Inferência de Tipos Dependentes em C

Marcus Rodrigues de Araújo
Universidade Federal de Minas Gerais
Belo Horizonte, MG - Brasil
maroar@dcc.ufmg.br

Leandro Terra Cunha Melo
Universidade Federal de Minas Gerais
Belo Horizonte, MG - Brasil
ltcmelo@dcc.ufmg.br

Fernando Magno Quintão Pereira
Universidade Federal de Minas Gerais
Belo Horizonte, MG - Brasil
fernando@dcc.ufmg.br

ABSTRACT

Tipos dependentes associam tipos a valores de forma a contribuir para a construção de programas mais seguros e eficientes. Este trabalho apresenta uma técnica para inferir tipos dependentes em C. A inferência de tipos usa uma Análise de Intervalos, que aproxima o conjunto de inteiros que cada variável pode assumir durante a execução do programa. Tal análise, juntamente com informações de tipos, nos habilita a anotar declarações em C com tipagem dependente. Nossa técnica permite que programadores usufruam das vantagens da tipagem dependente, sem que a linguagem C tenha que ser alterada. Além disso, desenvolvedores não precisam dominar a teoria de tipos dependentes para usar nosso método. Uma implementação de nossa técnica foi capaz de anotar com tipos dependentes a maioria dos arranjos e funções usadas nos 30 kernels de Polybench, um benchmark bem conhecido.

KEYWORDS

tipos dependentes, análise de intervalos, análise estática, linguagem C

1 INTRODUÇÃO

Tipos dependentes [2, 12] são tipos cuja definição depende de um valor. Além de fornecer um maior controle sobre os dados utilizados em um programa e uma melhor documentação, o uso desses tipos trás outras vantagens. Xi *et al.*, por exemplo, mostram como essa restrição de tipos pode ser usada para a realização de otimizações como: eliminação de verificação dos limites de acessos a vetores [20] e remoção de código morto [17]. O mesmo autor também afirma que tipos dependentes possibilitam que programadores capturem mais erros em tempo de compilação e obtenham propriedades de programas com maior acurácia [18].

Essa forma de atribuir tipos a entidades de um programa restringe ainda mais o universo de valores que variáveis podem assumir. Isso acontece através da adição, ou indexação, de valores às especificações de tipos das variáveis. Um exemplo clássico de emprego dessa tecnologia pode ser visto em situações em que se tem uma lista de N inteiros. Veja que neste caso o tipo da lista não é "lista de inteiros", mas "lista de N inteiros". Uma função que lê o primeiro elemento de uma lista assim pode apresentar, em sua especificação, uma restrição para não aceitar listas vazias. Dessa forma, a possibilidade de indexar incorretamente a lista é evitada já durante o processo de compilação de programas.

Tipos dependentes são utilizados em algumas linguagens de programação, como Idris [3], Twelf [16], ATS [4], Coq [5] e Epigram [9]. Ainda assim, em vista dos benefícios que oferecem, acreditamos que a noção de tipos dependentes é muito pouco explorada. Essa omissão deve-se, em grande parte, à dificuldade de se programar com essa tecnologia. Sendo mais restritivo, esse conceito proíbe a escrita de muitos programas que seriam válidos em linguagens que não apresentam tipos dependentes. É nossa visão, contudo, que tipos dependentes são úteis, mesmo em programação industrial, e que eles podem ser incorporados automaticamente a programas já prontos, sem a intervenção direta dos programadores.

As linguagens que atualmente usam tipos dependentes são essencialmente declarativas. Entretanto, Xi afirma que linguagens imperativas também podem se beneficiar dessa maneira de se atribuir tipos [18]. Partindo-se de tal pressuposto, este trabalho dá mais um passo na direção de trazer vantagens oferecidas por tipos dependentes à linguagens imperativas originalmente projetadas sem esse conceito. Com esse propósito, a seção 4 apresenta uma ideia de como esses tipos podem ser inferidos na linguagem C (C99). Essa nova técnica pode ser vista como uma forma de adicionar informação aos tipos de variáveis e funções existentes no código fonte. Entre vantagens que tipos dependentes podem trazer para essa linguagem, destacam-se: melhorar a geração de *Library Bindings* [8] entre linguagens de programação; apontar erros causados por acessos de posições de memórias fora dos limites de arranjos; oferecer suporte para criação de corpos falsos para funções (*stubs*) em teste de software.

Dada a natureza das aplicações mencionadas acima, é importante que a inferência de tipos dependentes seja feita automaticamente. Especificamente, queremos ser capazes de enriquecer os tipos das variáveis usadas no programa e tornar suas definições mais expressivas, sem exigir esforços por parte do programador. Neste artigo é mostrado como isso pode ser feito usando informações contidas no próprio código fonte do programa alvo.

Para definir os valores que serão associados aos tipos, na seção 3 mostramos como usar uma Análise de Intervalos (*Range Analysis*) para inferir limites para variáveis inteiras e regiões de memória. Esse tipo de análise já é bem conhecida na computação [1, 8, 10, 13], contudo, o uso que fazemos dela é original. Neste trabalho, ela é usada para enriquecer os tipos utilizados no código fonte C. É importante ressaltar que a Análise de Intervalos é realizada estaticamente, não exigindo que o programa seja executado, e, no caso deste trabalho, dá-se sobre a Árvore de Sintaxe Abstrata do código. Esse último fato, inclusive, distingue nossa contribuição de anteriores, em que a análise de intervalos é feita sobre a representação intermediária do programa alvo.

Para validar as ideias apresentadas, foi implementado um inferidor de tipos dependentes sobre o parser utilizado na IDE Qt Creator[1]. Nossa ferramenta anota declarações de funções, arranjos e variáveis inteiras com uma forma restrita de tipos dependentes, seção 4. Nossa análise foi capaz de associar corretamente expressões simbólicas à maioria dos arranjos analisados nos trinta programas disponíveis em *Polybench*[2]. Essas anotações são congruentes com o que se espera de um sistema de tipos: uma declaração é anotada somente se todos os usos do elemento declarado são consistentes com o tipo inferido. Tal experimento, junto com a ideia de aumentar C com anotações de tipos dependentes, constituem as contribuições deste trabalho, abaixo resumidas:

- **Tipos**: a seção 4 introduz uma extensão para o sistema de tipos da linguagem C que passa a considerar uma forma restrita de tipos dependentes;
- **Inferência**: partindo das ideias discutidas neste artigo, a seção 4.4 apresenta `grif-c`, uma ferramenta capaz de incrementar código em linguagem C com anotações baseadas em tipos dependentes;
- **Validação**: experimentos realizados com `grif-c` sustentam a validade das ideias discutidas, seção 5.

2 VISÃO GERAL

Em uma linguagem de programação que não possui tipos, como cálculo lambda não tipado, um programador tem a possibilidade de realizar qualquer forma de operação com qualquer valor. Essa liberdade, contudo, vem com um preço alto. Usuários de tal linguagem não podem contar com mecanismos estáticos para prevenção de falhas. Isso implica que uma grande quantidade de programas que irão gerar erros em tempo de execução são aceitos pela linguagem. Além disso, linguagens de programação que não possuem um sistema de tipos perdem a possibilidade de usar as regras de tipagem como forma de documentação do código [11]. Uma linguagem de programação que possui tipos deve respeitar restrições em relação ao uso de expressões como variáveis e constantes. Essas regras determinam a forma de operar sobre tais valores. Isso acaba por restringir o número de programas que são aceitos pela linguagem. No entanto, essa passa a contar com as vantagens descritas anteriormente.

Tipos dependentes [2, 12] são uma extensão dos sistemas de tipos simples. Nesse caso, os tipos são indexados por valores que geram **famílias de tipos**. Essa técnica restringe a quantidade de programas que são aceitos por determinada linguagem, eliminando, assim, a possibilidade de que diversas formas de erro aconteçam.

No presente trabalho é apresentada uma nova técnica de geração de tipos dependentes para linguagens que não os possuem nativamente. Em especial focamos na linguagem de programação C. Acreditamos que essas ideias são mais importantes para linguagens que não fornecem mecanismos de introspecção. Em Java, por exemplo, é possível obter o tamanho de um arranjo através do comando “`arranjo.length`”.

Nesta seção é mostrado um exemplo de como a inferência de tipos é feita para um trecho de programa. A demonstração é realizada por meio do código visto na figura 1a). Nela, temos o código de uma função, `foo`, que atribui um valor, 0, para determinada posição de um arranjo, a qual é definida a partir do resultado da avaliação de uma condição. O predicado que controla tal atribuição é regido pelo valor da variável `n`, que também é passada para a função como parâmetro.

```
void foo(int n,
         int *v)
{
  int i;
  if (n < 10) {
    i = 0;

  } else {
    i = 10;


  }

  v[i] = 0;
}
a)
```

```
1  void foo(int n,
2           int *v)
3  {
4    int i;
5    if (n < 10) {
6      i = 0;
7  // R(i) = [0, 0]
8    } else {
9      i = 10;
10 // R(i) = [10, 10]
11   }
12 // R(i) = [0, 10]
13   v[i] = 0;
14 }
15 b)
```

Figura 1: a) Definição da função `foo`. b) Análise de Intervalos para `foo`. *R* é um mapa que associa símbolos aos seus intervalos.

Na seção 3 descreveremos uma *Análise de Intervalos* que busca aproximar os intervalos de valores que podem ser atribuídos às variáveis do programa. Ao ser submetido para a Análise de Intervalos da seção 3, o código da figura 1a) retorna os intervalos mostrados na figura 1b), considerando os resultados para o símbolo `i`. As informações utilizadas ao

[1]Obtido em https://www.qt.io/ide/

[2]Obtido em http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

longo da análise surgem através das atribuições realizadas nas linhas 6, 9 e 13. É interessante notar que o resultado mostrado na linha 12 é obtido através da união dos valores possíveis para `i`, considerando que cada um dos possíveis desvios pode ser tomado (linhas 6 e 9).

O resultado da Análise de Intervalos nos leva a algumas conclusões quanto aos valores das variáveis usadas em `foo`. Por exemplo, como mostrado na figura 1b), o intervalo de possíveis valores assumidos por `i` varia de 0 a 10. Note que essa análise não infere, necessariamente, todos os números que podem ser atribuídos a uma variável, mas um limite inferior e superior para eles. Durante a análise, não é possível definir qual dos desvios do comando `if` (linha 5) será tomado. Assim, é necessário considerar todas as possibilidades. Essa abordagem conservadora leva ao resultado encontrado para o intervalo da variável `i` na linha 12.

A Análise de Intervalos nos permite, também, tirar conclusões em relação à área de memória apontada por `v`. No código considerado não existem informações explícitas relativas ao número de posições que podem ser acessadas por essa variável. No entanto, considerando que o programa analisado não contém comportamentos indefinidos, `v` deve apontar para uma sequência de posições em memória de tamanho mínimo 11. Isso pode ser inferido uma vez que o arranjo é acessado por uma variável cujo limite superior é 10. Em geral, considera-se que um array deve ter tamanho mínimo igual ao maior limite superior das variáveis que são usadas para acessá-lo. Dessa forma, como `v` é indexado somente por `i` (linha 13), conclui-se que, em `foo`, a variável aponta para um arranjo de no mínimo 11 posições.

O objetivo deste trabalho é enriquecer código fonte em C com informações mais detalhadas sobre o universo de valores usados no programa. Para tal, são inferidos tipos específicos para cada variável. Os tipos construídos são análogos há alguns dos que são apresentados por Xi [18]. A inferência é feita considerando-se o resultado obtido pela Análise de Intervalos, mais os tipos das variáveis e funções presentes no código. A partir das assinaturas que inferimos estaticamente, queremos ser capazes de associar arranjos a símbolos relacionando-os aos seus tamanhos. Queremos também apontar quando uma variável numérica pode ser vista como um valor constante ou intervalar. Neste último caso, queremos também ser capazes de especificar os possíveis limites dos valores assumidos pelas variáveis.

Um exemplo de uso para os tipos inferidos é mostrado na figura 2 que consiste no código da figura 1 adicionado de comentários contendo os tipos dependentes criados. Note que os tipos gerados fazem uso dos construtores *Const* e *Vector* que serão apropriadamente definidos na seção 4.

A figura 2 mostra não apenas uma aplicação de tipos dependentes, ela é um resultado concreto da ferramenta desenvolvida neste trabalho. Nesse caso, as anotações de tipos

```
// void foo(n:Const int n,
//          v:Vector int [11])
void foo(int n,
         int *v)
{
  int i;// i:Range int 0 10
  if (n < 10) {
    i = 0;
  } else {
    i = 10;
  }
  v[i] = 0;
}
```

Figura 2: Anotações baseadas em tipos dependentes geradas para a função foo.

dependentes foram inseridas para fins de documentação de código, utilizando a própria sintaxe de comentários oferecido pela linguagem C. Tais informações também poderiam ser incorporadas em ferramentas de análise estática para validar acessos a memória ou para enriquecer a geração casos de testes (através da criação de *stubs* mais significativos).

3 ANÁLISE DE INTERVALOS

A seção 4 explica como inferimos tipos dependentes para arranjos e variáveis inteiras de programas C. Um componente chave nesse processo de inferência é a **Análise de Intervalos**. A Análise de Intervalos [1, 8, 10, 13] é uma técnica capaz de encontrar os limites que variáveis numéricas podem assumir em diferentes pontos de um dado programa. Essa análise nos permite ter uma ideia melhor sobre a dimensão do universo de valores numéricos que são utilizados no código fonte. Além disso, ela é usada para diversos fins, vide seção 6. Neste trabalho, usamos os resultados dessa análise para a criação dos tipos dependentes.

As implementações da Análise de Intervalo encontradas na literatura [1, 8, 10, 13] funcionam com base no código intermediário gerado pelo LLVM [7]. Em geral, elas são implementadas como passes na etapa de otimização do *backend* dos compiladores. Diferente dessas, o algoritmo utilizado neste trabalho é implementado sobre a Árvore de Sintaxe Abstrata (AST : *Abstract Syntax Tree*) do programa. Até onde sabemos, esse é o primeiro trabalho a apresentar uma Análise de Intervalos simbólica com base na AST do código fonte.

Nossa análise percorre a AST e recolhe informações relacionadas aos usos de variáveis. A coleta de dados é feita com base nas regras da figura 3. Nesse esquema é possível verificar os tipos de declarações usadas para obter, restringir e derivar informação. Seja R um mapa, $R : V \mapsto E^2$, que associa váriáveis, V, a intervalos simbólicos, E^2 (definido

$$\mathbf{rg}(R, \mathsf{skip}) = R \qquad \frac{R' = R \setminus v \mapsto [n,n]}{\mathbf{rg}(R, v = n) = R'} \qquad \frac{R' = R \setminus v \mapsto [s,s]}{\mathbf{rg}(R, v = s) = R'} \qquad \frac{R(v_1) = [l,u] \qquad R' = R \setminus v \mapsto [l,u]}{\mathbf{rg}(R, v = v_1) = R'}$$

$$\frac{R(v_1) = [l_1,u_1] \qquad R(v_2) = [l_2,u_2] \qquad R' = R \setminus v \mapsto ([l_1 + l_2, u_1 + u_2])}{\mathbf{rg}(R, v = v_1 + v_2) = R'} \qquad \frac{\mathbf{rg}(R,S_1) = R_1 \qquad \mathbf{rg}(R_1,S_2) = R_2}{\mathbf{rg}(R, S_1; S_2) = R_2}$$

$$\frac{\begin{array}{c} R(v_a) = [l_a,u_a] \qquad R(v_b) = [l_b,u_b] \qquad R_t = (R \setminus v_a \to [l_a, \min(u_b - 1, u_a)]) \setminus v_b \to [\max(l_a + 1, l_b), u_b] \\ \mathbf{rg}(R_t,S_t) = R'_t \qquad R_f = (R \setminus v_a \to [\max(l_a,l_b), u_a]) \setminus v_b \to [l_b, \min(u_a,u_b)] \qquad \mathbf{rg}(R_f,S_f) = R'_f \qquad R' = R'_t \sqcup R'_f \end{array}}{\mathbf{rg}(R, \mathsf{if}(v_a < v_b)\ S_t \ \mathsf{else}\ S_f) = R'}$$

$$\frac{\mathbf{fp}(R, \mathsf{if}(v_a < v_b)\ S \ \mathsf{else\ skip}) = R'}{\mathbf{rg}(R, \mathsf{while}(v_a < v_b)\ S) = R'} \qquad \frac{\mathbf{rg}(R,S) = R}{\mathbf{fp}(R,S) = R} \qquad \frac{\mathbf{rg}(R,S) = R_1 \qquad R_1 \neq R \qquad \mathbf{fp}(R_1,S) = R'}{\mathbf{fp}(R,S) = R'}$$

Figura 3: Regras para coleta de informação da Análise de Intervalos Simbólica.

mais adiante), e S um comando do programa fonte. Sendo assim, a execução da análise é feita com o auxílio da relação $\mathbf{rg}(R,S)$ que usa as informações contidas em R e S para produzir novas associações entre variáveis e intervalos. Esses são representados por intervalos fechados, $[l,u]$, com um limite inferior, l, e outro superior, u.

A obtenção de informação dentro de laços é feita com o auxílio de técnicas de alargamento (*widening*) e estreitamento (*narrowing*). Inicialmente o corpo do comando de repetição é percorrido e todas as variáveis incrementadas ou decrementadas terão seus valores expandidos para valores extremos, ∞ ou $-\infty$ respectivamente. A checagem da mudança do conteúdo das variáveis é feita considerando os valores que elas possuíam antes e depois do laço. Em seguida, é realizada mais uma travessia no corpo do laço para estreitar os valores previamente alargados. É importante mencionar que, quando o algoritmo se depara com comandos de repetição aninhados, os intervalos relativos a laços mais externos são resolvidos primeiro. Isso faz com que a análise tenha que tomar decisões conservadoras em relação aos limites encontrados. Em troca, ao analisar um laço mais interno, podemos assumir que o contexto no qual ele está inserido já está resolvido.

Ao fim do algoritmo de Análise de Intervalos tem-se como saída uma associação de pontos de programa com os intervalos de cada uma das variáveis até aquele ponto. Os limites inferiores e superiores das variáveis são apresentados por meio de **expressões simbólicas**. Símbolos de programas são nomes encontrados no código fonte que não podem ser escritos em função de outros nomes. No contexto deste trabalho, consideramos um símbolo como qualquer nome que não tenha associado a ele um intervalo. Uma expressão simbólica é definida de acordo com a gramática abaixo, onde s é um símbolo e $n \in \mathbb{N}$:

$$\begin{array}{rcl} E & ::= & n \mid s \mid \min(E,E) \mid \max(E,E) \mid E - E \\ & & \mid E + E \mid E/E \mid E \bmod E \mid E \times E \\ & & \mid -\infty \mid +\infty \end{array}$$

Expressões simbólicas formam um semi-reticulado, cuja ordem parcial é dada pela relação $<$ entre símbolos. De modo geral essa ordenação pode ser vista como: $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots < +\infty$. É importante destacar que não existe uma ordem específica entre expressões que envolvem símbolos diferentes. Partindo dessas definições, nosso reticulado é definido como: $\mathcal{R} = \langle E^2, \sqsupseteq, \sqcap, \sqcup, \bot = \emptyset, \top = [-\infty, \infty] \rangle$, onde a ordem parcial é definida como $l_1 \leq l_2 \wedge u_1 \geq u_2 \Rightarrow [l_1,u_1] \sqsupseteq [l_2,u_2]$. O operador *meet* é visto como a interseção dos intervalos: $[l_1,u_1] \sqcap [l_2,u_2] = [\max(l_1,l_2), \min(u_1,u_2)]$; e o operador *join* como a união: $[l_1,u_1] \sqcup [l_2,u_2] = [\min(l_1,l_2), \max(u_1,u_2)]$.

Exemplo 3.1. *A figura 4 mostra um programa que possui dois laços aninhados. A figura 5 mostra o resultado produzido pela Análise de Intervalos sobre esse programa. Note que a mesma variável pode estar associada a intervalos diferentes, dependendo da região do programa. Por exemplo, a variável* `k` *possui o intervalo* $[0,0]$ *antes do laço mais externo,* $[0,99]$ *dentro daquele laço e* $[0,100]$ *depois dele.*

4 GERAÇÃO DE TIPOS DEPENDENTES

Construtores de tipos podem ser vistos como funções que recebem tipos como parâmetro e os usam para produzir outros tipos. Um exemplo de construtor de tipos é `Vector<T>`, que cria vetores polimórficos. Caso alimentemos este construtor com, por exemplo, o tipo `String`, teremos um tipo `Vector<String>`. É possível expandir esse conceito, e adicionar aos construtores de tipos a capacidade de receberem também valores. Assim, um construtor como `Vector<n:Int,T>` cria vetores com `n` instâncias de um certo tipo `T`. Dizemos que cada possível valor de `n` indexa uma família de tipos. Essa família de tipos é **dependente** de `n`.

Assim, é possível criar um construtor de tipos que associa arranjos a números naturais, que indicam o seu tamanho. Neste caso, a especificação do construtor que descreve tal

```
void foo() {
  int i, j, k = 0;
  while (k < 100) {
    i = 0;
    j = k - 1;
    while (i < j) {
      i++;
      j--;
    }
    k++;
  }
  print(k);
}
```

Figura 4: Programa passado como entrada para a Análise de Intervalos.

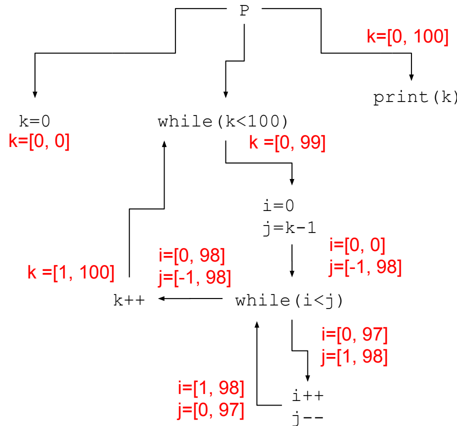

Figura 5: Resultado da Análise de Intervalos para o código da figura 4.

conjunto pode ser escrita como: $Vector :: Nat \to * \to *$, ou seja, tal gerador recebe um valor, de tipo Nat; um tipo, $*$; e produz um novo tipo, $*$. O número natural usado na definição anterior é o indexador que define uma família de tipos conforme mencionado acima (arranjos com valores do tipo T e de tamanho n). Tipos dependentes são tipos que usam valores em suas definições.

Um exemplo de função que gera arranjos de um dado tipo e que são limitados por um certo valor pode ter sua assinatura definida como `init` : $\Pi n : Nat \,.\, T \to Vector\ n\ T$. Ou seja, `init` recebe como parâmetros um número natural, `n`, e um tipo, `T`, para produzir um arranjo de `n` elementos de tipo `T`. O símbolo Π é análogo ao quantificador universal, $\forall$, de *System F* [11]. Porém, a quantificação de Π pode ocorrer sobre valores, ao invés de apenas sobre tipos. Tipos Π (Π-*types*) representam o produto de tipos, através da notação $\Pi x : S.T$. Essa notação indica que, para todo possível valor x, de tipo S, existe uma família de tipos T, que possivelmente faz uso de x em sua definição.

De forma análoga é possível definir os conjuntos de tipos dependentes que fazem uso do quantificador existencial $\exists$. Neste caso, os conjuntos definidos por esse quantificador são chamados de tipos Σ (Σ-*types*). Essas especificações podem ser vistas como os tipos dos pares dependentes. Em um par dependente, o **tipo** do segundo elemento da tupla depende do **valor** do primeiro elemento.

4.1 Assinaturas de Tipos Dependentes

Neste trabalho definimos novas assinaturas para as declarações de funções e variáveis em C. Tais assinaturas seguem a sintaxe defina na figura 6. Nessa gramática `B` representa os tipos *built-in* da linguagem e `T` mostra como gerar os novos tipos propostos. É possível ver que todos os tipos criados por meio da regra `T` possuem indexadores de famílias, representados por expressões simbólicas (`E`). Eles são construídos a partir de tipos presentes originalmente na linguagem e informações obtidas com a Análise de Intervalos.

$$
\begin{array}{lcll}
B & ::= & \textit{Any valid C type} & \\
T & ::= & \textit{Const } B\ E & \\
 & | & \textit{Range } B\ E_1\ E_2 & ;(E_1 < E_2) \\
 & | & \textit{Vector } B\ [E] & ;(E_i > 0)
\end{array}
$$

Figura 6: Tipos gerados nesse trabalho.

A seguir é mostrado como esses tipos dependentes são gerados. Em especial, são consideradas duas situações onde pretende-se reconstruí-los: para descrever os intervalos de valores assumidos por variáveis numéricas e para limitar o tamanho de arranjos. Além disso, é mostrado como essa forma mais restrita de tipos é usada para gerar anotações que documentam o código fonte. É importante lembrar que os tipos são atribuídos às variáveis considerando o escopo no qual elas estão sendo usadas.

4.2 Tipos Π

Um dos focos deste trabalho é identificar e encontrar limites para arranjos. Na linguagem C essa estrutura de dados consiste em uma sequência de uma ou mais posições de memória que armazenam algo do mesmo tipo. Neste artigo o tipo `Vector B [E]` é utilizado para caracterizar valores como esses quando seus limites são conhecidos. Uma vez que os tipos Π representam o produto de tipos, os tipos construídos a partir de `Vector` são tidos como os seus representantes. Tais tipos são parametrizados por um tipo base, `B`, e por uma lista de expressões simbólicas, `[E]`, com os limites de cada dimensão do arranjo considerado.

O tipo `Vector B [E]` é gerado para variáveis que são explicitamente declaradas como arranjos (`int a[10]`) ou para aquelas com tipo ponteiro (`int *a`) que são identificadas como vetores e têm um limite bem definido pela Análise de Intervalos. No primeiro caso, a inferência de tipos dependentes é trivial, pois os termos passados para o construtor de tipos têm valores conhecidos em tempo de compilação. No segundo caso, contudo, esses termos não estão representados explicitamente na sintaxe do programa. Faz-se necessário um processo de análise do escopo ao qual a variável pertence.

Para determinar se uma variável de tipo ponteiro será classificada como um candidato ao tipo `Vector B [E]` é feita uma análise de como essa variável é utilizada. Neste trabalho, um ponteiro é classificado como um candidato se o programa alvo contém sintaxe de indexação como "`v[E]`", em que `v` tem tipo ponteiro e `E` possui tipo numérico. No caso anterior, `v` é classificado como possível arranjo dependente e `E` como um parâmetro para o tipo produzido para `v`. É importante salientar que situações envolvendo outras aritméticas de ponteiros, como "`*(v + E)`", não são consideradas aqui. Isso não leva a resultados errados apesar de poder diminuir a quantidade de tipos `Vector B [E]` inferidos. Tal omissão é somente uma simplificação de nossa implementação, e será tratada posteriormente.

Uma vez que uma variável com tipo ponteiro tenha sido classificada como candidata ao tipo `Vector`, resta verificar se seus limites são bem definidos. Isso é feito através da verificação dos intervalos das expressões utilizadas para acessar (através da sintaxe de indexação) tal arranjo. Considerando que o código não contenha erros de acessos inválidos a memória, a ideia é a seguinte: em um dado escopo, atribuímos o tamanho de um vetor como sendo o maior limite superior das variáveis usadas para acessá-lo. Ressalta-se que a análise apresentada é conservadora, ou seja, os tamanhos inferidos para cada arranjo nunca são maiores que o tamanho real.

Exemplo 4.1. *Considere o caso em que uma variável* `a` *de tipo* `int*` *é acessada por meio de* `i`*, a[i], e* `j`*, a[j]. Além disso, considere que nesse ponto do programa o intervalo das variáveis i e j eram, respectivamente,* $[0, 5]$ *e* $[0, 9]$*. Se no código considerado estes são os únicos acessos à variável* `a`*, o tipo inferido para ele nesse escopo será* `Vector int [10]`.

4.3 Tipos Σ

Os tipos Σ são usados para especificar os intervalos de variáveis numéricas presentes no código. Em outras palavras, essa técnica é usada para descrever, quando possível, um limite inferior e superior para determinada variável. De modo geral, o que é feito é associar o resultado encontrado pela Análise de Intervalos com os tipos das variáveis. Na gramática mostrada anteriormente, esses tipos são representados como `Range B` E_1 E_2. Note que o tipo de `Range` depende de dois termos: um limite inferior E_1, e outro superior, E_2.

Exemplo 4.2. *Se a Análise de Intervalos conclui que uma dada variável* `x` *tem um intervalo* $r = [0, n]$ *e tipo* `int`*, o tipo final construído para essa variável será: "*`Range int 0 n`*". Veja que neste caso o tipo inferido para x depende de um valor inteiro* 0 *e de um símbolo* n*.*

Quando a Análise de Intervalos é inconclusiva ou pouco específica (produzindo, por exemplo $[-\infty, +\infty]$), não são geradas anotações para as variáveis. Esse fato acontece inclusive considerando o tipo `Vector`. Outro caso especial acontece quando alguma variável permanece inalterada em um determinado escopo. Nesses casos, os limites inferiores e superiores serão os mesmos. Isso indica que tal variável é constante naquele ambiente e é representada pelo tipo `Const B E`. Na ausência de especificadores que indicam esse fato, a anotação de tipos dependentes para essa variável diz ao programador que em dado ponto do programa o *qualificador* `const` (note a letra `c` inicial não-capitalizada) de C pode ser utilizado. Note que, neste caso, o construtor `Const` é parametrizado por um tipo e um termo (expressão simbólica).

Exemplo 4.3. *Se um parâmetro de função, e.x.* `n`*, permanece inalterado no corpo daquela função, o seguinte tipo é gerado para a variável* `n`*:* `Const int n`*, supondo que* `n` *seja inteiro.*

4.4 Anotador de código C

As ideias discutidas até aqui serviram de base para a construção de `grif-c`. Essa ferramenta é um anotador de código para a linguagem C. Detalhes sobre o seu funcionamento podem ser vistos na figura 7.

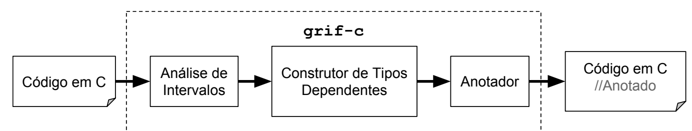

Figura 7: Esquema da ferramenta desenvolvida.

Essa ferramenta recebe como entrada um código em linguagem de programação C. Em seguida faz o *parsing* desse programa e gera a sua Árvore de Sintaxe Abstrata. Essa estrutura de dados é usada como entrada para a Análise de Intervalos. Em seguida, as informações de tipos e da análise anterior são unidas para gerar os tipos dependentes considerados neste trabalho. Por fim, um anotador de código recebe como entrada os tipos gerados e os insere em uma cópia do código fonte original. No fim do processo, tem-se como saída a cópia do código fonte original acrescido de informações sobre alguns dos tipos na forma de comentários.

5 RESULTADOS EXPERIMENTAIS

A fim de avaliar as ideias apresentadas neste trabalho, foram realizados experimentos com a ferramenta `grif-c`, seção 4.4. O objetivo deste experimento é verificar quantas assinaturas exatas nossa ferramenta é capaz de inferir para programas complexos. Com tal propósito, `grif-c` foi aplicada sobre todos os programas disponíveis em *Polybench*. Esse *benchmark* contém um conjunto de programas utilizados em álgebra linear, mineração de dados, processamento de imagens, entre outras áreas da matemática. Para realizar os experimentos de `grif-c`, foram utilizadas as 30 funções núcleo (*kernel*) dos programas encontrados nessa base de dados. O principal motivo para a escolha dessa coleção de métodos é que todos eles recebem, como parâmetro, arranjos e inteiros, estes relacionados aos tamanhos daqueles. Nesse experimento, sucesso é contabilizado toda vez que a ferramenta é capaz de gerar uma assinatura correta para o cabeçalho do núcleo de entrada, como mostrado no exemplo da figura 2. Mais especificamente, a metodologia usada para a realização dos testes consiste em: 1) pré-processar os códigos do *Polybench*; 2) extrair o *kernel* de cada programa; 3) substituir as dimensões dos arranjos, conhecidas em tempo de compilação, passados como parâmetros, ex. "`T f(T A[1][2])`", por uma versão que usa ponteiros, ex. "`T f(T **A)`"; 4) usar `grif-c` para anotar o código com os tipos dependentes; 5) verificar se os tipos encontrados pela ferramenta anterior são válidos.

A tabela da figura 8 apresenta um resumo das funções que foram usadas nesse experimento. No total, foram analisados 99 arranjos de 30 *kernels* diferentes. Ao analisar esses programas, `grif-c` foi capaz de inferir corretamente a assinatura de 24 *kernels*. Nossa inabilidade em encontrar tipos dependentes para os 6 procedimentos restantes deve-se a resultados imprecisos da Análise de Intervalos, o que ocorre devido ao uso de expressões não-afins (que não possuem o formato: $aX + b$, em que a e b são valores numéricos e X é um símbolo do programa) para indexar memória. A implementação dessa análise foi capaz de associar o tamanho das dimensões dos arranjos a expressões simbólicas em todos os testes realizados. Contudo, em 11% dos vetores não foi possível chegar a limites exatos. Nesses casos houve uma imprecisão por parte dos tamanhos dos arranjos com alguma variação do valor real. Assim, no momento de gerar as interfaces haviam tipos de vetores que não continham o tamanho exatamente igual ao que deveria ser inferido, apesar de apresentarem um resultado muito próximo. Mesmo assim, 89% dos arranjos tiveram seu tamanho inferido corretamente. Enfatizamos que cada um desses resultados foi avaliado via uma comparação manual entre as assinaturas produzidas por `grif-c` e as declarações presentes no programa original. Um detalhe a respeito dos sucessos é que em muitos casos, 62% do total de dimensões de arranjos corretamente inferidas, limites de dimensões são criados de forma simbólica, como `max(1, n)`.

Kernel	LoC	Nro. Vet	Maior Dim.
kernel_2mm	86	5	2
kernel_3mm	98	7	2
kernel_adi	102	4	2
kernel_atax	65	4	2
kernel_bicg	73	5	2
kernel_cholesky	56	1	2
kernel_correlation	103	4	2
kernel_covariance	75	3	2
kernel_deriche	130	4	2
kernel_doitgen	64	3	3
kernel_durbin	72	2	1
kernel_fdtd_2d	81	4	2
kernel_floyd_warshall	49	1	2
kernel_gemm	68	3	2
kernel_gemver	103	9	2
kernel_gesummv	77	5	2
kernel_gramschmidt	73	3	2
kernel_heat_3d	68	2	3
kernel_jacobi_1d	54	2	1
kernel_jacobi_2d	56	2	2
kernel_lu	53	1	2
kernel_ludcmp	91	4	2
kernel_mvt	70	5	2
kernel_nussinov	64	2	2
kernel_seidel_2d	50	1	2
kernel_symm	70	3	2
kernel_syr2k	74	3	2
kernel_syrk	63	2	2
kernel_trisolv	55	3	2
kernel_trmm	53	2	2

Figura 8: Programas usados no experimento. LoC: tamanho do programa em linhas de código; Nro. Vet: quantidade de arranjos recebidos como parâmetro; Maior Dim.: maior dimensão dentre os arranjos.

6 TRABALHOS RELACIONADOS

A Análise de Intervalos é utilizada para diversos fins em Ciência da Computação [1, 8, 10, 13]. Dentre suas inúmeras aplicações, destacamos: detecção de *overflow* de inteiros [13]; produção de *Library Bindings* automáticos [8]; desambiguação de ponteiros [1]; validação de acesso a memória [10]. Neste projeto, ela é usada para coletar informações sobre os usos de variáveis. É importante ressaltar que a análise apresentada neste artigo não exige que o programa seja executado e é computada simbolicamente com base na AST.

Na literatura é possível encontrar a especificação de diversas linguagens de programação que possuem, em sua essência, as ideias de tipos dependentes. Em um de seus trabalhos, Xi apresenta Xanadu que é uma linguagem imperativa que contém tipos dependentes [18]. Além disso, Xi *et al.* [19] já mostraram que é possível obter uma forma de *assembly* que faz uso dessa tecnologia. Diferente dessas linguagens, que possuem tipos dependentes como parte de suas especificações, propomos uma forma de inserir essas ideias em outras

linguagens imperativas que não lidam naturalmente com isso. Rondom *et al.* [?] já apresentaram uma forma de trazer tipos dependentes para linguagens imperativas de **baixo nível**. Nesse trabalho é mostrado como uma linguagem, *C-like*, que lida com inteiros e ponteiros pode tirar proveito dos *Liquid Types* [?]. Neste trabalho mostramos como tipos dependentes podem ser empregados na linguagem C mas nada impede que as técnicas mostradas aqui sejam aplicadas em outras linguagens imperativas. Apresentamos uma alternativa a forma de representação de um subconjunto dos tipos da linguagem C e uma forma de gerá-los, automaticamente, a partir de trechos de código fonte. Isso possibilita que programadores possam tirar proveito de vantagens oferecidas por tipos dependentes de uma forma mais simples.

A inferência de tipos é uma tecnologia muito presente em linguagens funcionais como é o caso de ML [11] e Haskell [6]. Ribeiro *et al.* [14] fazem uso da reconstrução de tipos para tratar do problema de tornar um trecho de código parcial em outro programa que possa ser compilável. Neste trabalho, é mostrada uma técnica para gerar os tipos apresentados na seção 4 para a linguagem de programação C. Esses tipos são baseados em tipos dependentes e são contruídos através da adição de informações a tipos *built-in* da linguagem.

7 CONCLUSÃO

Neste trabalho foi apresentado uma extensão para o sistema de tipos da linguagem C que faz uso de tipos dependentes. Demonstrou-se como tais tipos permitem enriquecer a documentação de interfaces nessa linguagem, produzindo programas mais seguros. O inferidor de tipos introduzido neste artigo foi usado para implementar a ferramenta `grif-c`. Essa ferramenta anota arranjos e variáveis inteiras em um programa C com declarações de tipos dependentes.

Limitações. A inferência de tipos descrita neste trabalho não anota todo o programa C. Isso acontece por que ela depende de uma análise estática para encontrar limites de regiões alocadas em memória. A definição de limites exatos para tais regiões é um problema indecidível. Tal resultado é conclusão imediata do Teorema de *Rice* [15]. Assim, nossa ferramenta, `grif-c`, anota – conservadoramente – somente a parte do programa que sua análise de intervalos é capaz de processar. Em programas de natureza científica, como *Polybench*, `grif-c` é capaz de anotar mais da metade de todas as declarações, conforme foi visto na Seção 5.

Trabalhos Futuros. Em trabalhos futuros pretendemos aplicar os conceitos discutidos neste artigo para resolver outros problemas. Um exemplo de situação que poderia se beneficiar dessas ideias é, por exemplo, a geração automática de casos de testes para programas escritos em C. Nesse caso, esperamos usar os tipos apresentados para melhorar a qualidade de dados criados para estratégias de teste *fuzzing*.

REFERÊNCIAS

[1] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime pointer disambiguation. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 589–606.

[2] Ana Bove and Peter Dybjer. 2009. Dependent types at work. In *Language engineering and rigorous software development*. Springer, 57–99.

[3] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593.

[4] Chiyan Chen and Hongwei Xi. 2005. Combining Programming with Theorem Proving. In *ICFP*. ACM, New York, NY, USA, 66–77.

[5] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.

[6] Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

[7] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*. IEEE, 75–86.

[8] Alisa J Maas, Henrique Nazaré, and Ben Liblit. 2016. Array length inference for C library bindings. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 461–471.

[9] Conor McBride and James McKinna. 2004. The View from the Left. *J. Funct. Program.* 14, 1 (2004), 69–111.

[10] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. 2014. Validation of memory accesses through symbolic analyses. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 791–809.

[11] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press, Chapter Higher-Order Polymorphism.

[12] Benjamin C Pierce. 2005. *Advanced topics in types and programming languages*. MIT press.

[13] Fernando Magno Quintao Pereira, Raphael Ernani Rodrigues, and Victor Hugo Sperle Campos. 2013. A fast and low-overhead technique to secure programs against integer overflows. In *CGO*. IEEE, 1–11.

[14] Rodrigo Geraldo Ribeiro, Leandro Terra Cunha Melo, Marcus Rodrigues de Araujo, and Fernando Magno Quintao Pereira. 2016. Compilacao Parcial de Programas Escritos em C. In *SBLP*. 16–31.

[15] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[16] Carsten Schurmann. 2009. The Twelf Proof Assistant. In *Springer*. Washington, USA, 79–83.

[17] Hongwei Xi. 1999. Dead Code Elimination through Dependent Types. In *The First International Workshop on Practical Aspects of Declarative Languages*. Springer-Verlag LNCS vol. 1551, San Antonio, 228–242.

[18] Hongwei Xi. 2000. Imperative programming with dependent types. In *Logic in Computer Science, 2000. Proceedings. 15th Annual IEEE Symposium on*. IEEE, 375–387.

[19] Hongwei Xi and Robert Harper. 2001. A dependently typed assembly language. *ACM SIGPLAN Notices* 36, 10 (2001), 169–180.

[20] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. In *PLDI*. ACM, 249–257.