$\frac{\Delta; \theta; \sigma; \textit{valor}_1 \Rightarrow_{\mathrm{c}} val \quad isFalse(val)}{\Delta; \theta; \sigma; \textbf{and(}\textit{valor}_1 \ldots \textit{valor}_m\textbf{)} \Rightarrow_{\mathrm{c}} isFalse(val)}$$

$$\frac{\Delta; \theta; \sigma; \textit{valor}_1 \Rightarrow_{\mathrm{c}} val \quad isTrue(val)}{\Delta; \theta; \sigma; \textbf{or(}\textit{valor}_1 \ldots \textit{valor}_m\textbf{)} \Rightarrow_{\mathrm{c}} isTrue(val)}$$

3.2 Algoritmo verificador de acesso de arranjo

Finalmente, nesta seção descreve-se o algoritmo que fará a verificação estática, ou seja, durante a compilação, de violações de acesso de vetor. Porém, antes de entrarmos na lógica em si, foi necessário algumas adições ao processo de compilação descrito anteriormente.

Foi dito que ao terminar a compilação criamos um arquivo binário que contém a memória de dados, a memória do *Stack* (de começo totalmente vazia), o índice da instrução em que o programa começa (o *Entry Point*) e as instruções em si. Adicionou-se dados complementares a este arquivo para auxiliar o processo de análise de acesso a arranjos: sempre que um bloco novo de instruções começasse (sempre que abrisse um novo *if*, *elif* ou *while*), sempre que um destes blocos terminasse e sempre que um comando completo terminasse era adicionado a uma lista o índice dessa instrução e essas listas foram anexadas ao final do arquivo binário para que pudessem ser lidas no processo de análise.

Assim, o algoritmo verificador é executado, depois da compilação normal, sobre o arquivo binário gerado por esta. Logo, neste ponto do algoritmo perdeu-se toda a informação do código fonte em si, o algoritmo lê apenas o código intermediário e os contextos (Γ, Δ e Θ) para verificar os acessos de arranjo.

Resumidamente, o algoritmo remove toda parte de código que não é relevante, analisa e define todas instruções em que ele pode rodar em tempo de compilação, executa estes comandos

com auxílio de código que guarda e verifica informações sobre acesso de arranjo e, por fim, garante que todo acesso na parte inexecutável será feito através de uma função que fecha o programa ao detectar acesso de vetor inválido. O algoritmo de verificação então pode ser dividido em quatro fases:

1. Remoção de código intermediário inutilizado;
2. Análise da executabilidade de instruções;
3. Interpretação auxiliada da parte executável com verificação de acesso de vetores;
4. Garantia de acesso de vetor seguro em código inexecutável.

3.2.1 Remoção de comandos inutilizados

Previamente à remoção é necessário identificar as funções criadas pelo desenvolvedor através do *processoFonc*, o índice que indica o começo dessa função e o que indica sua última instrução. Durante a compilação são guardados estes dados necessários em Θ: se a função é feita pelo usuário e ambos índices mencionados. Através disso cria-se um vetor que contém os dados Θ destas funções e um booleano indicando se esta foi chamada ou não. Estes booleanos são inicializados com "falso".

Logo depois, faz-se uma passada pelo código intermediário, começando do *Entry Point* até o último comando do *processoMain*, ambos índices também conhecidos no contexto Θ. Durante essa iteração, se encontramos uma chamada de função, esta criada pelo usuário, marcamos esta como "chamada", no vetor mencionado no parágrafo acima, e fazemos uma passada pelo corpo desta função agora para saber se esta chama outra função. Caso chame, repetimos o mesmo processo anterior de forma recursiva na função encontrada. Este processo continua até chegarmos no fim do *processoMain*, marcando assim no vetor acima todas funções que foram chamadas no programa. É importante dizer que não necessariamente estas funções marcadas serão executadas durante a interpretação, pois uma função apenas chamada dentro de um *if* ou *elif* pode não ser executada se a condição de tais expressões forem falsas.

Feito isso, percorre-se o vetor das funções definidas pelo usuário retirando os blocos de instruções relativos às funções que não são chamadas. Tal operação de remoção demanda certas modificações:

1. Remoção do código intermediário das funções removidas no vetor de instruções do programa;
2. Modifica-se o contexto Θ retirando-se as informações destas funções removidas;

3. Atualização dos vetores que contêm marcações de começo de bloco, fim de bloco e fim de expressão já que certos índices deverão ser removidos e os que aparecem depois dos removidos devem ter seus valores atualizados pela mudança de posição;

4. Atualização de todas funções no Θ que começam/terminam depois de qualquer uma das funções removidas já que sua posição foi movida para cima no vetor de instruções;

5. E, por fim, atualização de todas instruções que chamem as funções que começam/terminam depois das removidas por terem seus índices alterados.

Algorithm 1 Remoção de código intermediário inutilizado

```
1:  procedure REMOCAOCODINUTILIZADO(VetorMarcacoes, VetorInstrucoes)
2:      VetorFoncs ← ∅
3:      for funcao ∈ Θ do
4:          if funcao foi definida pelo usuario then
5:              VetorFoncs ← VetorFoncs ∪ (funcao, false)
6:          end if
7:      end for
8:      for i ← entryPoint; i < ultimoComandoIdx; i ← i + 1 do
9:          Comando ← COMANDOEM(i)
10:         if Comando == ChamadaFuncao_k then
11:             MARCAEVERIFICAFUNCAO(VetorFoncs, funcao_k)
12:         end if
13:     end for
14:     for funcao ∈ VetorFoncs do
15:         if VetorFoncs[funcao].chamada == false then
16:             REMOVEINSTRUCOES(VetorInstrucoes, funcao)
17:             Θ ← Θ − {funcao}
18:             ATUALIZAMARCACOES(VetorMarcacoes, funcao)
19:             ATUALIZADEFINICOESDEFUNCOES(Θ, funcao)
20:             ATUALIZACHAMADASDEFUNCOES(VetorInstrucoes, funcao)
21:         end if
22:     end for
23: end procedure
```

O pseudo-algoritmo acima (3.2.1) demonstra toda esta lógica que foi descrita textualmente e logo abaixo (3.2.1) temos a definição da função recursiva usada internamente *MarcaEVerifica-Funcao*:

Algorithm 2 Funcao: marcaEVerificaFuncao

```
1: function MARCAEVERIFICAFUNCAO(VetorFoncs, funcao_k)
2:     VetorFoncs[funcao_k].chamada ← true
3:     comeco ← Θ[funcao_k].comecoInstrucao
4:     fim ← Θ[funcao_k].fimInstrucao
5:     for i ← comeco; i < fim; i ← i + 1 do
6:         Comando ← COMANDOEM(i)
7:         if Comando == ChamadaFuncao_j then
8:             MARCAEVERIFICAFUNCAO(VetorFoncs, funcao_j)
9:         end if
10:    end for
11: end function
```

3.2.2 Análise da executabilidade de instruções

Durante a descrição acima, vê-se que a solução não foi descrita totalmente, algumas partes são omitidas para, através da simplicidade, trazer mais atenção à ideia principal do algoritmo. Um exemplo é a linha $Comando \leftarrow comandoEm(i)$ em que na prática envolve pegar uma quantidade fixa de bytes dependendo do cabeçalho do comando lido para determinar qual é o comando atual e seus argumentos e para que se saiba o desvio necessário para encontrar o próximo comando. Tais detalhes da implementação não são de tanta importância porém nesta seção será necessário mencionar mais da parte mais "baixa" da solução.

Assim como na parte anterior, nesta é necessário a criação de uma estrutura auxiliar, um vetor de booleanos com a mesma quantidade de elementos que a quantidade de comandos no vetor de instruções. Estes booleanos representam se o $comando_i$ é executável em tempo de compilação ou não e todos começam com o valor "falso". Nota-se que este valor inicial não importa, poderia ser "verdadeiro"e não faria diferença no algoritmo.

Foi adotado uma abordagem de "sujeira" para este algoritmo que será melhor entendida conforme o algoritmo é explicado. O que define cada comando ser executável em tempo de compilação é:

- *MovSV*s são sempre executáveis pois preenchem um argumento, valor ou campo com uma constante;
- *JmpT*s são sempre executáveis também pois sempre vão a uma instrução de índice constante;
- *Ret*s são sempre executáveis por simplicidade já que seu efeito só seria inexecutável caso algum comando da função em que se está inserido seja inexecutável logo não é necessário calcular sua executabilidade. Lembre-se que não se pode rodar *Ret* na função *main*;
- *JmpS*s e *Jmp2S*s são executáveis se o endereço usado em seu argumento não estiver sujo;

- *MovSD*s, *MovSS*s, *MovAddS*s e *MovAccessingAddSS*s são executáveis se o endereço usado em seu segundo argumento não estiver sujo e caso haja informação do primeiro argumento é necessário que este não esteja sujo também;
- *AccessingAddMovSS*s são executáveis se ambos argumentos de endereço não estão sujos;
- *UpdateStkF*s e *Call*s são executáveis se a função é executável e se seus endereços de argumentos não estão sujos.

Dito isto, como saber se uma função primitiva, já implementada naturalmente no compilador, é executável ou não? Esta informação se localiza também no contexto Θ e foi manualmente definida durante a implementação da linguagem seguindo algumas regras:

- Se a função usa-se internamente de algoritmos randômicos ela não pode ser executada durante compilação já que muitos destes algoritmos utilizam o tempo do computador como semente ao algoritmo fazendo-se necessário que se rodem apenas em execução. Ex: *randSetSeed*, *randPercentage*, *randNumber*;
- Se a função executa operações sobre arquivos ela não é executada em compilação, pois os dados dentro de um arquivo podem ser diferentes no momento de compilação e execução. Ex: *fileOpen*, *fileSeek*, *fileTell*, *fileClose*, *fileRead*, *fileWrite*;
- Se a função retorna o tempo do computador. Ex: *getTime*;
- Se a função possui algum efeito colateral para fora do programa, podendo ser uma comunicação com algum driver externo, um periférico, uma placa de vídeo por exemplo ou simplesmente o console. Ex: *printIntern*.

Futuramente pode ser desejável que versões executáveis de tais funções sejam definidas para diferentes propósitos que um desenvolvedor possa ter. De qualquer forma, ainda sobra definir como uma função, criada pelo usuário, pode ser executável ou não. Foi-se adotado a simplificação de que caso algum comando dentro de um bloco de comandos seja inexecutável logo o bloco inteiro também é, sendo uma função nada mais que um bloco de comandos a regra também se aplica aqui. Logo após se verificar que uma função criada pelo usuário é inexecutável este dado é guardado no contexto Θ.

Isto é uma simplificação pois existem formas, bem mais complexas de se implementar, de se executar comandos executáveis dentro de um bloco e deixar apenas os comandos inexecutáveis para serem executados em tempo de execução porém para isso seria necessário um algoritmo não trivial de inserção de código intermediário.

Esta simplificação também será aplicada no momento que formos executar o código: ao encontrar-se um comando inexecutável para-se a execução em tempo de compilação e guarda-se

o estado atual do programa no arquivo binário. É importante revelar isto aqui pois o algoritmo para verificar executabilidade de instruções não precisava ser tão completo quanto será mostrado aqui, foi feito assim visando executar a análise até o fim do programa mas ao analisar algo "sujo" é possível parar a análise e passar para o próximo passo do algoritmo verificador de acesso de arranjo.

Por fim, como saber se um endereço (não) está sujo? De começo, todos endereços estão limpos, tanto os do *Stack* quanto da memória de dados, eles se sujam quando são usados como argumento de qualquer função inexecutável, ou quando são retornados de uma função inexecutável. Ademais, quando se passa um endereço de uma variável que representa um tipo composto é necessário verificar nos contextos Γ e Δ se algum dos campos dessa variável está sujo, caso esteja este endereço também está.

Por completude do algoritmo, foi-se implementado uma rotina que propaga a inexecutabilidade de endereços e funções para outros endereços a fim de que a análise pudesse ser feita pelo programa inteiro. Mas, como já foi dito, não seria necessário: após encontrar-se algo inexecutável é possível parar a execução da análise e ir para o próximo passo. Por conhecimento e curiosidade o algoritmo funciona, resumidamente, assim:

Quando uma função, primitiva ou não, é avaliada como inexecutável, seja pela sua natureza ou por algum argumento sujo, volta-se no código intermediário até o ponto em que seus argumentos foram setados e propaga a sujeira pelos endereços que foram usados. Caso um destes foi preenchido através de outra chamada de função a propagação se torna recursiva até que os endereços originários sejam sujos.

Para tal lógica é necessário usar uma estrutura auxiliar, desta vez um vetor que contém o endereço, um booleano pro seu tipo (se é endereço de *Stack* ou da memória de dados), um booleano representando se este endereço está sujo ou não e um valor que indica o nível de bloco em que este endereço foi setado. O valor de nível de bloco é apenas para saber quando que este dado de endereço deve ser retirado do vetor quando se sai de um bloco, já que ao sair os espaços de memória inicializados no escopo serão retirados do *Stack*.

Por fim, utilizando-se desse vetor e do algoritmo de propagação de sujeira, faz-se uma passagem do início ao fim sobre a lista de instruções. No começo, adiciona-se ao vetor os dados das variáveis globais do contexto Γ, tanto constantes quanto vetores globais. Sobre estes vetores globais é importante ressaltar que, contraintuitivamente, eles só podem ser inicializados imediatamente anterior ao *processoMain*, logo, eles não podem ser usados em outras funções diretamente. Isto foi feito pois ter uma variável global de vetor (este podendo ser sujo, diferente das constantes globais) sendo usada em qualquer função levaria a uma sujeira imensurável já que a qualquer momento que a função rodasse não se teria como identificar se o vetor estivesse sujo ou não inviabilizando, assim, o algoritmo.

Após inseridas as variáveis globais, obviamente como executáveis neste momento,

identifica-se a qual função a instrução atual se localiza, em *processoMain* ou em uma função criada pelo usuário, dados contidos em Θ. Para lidar com possíveis recursões, esta função a ser analisada é setada como executável para que, se for chamada novamente dentro de si, a análise possa continuar. Assim, é inserido no vetor os argumentos desta função, de novo dados contidos em Θ, estes como executáveis pois é uma suposição necessária ao algoritmo já que se não forem o algoritmo detectará durante a chamada de função fazendo com que ela não se execute.

Dentro da função, verifica-se instrução a instrução e a todo momento se verifica no vetor de marcações se iniciou-se um bloco de instruções, caso sim continuamos fazendo a mesma rotina, verificando instrução a instrução, porém no momento que encontrar um comando inexecutável roda-se uma rotina que suja o bloco inteiro e endereços de fora que foram utilizados dentro do bloco. Ao sair de um bloco, endereços que foram inicializados neste são retirados do vetor de endereços.

Durante a análise de instruções temos remoções de endereços também quando estes são utilizados em chamadas de funções, primitivas ou não. Lembre-se aqui que estes endereços removidos são apenas aqueles que são colocados no *Stack* unicamente para preencher os argumentos das funções, logo, variáveis mantidas no *Stack* definidas por *processoNew* não são removidas assim. Os pseudo-algoritmos abaixo (3.2.2) (3.2.2) representam a lógica que foi expressa textualmente:

Algorithm 3 Análise da executabilidade de instruções

```
1: procedure analiseExecInsts(instrucoesQnt, vetorEnderecos, vetorMarcacoes)
2:     for i ← 0; i < instrucoesQnt; i++ do
3:         vetorCmdExecutavel[ i ] ← false
4:     end for
5:     adicionarEnderecosGlobais(vetorEnderecos)
6:     i ← 0
7:     while i < instrucoesQnt; do
8:         funcao_i ← acharFuncaoEm(i)
9:         Θ[funcao_i].executavel ← true
10:        adicionarEnderecosDeParametros(vetorEnderecos, funcao_i)
11:        analisarFuncao(vetorCmdExecutavel, funcao_i, i, vetorMarcacoes)
12:    end while
13: end procedure
14:
15: function analisarFuncao(vetorCmdExecutavel, funcao_i, i, vetorMarcacoes)
16:    while i ≤ Θ[funcao_i].fimInstrucao do
17:        if i ∈ vetorMarcacoes.indicesComecoBloco then
18:            exec ← analisarBloco(vetorCmdExecutavel, i, vetorMarcacoes, 2)
19:                                                ▷ nivelBloco 0 é global, 1 é função
20:        end if
21:        exec ← analisarInstrucao(vetorCmdExecutavel, vetorEnderecos, i)
22:        if exec == false then
23:            Θ[funcao_i].executavel ← false
24:        end if
25:    end while
26:    removerEnderecosDeParametros(vetorEnderecos, funcao_i)
27: end function
28:
29: function analisarBloco(vetorCmdExecutavel, i, vetorMarcacoes, nivelBloco)
30:    iInicial ← i; blockExec ← true
31:    while true do
32:        if i ∈ vetorMarcacoes.indicesComecoBloco and i ≠ iInicial then
33:            exec ← analisarBloco(vetorCmdExecutavel, i, vetorMarcacoes,
   nivelBloco + 1)
34:        end if
35:        exec ← analisarInstrucao(vetorCmdExecutavel, vetorEnderecos, i)
36:        if exec == false then
37:            blockExec ← false
38:        end if
39:        if i ∈ vetorMarcacoes.indicesFimBloco then
40:            removerEnderecosBloco(vetorEnderecos, nivelBloco)
41:            if blockExec == false then
42:                sujarTodasInstsEEnderecosDoBloco(vetorEnderecos)
43:            end if
44:        end if
45:    end while
46: end function
```

Algorithm 4 Funcao: analisarInstrucao

```
1: function ANALISARINSTRUCAO(vetorCmdExecutavel, vetorEnderecos, i)
2:     exec ← true
3:     comando ← COMANDOEM(i)
4:     if comando == (MovSS or MovAddS or MovSD or MovAccessingAddSS) then
5:         exec ← EXECUTABILIDADE(comando.segundoArg)
6:         if exec == true and EHTIPOCOMPOSTO(comando.segundoArg) then
7:             exec ← EXECUTABILIDADEDOTIPOCOMPOSTO(comando.segundoArg)
8:         end if
9:         if EXECUTABILIDADE(comando.primeiroArg) == false then
10:            exec ← false
11:        end if
12:        NOVOENDERECO(vetorEnderecos, comando.primeiroArg, exec)
13:    else if comando == MovSV then
14:        NOVOENDERECO(vetorEnderecos, comando.primeiroArg, exec)
15:    else if comando == (Ret or Jmpt) then
16:    else if comando == AccessingAddMovSS then
17:        exec1 ← EXECUTABILIDADE(comando.primeiroArg)
18:        exec2 ← EXECUTABILIDADE(comando.segundoArg)
19:        exec ← exec1 and exec2
20:        if exec == false then
21:            if exec1 == true then
22:                PROPAGARSUJEIRA(vetorEnderecos, comando.primeiroArg)
23:            else if exec2 == true then
24:                PROPAGARSUJEIRA(vetorEnderecos, comando.segundoArg)
25:            end if
26:        end if
27:        REMOVERENDERECO(vetorEnderecos, comando.primeiroArg)
28:        REMOVERENDERECO(vetorEnderecos, comando.segundoArg)
29:    else if comando == (JmpS or Jmp2S) then
30:        exec ← EXECUTABILIDADE(comando.primeiroArg)
31:        REMOVERENDERECO(vetorEnderecos, comando.primeiroArg)
32:    else if comando == (Call or UpdateStkF) then
33:        exec ← EXECUTABILIDADEDAFUNCAOESEUSARGS(comando)
34:        REMOVERENDERECODOSARGS(vetorEnderecos, comando)
35:        if exec == false then
36:            PROPAGARSUJEIRAPELOSARGS(vetorEnderecos, comando)
37:        end if
38:    end if
39:    vetorCmdExecutavel[ i ] ← exec
40:    i++
41: end function
```

3.2.3 Interpretação auxiliada com verificação de acesso de vetores

Até este ponto, removeu-se todo o código intermediário inutilizado pelo programa e marcou-se todos comandos que podem ser executados durante compilação. Antes de rodarmos o

código executável, novamente, precisamos de uma estrutura auxiliar.

Para definir se um acesso de vetor é inválido é necessário conhecer o tamanho de um vetor, assim como foi mostrado em 2.1.1. Logo, um elemento dessa estrutura auxiliar contém dados referentes a um vetor no programa FonC: sua localização (endereço, se está no *Stack* ou não, e escopo) e seu tamanho. Ademais, cada elemento pode conter também a referência para uma lista de dados sobre outros vetores dentro do programa (será justificado mais tarde). A ideia por trás dessa estrutura é que ela sirva de representação sobre qual vetor um endereço no *Stack*, ou na memória de dados, ela se refira.

Desta forma, inicializa-se a estrutura auxiliar, que é um vetor, acessando o contexto Γ e adiciona-se elementos que correspondam às variáveis globais que foram inicializadas com o comando *array*, que pré-aloca memória durante a compilação e retorna um ponteiro com o endereço do seu início. Coloca-se assim no elemento o endereço desse vetor, que é constante por ser inicializado globalmente, seu tipo, um booleano que representa que se localiza na memória de dados, um nível de escopo 0 e seu tamanho, também constante, retirado de Γ.

Então, calcula-se o índice da primeira instrução inexecutável no programa a partir do *Entry Point*. Isso é alcançado percorrendo o vetor de booleanos resultante do passo anterior enquanto o valor iterado se manter em "verdadeiro", quando chegar em um "falso" este é o índice da primeira instrução inexecutável. Assim, o próximo passo é executar instruções, como se o programa estivesse rodando normalmente, porém propagando a informação dos vetores correspondentes e fazendo uma verificação de acesso de vetores quando houver um acesso.

A verificação é simples: sempre que for encontrado um comando *Call* que chama a função interna de aritmética de ponteiro faz-se uma busca na estrutura auxiliar para encontrar o vetor relacionado ao endereço do primeiro argumento, verifica-se o valor do argumento de índice e se esse for maior ou igual ao tamanho do vetor encontrado o programa fecha e mostra a mensagem de acesso de vetor inválido. Já a propagação de informação dos vetores ocorre de forma diferente para cada comando encontrado:

- Ao encontrar um *MovSD* ou *MovSS*, verifica-se se o endereço do segundo argumento é de uma variável de tipo composto. Caso seja para cada endereço relativo aos seus campos verifica-se se há informação contida na estrutura auxiliar que aponta aquele endereço a algum vetor e caso tenha propague a informação ao novo endereço relativo para onde a informação está sendo copiada. Caso este segundo argumento não seja um tipo composto verifica-se se há informação contida na estrutura auxiliar sobre seu próprio endereço e, caso haja, essa informação é copiada ao novo endereço. Além disso, itera-se também sobre endereços do segundo argumento até $segundoArgumento + terceiroArgumento$, sendo este terceiro argumento dos *Mov*s o tamanho do dado que está sendo copiado do segundo argumento ao primeiro argumento. Isto acontece pois pode haver informações de vetor no meio do dado a ser copiado quando se preenche um argumento através de partes que,

juntas, formam uma estrutura completa. Caso encontre-se uma informação de arranjo no meio do dado é necessário propagá-la ao correto endereço relativo ao primeiro argumento e remover o endereço, de onde se copiou o valor, da estrutura auxiliar;

- Ao encontrar um *Call* que chama a função interna de aritmética de ponteiro faz-se a verificação, como descrita no parágrafo anterior, e cria-se uma informação nova no vetor auxiliar sobre o ponteiro que é retornado. Esta informação é como uma propagação mas o tamanho do vetor retornado é $tamanho = tamanhoOriginal - valorIndice$. Isso pois ao se acessar um vetor em uma posição $valorIndice$ é como se movesse o ponteiro do vetor original para frente em $valorIndice$ posições;

- Ao encontrar um *Call* qualquer, incluindo o caso anterior, é necessário retirar da estrutura auxiliar a informação de ponteiros que foram utilizadas na chamada dessa função interna;

- Ao encontrar uma chamada de função criada pelo usuário, que é um *JmpT* seguido de *UpdateStkF*, também retira-se do vetor auxiliar ponteiros usados na chamada. Além disso, propaga-se estes valores removidos do escopo atual para o escopo de *workspace* da função e incrementa-se o valor do escopo atual. Este valor é um inteiro que indica o nível de profundidade do escopo atual e é usado durante a inserção de novos endereços, relacionados a vetores, à estrutura auxiliar. Este serve como um identificador único do escopo onde se está. Ele é necessário para que saibamos quais endereços remover da estrutura quando sairmos de uma função e para que, ao buscar um valor que esteja relacionado a um endereço X, não haja problemas de *overlapping* nos valores já que cada *workspace* de função começa em 0 e teria-se muitos endereços com mesmo valor;

- Ao encontrar um *Ret* verifica-se se a função do escopo onde se acabou de sair retorna um valor que possui informação de arranjo associado, sendo um tipo composto ou não, e, caso assim seja, propaga-se essa informação ao escopo para onde se retorna. Além disso, faz-se uma passada na estrutura auxiliar encontrando valores no nível do escopo atual e os remove do vetor. Decrementa-se o valor de escopo atual;

- Ao encontrar um *AccessingAddMovSS* retira-se informação de vetor da estrutura relacionada ao primeiro argumento;

- Ao encontrar um *MovAddS* cria-se informação na estrutura auxiliar, usando o endereço do primeiro argumento, sobre um novo vetor de tamanho 1, pois passar o endereço de uma variável é equivalente a isto. Porém, caso o endereço usado, aquele que está no segundo argumento, estiver relacionado a um vetor dentro da estrutura auxiliar uma lógica a mais é necessária e tem-se dois caminhos:

    1. O segundo argumento é o endereço de um vetor. Dessa forma, cria-se uma lista, de tamanho 1, com um tipo de dado que guarda apenas tamanho e uma referência a uma lista de dados de outros vetores. O valor do tamanho do único elemento dessa lista

será o tamanho do vetor ao qual está se criando um ponteiro a ele e o valor da lista não precisa ser setado. E aqui se justifica a referência de uma lista de dados sobre vetores que foi adicionada ao tipo de dado do vetor auxiliar: usará-se a referência, vazia no momento, do novo vetor recém-criado pelo *MovAddS* para apontar para essa lista que acabou-se de criar. A ideia de se ter uma referência a uma lista de dados de outros vetores é para momentos em que se usa o *MovAddS* e durante operações de acesso de vetor sobre acesso de vetor, consiga-se usar os dados do vetor interno para verificação de acesso;

2. O outro caminho é se o endereço é de um tipo composto. Dessa forma, cria-se uma lista do tamanho dos campos desse tipo que contém informações sobre algum vetor e preenche-se essa lista com o tamanho de respectivos vetores. Aqui, porém, usa-se um dado a mais nessa estrutura de lista: cada elemento estará associado ao seu respectivo desvio em relação ao início do seu tipo. Ou seja, se o tipo é (u32* x u32* y), o primeiro elemento possui desvio 0 e o segundo 4, pois lembre-se que ponteiros em FonC possuem 4 bytes. E, novamente, seta-se a referência contida nos dados do novo vetor criado por *MovAddS* a esta lista criada;

- Ao encontrar um *MovAccessingAddSS* seu segundo argumento deve estar no vetor auxiliar, porém verifica-se se possui a referência de lista dentro de si setada. Caso não tenha, simplesmente remove-se seu dado inteiro da estrutura auxiliar. Caso tenha uma referência em si, propaga-se estes dados ao endereço do primeiro argumento e seta-se uma flag que significa que no próximo *MovAccessingAddSS* o dado de vetor que será copiado ao dado atual no momento de acesso será este que está no primeiro lugar da lista referenciada. E aqui, caso o endereço do primeiro e segundo argumentos forem diferentes deleta-se o segundo argumento da estrutura auxiliar;

- Por fim, existe uma função interna, chamada "incremento de ponteiro", que é usada em casos como: arr[4].count, ou seja, incrementa-se ao ponteiro gerado pela aritmética de ponteiro um valor de desvio para que se acesse um campo do tipo composto de um ponteiro. Pois então, ao encontrar um *Call* da função interna de incremento de ponteiro, pega-se o dado sobre o vetor referente a este incremento através do endereço de seu segundo argumento e caso este tenha a sua referência de lista setada procura-se o elemento referente ao valor de desvio utilizado nesta chamada, coloca-o em primeiro desta lista e seta-se uma variável no tipo deste vetor para sinalizar que na próxima leitura de vetor (*MovAccessingAddSS*) é este dado de vetor que será copiado ao dado de vetor que estiver no momento do acesso.

Os pseudo-algoritmos abaixo (3.2.3) (3.2.3) representam a lógica que foi expressa textualmente:

Algorithm 5 Interpretação com verificação de acesso de vetores : Parte 1

```
1: procedure INTERPRETACAOCOMVERIFICACAOACESSOVETOR(vetorCmdExecutavel, comandoEntryPoint, comandos, comandosQnt)
2:     arraysInfo ← ∅
3:     nivelEscopo ← 0
4:     for var ∈ Γ do
5:         if EHVETORGLOBAL(var) then
6:             arraysInfo ← arraysInfo ∪ (Γ[var].endereco, Γ[var].tipoEndereco, 0, Γ[var].tamanho, NULL)
7:         end if
8:     end for
9:     novoEntryPoint ← comandoEntryPoint
10:    while novoEntryPoint < comandosQnt do
11:        if vetorCmdExecutavel[ novoEntryPoint ] == false then
12:            BREAK
13:        end if
14:        novoEntryPoint + +
15:    end while
16:    i ← comandoEntryPoint
17:    while i < novoEntryPoint do
18:        comando ← comandos[ i ]
19:        INTERPRETARINSTRUCAO(comando)
20:        if comando == (MovSD or MovSS) then
21:            for j ← 0; j < comando.terceiroArg; j + + do
22:                endFonte ← comando.segundoArg + j
23:                endDestinacao ← comando.primeiroArg + j
24:                if EHTIPOCOMPOSTO(comando, endFonte) then
25:                    COPIARARRAYINFOCAMPOACAMPO(endDestinacao, endFonte)
26:                else if ARRAYSINFOCONTEM(comando, endFonte) then
27:                    PROPAGARARRAYINFO(arraysInfo, endDestinacao, endFonte)
28:                end if
29:            end for
30:        else if comando == Call and comando.primeiroArg == aritmPtrID then
31:            ai ←ARRAYINFODESSEARGUMENTO(comando.segundoArg)
32:            indice ←VERIFICAACESSOVETOR(ai)
33:            ADICIONARARRAYINFO(arraysInfo, comando.segundoArg, ai.tamanho − indice)
34:        else if comando == Call and comando.primeiroArg == incremPtrID then
35:            ai ←ARRAYINFODESSEARGUMENTO(comando.segundoArg)
36:            if ai.list ≠ NULL then
37:                desvio ← LERARGDEDESVIO(comando)
38:                SELECIONARELEMENTOPARAPROXLEITURA(ai, desvio)
39:            end if
40:        else
41:            SEGUNDAPARTEDESDEALGORITMO(comandos, arraysInfo, comando, nivelEscopo, i)
42:        end if
43:    end while
44: end procedure
```

Algorithm 6 Interpretação com verificação de acesso de vetores : Parte 2

```
1: function segundaParteDesdeAlgoritmo(comandos, arraysInfo, comando, nivelEscopo, i)
2:     if comando == Call then
3:         retirarArrayInfoUsadosPorFuncao(arraysInfo, comando)
4:     end if
5:     if comando == Ret then
6:         if funcaoRetornaValorComArrayInfoAssociado(comando) then
7:             propagarRetArrayInfo(arraysInfo, comando)
8:         end if
9:         retirarArrayInfoCriadosNesteEscopo(arrayInfo, nivelEscopo)
10:        nivelEscopo − −
11:    end if
12:    if comando == JmpT and comandos[ i + 1 ] == UpdateStkF then
13:        retirarArrayInfoUsadosPorFuncao(arraysInfo, comando)
14:        propagarArrayInfoUsadosPorFuncaoParaNovoEscopo(arraysInfo, comando)
15:        nivelEscopo + +
16:    end if
17:    if comando == MovAddS then
18:        list ← NULL
19:        if ehTipoComposto(comando.segundoArg) then
20:            list ← criarListaTipoComposto(comando.segundoArg)
21:        else if arraysInfoContem(comando.segundoArg) then
22:            list ← criarListaTipoSimples(comando.segundoArg)
23:        end if
24:        adicionarArrayInfoTam1ComLista(arraysInfo, comando.segundoArg, list)
25:    end if
26:    if comando == AccessingAddMovSS then
27:        retirarArrayInfo(arraysInfo, comando.primeiroArg)
28:    end if
29:    if comando == MovAccessingAddSS then
30:        if ehArrayInfoComLista(arraysInfo, comando.segundoArg) then
31:            propagarArrayInfoMarcandoFlag(arraysInfo, comando.primeiroArg, comando.segundoArg)
32:            if comando.primeiroArg ≠ comando.segundoArg then
33:                retirarArrayInfo(arraysInfo, comando.segundoArg)
34:            end if
35:        else
36:            retirarArrayInfo(arraysInfo, comando.segundoArg)
37:        end if
38:    end if
39: end function
```

3.2.4 Garantir acesso de vetor seguro em código inexecutável

Depois que a análise garantiu que, na parte executável, não há violações de acesso de vetor, é necessário que garanta-se o acesso de vetor seguro em código que não foi executado.

Para isto, o desenvolvedor que usa *FonC* deve efetuar acessos de vetor com a função interna da linguagem *ig*, abreviação de *index Guarantee* (garantia de índice).

Esta função recebe três argumentos: o índice, o tamanho do vetor e um vetor de caracteres (*string*). Internamente o que ela faz é verificar se o índice é maior ou igual ao tamanho do vetor e, caso seja, fecha o programa mostrando a mensagem que foi colocada em seu terceiro argumento. Caso não seja, simplesmente retorna o índice para efetuar o acesso de vetor.

Assim, todo acesso de vetor, seja em *processoMain* ou em qualquer função criada pelo usuário, que ocorre depois da parte executável deve, obrigatoriamente, usar essa função:

```
1    fonc(void setIdx1 u8* buf){
2        set(u8 buf[1] 67)
3    }
4    fonc(void setIdx2 u8* buf){
5        // Caso retire o ig() o compilador solta um erro
6        set(u8 buf[ig(2 10 "accessViolation\0")] 68)
7    }
8    data(u8* buf array(10))
9    main(){
10       set(u8 buf[0] 66) // Parte executavel nao precisa de ig()
11       setIdx1(buf) // Parte executavel nao precisa de ig()
12       printIntern(buf 2) // Parte inexecutavel, daqui para frente deve
     ser usado ig()
13       setIdx2(buf)
14       printIntern(buf 3)
15   }
```

Listing 3.6 – Exemplo de acessos de arranjo válidos com *ig*

Um fato importante é que o *ig* deve ser chamado imediatamente após a abertura de colchetes. Não é considerado um *ig* que retorna para uma variável e esta ser usada no acesso de vetor, nem que é retornado através de uma chamada de função criada pelo usuário:

```
1    fonc(void setIdx2 u8* buf){
2        new(u32 idx ig(2 10 "accessViolation\0"))
3        set(u8 buf[idx] 68) // Erro
4    }
```

Listing 3.7 – Uso indevido de *ig* em variável

```
1    fonc(u32 safeIdx){
2        ret(ig(2 10 "accessViolation\0"))
3    }
4    fonc(void setIdx2 u8* buf){
5        set(u8 buf[safeIdx()] 68) // Erro
6    }
```

Listing 3.8 – Uso indevido de *ig* em função

Para se garantir que o usuário usará *ig* sempre que necessário usa-se a estrutura auxiliar, criada no primeiro passo de remoção de código inutilizado, em que tem-se os dados Θ das funções criadas pelo usuário e o booleano indicando se esta foi chamada ou não. Inicializa-se todos os booleanos como "falso" novamente e efetua-se uma passada pelos comandos a partir do primeiro comando inexecutável, o $novoEntryPoint$ calculado no passo anterior, até o último dos comandos.

Durante as iterações, procura-se chamadas de funções criadas pelo usuário, recursivamente assim como no primeiro passo, até marcar todas funções que são chamadas no código inexecutável. Então, itera-se sobre a estrutura auxiliar garantindo que todo acesso de vetor nestas funções usa-se o *ig* e, por fim, verifica-se se todo acesso de vetor no código inexecutável em *processoMain* também se usa de *ig*. Os pseudo-algoritmos abaixo (3.2.4) (3.2.4) representam a lógica que foi expressa textualmente:

Algorithm 7 Garantia de acesso de vetor seguro em código inexecutável

procedure GARANTIA DE ACESSO DE VETOR SEGURO EM CÓDIGO INEXECUTÁVEL($VetorFoncs, novoEntryPoint, comandos, comandosQnt$)
  for $v \in VetorFoncs$ **do**
    $v.chamada \leftarrow false$
  end for
  for $i \leftarrow novoEntryPoint; i < comandosQnt; i++$ **do**
    if $comandos[\, i \,] == ChamadaFuncao_k$ **then**
      MARCAEVERIFICAFUNCAO($VetorFoncs, funcao_k$)
    end if
  end for
  for $v \in VetorFoncs$ **do**
    if $v.chamada$ **then**
      $comandoInicio \leftarrow$ COMANDOEM($v.\Theta.comecoInstrucao$)
      $comandoFim \leftarrow$ COMANDOEM($v.\Theta.fimInstrucao$)
      VERIFICAIGEMACESSOSVETOR($comandos, comandoInicio, comandoFim$)
    end if
  end for
  VERIFICAIGEMACESSOSVETOR($novoEntryPoint, comandosQnt - 1$)
end procedure

Algorithm 8 Funcao : verificaIGEmAcessosVetor

function VERIFICAIGEMACESSOSVETOR($comandos, comandoInicio, comandoFim$)
  while $comandoInicio < comandoFim$ **do**
    $comando \leftarrow comandos[\, comandoInicio \,]$
    if $comando == Call\ and\ comando.primeiroArg == aritmPtrID$ **then**
      $usa \leftarrow$ ESSACHAMADAUSAIG($comando$)
    end if
    $comandoInicio++$
  end while
end function

4 Resultados

Construído o compilador e o interpretador desta linguagem, na linguagem C, procurou-se validar a análise estática através da escrita de algoritmos que se utilizam de acesso de arranjo. Os exemplos que serão apresentados nesta seção procuram explorar todas as possíveis formas de se acessar arranjos na linguagem FonC.

4.1 Exemplos de acesso inválidos detectados

4.1.1 Acesso com índice constante

```
1    data(u8* buf array(10))
2    main(){
3        set(u8 buf[10] 66)
4    }
```

Listing 4.1 – Acesso inválido de arranjo com índice constante

O exemplo acima (4.1) mostra o tipo de acesso de vetor mais comum e básico que existe. Declara-se um vetor *buf* através da função interna *array*, que pré-aloca memória no campo de dados e constantes e retorna um ponteiro pro início deste espaço. Logo após, no *processoMain*, procura-se alterar o valor da posição 10, uma posição inválida pois o vetor possui apenas 10 posições sendo a 9 a última. A mensagem que o compilador retorna, que é a mensagem que será retornada para todo exemplo mostrado nesta seção, é a seguinte:

```
Invalid pointer access detected, compile interpretation canceled
```

Listing 4.2 – Mensagem de erro, acesso de arranjo inválido

4.1.2 Acesso com constante NULL

```
1    main(){
2        set(u8 NULL[10] 66)
3    }
```

Listing 4.3 – Uso indevido da constante **NULL**

Outro exemplo (4.3), tão básico quanto o anterior porém extremamente raro, é o uso indevido da constante **NULL** presente na linguagem. Por ser uma constante, com o valor de endereço 0, e possuir um tipo de ponteiro, o acesso de vetor aplicado a ele está semanticamente correto, sem a extensão da análise semântica que desenvolveu-se neste projeto. Porém, internamente, é feito com que o tamanho de seu vetor seja igual a 0 logo qualquer tentativa de acesso sobre ele, ou sobre variáveis que herdaram seu valor, causa erro.

4.1.3 Acesso com constante NULL, verificação extra *ig*

```
main(){
    set(u8 NULL[ig(4 10 "acessoInvalido\0")] 66)
}
```

Listing 4.4 – Exemplo sobre a verificação extra de *ig*

Falando mais sobre a constante **NULL**, a função *ig* também faz uma verificação extra antes mesmo de verificar o valor de índice usado sobre o tamanho do vetor passado como argumento. O *ig* verifica se o ponteiro usado, no momento do acesso de arranjo, possui o valor 0. Isso pode parecer um código que vai rodar apenas no caso acima (4.4) mas é um caso que pode ser muito comum ao se programar, não só na linguagem FonC.

É comum desenvolvedores inicializarem um valor de referência como NULL, tanto para terem um comportamento especial nesse caso ou um valor inicial de referência não setada. E por engano esse valor pode ser usado durante acesso de vetores e pode causar um acesso indevido mesmo com valor de índice correto. Portanto, a função *ig* detecta casos assim, em tempo de execução, e fecha o programa apresentando a mensagem passada para seu terceiro argumento junto da mensagem *NULL pointer exception*. Ao rodar o programa acima é esta a saída que se vê:

```
NULL pointer exception: acessoInvalido
```

É importante lembrar que, sendo *ig* uma função executável, esta exceção acima se retorna ainda em tempo de compilação já que usa-se o *ig* antes da parte inexecutável do programa. Mesmo se não usasse o *ig* neste caso o algoritmo estático de compilação já detectaria o erro sobre o ponteiro nulo (no exemplo anterior: 4.3).

4.1.4 Acesso com índice variável

```
data(u8* buf array(2))
main(){
    new(u32 index 0)
    set(u8 buf[index] 66)
    set(u32 index u32add(index 1))
    set(u8 buf[index] 66)
    set(u32 index u32add(index 1))
    set(u8 buf[index] 66) // Acesso invalido
}
```

Listing 4.5 – Exemplo de violação em acesso de arranjo com variável

Mais um caso comum, uso de variáveis para acesso de arranjo. Percebe-se como, por ter 2 elementos, após o índice ter seu valor incrementado duas vezes um acesso inválido foi detectado, já que *index* teria então um valor de 2.

4.1.5 Acesso com índice de retorno de função

```
fonc(u32 multBy2 u32 x){
    ret(u32mul(x 2))
}
data(u8* buf array(3))
main(){
    new(u32 index 2)
    set(u8 buf[index] 67)
    set(u8 buf[multBy2(index)] 67) // Acesso invalido
}
```

Listing 4.6 – Exemplo de violação em acesso de arranjo com índice de retorno de função

Um caso menos comum mas possível, usar-se de uma função para gerar, ou filtrar, um índice que será usado para acessar o arranjo. Também ocorre quando se usa de uma função interna:

```
data(u8* buf array(3))
main(){
    new(i32 index -2)
    set(u8 buf[i32tou32(index)] 67) // Acesso invalido
    // Lembre-se que -2 em u32 eh o mesmo que 4294967293
}
```

Listing 4.7 – Exemplo de violação em acesso de arranjo com índice de retorno de função interna

4.1.6 Acessos anteriores com propagação de informação de arranjo

```
fonc(u32 returnIndex){
    ret(2)
}
data(u8* buf array(4))
main(){
    new(u8* buf2 buf)
    new(u32 index 1)
    set(u8 buf2[0] 65)
    set(u8 buf2[index] 65)
    set(u8 buf2[returnIndex()] 65)
    set(u8 buf2[i32tou32(3)] 65)
    set(u8 buf2[4] 65) // Acesso invalido
}
```

Listing 4.8 – Casos de acesso de arranjo anteriores com propagação de informação de arranjo

Este exemplo apresenta a justificativa de se ter um algoritmo de propagação de informação de arranjo. Criou-se a variável *buf2* e a colocou com o mesmo valor que *buf*, fazendo com que se aponte ao mesmo endereço, ou seja, ao mesmo arranjo. É fato que, se modificarmos a constante

na linha 8, o *index* na linha 7, o retorno de *returnIndex* ou a constante dentro de *i32tou32* na linha 11 para qualquer valor acima de 3 acontecerá um acesso inválido de arranjo, assim como esperado. Esta propagação, como dito anteriormente nessa monografia, também ocorre em tipos compostos:

```
1  type(Arr u8* array u32 count u32 size)
2  fonc(u32 returnIndex){
3      ret(2)
4  }
5  data(u8* buf array(4))
6  main(){
7      new(Arr u8Buf (buf 0 4))
8      new(u32 index 1)
9      set(u8 u8Buf.array[0] 65)
10     set(u8 u8Buf.array[index] 65)
11     set(u8 u8Buf.array[returnIndex()] 65)
12     set(u8 u8Buf.array[i32tou32(3)] 65)
13     set(u8 u8Buf.array[4] 65) // Acesso invalido
14 }
```

Listing 4.9 – Propagação de informação de arranjo em tipo composto

E esta propagação também ocorre de tipos compostos a outros tipos compostos:

```
1  type(Arr u8* array u32 count u32 size)
2  fonc(u32 returnIndex){
3      ret(2)
4  }
5  data(u8* buf array(4))
6  main(){
7      new(Arr u8Buf (buf 0 4))
8      new(Arr u8Buf2 u8Buf)
9      new(u32 index 1)
10     set(u8 u8Buf2.array[0] 65)
11     set(u8 u8Buf2.array[index] 65)
12     set(u8 u8Buf2.array[returnIndex()] 65)
13     set(u8 u8Buf2.array[i32tou32(3)] 65)
14     set(u8 u8Buf2.array[4] 65) // Acesso invalido
15 }
```

Listing 4.10 – Propagação de informação de arranjos entre tipos compostos

4.1.7 Acessos consecutivos

```
1  data(u8* buf array(12))
2  main(){
3      new(u8* ptr &buf[6])
4      new(u8* ptr2 &ptr[4])
```

```
5         new(u8* ptr3 &ptr2[1])
6         new(u8* ptr4 &ptr2[2]) // Acesso invalido
7     }
```

Listing 4.11 – Exemplo de acessos consecutivos em arranjo

buf possui 12 elementos, *ptr* ao acessar *buf* na posição 6 aponta para um sub-arranjo de 6 elementos, $novoTamanho = tamanhoOriginal - indice$. Portanto, *ptr2* aponta para um sub-arranjo com $6 - 4 = 2$ elementos. *ptr3* aponta para um arranjo de 1 elemento enquanto que *ptr4* passou dos limites do arranjo original.

4.1.8 Acesso interno em função

```
1     fonc(void receivingPointer u8* ptr){
2         new(u8 x ptr[1])
3     }
4     fonc(void receivingPointer2 u8* ptr){
5         new(u8 x ptr[3])
6     }
7     data(u8* buf array(3))
8     main(){
9         receivingPointer(buf)
10        receivingPointer2(buf) // Acesso invalido
11    }
```

Listing 4.12 – Exemplo de acesso de arranjo internamente em funções

Talvez o caso de propagação de informação de arranjo mais frequente durante o desenvolvimento. Também acontece usando-se de variáveis ou em qualquer variação já apresentada aqui porém dentro de funções:

```
1     fonc(void receivingPointer3 u8* ptr u32 idx){
2         new(u8 x ptr[idx])
3     }
4     data(u8* buf array(3))
5     main(){
6         receivingPointer3(buf 0)
7         receivingPointer3(buf 3) // Acesso invalido
8     }
```

Listing 4.13 – Exemplo de acesso de arranjo internamente em funções com variável

A solução também é flexível para os casos em que se envia um tipo composto com algum campo de ponteiro:

```
1     type(Arr u32 count u8* arr u32 size)
2     fonc(void receivingType Arr cB){
3         set(u8 cB.arr[3] 67) // Acesso invalido
4     }
```

```
5     data(u8* buf array(3))
6     main(){
7         new(Arr charBuffer (0 buf 3))
8         receivingType(charBuffer)
9     }
```

Listing 4.14 – Exemplo de propagação de informação de arranjo campo a campo

4.1.9 Acesso sobre acesso de arranjo

```
1     data(u8* buf array(3))
2     main(){
3         new(u8* ptr buf)
4         new(u8** ptr1 &ptr)
5         new(u8*** ptr2 &ptr1)
6         new(u8**** ptr3 &ptr2)
7         set(u8 ptr3[0][0][0][3] 3) // Acesso invalido
8     }
```

Listing 4.15 – Exemplo de acesso sobre acesso de arranjo

Esse é um dos casos que justifica a lista, de informação sobre um arranjo interno, dentro de um dado sobre a informação de um arranjo externo. É possível acessar arranjo de arranjo de arranjo, *ad infinitum*, por conta disto.

4.1.10 Acesso sobre acesso em função

```
1     type(Arr u32 count u8* arr u32 size)
2     fonc(void receivingPointerType Arr* cB){
3         set(u8 cB[0].arr[3] 67) // Acesso invalido
4     }
5     data(u8* buf array(3))
6     main(){
7         new(Arr charBuffer (0 buf 3))
8         receivingPointerType(&charBuffer)
9     }
```

Listing 4.16 – Exemplo de acesso sobre acesso internamente em função

Juntando-se ambos os casos anteriores, de acesso sobre acesso e acesso internamente em função, tem-se este que é extremamente comum durante o desenvolvimento quando se envia endereços de tipos compostos a argumentos de funções. Além disso, o exemplo a seguir demonstra a importância dos dados contidos dentro da lista, esta que está dentro de um dado de arranjo na implementação, serem selecionados através do desvio de endereço em relação à cabeça de um tipo composto, já que o exemplo a seguir possui um tipo que pode conter mais de um arranjo dentro de si:

```
type(Arr2 u32 count u8* arr u8* arr2)
fonc(void receivingPointerType2 Arr2* cB){
    set(u8 cB[0].arr[2] 67)
    set(u8 cB[0].arr2[1] 68)
    set(u8 cB[0].arr[3] 67) // Acesso invalido
    set(u8 cB[0].arr2[2] 68) // Acesso invalido
}
data(u8* buf array(3))
data(u8* buf2 array(2))
main(){
    new(Arr2 charBuffer (0 buf buf2))
    receivingPointerType2(&charBuffer)
}
```

Listing 4.17 – Propagação de informação de mais de um arranjo no mesmo tipo composto

4.1.11 Acessos com ponteiro retornado de função

Assim como na seção sobre a verificação extra do *ig* (4.1.3), nesta será apresentado o resultado de verificações estáticas do algoritmo não sobre índices mas sobre o arranjo em que usa para fazer o acesso. Isso é necessário pois geralmente em linguagens de programação é possível acessar áreas de memória inválidas, normalmente fechando o programa assim, através de índices corretos porém com um valor de arranjo incorreto, como, por exemplo, acesso à constante *NULL* ou ponteiros com valor nulo, ou endereços em escopos inferiores de variáveis que eram de escopos superiores e, por isso, não existem mais.

```
fonc(u8* retPtr){
    ret(NULL)
}
main(){
    new(u8* ptr retPtr())
    set(u8 ptr[0] 3) // Acesso invalido
}
```

Listing 4.18 – Tentativa de acesso à constante NULL através de retorno de função

O caso mais básico é demonstrado acima em que *ptr* possui um valor de arranjo inválido que aponta para o endereço 0, da constante *NULL*, que apenas serve semanticamente ao programa e não direciona a um endereço válido. Abaixo é apresentado casos mais comuns, normalmente visto em inicializações de variáveis e tipos compostos.

```
fonc(u8* retPtr u8* arr){
    ret(&arr[1])
}
data(u8* buf array(3))
main(){
    new(u8* ptr retPtr(buf))
```

```
7         set(u8 ptr[2] 3) // Acesso invalido
8     }
```

Listing 4.19 – Acesso inválido de arranjo através de arranjo retornado de função

```
1     type(Arr u32 count u8* arr)
2     fonc(Arr ArrInit u8* arr2){
3         ret((0 &arr2[3]))
4     }
5     data(u8* buf array(7))
6     main(){
7         new(Arr abc ArrInit(&buf[3]))
8         set(u8 abc.arr[1] 67) // Acesso invalido
9     }
```

Listing 4.20 – Acesso inválido de arranjo através de tipo com arranjo retornado de função

4.2 Arranjos inválidos

Finalmente, tem-se o último tipo de erro semântico sobre arranjos, bem diferente dos anteriores. Observe o código a seguir:

```
1     fonc(u32* retPtrLocalVar u32 x){
2         new(u32 localVar x)
3         ret(&localVar)
4     }
5     main(){
6         new(u32* x retPtrLocalVar(5)) // x possuiria um endereco com um
    valor imprevisivel
7     }
```

Listing 4.21 – Exemplo de propagação de arranjo inválido

Ele possui um problema sério, é retornado o endereço de uma variável que, ao se sair do escopo da função, será "deletada" e o endereço então apontará para um dado imprevisível. Foi-se adicionado à logica do comando *MovSS*, durante a fase 3 do algoritmo (3.2.3), para que em casos que se esteja setando o dado de retorno de uma função verifique-se se esta possui algum dado de endereço sobre uma variável que existe apenas no escopo atual e, caso assim seja, o programa retorna o seguinte erro:

```
    Returning invalid pointer
```

Assim sendo, tanto o código apresentado acima quanto os próximos não passam pela fase de compilação, sendo apresentado ao usuário a mensagem de erro respectiva. O exemplo a seguir demonstra que o algoritmo suporta propagação de informação de arranjo e, o próximo deste, mostra que o algoritmo suporta sobrescrita de informação de arranjo também:

```
fonc(u32* retPtrLocalVar u32 x){
    new(u32* ptr &x)
    ret(ptr) // Ponteiro invalido
}
main(){
    new(u32* x retPtrLocalVar(5))
}
```

Listing 4.22 – Propagação de arranjo inválido

```
fonc(u8* retPtrLocalVar u8* x){
    new(u8 y 0)
    set(u8* x &y)
    ret(x) // Ponteiro invalido
}
main(){
    new(u8* x retPtrLocalVar(NULL))
}
```

Listing 4.23 – Sobreescrita de arranjo inválido

Uma observação importante é que, assim como apresentados na última sessão, em 4.19 e 4.20, casos em que se usa de acesso de arranjo e então retorna-se uma nova informação de arranjo são totalmente razoáveis e corretos, pois não retornam endereços locais.

4.3 Comparação com Trabalhos Relacionados

Foi apresentado no capítulo de Trabalhos Relacionados (2.2), exemplos de programas que tratam os casos de acessos de arranjo. Mostrou-se que vários possuem problemas como:

- Falsos positivos e falsos negativos;

  Efetivamente, o algoritmo apresentado não possui falsos positivos e falsos negativos em casos comuns pois consegue-se verificar todos tipos de acesso de vetor possíveis na linguagem em código executável. Ademais, caso um código não seja executável é necessário a adição da função *ig* que, por definição, roda em tempo de execução e faz-se uma verificação de acesso de arranjo que apenas falha caso o tamanho do vetor inserido pelo programador esteja errado. Logo, o algoritmo apenas falha quando o problema é causado pelo próprio desenvolvedor, o que é impossível de se remediar totalmente;

- Deixam o código lento;

  O algoritmo apresentado não torna o código mais lento significativamente. O tempo de execução de *ig* é aproximadamente de 2 condicionais, ou seja, extremamente rápido;

- Precisam de um trabalho significativo para configurar ou muita escrita para funcionarem;

Pode-se dizer que este algoritmo possui este problema pois para cada acesso de arranjo em área inexecutável é necessário a adição de *ig* o que pode causar a escrita extensiva desta função;

- Não detectam todos erros;

  Este algoritmo pode detectar todos erros com o uso correto da função *ig* pelo desenvolvedor;

- Algoritmos que levam muito tempo para executar ou que precisam de muitos recursos computacionais;

  O tempo de compilação, por definição, será o tempo de execução mais o tempo de se verificar os acessos, sendo este muito rápido por ser escrito em C. A única forma de levar muito tempo seria ter um tempo de execução, sobre a parte executável, extremamente grande;

- Não conseguem lidar com acessos de arranjos dentro de *loops*.

  Nosso algoritmo lida com estes casos.

4.4 Repositório do Projeto

Tanto o código da implementação do compilador, interpretador e algoritmo verificador de acesso de arranjos estão localizados no repositório *FoncEnv*, no *GitHub* ((Compiler/Interpreter. . . , 2025)). Atualmente, o projeto possui 4511 linhas de código em C.

5 Conclusão

Neste trabalho, assim como explicitado nos Objetivos (1.2), documentou-se a linguagem FonC, sua semântica estática e operacional, sua máquina virtual e contexto inicial e o algoritmo verificador de acesso de arranjo. Ademais, foi apresentado todos os casos de acesso de arranjo que o algoritmo detecta, além de detectar arranjos inválidos.

Logo, este trabalho possui importância acadêmica já que mostrou com sucesso uma solução para o problema de acesso de arranjo. Todos tipos de acesso de arranjo possíveis são garantidos na parte executável e os inexecutáveis são garantidos através da função *ig* que apenas falha por erro humano, este intratável.

5.1 Trabalhos Futuros

Para possíveis trabalhos futuros sobre o algoritmo verificador, seria interessante executar o algoritmo de executabilidade de instruções pensando em rodar todas partes executáveis mesmo que apareçam depois de uma inexecutável. Isto seria possível através de um algoritmo que aloca espaços de memória durante compilação para guardar o(s) estado(s) executáveis em certas áreas da memória de dados e alocar espaço os resultados inexecutáveis que serão gerados durante execução do código.

Referências

AHO RAVI SETHI, J. D. U. A. V. **Compiladores. Princípios E Técnicas**. 1. ed. [S.l.]: LTC, 1995. ISBN 8521610572,9788521610571.

AYEWAH, N. *et al.* Using static analysis to find bugs. **IEEE Software**, v. 25, n. 5, p. 22–29, 2008.

CHECKMARX Static Application Security Testing. 2024. <https://checkmarx.com/cxsast-source-code-scanning/>. Accessed: 30/01/2024.

CHIMDYALWAR, B. Survey of array out of bound access checkers for c code. In: **Proceedings of the 5th India Software Engineering Conference**. New York, NY, USA: Association for Computing Machinery, 2012. (ISEC '12), p. 45–48. ISBN 9781450311427. Disponível em: <https://doi.org/10.1145/2134254.2134262>.

COMPILER/INTERPRETER for FonC Programming Language. 2025. <https://github.com/BernardoSoD/FoncEnv/tree/master>. Accessed: 12/08/2025.

FORTIFY Static Code Analyzer. 2024. <https://www.microfocus.com/pt-br/cyberres/application-security/static-code-analyzer>. Accessed: 30/01/2024.

GAO, F. *et al.* Static checking of array index out-of-bounds defects in c programs based on taint analysis. **International Journal of Software and Informatics**, v. 11, p. 121–147, 01 2021.

ISO. ISO, **Information technology - Programming languages - C**. 2011.

KELLOGG, M. *et al.* Lightweight verification of array indexing. In: **Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 2018. p. 3–14.

MARJAMAKI, D. **Cppcheck - A tool for static C/C++ code analysis**. 2024. <https://cppcheck.sourceforge.io/>. Accessed: 30/01/2024.

NETHERCOTE, N.; SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: . [S.l.: s.n.], 2007. v. 42, p. 89–100.

NGUYEN, T. V. N.; IRIGOIN, F. Efficient and effective array bound checking. **ACM Trans. Program. Lang. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 27, n. 3, p. 527–570, may 2005. ISSN 0164-0925. Disponível em: <https://doi.org/10.1145/1065887.1065893>.

RITCHIE, D.; KERNIGHAN, B. **C, a Linguagem de Programação: padrão ANSI**. [S.l.]: Editora Campus, 1989. ISBN 8570015860.

SEREBRYANY, K. *et al.* Addresssanitizer: a fast address sanity checker. In: . [S.l.: s.n.], 2012. p. 28–28.

THE Z3 Theorem Prover. 2024. <https://github.com/Z3Prover/z3>. Accessed: 30/01/2024.

VENET, A.; BRAT, G. Precise and efficient static array bound checking for large embedded c programs. In: **Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation**. New York, NY, USA: Association for Computing Machinery, 2004. (PLDI '04), p. 231–242. ISBN 1581138075. Disponível em: <https://doi.org/10.1145/996841.996869>.

ÖZKAN, S. **CVE Details**. 2023. <https://www.cvedetails.com/vulnerabilities-by-types.php>. Accessed: 21/10/2023.