Compilação Parcial de Programas Escritos em C

Rodrigo Geraldo Ribeiro[1], Leandro Terra Cunha Melo[2], Marcus Rodrigues de Araújo[2], e Fernando Magno Quintão Pereira[2]

[1] ICEA-UFOP – R. Trinta e Seis, 115, 35.931-008, João Monlevade, MG, Brasil.
rodrigo@decsi.ufop.br

[2] UFMG – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte, MG, Brasil.
{ltcmelo,maroar,fernando}@dcc.ufmg.br

Resumo Há situações em que é desejável compilar programas cujo código fonte não está totalmente disponível. Tal necessidade aparece em ambientes integrados de desenvolvimento que auto-completam partes do programa, por exemplo, e já levou à criação de compiladores parciais para linguagens como Java e Python. Por outro lado, até o momento não existe um compilador parcial de programas escritos em C. Este artigo preenche essa lacuna. Com tal propósito, descreve-se um parser capaz de escanear código fonte C incompleto, e um inferidor de tipos capaz de reconstruir definições de tipos cujas declarações não estão presentes naquele código fonte. Ao contrário de algoritmos de inferência de tipos usados em linguagens funcionais, a técnica proposta neste trabalho reconstrói declarações ausentes de tipos algébricos, criando novas definições que tendem a aproximar aquelas presentes no programa original. Um protótipo descrevendo a nova abordagem é apresentado. Tal protótipo é capaz de reconstruir trechos não triviais de programas parcialmente disponíveis, permitindo a geração de código binário executável a partir deles.

1 Introdução

Um programa é dito *parcialmente disponível* quando parte de seu código fonte não está disponível no ambiente em que esse programa deveria ser compilado. Partes faltantes vão desde trechos de expressões, até módulos inteiros. Existem várias situações em que é desejável compilar código parcialmente disponível. A título de exemplo, descrevem-se três cenários. Primeiro, uma capacidade interessante de ambientes de desenvolvimento integrados (IDEs) é a habilidade de auto-completar trechos de código, a fim de aumentar a produtividade de programadores. Envisiona-se neste trabalho uma IDE expressiva o suficiente para prover declarações de tipos a partir do uso de instâncias desses tipos em código já escrito. Segundo, ferramentas de análises dinâmicas, como *profiling* [2], simulação [14] e instrumentação [12] requerem código executável. Entretanto, nem todo programa pode ser portado para o ambiente onde tais ferramentas estão disponíveis, devido à existência de trechos de *assembly* integrado ao fonte, módulos com direitos de propriedade, descontinuidade de versões, etc. Terceiro, em muitas situações deseja-se reconstituir um *bug* encontrado em um programa, porém, por conveniência, prefere-se descartar as partes do programa não envolvidas com o

problema descoberto. Em tal cenário, programadores poderiam beneficiar-se da capacidade de extrair e compilar um trecho específico de um programa maior.

A necessidade de compilar código parcialmente disponível é tão real que a comunidade de linguagens de programação já propôs soluções para tal problema [3,7]. Entretanto, ainda não existem técnicas de reconstrução de código C, embora a literatura já descreva parsers para programas incompletos [10]. Um dos principais entraves à compilação parcial de programas C é a ausência de técnicas de inferência de tipos para essa linguagem. Inferência de tipos é fundamental para a compilação parcial de programas devido à necessidade de suprir declarações faltantes. Contudo, a inferência de tipos em C não é uma tarefa trivial, devido, primeiro, ao seu sistema de tipagem fraca, e segundo, a uma série de idiosincrasias da linguagem, como funções com número variável de argumentos, a extensa lista de coerções possíveis e ponteiros para funções.

Esse artigo apresenta uma solução para tais problemas. Com tal intuito, descreve-se na seção 3 um parser para código C parcialmente disponível, e na seção 4 um algoritmo de inferência de tipos para a linguagem. Esse algoritmo, ao contrário de abordagens mais tradicionais, usadas em linguagens funcionais, não assume a existência de tipos algébricos que guiem o processo de inferência. Ao contrário, a abordagem proposta *reconstrói* tais uniões algébricas, produzindo declarações de registros variantes (*struct*) e arranjos a partir de restrições criadas pelo uso de instâncias de tais tipos no código disponível. Devido ao caráter fracamente tipado da linguagem, não é possível, em geral, prover garantias de progresso e preservação a partir dos tipos inferidos. Todavia, assumindo-se que o programa original não possui comportamento indefinido, tais garantias podem ser alcançadas, conforme demonstra-se na seção 4 to presente trabalho.

2 Visão Geral

Nesta seção descrevem-se, de forma intuitiva, as idéias posteriormente formalizadas neste trabalho. Todas as explicações desta seção dizem respeito ao programa visto na figura 1. Com relação a tal programa, considera-se a seguinte questão: "Caso somente a função `main.c` esteja disponível para compilação, quanto do programa original pode ser reconstruído?" A meta de tal desafio é produzir implementações para `new_node` e `createRandomPoints` a partir de informações estaticamente disponíveis nas demais funções que compõem o programa.

O desafio que propõe-se ao leitor não é artificial: os autores do presente trabalho depararam-se com ele enquanto prestando serviço para a indústria automotiva. Naquela ocasião, fazia-se necessário testar um sistema de rastreamento de veículos, implementado em C, com *assembly* embutido para processadores ARM. Entretanto, a ferramenta de teste disponível, a saber, Valgrind [14], não existe para ARM. A fim de contornar tal problema, foi feito um esforço para compilar, automaticamente, o sistema de rastreamento para x86. Muito desse esforço consistiu na substituição de código *assembly* integrado ao programa C por *stubs* gerados automaticamente. Outro desafio superado foi a reconstrução de

list.h
```
typedef struct Node* List;                              1
struct Node {                                           2
  double data;                                          3
  List next;                                            4
};                                                      5
List new_node(List, float);                             6
                                                        7
```

list.c
```
List new_node(List anchor) {                            8
  List node = (List)malloc(sizeof(struct Node));        9
  node->next = anchor;                                  10
  return node;                                          11
}                                                       12
                                                        13
```

points.h
```
typedef struct {                                        14
  float x, y;                                           15
} Point;                                                16
Point* createRandomPoints(int);                         17
                                                        18
```

points.c
```
Point* createRandomPoints(int lim) {                    19
  if (lim <= 0) return NULL;                            20
  Point* points =                                       21
          (Point*)malloc(lim*sizeof(Point));            22
  do {                                                  23
    lim--;                                              24
    points[lim].x = rand() / (RAND_MAX + 1.);           25
    points[lim].y = rand() / (RAND_MAX + 1.);           26
  } while (lim);                                        27
  return points;                                        28
}
```

main.c
```
                                                        29
List x2list(Point* points, int N) {                     30
  int i;                                                31
  List node = 0;                                        32
  for (i = 0; i < N; i++) {                             33
    node = new_node(node);                              34
    node.data = points[i].x;                            35
  }                                                     36
  return node;                                          37
}                                                       38
int main(int argc, char** argv) {                       39
  Point* points = createRandomPoints(argc);             40
  List L = x2list(points, argc);                        41
  while (L) {                                           42
    printf("%lf\n", L->data);                           43
    L = L->next;                                        44
  }                                                     45
}                                                       46
```

Figura 1: Desafio: produzir código binário executável para a função `main.c`, assumindo-se que somente o arquivo `main.c` esteja disponível para compilação.

algumas declarações de tipos ausentes. Tais declarações faziam parte de arquivos de cabeçalho embutidos no compilador usado pela empresa em questão.

A fim de reconstruir o ambiente necessário para executar a função `main`, procede-se em dois passos. O primeiro deles é a reconstrução de tipos. No exemplo da figura 1, é preciso inferir a estrutura dos tipos `Point` e `List`. O algoritmo de inferência de tipos descrito na seção 4 é capaz de descobrir que `Point` possui um campo `x` de tipo `double`. Tal é possível por que a linha 35 da figura 1 explicita esse campo. Aquela linha também deixa claro que os tipos de `Point.x` e `List.data` são idênticos, módulo coerção, isto é: esses são tipos mutuamente atribuíveis. Finalmente, na linha 43 da figura 1 aprende-se que `List.data` possui tipo `double`, vista a forma como tal campo é usado na função `printf`. Posto que o campo `y` de `Point` não foi usado no corpo de `main`, a reconstrução de tipos segue desavisada dele. Por outro lado, as linhas 43 e 44 permitem àquele algorimo reconstruir a estrutura completa de `List`. O segundo passo do processo de geração de código consiste na criação de *stubs*. Um *stub* é uma representação minimalista de uma função, cuja assinatura garante a compilação do programa alvo. O foco do presente artigo é o passo 1 desse processo. Assim, por limitação de espaço, omite-se deste documento os pormenores do passo 2.

3 Parsing

Linguagens como Java, C e C++ impõem um desafio adicional para inferência de tipos em programas parciais. Elas são sensíveis ao contexto em um aspecto não óbvio à primeira vista: suas gramáticas são ambíguas e contêm produções

nas quais o parser precisa distinguir se determinado identificador do programa nomeia um tipo ou um valor. Um exemplo de C e C++ é a ambiguidade oriunda da sintaxe "`x*y;`" [18]. Se `x` nomeia um tipo, temos a declaração de um ponteiro. Mas se `x` nomeia um valor, temos uma expressão de multiplicação. A linguagem D [6], apesar de conter a mesma construção sintática, elimina essa ambiguidade com a regra de sempre favorecer declarações. Porém, em C e C++ o parser precisa consultar uma tabela de símbolos para tomar sua decisão. No entanto, programas parciais podem não conter todas as declarações de tipos, tornando essa análise semântica inviável.

Uma alternativa para lidar com programas parciais é a utilização de *fuzzy parsers* [11], os quais têm como principal característica a capacidade de lidar com programas incompletos e/ou com grande incidência de erros sintáticos. Para atingirem seus objetivos fuzzy parsers não reconhecem, intencionalmente, toda a linguagem sob análise, mas apenas construções sintáticas pré-determinadas que permitam a extração de alguma informação sobre o programa em questão. Por exemplo, um fuzzy parser de C ou C++ poderia ser construído de forma que reconheça apenas declarações de funções e/ou de tipos definidos pelo usuário a fim de popular o navegador de nomes em uma IDE [1]. A ausência de declarações de tipo em programas parciais requer fuzzy parsing. Contudo, fuzzy parsing em sua instância original não é suficiente para que possamos compilar programas parciais. Nosso parser precisa produzir uma *Abstract Syntax Tree* (AST). Logo, ele deve ser capaz de reconhecer toda a linguagem, e não apenas parte dela.

Knapen [10] apresenta uma solução que combina características de fuzzy parsing com o requisito de reconhecimento completo da linguagem. Nosso parser adota essa estratégia, que consiste da adição de um *nó de bifurcação* na AST sempre que uma ambiguidade é encontrada. A esse nó são adicionadas as possíveis interpretações da sintaxe em questão. A medida que o parser prossegue e descobre, via outras construções sintáticas não-ambíguas, o significado de identificadores desconhecidos, ele aplica uma *resolução de desambiguação* - essa técnica é também utilizada em [15] e na IDE Eclipse CDT. Voltando à sintaxe `x*y;`, se o parser a encontra sem conhecer uma declaração de `x`, uma bifurcação será criada na AST. Esse nó será considerado uma declaração caso exista, dentro do mesmo escopo, a sintaxe `x v;`, a qual inquestionavelmente caracteriza `x` como um nome de tipo. Similarmente, o nó será uma expressão, caso exista, dentro do mesmo escopo, uma sintaxe como `x+v;`, a qual caracteriza `x` como um nome de variável. Em ambas resoluções, nosso parser depende de uma *sintaxe desambiguadora* para auxiliar sua decisão. Caso a AST contenha nós ambíguos após o parsing, o processo não prossegue para a inferência de tipos e a análise termina sem sucesso. Falhas acontecem devido a ambiguidades inerentes em alguns programas parciais, como `void f(){x*y;}` ou por que nosso parser não encontra uma sintaxe desambiguadora que o auxilie.

As ambiguidades com as quais nosso parser precisa lidar são um subconjunto daquelas mencionadas por Knapen *et al.* [10]: Todavia, há uma diferença relevante entre o nosso parser e o de Knapen *et al.*. Ele utiliza a AST produzida exclusivamente para análise estática – execução não é um requerimento. Diante

Sintaxe Ambígua	**Sintaxe Desambiguadora**	**Resolução de Desambiguação**
`x*y;`	`x z;`	Declaração de ponteiro.
	`x+z;`	Expressão multiplicativa.
`x(y);`	`int y;`	Chamada de função.
	`y z;`	Declaração de função (tipo de retorno implícito).
	`x z;`	Declaração de variável.

Tabela 1: Sintaxes ambíguas, sintaxes desambiguadoras e resolução de desambiguação, quando `x` é inicialmente desconhecido.

Símbolo	Significado	Símbolo	Significado
l	literal	x	variável
f	função	τ	variável de tipo
$\mathbf{t}$	tipo concreto	$\oplus$	operador binário
$\rho : l \to \tau$	mapeia literal para tipo	p	programa
d	declaração	c	comando
e	expressão		

Figura 2: Metavariáveis

da inexistência de sintaxes desambiguadoras, Knapen *et al.* utiliza heurísticas baseadas em convenções e/ou estilos de programação. Por exemplo, considere a ambiguidade oriunda da sintaxe "`x(y);`". Temos uma chamada de função se o identificador `x` nomeia um valor, ou uma declaração de variável caso `x` nomeie um tipo. Com o intuito de sempre produzir uma AST, se essa ambiguidade não estiver resolvida até o final do parsing, Knapen *et al.* a definirá como uma chamada de função. Nosso parser precisa produzir uma AST correta, que permita a inferência de tipos sem alteração semântica do programa original. Sendo assim, não podemos nos apoiar em heurísticas, pois resultados incorretos são inadmissíveis. A Tabela 1 ilustra exemplos das sintaxes ambíguas discutidas, possíveis sintaxes desambiguadoras e a resolução de desambiguação, considerando que a declaração de `x` seja inicialmente desconhecida.

4 Inferência de Tipos

Para compilar código C parcialmente disponível é necessário reconstruir declarações de sinônimos de tipos definidos usando a construção `typedef`. Para inferir tais declarações, usa-se um algoritmo de inferência de tipos comumente encontrado em linguagens funcionais modernas [8,13]. Para apresentação deste algoritmo, considera-se uma versão reduzida da linguagem C, denominada Mini-C, cuja sintaxe é vista na figura 3. A figura 2 indica as meta-variáveis utilizadas em toda formalização do algoritmo.

$$
\begin{array}{lll}
p & ::= p\,d \mid d & ;\text{Programa} \\
d & ::= \texttt{typedef}\ \tau\ x & ;\text{Declaração de tipo} \\
 & \mid\ \tau\ f\ \texttt{(}\tau\ x\texttt{)}\ \texttt{\{}s'\texttt{\}} & ;\text{Declaração de função} \\
s & ::= \tau\ x & ;\text{Variável local} \\
 & \mid\ x\ \texttt{=}\ e & ;\text{Atribuição} \\
 & \mid\ \texttt{return}\ e & ;\text{Retorno} \\
 & \mid\ \texttt{while}\ \texttt{(}e\texttt{)}\ \texttt{\{}s'\texttt{\}} & ;\text{Loop} \\
 & \mid\ \texttt{if}\ \texttt{(}e\texttt{)}\ \texttt{\{}s'\texttt{\}}\ \texttt{else}\ \texttt{\{}s'\texttt{\}} & ;\text{Condicional} \\
s' & ::= s'\,s \mid s & \\
e & ::= \ell & ;\text{Literal} \\
 & \mid\ x & ;\text{Variável} \\
 & \mid\ \texttt{malloc}(e) & ;\text{Alocação de memória} \\
 & \mid\ e.x & ;\text{Acesso a campo} \\
 & \mid\ e\texttt{->}x & ;\text{Acesso a campo (Ponteiro)} \\
 & \mid\ \texttt{*}e & ;\text{Deref. (Ponteiro)} \\
 & \mid\ e\,[e_1] & ;\text{Arranjo } \star(e+e_1) \\
 & \mid\ (\tau)\,e & ;\text{Coerção} \\
 & \mid\ e \oplus e' & ;\text{Operador binário} \\
 & \mid\ \&\,e & ;\text{Endereço} \\
 & \mid\ f(e_i)^{i=0..n} & ;\text{Chamada de função} \\
\tau & ::= \mathbf{B} & ;\text{construtores de tipos: int, bool, etc.} \\
 & \mid\ \tau\texttt{*} & ;\text{ponteiros} \\
 & \mid\ \{x_i : \tau_i\}^{i=1..n} & ;\text{registros} \\
 & \mid\ \tau \to \tau & ;\text{tipos de funções} \\
 & \mid\ \alpha & ;\text{variáveis de tipos}
\end{array}
$$

Figura 3: Sintaxe da linguagem Mini-C

O significado da maioria dos elementos sintáticos de Mini-C é imediato. Denota-se por $fv(\tau)$ o conjunto de variáveis livres que ocorrem no tipo τ. A meta-variável $\mathbf{t}$ representa tipos que não possuem variáveis livres, i.e. $fv(\mathbf{t}) = \emptyset$. Representamos por $fields(\tau)$ o conjunto de campos presentes na definição do tipo τ. Caso τ não seja um tipo registro, $fields(\tau) = \emptyset$. Dois tipos τ e τ' são considerados iguais módulo a relação de conversão entre tipos da linguagem C [9]. Assim, é possível unificar os tipos `int` e `short`, por exemplo.

Seja $X = \{x_1, ..., x_n\}$ um conjunto contendo $n \geq 0$ elementos. Denotamos uma sequência de $m \geq 0$ elementos de X como $x_i^{i=1..n}$. Quando o índice i não for relevante para a expressão, este será omitido como em $x^{0..n}$. Representamos a sequência vazia por $\bullet$ e a operação de inclusão de um item x em uma sequência xs por $x : xs$. Além disso, nos permitimos um pequeno abuso de notação ao usar sobre sequências operações sobre conjuntos, cujo significado é imediato.

Contextos de tipos, representados pela meta-variável Γ, são conjuntos de pares $x : \tau$. Definimos as seguintes operações sobre contextos de tipos:

$$
\begin{array}{c}
\Gamma(x) = \{\tau \mid x : \tau \in \Gamma\} \\
\Gamma,\, x : \tau = (\Gamma - \Gamma(x)) \cup \{x : \tau\}
\end{array}
$$

$$
\begin{array}{lll}
K ::= & \tau \equiv \tau & \text{Igualdade} \\
\mid & x = \tau & \text{O símbolo } x \text{ possui o tipo } \tau \\
\mid & typedef\ \tau\ as\ \tau' & \text{O tipo } \tau \text{ é um sinônimo para o tipo } \tau' \\
\mid & has(\tau, x' : \tau') & \text{O registro } \tau \text{ possui um campo de nome } x \text{ de tipo } \tau' \\
\mid & def\ x : \tau\ \text{ in } K & \text{Definição que símbolo } x \text{ possui o tipo } \tau \\
\mid & \exists\, \alpha.\, K & \text{Definição de uma nova variável } (\alpha) \\
\mid & K \wedge K & \text{Conjunção} \\
\mid & \top & \text{Tautologia} \\
\mid & \bot & \text{Contradição}
\end{array}
$$

Figura 4: Sintaxe concreta de restrições

$$
\frac{}{\phi; \varphi; \Theta \models \top}\ (CEmpty) \qquad \frac{\phi(\tau_1) = \phi(\tau_2)}{\phi; \varphi; \Theta \models \tau_1 = \tau_2}\ (CEq)
$$

$$
\frac{\phi; \varphi; \Theta \models K_1 \quad \phi; \varphi; \Theta \models K_2}{\phi; \varphi; \Theta \models K_1 \wedge K_2}\ (CConj) \qquad \frac{\phi[\alpha \mapsto \mathbf{t}]; \varphi; \Theta \models K}{\phi; \varphi; \Theta \models \exists \alpha.K}\ (CExists)
$$

$$
\frac{\varphi(x) = \phi(\tau)}{\phi; \varphi; \Theta \models x = \tau}\ (CVar) \qquad \frac{\phi; \varphi[x \mapsto \phi(\tau)]; \Theta \models K}{\phi; \varphi; \Theta \models def\ x : \tau\ in\ K}\ (CDef)
$$

$$
\frac{x : \tau' \in fields(\Theta(\tau))}{\phi; \varphi; \Theta \models has(\tau, x : \tau')}\ (CHas) \qquad \frac{\phi; \varphi; \Theta \models \Theta(\phi(\tau)) = \phi(\tau')}{\phi; \varphi; \Theta \models typedef\ \tau\ as\ \tau'}\ (CTyDef)
$$

Figura 5: Semântica de restrições

Utilizamos o símbolo $\supset$ para representar implicação lógica, ao invés de $\rightarrow$ que usamos para representar tipos funcionais.

4.1 Sintaxe e Semântica de Restrições

Seguindo a literatura atual [16], descreveremos o algoritmo de inferência de tipos como dois processos: um de geração e outro de resolução de restrições, cuja sintaxe é apresentada na Figura 4. Como restrições são fórmulas da lógica de primeira ordem, sua satisfazibilidade é representada em termos de um modelo [19]. A interpretação de restrições é dada por um julgamento $\phi; \varphi; \Theta \models K$ em que ϕ é uma função finita que associa variáveis (de tipos) a tipos, φ associa variáveis de programa a tipos e Θ é uma função que associa nomes de tipos a sua definição. Dizemos que uma restrição K é satisfazível se existem ϕ, φ e Θ tais que $\phi; \varphi; \Theta \models K$ é provável usando as regras da Figura 5.

Note que as funções finitas utilizadas para definir a semântica de restrições, representam as declarações de um programa Mini-C. De maneira simples, a restrição $\top$ é satisfazível e $\bot$ não pode ser satisfeita em quaisquer ϕ, φ e Θ. Uma restrição $\tau \equiv \tau'$ é satisfazível se e somente se, os tipos τ e τ' forem equivalentes. Dizemos que $x = \tau$ é satisfeita se o tipo associado ao símbolo x (definido em

$$\frac{l_i^{i=1..n} \subseteq k_j^{1..m} \quad k_j = l_i \supset \tau_j \preceq \tau_i}{\{k_j : \tau_j\}^{j=1..m} \preceq \{l_i : \tau_i\}^{i=1..n}} \; \{SStruct\}$$

Figura 6: Relação de subtipagem para Mini-C (regras de reflexividade e transitividade omitidas)

φ) é equivalente ao tipo τ. A conjunção de duas restrições K_1 e K_2, $K_1 \wedge K_2$ é satisfeita se ambos seus componentes forem satisfeitos. Restrições contendo o quantificador existencial, $\exists\alpha.K$, são satisfeitas se K for satisfeita por $\phi[\alpha \mapsto \mathbf{t}]$. O mesmo aplica-se a restrições de definições de símbolos (def). Uma restrição $has(\tau, x : \tau')$ é satisfeita se o tipo τ possui um campo de nome x de tipo τ', isto é, $x : \tau' \in fields(\Theta(\tau))$. Dizemos que uma restrição K_1 implica uma restrição K_2, $K_1 \Vdash K_2$, se e somente se para todo ϕ, φ e Θ, $\phi; \varphi; \Theta \models K_1$ implica $\phi; \varphi; \Theta \models K_2$. Finalmente, restrições K_1 e K_2 são equivalentes, $K_1 \approx K_2$

4.2 Sistema de Tipos

Um problema para compilação de programas parciais é a inferência de tipos registros: nem sempre o código disponível possui informação suficiente para inferir toda estrutura do registro. Para garantir que os resultados do algoritmo de inferência sejam corretos em relação ao sistema de tipos de Mini-C, utilizamos uma relação de subtipagem. Essa relação assegura que registros cuja estrutura não é possível de ser completamente inferida sejam supertipos de seus respectivos tipos completos. A relação de subtipagem é o fecho reflexivo e transitivo (módulo regras de conversão implícita de tipos) da regra apresentada na figura 6.

O sistema de tipos para Mini-C é apresentado nas Figuras 7,8 e 9. As regras de tipo para expressões são definidas pelo sistema de provas $K, \Gamma \vdash e : \tau$, denotando que a expressão e possui o tipo τ no contexto Γ e restrições K. Tipagem de literais é expressa em termos da função ρ. Um literal é tipável se ele possui uma definição no contexto de tipos. Uma expressão de acesso a campo ($e.x$ ou $e \rightarrow x$) é correta caso a expressão e possua um tipo τ com um campo de nome x. Regra para subtipagem ($\preceq$) não é dirigida pela sintaxe. Essa regra lida com a relação de subtipos registros (Figura 6). Um comando c é bem tipado em um contexto Γ e restrições K, se $K, \Gamma \vdash c$ é provável de acordo com as regras da Figura 8 e utiliza as regras para atribuir tipos a expressões (Figura 7). A Figura 9 apresenta regras para declarações, cujo significado é imediato.

4.3 Geração de Restrições

O gerador de restrições é o componente responsável por reduzir o problema de inferência de tipos para Mini-C a um problema de satisfazibilidade de restrições. De maneira simples, o problema de inferência de tipos deve, a partir de um par formado por um contexto de tipos Γ e um programa $p = ds$, determinar uma restrição K tal que $K, \Gamma \vdash ds$ é provável utilizando as regras das Figuras 7, 8

$$\frac{K,\,\Gamma \vdash e : \tau \quad K \Vdash \tau \equiv \texttt{int}}{K,\,\Gamma \vdash \texttt{malloc(}e\texttt{)} : \texttt{int} \to \texttt{void}^{\star}} \qquad \frac{\Gamma(x) = \tau \quad K \Vdash \exists\alpha.\alpha \equiv \tau}{K,\,\Gamma \vdash x : \tau} \qquad \frac{}{K,\,\Gamma \vdash l : \rho(l)}$$

$$\frac{K,\,\Gamma \vdash e : \tau' \quad K \Vdash has(\tau', x : \tau)}{K,\,\Gamma \vdash e.x : \tau} \qquad \frac{K,\,\Gamma \vdash e : \tau' \quad K \Vdash \tau \equiv \tau'}{K,\,\Gamma \vdash (\tau)\, e : \tau} \qquad \frac{K,\,\Gamma \vdash e : \tau' \quad K \Vdash \tau' \equiv \tau^{\star}}{K,\,\Gamma \vdash \star e : \tau}$$

$$\frac{\begin{array}{cc} K,\,\Gamma \vdash e : \tau' & K \Vdash \tau' \equiv \tau[n] \\ K,\,\Gamma \vdash e' : \texttt{int} & K,\,\Gamma \vdash n : \texttt{int} \end{array}}{K,\,\Gamma \vdash e[e'] : \tau} \qquad \frac{\begin{array}{c} K,\,\Gamma \vdash e_i : \tau_i^{i=0..n} \\ \Gamma(f) = \tau_i^{i=0..n} \to \tau \end{array}}{K,\,\Gamma \vdash f(e_i)^{0..n} : \tau} \qquad \frac{\begin{array}{c} K,\,\Gamma \vdash e : \tau_1 \quad K,\,\Gamma \vdash e' : \tau_2 \\ \Gamma(\oplus) = \tau_1 \to \tau_2 \to \tau \end{array}}{K,\,\Gamma \vdash e \oplus e' : \tau}$$

$$\frac{\begin{array}{c} K,\,\Gamma \vdash e : \tau' \\ K \Vdash \exists\alpha.\tau' \equiv \alpha* \wedge has(\alpha, x : \tau) \end{array}}{K,\,\Gamma \vdash e\texttt{->}x : \tau} \qquad \frac{K,\,\Gamma \vdash e : \tau}{K,\,\Gamma \vdash \&\, e : \texttt{long}} \qquad \frac{K,\,\Gamma \vdash e : \tau' \quad \tau' \preceq \tau}{K,\,\Gamma \vdash e : \tau}$$

Figura 7: Regras de tipo para expressões Mini-C

$$\frac{\Gamma(x) = \tau \quad K,\,\Gamma \vdash e : \tau}{K,\,\Gamma \vdash_{\tau'} x \;=\; e} \qquad \frac{K,\,\Gamma \vdash_{\tau'} e : \tau'}{K,\,\Gamma \vdash_{\tau'} \texttt{return}\ e} \qquad \frac{}{K,\,\Gamma \vdash_{\tau} \bullet}$$

$$\frac{K,\,\Gamma \vdash e : \texttt{bool} \quad K,\,\Gamma \vdash_{\tau'} c \quad K,\,\Gamma \vdash_{\tau'} c'}{K,\,\Gamma \vdash_{\tau'} \texttt{if (}e\texttt{) c else c'}} \qquad \frac{K,\,\Gamma \vdash e : \texttt{bool} \quad K,\,\Gamma \vdash_{\tau'} c}{K,\,\Gamma \vdash_{\tau'} \texttt{while (}e\texttt{) c}}$$

$$\frac{K \Vdash \exists\alpha.typedef\ \tau\ as\ \alpha \quad K, \Gamma, x : \tau \vdash cs}{K,\,\Gamma \vdash_{\tau'}\ \tau\ x\ :\ cs} \qquad \frac{K,\,\Gamma \vdash_{\tau'} c \quad K,\,\Gamma \vdash_{\tau'} cs}{K,\,\Gamma \vdash_{\tau'} c : cs}$$

Figura 8: Regras de tipo para comandos Mini-C

$$\frac{\begin{array}{c} K \Vdash \exists\alpha.typedef\ \ \tau\ \ \alpha \qquad K \Vdash (\exists\alpha_i.typedef\ \tau_i\ \ as\ \ \alpha_i)^{i=0..n} \\ K,\,\Gamma, f : \tau^{i=0..n} \to \tau, \{x_i : \tau_i\}^{i=0..n} \vdash_{\tau} c \end{array}}{K,\,\Gamma \vdash\ \tau\ f\ (\tau_i\ x_i)^{i=0..n}\{c\}}$$

$$\frac{K \Vdash \exists\alpha.typedef\ \ \tau\ \ as\ \ \alpha \qquad K,\,\Gamma, x : \tau \vdash ds}{K,\,\Gamma \vdash \tau\ x : ds}$$

$$\frac{K \Vdash typedef\ \texttt{x}\ as\ \tau}{K,\,\Gamma \vdash \texttt{typedef}\ \tau\ \texttt{x}} \qquad \frac{K,\,\Gamma \vdash d \quad K,\,\Gamma \vdash ds}{K,\,\Gamma \vdash d : ds} \qquad \frac{}{K,\,\Gamma \vdash \bullet}$$

Figura 9: Regras de tipo para declarações Mini-C

e 9. O problema de satisfazibiliade de restrições possui como parâmetro uma restrição K. Esse problema consiste em determinar se K é ou não satisfazível.

Para reduzir um problema de inferência de tipos (Γ, p) a um problema de satisfazibilidade de restrições devemos produzir um conjunto de restrições K, *necessário* e *suficiente*, para que $K,\,\Gamma \vdash p$ seja provável. Denotamos a restrição construída a partir de um programa p por $\langle\langle\, p \,\rangle\rangle_d$. Para suficiência, uma restrição

$$
\begin{array}{ll}
\langle\langle\, l : \tau \,\rangle\rangle_e & = \rho(l) \equiv \tau \\
\langle\langle\, x : \tau \,\rangle\rangle_e & = x \equiv \tau \\
\langle\langle\, \oplus : \tau \,\rangle\rangle_e & = \oplus \equiv \tau \\
\langle\langle\, f : \tau \to \tau' \,\rangle\rangle_e & = f \equiv \tau \to \tau' \\
\langle\langle\, e.x : \tau \,\rangle\rangle_e & = \exists \alpha_1\, \alpha_2 . \langle\langle\, e : \alpha_1 \,\rangle\rangle_e \wedge \langle\langle\, x : \alpha_2 \,\rangle\rangle_e \wedge has(\alpha_1, x : \alpha_2) \wedge \tau \equiv \alpha_2 \\
\langle\langle\, e[e_1] : \tau \,\rangle\rangle_e & = \exists \alpha_1 \alpha_2 \alpha_3 . \langle\langle\, e : \alpha_1 \,\rangle\rangle_e \wedge \langle\langle\, e_1 : \alpha_2 \,\rangle\rangle_e \wedge \alpha_1 \equiv \star\alpha_3 \wedge \tau \equiv \alpha_3 \wedge \alpha_2 \equiv int \\
\langle\langle\, (\tau')\, e : \tau \,\rangle\rangle_e & = \exists \alpha . typedef\ \tau'\ as\ \alpha \wedge \tau \equiv \tau' \\
\langle\langle\, f(e^{i=1..n}) : \tau \,\rangle\rangle_e & = \exists \alpha^{i=1..n} . \bigwedge_{i=1..n} \langle\langle\, e_i : \alpha_i \,\rangle\rangle_e \wedge \langle\langle\, f : \alpha^{i=1..n} \to \tau \,\rangle\rangle_e \\
\langle\langle\, e \oplus e' : \tau \,\rangle\rangle_e & = \exists \alpha_1 \alpha_2 . \langle\langle\, e : \alpha_1 \,\rangle\rangle_e \wedge \langle\langle\, e' : \alpha_2 \,\rangle\rangle_e \wedge \langle\langle\, \oplus : \alpha_1 \to \alpha_2 \to \tau \,\rangle\rangle_e \\
\langle\langle\, \&\, e : \tau \,\rangle\rangle_e & = \exists \alpha\, \alpha' . \langle\langle\, e : \alpha' \,\rangle\rangle_e \wedge \alpha \equiv \star\alpha' \wedge \tau \equiv \alpha \\
\langle\langle\, \star e : \tau \,\rangle\rangle_e & = \exists \alpha . \langle\langle\, e : \alpha \,\rangle\rangle_e \wedge \alpha \equiv \star\tau \\
\langle\langle\, e \to x : \tau \,\rangle\rangle_e & = \exists \alpha_1 \alpha_2 \alpha_3 . \langle\langle\, e : \alpha_1 \,\rangle\rangle_e \wedge \langle\langle\, x : \alpha_3 \,\rangle\rangle_e \wedge \alpha_1 \equiv \star\alpha_2 \wedge has(\alpha, x : \alpha_3) \wedge \tau \equiv \alpha_3
\end{array}
$$

Figura 10: Geração de restrições para expressões.

deve ser tal que $\langle\langle\, p \,\rangle\rangle_d,\, \Gamma \vdash p$ é provável. A propriedade de correção do gerador de restrições (Teorema 1) garante que restrições geradas são de fato suficientes. Dizemos que $\langle\langle\, p \,\rangle\rangle_d$ é necessário se, para toda restrição K, a validade de $K,\, \Gamma \vdash p$ implica $K \Vdash \langle\langle\, p \,\rangle\rangle_d$. Isto é, toda restrição K que garante a tipabilidade de p é pelo menos tão específica quanto $\langle\langle\, p \,\rangle\rangle_d$. O Teorema 2 garante que esta propriedade é verdadeira.

O processo de geração de restrições é apresentado nas Figuras 10, 11 e 12 como funções recursivas sobre a estrutura sintática de programas Mini-C. A função para geração de restrições para uma expressão e, isto é: $\langle\langle\, e : \tau \,\rangle\rangle_e$, recebe como parâmetro a expressão analisada e, mais o tipo τ que ela deve possuir. Para literais, variáveis, operadores e funções são geradas restrições de igualdade. No que tange expressões envolvendo acesso a campos, restrições de acesso são geradas e sempre que um tipo não conhecido é encontrado um quantificador existencial é utilizado para representar novas variáveis de tipos.

A função para cálculo de restrições para um comando c recebe como parâmetro, além do comando c, um tipo τ que representa o tipo de retorno da função onde o comando c ocorre. Este tipo τ é utilizado como parâmetro para a geração de restrições para expressões contidas em comandos `return`. Tanto blocos de comandos quanto declarações são representados como sequências. Esta estrutura é utilizada para modelar a visibilidade de declarações.

Teorema 1 (Correção da geração de restrições)**.** *Se p é um programa Mini-C válido então existe Γ tal que $\langle\langle\, p \,\rangle\rangle_d, \Gamma \vdash p$ é provável.*

Demonstração. Por indução sobre a estrutura de p usando as definições do gerador de restrições e das regras do sistema de tipos. □

Teorema 2 (Completude da geração de restrições)**.** *Suponha que $K,\, \Gamma \vdash p$. Então, $K \Vdash \langle\langle\, p \,\rangle\rangle_d$.*

Demonstração. Por indução sobre a estrutura de $K,\, \Gamma \vdash p$. □

$$
\begin{array}{ll}
\langle\langle \bullet, \tau \rangle\rangle_s & = true \\
\langle\langle \tau'\ x; cs, \tau \rangle\rangle_s & = typedef\ \tau'\ as\ \alpha \wedge def\ x : \tau'\ in\ \langle\langle cs, \tau \rangle\rangle_s \\
\langle\langle x := e; cs, \tau \rangle\rangle_s & = \exists\alpha.\langle\langle x : \alpha \rangle\rangle_e \wedge \langle\langle e : \alpha \rangle\rangle_e \wedge \langle\langle cs, \tau \rangle\rangle_s \\
\langle\langle \texttt{while (}e\texttt{)\{}cs\texttt{\}}; cs', \tau \rangle\rangle_s & = \langle\langle e : bool \rangle\rangle_e \wedge \langle\langle cs, \tau \rangle\rangle_s \wedge \langle\langle cs', \tau \rangle\rangle_s \\
\langle\langle \texttt{if (}e\texttt{)}\ cs_1\ \texttt{else}\ cs'_1; cs, \tau \rangle\rangle_s & = \langle\langle e : bool \rangle\rangle_e \wedge \langle\langle cs_1, \tau \rangle\rangle_s \wedge \langle\langle cs'_1, \tau \rangle\rangle_s \wedge \langle\langle cs, \tau \rangle\rangle_s \\
\langle\langle \texttt{return}\ e, \tau \rangle\rangle_s & = \langle\langle e : \tau \rangle\rangle_e
\end{array}
$$

Figura 11: Geração de restrições para comandos.

$$
\begin{array}{ll}
\langle\langle \bullet \rangle\rangle_d & = true \\
\langle\langle \texttt{typedef t}\ x\ , : ds \rangle\rangle_d & = typedef\ x\ as\ \texttt{t} \wedge \langle\langle ds \rangle\rangle_d \\
\langle\langle \tau\ f\ \texttt{(}(\tau\ x)^{i=1..n}\texttt{)\{}cs\texttt{\}}, ds \rangle\rangle_d & = \exists\alpha^{i=1..n}.(typedef\ \tau_i\ as\ \alpha_i)^{1..n} \wedge def\ f : \tau^{i=1..n} \rightarrow \tau\ in \\
 & \quad (def\ x_i : \tau_i)^{i=1..n}\ in\ \langle\langle cs, \tau \rangle\rangle_s \wedge \langle\langle ds \rangle\rangle_d
\end{array}
$$

Figura 12: Geração de restrições para declarações.

4.4 Resolvendo Restrições

O algoritmo de resolução de restrições consiste em 4 etapas: 1) coleta de definições de tipos, variáveis e funções; 2) expansão de restrições de tipos de símbolos; 3) unificação das restrições de igualdade e 4) construção de registros a partir das restrições de campos de registros. A primeira etapa do resolvedor consiste em coletar restrições de definições de tipos (*typedef*) e definições de símbolos (*def*) para construção de ambientes a serem utilizados em etapas posteriores. Na etapa 2 ocorre a substituição de restrições $x = \tau$ por restrições de igualdade $\tau' \equiv \tau$, em que τ' é o tipo do símbolo x. A etapa principal de resolução de restrições é a unificação de restrições de igualdade. Esta etapa segue apresentações tradicionais de algoritmos para unificação [16,17]. A substituição resultante da unificação é utilizada em conjunto com as restrições de campos de registros para determinar a estrutura de definições de tipos inferidos.

5 Resultados Experimentais

A fim de avaliar a utilidade das idéias discutidas neste artigo, optou-se por usar o compilador parcial juntamente com `aprof` [2] e verificar se os resultados obtidos por esta ferramenta nos códigos reconstruídos são os mesmos quando comparados aos resultados dos códigos originais. `aprof` é uma programa usado para estimar a complexidade assimptótica de programas via *profiling*. Com tal propósito, essa tecnologia rastreia o tamanho da entrada de cada função de um programa, e relaciona esse valor ao número de operações executadas por aquela função. Um requisito essencial de `aprof` é que a função seja executada diversas vezes. Doutro modo, o algoritmo de regressão polinomial não encontra pontos suficientes para estabelecer a complexidade da função de interesse. Essa limitação, reconhecida

pelos autores de `aprof`, é bastante séria: caso uma função seja invocada somente uma ou duas vezes durante a execução de um programa, então sua complexidade não poderá ser estimada com elevado grau de confiança. E, infelizmente, a maior parte das funções em programas reais tende a ser chamada somente umas poucas vezes. Por exemplo, Costa *et al.* [4] observaram que mais de 50% das funções JavaScript observadas em uma sessão típica do navegador Internet Explorer foram invocadas somente uma vez.

Com o intuito de contornar tal limitação, nesta seção propõe-se a seguinte metodologia: para cada função de interesse f, que existe dentro de um programa p, cria-se um novo programa p', que contém somente uma função `main` e f. Da análise de p', o compilador parcial descrito nesse trabalho reconstrói o contexto necessário para que f possa ser invocada a partir da função `main`. Em seguida, dentro desta função, são alocadas e inicializadas, com dados aleatórios, as estruturas que serão passadas como parâmetro para a função f. Cada execução de p' é analisada por `aprof`. As tuplas $(\textit{size} \times \textit{operations})$ são então usadas para gerar uma curva de complexidade para f. A fim de exercitar as capacidades de nosso gerador de tipos, faz-se ainda uma alteração em f: todas as declarações de tipos foram removidas. Nosso objetivo é inferir cada uma dessas declarações.

Benchmarks: esse experimento será executado sobre os benchmarks utilizados para validar a ferramenta `Asymptus` [5]. `Asymptus` implementa uma técnica de inferência de complexidade que compete com `aprof`. Utilizou-se os benchmarks do projeto `Asymptus` porque um dos autores participou da realização daqueles experimentos, e por isso ele tem acesso aos programas testados. Esse conjunto de testes consiste de 16 programas que implementam algoritmos simples, todos retirados de polybench, um benchmark tipicamente utilizado em otimizações poliédricas. Os testes incluem algoritmos de ordenação quadráticos como *bubblesort* e *insertion-sort*, algoritmos de álgebra linear como: multiplicação de matrizes, soma matricial, decomposição LU, etc; algoritmos em grafos como Bellman-Ford e Floyd-Warshall, dentre outros. A figura 13 (a) mostra um desses programas. Note que as declarações de tipos foram removidas, exatamente para exercitar o inferidor apresentado neste trabalho.

A figura 13 (b) mostra a curva de complexidade produzida por `aprof` a partir de 100 execuções da função reconstruída. E a figura 13 (c) mostra os tipos inferidos – automaticamente – para a função `kernel_fdtd_2d`. Cada um dos testes de complexidade usa entradas aleatórias, geradas pela função `main` presente em cada benchmark. Essa rotina possui um laço em que a função de interesse, por exemplo, `kernel_fdtd_2d` na figura 13 (a), é invocada com entradas diferentes.

A figura 14 resume os resultados obtidos neste experimento. Dentre os 16 programas, pudemos executar corretamente 13 deles. As omissões devem-se a impossibilidade de reconstruir tipos: ao remover todas as declarações de tipos, o benchmark não forneceu sintaxe suficiente para que o inferidor de tipos pudesse completar seu trabalho com sucesso. Em geral, isso ocorre devido a `struct`s cujos campos não são utilizados em nenhum ponto do programa incompleto. Assume-se que esse tipo de situação não é comum em cenários reais: ele surgiu em nosso ambiente controlado porque optou-se por remover todas as declarações

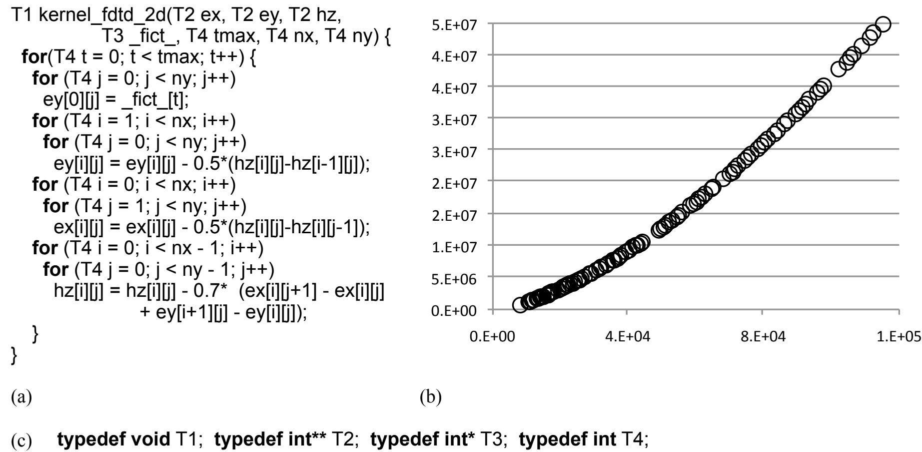

Figura 13: (a) Versão do método de diferenças finitas (*Finite-difference time-domain*) em que as declarações de tipos foram removidas. (b) Curva de complexidade da função `kernel_fdtd_2d`, produzida por *aprof* a partir de pontos obtidos de 100 execuções. (c) Tipos inferidos pela técnica deste artigo.

de tipos dos programas alvo. Os demais benchmarks – 13 programas – puderam ser compilados e analisados por `aprof` com sucesso, e para cada um deles foi produzida uma curva de complexidade, similar àquela vista na figura 13 (c).

6 Trabalhos Relacionados

Bischofberger [1] propôs as primeiras técnicas para efetuar o parsing de programas C ou C++ incompletos, com o objetivo de suportar o desenvolvimento de melhores IDEs. Foi ele quem introduziu a noção de fuzzy parsing como suporte para a ferramenta Sniff C++. Uma caracterização mais precisa, porém ainda semi-formal, do que seria fuzzy parsing foi data por Koppler *et al.* [11]. Um fuzzy parser que precisa reconhecer toda a linguagem é proposta por Knapen *et al.* [10]. Tratando-se especificamente de parsing, nosso trabalho segue essa abordagem. A diferenciação acontece ao final do parsing: Knapen *et al.* buscam obter uma AST qualquer que representa a sintaxe de um programa parcial; a obtenção de uma AST em nosso trabalho é uma etapa intermediária. Nosso objetivo final é inferir tipos em um programa parcial. O trabalho de Dagenais *et al.* [3] tem objetivos semelhantes aos nossos, sendo porém voltado para Java. Como o parser de Knapen *et al.*, o parser de Dagenais *et al.* também adota fuzzy parsing para lidar com programas parciais. Assim como nosso trabalho, Dagenais *et al.* também produz uma AST para compilar um programa parcial. Contudo, as linguagens Java e C possuem diferenças sintáticas e semânticas extensas o suficiente para impedir-nos de usar as técnicas de Dagenais. Por exemplo, nosso algoritmo de inferência de tipos usa um sistema de restrições diferente daquele

Programa	LoC	Lacunas	Acertos	Struct	Globais	Executável?
`bellman_ford`	29	4	4	Não	Não	Sim
`bubble_sort`	10	3	3	Não	Não	Sim
`closest_pairs`	15	10	10	Sim	Sim	Sim
`closest_point`	12	10	7	Sim	Sim	Não
`insertion_sort`	11	3	3	Não	Não	Sim
`kernel_2mm`	16	5	5	Não	Não	Sim
`kernel_fdtd_2d`	16	4	4	Não	Não	Sim
`kernel_floyd_warshall`	7	3	3	Não	Não	Sim
`kernel_jacobi_2d_imper`	11	4	4	Não	Não	Sim
`kernel_lu`	8	3	2	Não	Não	Sim
`kernel_seidel`	7	3	3	Não	Não	Sim
`matrix_mul`	16	3	3	Não	Não	Sim
`multiply`	6	2	2	Não	Não	Sim
`selection_sort`	13	3	3	Não	Não	Sim
`sub_list`	19	3	0	Sim	Não	Não
`sum`	9	3	0	Não	Não	Não

Figura 14: Resumo dos resultados obtidos ao se aplicar o inferidor de tipos aos benchmarks. As colunas constam, respectivamente, das seguintes informações: **Programa**: nome do programa; **LoC**: número de linhas do programa de entrada; **Lacunas**: quantidade de tipos indefinidos no código; **Acertos**: quantidade de tipos inferidos corretamente; **Struct**: benchmarks que contêm instâncias de `struct`; **Globais**: benchmarks que contêm variáveis globais; **Executável**: benchmarks que pudemos executar com sucesso.

usado em Java. Além disso, a sintaxe de C possui ambiguidades inexistentes em Java. Finalmente, o sistema de tipos de Java provê garantias de progresso e preservação que não são possíveis em C. Muito de nossas decisões acerca da reconstrução de tipos foi norteada por essa observação.

7 Conclusão

Este artigo apresentou um conjunto the técnicas para compilar código C disponível parcialmente, incluindo o primeiro sistema de inferência e reconstrução de tipos para C. A compilação de código parcial é necessária em vários cenários como teste, depuração e criação de APIs de desenvolvimento. O compilador parcial descrito neste artigo está disponível publicamente, e é capaz de reconstruir programas complexos, contendo tipos compostos como `struct`s e arranjos.

Agradecimentos

Leandro Melo recebe uma bolsa DTC mantida pela Intel Semicondutores. Este projeto é financiado pela FAPEMIG, pelo CNPq e pela CAPES.

Referências

1. Walter R Bischofberger. Sniff (abstract): a pragmatic approach to a c++ programming environment. *OOPS*, 4(2):229, 1993.
2. Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *PLDI*. ACM, 2012.
3. Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *OOPSLA*, pages 313–328. ACM, 2008.
4. Igor Rafael de Assis Costa, Pericles Rafael Oliveira Alves, Henrique Nazare Santos, and Fernando Magno Quintao Pereira. Just-in-time value specialization. In *CGO*, pages 1–11. ACM, 2013.
5. Francisco Demontiê, Junio Cezar, Mariza Andrade da Silva Bigonha, Frederico Campos, and Fernando Magno Quintão Pereira. Automatic inference of loop complexity through polynomial interpolation. In *SBLP*, pages 1–15. Springer, 2015.
6. Walter Bright et al. The D programming language, 2016. http://dlang.org.
7. Patrice Godefroid. Micro execution. In *ICSE*, pages 539–549. ACM, 2014.
8. Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *HOPL III*, pages 12–1–12–55. ACM, 2007.
9. Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.
10. Gregory Knapen, Bruno Laguë, Michel Dagenais, and Ettore Merlo. Parsing c++ despite missing declarations. In *IWPC*, pages 114–125. IEEE, 1999.
11. Rainer Koppler. A systematic approach to fuzzy parsing. *Software-Practice and Experience*, 27(6):637–650, 1997.
12. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200. ACM, 2005.
13. Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.
14. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.
15. J Overbey, Matthew D Michelotti, and R Johnson. Toward a language-agnostic, syntactic representation for preprocessed code. In *WRT*. ACM, 2009.
16. François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
17. Rodrigo Ribeiro and Carlos Camarão. A mechanized textbook proof of a type unification algorithm. In *SBMF*, pages 127–141. Springer, 2015.
18. ISO Standard. 9899. *C Programming Language*, 2011.
19. Dirk van Dalen. *Logic and structure (3.ed)*. Universitext. Springer, 1994.