Otimização de Tipos em Linguagem $\mathcal{LL}$

Clevan Ricardo da Costa

Otimização de Tipos em Linguagem $\mathcal{LL}$

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Clevan Ricardo da Costa e aprovada pela Comissão Julgadora.

Campinas, 11 de dezembro de 1995.

Prof. Tomasz Kowaltowski
Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Costa, Clevan Ricardo da

C823o Otimização de tipos em linguagem LL/Clevan Ricardo da Costa. -- Campinas, [S.P. :s.n.], 1995..

Orientador : Tomasz Kowaltowski

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Ciência da Computação.

1. Compiladores (Computadores). 2. Linguagem de programação (Computadores). 3.*Otimização de código. I. Kowaltowski, Tomasz II. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Ciência da Computação. III. Título.

Tese de Mestrado defendida e aprovada em 11 de Dezembro de 1995 pela Banca Examinadora composta pelos Profs. Drs.

Prof (a). Dr (a).

Cecília Mary Fischer Rubira

Prof (a). Dr (a). CECILIA MARY FISCHER RUBIRA

Prof (a). Dr (a).

Otimização de Tipos em Linguagem $\mathcal{LL}$[1]

Clevan Ricardo da Costa[2]

Departamento de Ciência da Computação
IMECC – UNICAMP

Banca Examinadora:

- Luiz Eduardo Buzato (Suplente)[3]
- Tomasz Kowaltowski (Orientador)[3]
- Roberto da Silva Bigonha[4]
- Cecília Mary Fischer Rubira[3]

[1]Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

[2]O autor é Bacharel em Informática pela Universidade Federal do Paraná.

[3]Professor(a) do Departamento de Ciência da Computação - IMECC - UNICAMP.

[4]Universidade Federal de Minas Gerais.

Agradecimentos

Ao CNPq, à FAPESP e à FAEP pelo apoio financeiro recebido.

Ao prof. Tomasz Kowaltowski pela orientação e dedicação. Aos amigos do projeto: Cristina e Apuã. Ao Evandro, criador do compilador de $\mathcal{LL}$ e amigo para todas as horas. Aos professores Cecília Rubira e Roberto Bigonha e os amigos Letícia e Fabiano pela a atenção dispensada a este trabalho e seus comentários valiosos.

Aos professores e funcionários do DCC e do IMECC pela dedicação e profissionalismo. Aos professores da UFPr pelo incentivo e apoio.

A cada um dos amigos que tive o prazer de conhecer em Campinas (brasileiros por nascimento ou de coração). Ao pessoal da UNICAMP, do DCC, do "Predinho", do volei. Gostaria realmente de agradecer a cada um pela amizade e companheirismo durante este tempo.

Aos colegas de salinha – Fábio, Mateus, Ronaldo, Otávio e Luiz; Chris, Dalmi e Flávio – e aos colegas de turma: testemunhas, incentivadores e colaboradores do trabalho de cada dia.

Aos amigos de moradia – Anderson, Jaime e Júnior; Diogo, Escobar, Edson e Orlando – e aqueles que consideram nossa casa como sua. Com sua amizade sempre presente ajudaram a minimizar a lacuna deixada pela saudade da família.

A minha família, principalmente meus pai – Inês e Rogério – e irmãos – Clayton e Renan, pelo amor e pelo apoio incondicional e irrestrito.

A Deus por tudo.

Resumo

Este trabalho consiste na introdução de uma fase de otimização no compilador para a linguagem $\mathcal{LL}$. O seu principal objetivo é a inferência de tipos, realizada através de uma análise de fluxo de dados.

São apresentados diversos métodos para inferência de tipos em linguagens orientadas a objetos, bem como uma revisão sobre análise de fluxo de dados. O método de Kaplan e Ullman é apresentado com mais detalhes, juntamente com sua adaptação para $\mathcal{LL}$, sua implementação e junção ao compilador desenvolvido para a linguagem.

Finalmente são apresentadas as conclusões, os resultados obtidos e as propostas de extensões futuras para o trabalho.

Abstract

We describe in this thesis an optimization phase for a compiler for the language $\mathcal{LL}$. Its main goal is type inference achieved through data flow analysis.

Several methods for type inference in object-oriented languages are described, including a description of data flow analysis. The Kaplan and Ullman method is described in more detail. Its adaptation to $\mathcal{LL}$ and the implementation within an existing compiler are also described.

We present also some final conclusions, including examples and possible extensions.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

$\mathcal{LL}$[1], uma linguagem em desenvolvimento no DCC, é caracterizada por ter objetos com tipos fortes e por outro lado variáveis com tipos fracos, ou seja, com tipos especificados apenas opcionalmente. Possui um conjunto pequeno de comandos, uma sintaxe simples e facilidades para manipulação de bibliotecas.

Se por um lado linguagens como $\mathcal{LL}$, onde variáveis tem tipos fracos, facilitam o trabalho de programação, a prototipação e manutenção de sistemas e dão maior flexibilidade ao programador, por outro lado os programas gerados pelo compilador de tais linguagens, em geral, não são tão eficientes como os programas gerados de linguagens de variáveis com tipos fortes.

Idealmente, compiladores deveriam produzir um código tão eficiente (isto é, executar tão rápido e ocupar tanto espaço) quanto algum programa escrito diretamente em linguagem objeto. Em linguagens com variáveis de tipos fracos este ideal é ainda mais forte. Compiladores de tais linguagens, por serem mais próximas à representação do algoritmo e mais distantes da representação interna na máquina, tendem a gerar um código ainda menos eficiente. $\mathcal{LL}$ não foge à regra.

Como os compiladores atuais não são tão eficientes, ganhos podem ser obtidas por meio de rotinas que analisam a estrutura intermediária gerada pelo *front end* do compilador e descobrem como produzir melhorias no código a ser gerado. Tais rotinas são chamadas de "otimizadores" (apesar de nem sempre produzirem o resultado "ótimo") [ASU86].

No caso de $\mathcal{LL}$, o objeto principal do trabalho aqui proposto, ganhos em velocidade podem ser obtidos por um otimizador de chamadas de rotinas. Quando um método é invocado é necessário, durante a execução, fazer diversos testes para se descobrir o tipo do primeiro argumento antes de se fazer a chamada da rotina que implementa tal método. Estudos demonstram que grande parte desses testes pode ser eliminada por uma análise estática anterior à geração de código.

O objetivo deste trabalho é fazer a inferência de tipos para $\mathcal{LL}$: determinar em cada ponto do programa, o tipo (ou o conjunto de tipos) possível para cada variável. Isso será obtido através de uma análise de fluxo de dados.

Certos critérios devem ser levados em conta quando se projeta um otimizador [ASU86].

Primeiro, o otimizador deve preservar o "significado" de um programa, ou seja, para uma

[1] Library Language

mesma entrada, o programa otimizado deve produzir o mesmo resultado do programa não otimizado. Isto implica que ao tomar uma decisão na alteração de um programa, o otimizador deve sempre optar pela abordagem segura.

Segundo, na média o otimizador deve acelerar os programas (embora em alguns casos o objetivo seja diminuir o tamanho do código, independente do tempo de execução).

O esforço para se escrever o otimizador deve valer a pena, isto é, as melhorias devem ser importantes para o tempo de execução ou tamanho do código.

No capítulo 2 são descritas as características principais de $\mathcal{LL}$, sobretudo no que diz respeito à invocação de rotinas e otimização de código. No capítulo 3, uma breve discussão sobre análise de fluxo de dados visa, além de dar noções gerais sobre esta ferramenta de otimização, unificar a notação dos métodos a serem descritos nos capítulos seguintes. No capítulo 4 temos a descrição de alguns métodos de inferência de tipos, representativos de algumas categorias de otimizadores. Os capítulos 5 e 6 contêm a descrição do método utilizado para a inferência de tipos e sua adaptação para a linguagem $\mathcal{LL}$. A descrição da forma como este método foi implementado em C++ pode ser encontrada no capítulo 7 e resultados e conclusões no capítulo 8.

Capítulo 2

Características principais de $\mathcal{LL}$

$\mathcal{LL}$ possui um conjunto pequeno de conceitos de programação. Sua sintaxe é simples e possui um mecanismo eficiente de interface com bibliotecas (o que inspirou o seu nome).

$\mathcal{LL}$ é totalmente baseada no conceito de objetos: todos os valores manipulados pela linguagem têm um tratamento uniforme. Isto é verdade mesmo para os tipos mais comuns, como inteiros, reais ou cadeias de caracteres, o que quer dizer que mesmo estes últimos podem ser redefinidos, conforme a aplicação[1].

A linguagem é dotada de um mecanismo simples de herança. Através deste mecanismo, subtipos podem receber novas rotinas, as quais podem ser introduzidas ou sobrepostas às do tipo "pai" (permite sobrecarga). Outras características de herança não foram incluídas pois $\mathcal{LL}$ não "sabe" nada sobre a representação interna dos objetos no sistema subjacente.

Variáveis em $\mathcal{LL}$ não precisam ser declaradas e também não precisam ter tipo fixo. Quando variáveis forem declaradas com tipo, a verificação será feita, a princípio, em tempo de execução. Operadores denotam métodos e, como funções, procedimentos, construtores e métodos podem ser sobrecarregados[2]. Expressões, assim como as construções da linguagem — repetições, condições, seleções — e construção de rotinas são similares a outras linguagens do mesmo tipo. A unidade básica de compilação e visibilidade é o módulo. Cláusulas **import** e **export** são providas para especificar os nomes acessíveis entre módulos distintos.

Uma descrição mais detalhada da linguagem pode ser encontrada nas referências [KB94a] e [KB94b]. Um compilador para a linguagem, gerando código em C++, está disponível em [Bac95].

Nas figuras 2.1 e 2.2 podem ser encontrados dois exemplos de utilização da linguagem.

[1] Tais tipos são normalmente implementados no módulo especial *System*.
[2] *overloaded* em inglês

```
procedure BubbleSort(L) is
   forall i in -(Succ(First(L)) .. Last(L)) do
      forall j in First(L) .. Pred(i) do
         if L[j]>L[Succ(j)] then
            t ← L[j]; L[j] ← L[Succ(j)]; L[Succ(j)] ← t
         endif
      endforall
   endforall
endprocedure;
```

Figura 2.1: Exemplo de programa em $\mathcal{LL}$: *sort*

```
function MaxElement(L) is
   max ← L[First(L)];
   forall v in Values(L) do
      if max<v then max ← v endif
   endforall;
   return max
endfunction;
```

Figura 2.2: Exemplo de programa em $\mathcal{LL}$: *maxelement*

Nestes exemplos podem ser vistos alguns dos benefícios da não declaração de tipos. Como parâmetro L, podem ser passados objetos de quaisquer tipos que implementem os métodos *First, Last, Succ, Values* e *Pred* e os operadores $[\,]$, $<$, $>$, "–" e "..". Por exemplo, a rotina pode ser usada para ordenar uma lista ou um vetor. É interessante notar como o operador unário "–" inverte o intervalo na segunda linha do *BubbleSort*.

Como conseqüência da sua concepção, a execução de programas em $\mathcal{LL}$ requer identificação (*binding*) dinâmica dos métodos aplicados aos objetos. Quer dizer, em tempo de execução deve-se inferir o tipo do primeiro argumento de uma invocação a um método para se identificar qual implementação deve ser chamada. Isto acontece mesmo no caso de tipos mais comuns como inteiros ou reais. Este fato pode acarretar um aumento apreciável no tempo de execução de programas quando comparados com programas escritos diretamente em linguagens como C++.

O objetivo do projeto aqui proposto é a introdução de uma fase de otimização no compilador $\mathcal{LL}$, cuja função será a identificação estática, sempre que possível, dos tipos associados às variáveis e às expressões do programa. Esta identificação, quando possível, resultará em geração de código estático para as chamadas de métodos. O problema básico a ser tratado é a inferência de tipos para variáveis e expressões da linguagem. Isto se dará através de uma análise de fluxo de dados.

Como resultado da otimização poderíamos obter simplificações exemplificadas a seguir:

```
module Example is
...
begin
   ...
   a ← a + b;
   ...
endmodule
```

O código gerado em C++ para o exemplo é mostrado na Figura 2.3 e trata-se de uma tradução trivial para uma máquina de pilha. Supõe-se que nenhuma otimização é executada. Notar que, em princípio, todos os objetos em $\mathcal{LL}$ são implementados como pares (*LLType*) compostos de uma marca de tipo e o valor ou o seu apontador.

```
void _Init_Example ()
{
  LLtype a;
  LLtype b;
  ...
  RTS_push(a);
  RTS_push(b);
  BINDER(OperatorPlus_f2c,2);
  a = RTS_pop();
  ...
}
```

Figura 2.3: Tradução do corpo do módulo

Nas Figuras 2.4 e 2.5 são mostrados os corpos das funções que implementam o interpretador para o exemplo, tanto o geral como o específico para o tipo *integer*.

```
void BINDER(int method_code, int nargs)
{
  switch (RTS_get_top_type_code(nargs))
  {
    case 1: ...
    ...
    case _t_System_integer_c:
              _System_integer_binder(method_code,nargs);
              break;
    ...
  }
}
```

Figura 2.4: O interpretador geral para o exemplo

O interpretador geral (BINDER) descobre o tipo do primeiro argumento e, caso seja *inte-*

ger, chama o interpretador particular (`_System_integer_binder`)[3]. Este segundo descobre a operação solicitada e chama a função que faz a soma dos dois inteiros e coloca o resultado na pilha de execução. A função `_System_integer_OperaratorPlus_f2c` é gerada a partir da especificação da implementação do método "+" para o tipo *integer* fornecida ao sistema.

```
void _System_integer_binder (int method_code, int nargs)
{
  switch (Applicable (method_code,_t_System_integer_c))
  {
    case 0: RunTimeError();
    case 1: ...
    ...
    case binary_plus_code: _System_integer_OperatorPlus_f2();
                           break;
  }
}


void _System_integer_OperatorPlus_f2()
{
  implIntegerPlus();
}
```

Figura 2.5: O interpretador para o tipo *integer*

Se o otimizador puder inferir que o tipo da variável *a* é *integer* no ponto do programa imediatamente anterior à atribuição, pode-se fazer a substituição da chamada ao "binder" pela rotina apropriada. Note que esta otimização é independente da interpretação semântica do operador "+" (Figura 2.6).

```
void _Init_Example ()
{
  LLtype a;
  LLtype b;
  ...
  RTS_push(a);
  RTS_push(b);
  _System_integer_OperatorPlus_f2c();
  a = RTS_pop();
  ...
}
```

Figura 2.6: Otimização sem a interpretação semântica

Uma otimização mais profunda poderia gerar simplesmente o código C++, como pode ser visto na Figura 2.7. Neste caso, o otimizador precisaria descobrir que as variáveis *a* e *b* são

[3]Note-se que o argumento poderia ser, por exemplo, do tipo *string*, com a operação "+" denotando a concatenação.

integer, não somente no ponto da atribuição, mas em toda a rotina, além de "saber" que o tipo *integer* utilizado corresponde ao tipo `int` em C++. Este seria um estágio posterior de otimização.

```
void _Init_Example ()
{
  int a;
  int b;
  ...
  a = a + b;
  ...
}
```

Figura 2.7: Otimização com interpretação do tipo *integer*

Existem vários métodos para inferir tipos em linguagens de programação, sendo alguns apresentados no capítulo 4. Neste trabalho, adotamos o método de Kaplan-Ullman apresentado nos capítulos 5 e 6.

Capítulo 3

Análise de Fluxo de Dados

Uma ferramenta importante na otimização de código é a análise de fluxo de dados (AFD)[1]. Esta análise consiste em um levantamento de informações úteis, distribuídas em todo o corpo de um procedimento ou programa, veiculando-as até pontos em que elas possam ser utilizadas [Sil84]. Tais informações podem ser expressas por meio de objetos (geralmente conjuntos) e propagadas pelo programa através de equações [JM76].

Um programa, para muitos tipos de problemas, pode ser expresso em termos de um grafo dirigido. Neste grafo, os vértices são seqüências de comandos com uma única entrada e uma única saída (chamados de blocos básicos). Arestas indicam fluxos de execução entre blocos: se b_2 pode seguir diretamente b_1 em alguma execução então existe uma aresta dirigida de b_1 para b_2. Diz-se que b_2 segue b_1 (b_1 precede b_2). Um bloco pode ter mais de um sucessor ou predecessor. Ao conjunto de sucessores e predecessores de um bloco b chamamos Δ_b e ∇_b respectivamente (ver Figura 3.1). Blocos básicos são seqüências (preferencialmente maximais) de comandos com as seguintes propriedades:

1. Nenhum comando é um desvio, a não ser possivelmente o último.

2. Qualquer desvio para esta seqüência se dará somente para o primeiro comando.

Cada lugar, antes e depois de um vértice, é chamado "ponto" de um programa, também referenciados por "ponto de entrada" e "ponto de saída" de um bloco.

Para ilustrar tais conceitos, um exemplo interessante é o "problema das variáveis ativas". Tal problema consiste em determinar para cada ponto do programa quais as variáveis que precisam ser armazenadas. Uma variável v é dita "ativa" em um determinado ponto p se existe um caminho no grafo do programa, iniciando em p e terminando em algum ponto q onde há um uso do valor de v, sem que exista neste caminho, alguma atribuição a v. Decorre que v precisa ter espaço para armazenamento em p pois seu conteúdo será usado em q. Se todos os caminhos que levam de p para algum uso de v possuem uma atribuição a v, diz-se que v não está ativa em p, pois seu valor será sobreposto até a próxima utilização, qualquer que seja o fluxo de

[1] Análise de fluxo de dados está descrita de várias maneiras em diversos trabalhos, entre eles [ASU86]. Utilizaremos a notação empregada por [Kow86].

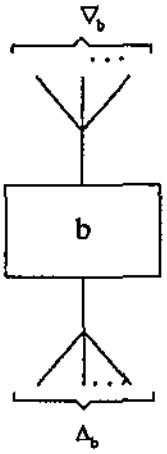


Figura 3.1: Bloco básico

execução. Entende-se por utilização, o uso do valor da variável em alguma expressão. A solução do problema consiste em encontrar o conjunto de variáveis ativas para cada ponto do programa.

Sejam X_b e Y_b os conjuntos das variáveis ativas no início e no final do bloco básico b respectivamente. Define-se h_b como sendo a contribuição do bloco b para o conjunto das variáveis ativas, isto é, as variáveis usadas em b tal que seu uso não é precedido de uma atribuição no próprio bloco b; e g_b como sendo o conjunto das variáveis preservadas em b, isto é, as variáveis que não são atribuídas em b. Os conjuntos g_b e h_b podem ser determinados simplesmente pela inspeção dos comandos do bloco básico b. Pode-se relacionar então os conjuntos das variáveis ativas no início e no final de cada bloco:

$$\begin{aligned} X_b &= Y_b \cap g_b \cup h_b \\ Y_b &= \bigcup_{b' \in \Delta_b} X_{b'} \end{aligned}$$

Um esquema pode ser visto na Figura 3.2.

A forma como as equações são montadas depende do tipo do problema a ser tratado. Em certos tipos de problemas a informação flui de cima para baixo, isto é, para se concluir algo sobre algum comando utiliza-se das informações dos comandos que o antecedem; em outros o processo ocorre ao contrário. Estes problemas são conhecidos como progressivos e regressivos, respectivamente.

O problema de "variáveis ativas" é um problema regressivo. As equações para o problema progressivo são similares. Como a informação propaga-se de cima para baixo, o conjunto Y_b torna-se a união dos predecessores de b. As demais equações permanecem iguais (veja Figura 3.3).

Outros tipos de problemas podem ter outras variações nas equações. Uma outra possibilidade é quando deseja-se que o valor "recebido" por um bloco básico seja a *intersecção* dos posteriores (ou antecessores) e não a união como foi mostrado acima. Tais problemas são chamados *disjuntivos*, enquanto os primeiros são chamados *conjuntivos*.

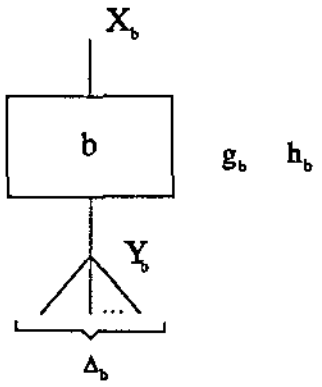


Figura 3.2: Bloco básico com seus coeficientes

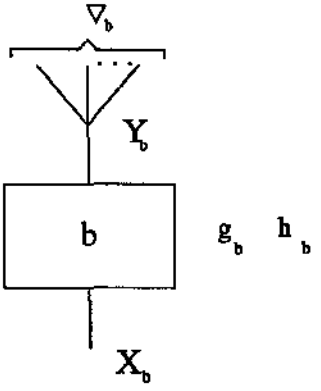


Figura 3.3: Bloco básico com seus coeficientes para problemas progressivos

As modalidades de problemas podem ser resumidas na tabela da Figura 3.4. Na verdade, aplicando-se algumas transformações, todos os problemas podem ser reduzidos uns aos outros [Kow86].

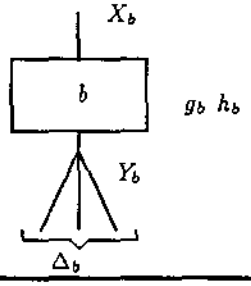
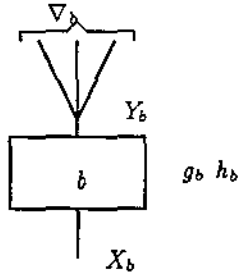
	Esquema	Disjuntivo	Conjuntivo
Regress.		$X_b = g_b \cap Y_b \cup h_b$ $Y_b = \bigcup_{b' \in \Delta_b} X_b$	$X_b = g_b \cap Y_b \cup h_b$ $Y_b = \bigcap_{b' \in \Delta_b} X_b$
Progres.		$X_b = g_b \cap Y_b \cup h_b$ $Y_b = \bigcup_{b' \in \nabla_b} X_b$	$X_b = g_b \cap Y_b \cup h_b$ $Y_b = \bigcap_{b' \in \nabla_b} X_b$

Figura 3.4: Equações para análise de fluxo de dados

Tem-se então um sistema de equações, o qual pode ser resolvido de forma iterativa ou direta [KD94]. Em ambos os casos calculam-se primeiramente os coeficientes g e h.

No caso iterativo começa-se iterando com valores apropriados para os conjuntos X e Y (geralmente conjunto vazio para o problema disjuntivo e conjunto universo para o problema conjuntivo). Como as operações de transformação são monotônicas pode-se garantir que se obtém soluções maximais (ou minimais) em um número finito de iterações. Tais soluções são únicas.

Na abordagem direta montam-se sistemas de equações. Estas equações são associadas a cada um dos pontos do programa. Resolvendo-se este sistema, tem-se os conjuntos X e Y para cada comando. Esta solução é garantida ser ótima (maximal ou minimal) e é única [Kow86]. Tal método pode ser usado somente em certas condições. Em [Kow86], por exemplo, o programa deve ser estruturado para o método funcionar.

Nem todos os problemas de análise de fluxo de dados têm formulação tão simples por meio de equações booleanas como no problema das variáveis ativas. O problema da inferência de tipos é um exemplo. Porém algumas propriedades do sistema de equações permitem chegarmos às mesmas conclusões [KU77]. Nesta formulação as equações podem assumir formas mais gerais,

como por exemplo (para o problema regressivo):

$$\begin{aligned} X_b &= f_b(Y_b) \\ Y_b &= \bigcup_{b' \in \Delta_b} X_{b'} \end{aligned}$$

onde tanto f_b quanto $\bigcup$ têm certas propriedades que garantem a aplicabilidade do processo iterativo [KU77].

Capítulo 4

Métodos de inferência de tipos

4.1 Método de Palsberg e Schwartzbach

Uma forma de se fazer a inferência de tipos, descrita por Palsberg e Schwartzbach em [PS94], é por meio de sistema de restrições.

Em linguagens com declaração de tipos, como Pascal, anotações de tipos são encontradas em diversos pontos do programa. Para declaração de variáveis, procedimentos e seus parâmetros tais anotações de tipos são obrigatórias. Constantes têm seu tipo inferido no ponto da sua declaração. Informações sobre os tipos de expressões ao longo do programa podem ser deduzidas diretamente destas declarações.

A verificação da correção dos tipos utilizados pode ser feita, em geral, estaticamente e os erros são apontados em tempo de compilação. Tal verificação pode ser feita por meio de um conjunto de restrições deduzidas a partir das declarações e de cada ponto onde variáveis e constantes são utilizadas no programa.

A idéia é definir restrições, sobre todas as variáveis e expressões, que possam captar:

- informação sobre tipos declarados;
- a propagação de tipos ao longo do corpo do programa e
- requerimentos para correção estática de tipos.

Para expressar tais restrições, a seguinte notação é utilizada: $[E]$ denota a variável que contém o conjunto de tipos de E, onde E pode ser uma variável de programa, constante ou expressão mais complexa. Para ilustrar o método seguem algumas das regras de restrições para uma linguagem como Pascal:

"$expr_1 + expr_2$": A adição e demais operações aritméticas binárias produzem as seguintes restrições:

$$\begin{aligned} [expr_1] &= \{integer\} \\ [expr_2] &= \{integer\} \\ [expr_1] &= [expr_2] \\ [expr_1 + expr_2] &= \{integer\} \end{aligned}$$

if $expr_1$ **then** $expr_2$ **else** $expr_3$ **end:** O condicional produz as seguintes restrições:

$$\begin{array}{lcl} [expr_1] & = & \{boolean\} \\ [\textbf{if}\ expr_1 \textbf{then}\ expr_2 \textbf{else}\ expr_3 \textbf{end}] & = & [expr_2] = [expr_3] \end{array}$$

Por exemplo para o seguinte trecho de programa em uma linguagem como Pascal:

```
var x, y : integer;
var z : array [1..100] of boolean;

writeln(x+y);
z[87] := true;
```

Valem as seguintes restrições:

1. $[x] = integer$
2. $[y] = integer$
3. $[z] = array\ [1..100]\ of\ boolean$
4. $[x] = [y]$
5. $([x] = integer) \Rightarrow [x + y] = integer$
6. $([x] = real) \Rightarrow [x + y] = real$
7. $[87] = integer$
8. $[true] = boolean$
9. $[z[87]] = [true]$
10. $[z] = array\ [1..100]\ of\ [z[87]]$

As três primeiras restrições são deduzidas das declarações das variáveis. As demais, são conseqüência da utilização de expressões no corpo do programa. Note como a sobrecarga do operador + é captada pelas restrições condicionais 5 e 6.

O programa estará com utilização correta de tipos se o sistema tiver solução, em princípio, única, como é o caso do sistema acima. Note que se, por exemplo, a atribuição a `z[87]` fosse substituída por `z[87]:=10;`, a restrição $[z[87]] = integer$ seria inserida. Esta restrição é incompatível com a restrição gerada pela declaração, tornando o sistema sem solução.

Em linguagens com declaração obrigatória de tipos é simples fazer a verificação estática, visto que todas as expressões dependem de tipos conhecidos por serem declarados. O procedimento para tal verificação faz simplesmente um percurso pela árvore de programa mantendo informações globais e locais.

Algumas linguagens orientadas a objetos não permitem declarações com tipos de objetos ou não obrigam a sua utilização, por exemplo, SMALLTALK[GR84] e $\mathcal{LL}$. Nestas linguagens pode-se utilizar a mesma técnica usada na verificação da correção de tipos para se fazer a inferência de tipos. Quando o sistema de restrições for montado, ele não terá uma única solução.

Um método para resolver o sistema de restrições é descrito a seguir.

Soluções para o problema são tuplas de conjuntos de tipos: cada elemento da tupla corresponde a um conjunto de tipos possível para uma variável. Soluções parciais formam uma ordem parcial para a operação de inclusão de conjuntos ($\subseteq$). O menor elemento desta ordem é a tupla onde cada elemento é conjunto vazio; o maior elemento é o conjunto onde todas as variáveis podem ser de todos os tipos $(U, U, \ldots, U)$ (U denota o conjunto de todos os tipos) (ver Figura 4.1).

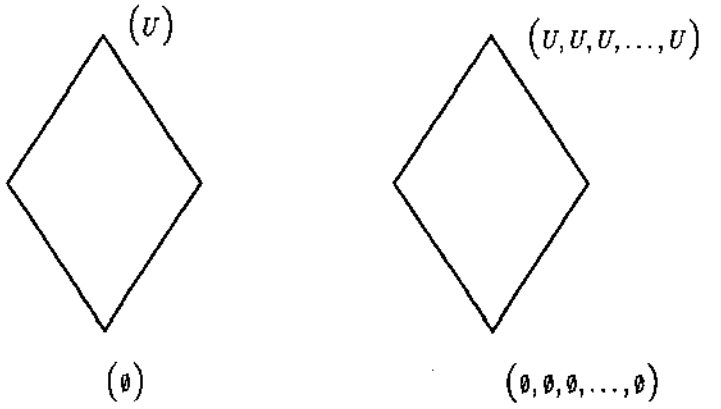


Figura 4.1: Ordem parcial das soluções

A tarefa é associar elementos dessa ordem parcial, isto é, possíveis soluções, ao conjunto de variáveis, de forma a manter a consistência. Satisfazer ao conjunto de restrições não é suficiente: deve-se procurar a solução mínima.

Palsberg e Schwartzbach demonstram que a operação de interseção é fechada para o conjunto de soluções (onde interseção para tuplas é entendida como a interseção entre cada um dos elementos da tupla).

Ainda sem conhecer a solução mínima para o problema, pode-se imaginar um conjunto $S = \{L_1, L_2, \ldots, L_m\}$ como o conjunto de todas as soluções para o sistema.

Uma possível solução é

$$L = L_1 \cap L_2 \cap \ldots \cap Lm.$$

Como L é menor ou igual a cada L_i e, pelo enunciado acima, a interseção entre duas soluções é uma solução para o problema, L é a solução mínima.

Um possível algoritmo consiste em calcular todas as combinações de tipos, extrair aquelas que são soluções e fazer a interseção entre elas. Este algoritmo é impraticável, pois considera todas as combinações, isto é, um número exponencial delas. Uma outra abordagem iterativa, parte de uma solução inicial (a solução onde cada variável possui um conjunto vazio de tipos,

ou seja, o menor elemento do reticulado da figura 4.1) e ir acrescentando, uma a uma, cada uma das restrições. A cada inserção a solução mínima é procurada. Quando todas as restrições estiverem sido acrescentadas o algoritmo terá a solução minimal.

Este método funciona bem para linguagens onde as variáveis tem tipos fixos ao longo de todo o programa, pois a análise chega a um conjunto de tipos para cada variável, independente do ponto do onde ela se encontra. Em linguagens como $\mathcal{LL}$, onde variáveis podem ter tipos diferentes em cada ponto do programa, este tipo de análise não traria bons resultados, pois chegaria a um conjunto possivelmente maior de tipos em alguns pontos do programa.

Para sua utilização em tais linguagens uma adaptação seria necessária. Cada variável teria um "nome diferente" (seria considerada uma outra variável) quando se sabe que ela pode ter mudado de tipo (o que acontece em uma atribuição). Uma segunda análise precisaria ser feita: para cada definição de uma variável (atribuição de valor), uma análise levantaria quais os pontos que esta definição alcança. Para cada variável, o programa pode ser particionado entre conjuntos de pontos que cada atribuição alcança. Em cada partição a variável pode ter um nome diferente e então o método pode ser utilizado.

4.2 Método de Jones e Muchnick

Outro método para resolver problemas de inferência de tipos baseado em equações é descrito por Jones e Muchnick [JM76]. Em seu artigo é descrita uma nova abordagem para desenvolvimento de linguagens de programação, usando a linguagem TEMPO como exemplo. Tal abordagem sugere inclusive o projeto de um tipo de processador que facilite a tradução de características da linguagem. Porém sua maior inovação é a consideração do *binding time* como um parâmetro que pode variar de programa para programa ou mesmo em pontos de um programa, isto é, considerar que uma variável pode ser atrelada a seu tipo em um momento posterior ao da compilação.

Tipos de dados são uma variedade de uma larga categoria de informações sobre variáveis as quais são chamadas atributos. Além do tipo, outros atributos importantes são a informação se a variável tem um tipo, se seu valor será utilizado mais tarde e o seu tamanho. Nesta descrição nos deteremos no aspecto *inferência de tipos*. Estas informações propagam-se tanto progressivamente quanto regressivamente. Por exemplo, da seguinte atribuição

$$A \leftarrow B + 4$$

podemos deduzir que A será inteira após a execução e B precisa ser inteiro antes do comando[1]. Um método para propagação de informações é descrito em [GW75]. Jones e Muchnick propõem um algoritmo iterativo que, explorando as características progressivas e regressivas e as características dos operadores, propaga as informações enumeradas acima para todos os pontos do programa.

A análise será útil para minimizar o espaço de alocação e o número de testes de tipo ao longo da execução do programa. O resultado será um par de funções F e B que para cada ponto do programa retornam um mapeamento entre variáveis e tipos. A definição de tais funções utiliza

[1] Supondo uma interpretação óbvia do operador "+".

os seguintes conjuntos

$$UC_c(X) = \{t \in S \mid c \text{ será executado corretamente com X sendo do tipo } t\}$$

$$SC_c(X) = \{t \in S \mid \text{a execução de } c \text{ pode colocar valores do tipo } t \text{ em X}\}$$

onde

c é um comando do programa;

X é uma variável;

t é um tipo;

S é um conjunto de tipos.

Por exemplo, na atribuição

$$c : A \leftarrow B + length(A)$$

temos:

- $UC_c(A) = \{string\}$;
- $UC_c(B) = \{integer\}$
  pois c só pode ser executado corretamente se A for do tipo *string* e B do tipo *integer*;
- $SC_c(A) = \{integer\}$;
- $SC_c(B) = \emptyset$.

A função F pode ser vista como uma função para monitorar a execução, verificando a utilização dos tipos. Por exemplo, se $F(I,X) - UC_c(X) \neq \emptyset$ então é necessário inserir um teste no programa executável para verificar a pertinência dos tipos de X no conjunto $UC_c(X)$ (I é o ponto que precede o comando c). É interessante que F seja suficientemente grande para ser segura e evitar erros mas ser ao mesmo tempo o menor possível para evitar testes inúteis (figura 4.2).

A função B é usada para verificar se os valores assinalados são compatíveis com futuros usos. Por exemplo, se $SC_c(X) - B(J,X) \neq \emptyset$ então após executar c um teste de pertinência de X em $B(J,X)$ é necessário (J é o ponto após c). Se por um lado deve-se procurar o menor F, quanto maior for B menor o número de testes requeridos (figura 4.3).

F é definida a seguir:

1. $F(I,X) = indefinido$ se I é o ponto inicial do programa.
2. $F(J_j,X) = \begin{cases} F_c^e(X) \text{ caso } X \text{ seja uma variável que não recebe atribuição em } c \\ B(J_j,X) \cap F_c^e(X) \text{ para atribuições do tipo "} X \leftarrow Y \text{"} \\ B(J,X) \cap SC_c(X) \text{ se } X \text{ recebe atribuição em } c \end{cases}$
   para cada ponto J_j posterior ao comando c.

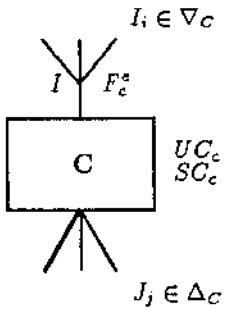


Figura 4.2: Diagrama da função F

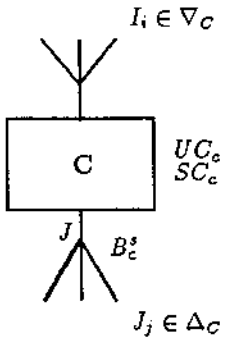


Figura 4.3: Diagrama da função B

onde $F_c^s = \bigcup_{I_i \in \nabla_c} F(I_i)$

A definição de B é:

1. $B(I_i, X) = \begin{cases} B_c^s \text{ se } X \text{ não aparece em } c \\ U \text{ se } X \text{ é atribuído em } c \text{ mas não é usado na expressão} \\ UC_c(X) \text{ se } X \text{ é usado e atribuído em } c \\ B_c^s(X) \cap B_c^s(Y) \text{ se } c \text{ é da forma "} Y \leftarrow X \text{"} \\ B_c^s(X) \cap UC_c(X) \text{ se } X \text{ é usado em } c \text{ mas não atribuído} \end{cases}$
   para cada ponto I_i anterior ao comando c

2. $B(J_j, X) = U$ se J_j é um ponto final do programa.

onde U é o conjunto de todos os tipos e $B_c^s = \bigcup_{J_j \in \Delta_c} B(J_j)$. Em alguns casos, esta equação pode ser substituída por $B_c^s = \bigcap_{J_j \in \Delta_c} B(J_j)$.

Os conjuntos SC_c e UC_c, ao contrário das funções, são dependentes da linguagem em questão. No caso de $\mathcal{LL}$ são dependentes, inclusive da definição do módulo System (Capítulo 2 e [Bac95]).

Uma definição possível de SC e UC para uma linguagem como Pascal poderia ser:

$SC_c(X) = t$ se c é do tipo "$X \leftarrow constante$" e t é o tipo da constante;

$SC_c(X) = U$ se c é uma atribuição de uma entrada para X;

$SC_c(X) =$ conjunto de tipos possíveis para o operador "op" se c é da forma "$X \leftarrow expr_1\ op\ expr_2$".

Definição de UC:

$UC_S(X) = UC_{expr}(X)$ se c é "$Y \leftarrow expr$", "$output \leftarrow expr$" ou 'if $expr$";

$UC_{expr}(X) = gen$ se X não aparece em $expr$ ou X é $expr$;

$UC_{expr_1\ op\ expr_2}(X) = UC_{expr_1}(X) \cap UC_{expr_2}(X)$ se X não for nenhuma das duas expressões;

$UC_{X\ op\ expr}(X) = UC_{expr}(X) \cap leftoperandtypes(op)$;

$UC_{expr\ op\ X}(X) = UC_{expr}(X) \cap rigthoperandtypes(op)$;

$UC_{X\ op\ Y}(X) = gen$ para operadores de comparação;

$UC_{X\ op\ expr}(X) =$ tipo de $expr$ para operadores de comparação e se $expr$ não é uma variável.

Com o programa a ser analisado em mãos e as definições acima monta-se o sistema de equações, que deve ser resolvido iterativamente. A função B é resolvida primeiro, por ser independente de F. Com o seu resultado pode-se então calcular os valores da função F para cada ponto do programa. Podemos expressar este processo por

$$K = F(B(\emptyset))$$

Kaplan-Ullman, como se vê no capítulo 5, faz cálculos parecidos com os vistos aqui, porém a função resultado é

$$K = (B^*\ F^*)^*(\emptyset)$$

As aproximações feitas em cada aplicação de F são imediatamente aproveitadas em B e vice-versa. A análise regressiva não é menos importante que a progressiva. Um outro ponto onde Kaplan-Ullman podem produzir resultados mais precisos é o aproveitamento da relação que há entre os argumentos. Do exemplo:

$$c : ST \leftarrow ST\ \ concat\ \ Y$$

podemos deduzir que tanto ST quanto Y são cadeias de caracteres no ponto posterior à atribuição. Jones e Muchnick, por só considerarem a variável atribuída (ST no exemplo), não restringirá os tipos de Y (ver ponto 1 da definição de F).

4.3 Outras abordagens

4.3.1 Abordagem de Chambers e Ungar

Self [CU89, US87], uma linguagem desenvolvida em Stanford, é um exemplo de uma linguagem orientada a objetos pura com tipos dinamicamente escolhidos. Contém, segundo os autores, ainda menos informação estática que SMALLTALK. Objetos em Self são conjuntos de *slots*, cada qual com referência para um outro objeto os quais podem ter código (neste caso são chamados métodos) ou não (neste caso funcionam como variáveis). Alguns *slots* são designados como *parent slots* para possibilitar a herança. Variáveis são declaradas sem tipo e não precisam apontar sempre para a mesma classe de objetos. Nem mesmo *parent slots* precisam apontar sempre para o mesmo objeto.

A abordagem usada para a inferência de tipo é iterativa, embora sem considerar a propagação bidirecional das informações (considera somente a propagação progressiva) [CU90]. Tais informações são usadas para se evitar a consulta a tabelas na chamada de métodos (evitar o *binding* dinâmico). Outra melhoria que pode ser obtida com a inferência de tipos é a possibilidade de *splitting*: quando o conjunto de tipos para uma variável é pequeno e invariável em um bloco (por exemplo um *loop*) pode-se gerar um versão deste bloco para cada tipo do conjunto, acrescentando um seletor.

Uma outra alternativa usada por Chambers e Ungar para contornar o *bindig* dinâmico é a previsão de tipos. Quando não é possível descobrir o conjunto de tipos para uma variável o compilador ainda assim pode fazer algumas assertivas. Por exemplo, em uma operação como $x + y$ há uma chance muito maior de os operadores serem inteiros. Pode-se gerar uma versão do bloco onde a operação se encontra considerando o tipo como sendo inteiro e outra versão para os demais tipos.

4.3.2 Abordagem de Vitek, Horspool e Uhl

Vitek et al. [VHU92] propõem um algoritmo de inferência de tipos genérico para linguagens de variáveis com tipos fracos. As informações obtidas podem ser usadas para:

- transformar as chamadas de métodos dinâmicas em estáticas;
- fazer testes de parâmetros em tempo de compilação;
- fazer *inline* de corpo de métodos.

A análise, realizada somente no sentido progressivo, faz a associação de cada ponto do programa a um grafo dirigido no qual nós são objetos e classes e arestas (rotuladas) indicam possibilidade de tipo: uma aresta do nó X para o nó Y com rótulo z indica que o atributo z do objeto X pode ser da classe Y. Cada nó é visitado e o grafo é atualizado de acordo com as informações provenientes dos seus predecessores iterativamente até atingir um ponto fixo.

Um aspecto interessante deste trabalho é a forma como chamadas de procedimentos são tratadas. Os procedimentos são analisados na ordem e na forma na qual eles são invocados e

não na ordem na qual eles são declarados. Ou seja, quando uma invocação de procedimento é encontrada o corpo deste é analisado; se uma outra invocação é encontrada a análise do atual é suspensa e outra análise é iniciada, gerando uma cadeia de invocações. Tal cadeia é possivelmente infinita. Para resolver este problema os autores usam um número K limitante do crescimento da cadeia. A escolha de K é uma relação de compromisso entre tempo de execução e precisão dos resultados.

Esta abordagem apresenta a desvantagem, em relação a abordagem de Kaplan e Ullman, de fazer somente a análise progressiva.

4.4 Escolha do método

Embora cada uma das abordagens apresente vantagens e desvantagens (apresentadas acima), consideramos que a propagação regressiva das informações era um dos aspectos mais importantes a ser levado em conta na escolha do método. Outro aspecto importante é descrito por Khedker e Dhamdhere em [KD94] diz respeito a utilização de métodos diretos em problemas biderecionais. Segundo eles para este tipo de problema somente métodos iterativos podem ser utilizados.

Estas constatações nos levaram a adotar o método de Kaplan e Ullman (Capítulo 5) para a inferência de tipos em linguagem $\mathcal{LL}$.

Capítulo 5

Abordagem de Kaplan e Ullman

Kaplan e Ullman [KU78, KU80] propõem um esquema para inferência de tipos em linguagens como APL, SETL e SNOBOL — linguagens onde objetos possuem tipos fortes mas variáveis não os possuem, analogamente a $\mathcal{LL}$. O objetivo deste enfoque é fazer um mapeamento entre variáveis e subconjuntos de tipos para cada ponto do programa.

Dependendo do resultado da inferência de tipos, o compilador pode gerar um código mais eficiente nos seguintes pontos:

"binding" estático: gerar código onde as chamadas de procedimentos sejam feitas diretamente, sem a necessidade de se chamar um "binder" o qual fará a decisão de qual é a rotina desejada.

alocação eficiente: conhecendo-se o tipo que uma variável pode assumir ao longo de todo o procedimento pode-se fazer a alocação de espaço diretamente, sem a necessidade de indireção (tipo *LLType*).

Somente os comandos de atribuição são tratados pela abordagem. Qualquer outro comando pode ser reduzido a seqüências de comandos de atribuição. Isto é feito por meio de variáveis fictícias, como no caso de comandos condicionais exemplificado na Figura 5.1, onde o comando **if** foi transformado em um comando de atribuição com dois sucessores.

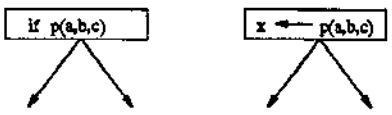


Figura 5.1: Comando condicional **if**

A seguir são apresentadas algumas notações usadas na abordagem a ser discutida ao longo deste capítulo.

1. T – um conjunto finito de tipos.

2. S – conjunto de subconjuntos de T.
3. V – conjunto finito de variáveis de um programa ou rotina.
4. $t : V \rightarrow S$ – mapeamento dos tipos possíveis das variáveis em um dado um ponto do programa.
5. $R_{\otimes}$ – para cada operador k-ário $\otimes$, supõe-se que é dada uma função

$$R_{\otimes} : T^k \rightarrow S$$

Esta função retorna, em função dos tipos dos k operandos, os tipos possíveis do resultado da operação $\otimes$.

6. $R_{i\otimes} : T^k \rightarrow S$ – retorna os tipos possíveis para o i-ésimo operando, em função dos tipos dos demais.
7. $T_{i\otimes} : T^k \rightarrow S$ – retorna os tipos possíveis para o i-ésimo operando, em função dos tipos do resultado esperado para a expressão e dos tipos dos demais operandos.

5.1 Problema de inferência de tipos

O ponto inicial no processo de inferência de tipos é a determinação dos tipos de expressões. Tendo esta fase como base pode-se inferir tipos para outros pontos de interesse.

Conhecendo-se os tipos de X_i pode-se usar a função $R_{\otimes}$ para se deduzir o tipo de $\otimes(X_1, X_2, \cdots, X_n)$. Este tipo de dedução geralmente é usado para deduzir tipo de variáveis a partir do tipo da expressão. Em alguns casos o que tem-se é exatamente o contrário. Sabe-se o tipo esperado para uma variável que está no lado esquerdo de uma atribuição, isto é, sabe-se qual o tipo que ela deve ter após o comando, e deseja-se inferir alguma coisa sobre os tipos das variáveis que estão do lado direito desta atribuição no ponto anterior à atribuição.

A conclusão a que se chega destas duas considerações é que o problema de inferência de tipos é um problema progressivo e ao mesmo tempo regressivo[1]. Deve-se fazer a combinação de duas análises de fluxo de dados para se obter a solução mais precisa.

A análise progressiva faz a inferência dos tipos resultantes para as variáveis, isto é, dado um certo comando Q, esta análise descobre o tipo que as variáveis devem ter após a execução de Q. Para isso, deve levar em consideração as suas operações. Por exemplo, dado o comando $(Q : v \leftarrow a + b)$, a variável a aparece em uma expressão, operada com o operador $+$. Pode-se inferir que a será de um tipo numérico ou cadeia de caracteres (mesmo que nada se saiba sobre seus tipos).[2]

A análise regressiva é necessária para inferir os tipos requeridos para variáveis referidas, isto é, olhando-se de "trás para frente", sabendo o tipo que uma variável atribuída deve ter após um certo comando, inferir os tipos das outras variáveis antes deste comando. Em um exemplo: se em um comando de atribuição $(Q : v \leftarrow a + b)$ sabe-se que v deve ser inteira após o comando, pode-se inferir que tanto a variável a quanto b devem ser inteiras antes de Q.

[1] Esta mesma propriedade já foi constatada na seção 4.2 (Jones e Muchnick).

[2] Supondo que o operador $+$ exista tanto para soma de inteiros e reais como para concatenação de cadeias.

5.2 Problema progressivo

Este problema consiste no seguinte: dado um certo comando $Q : v \leftarrow \otimes(x_1, x_2, \cdots, x_n)$ do programa e um mapeamento t entre variáveis e tipos no ponto imediatamente anterior a Q, define-se uma função $f_Q : (V \rightarrow S) \rightarrow (V \rightarrow S)$ a qual retorna o mapeamento para os tipos das variáveis no ponto imediatamente posterior a Q[3]. A definição de f_Q é:

1. $f_Q(t)(z) = t(z)$ se z não ocorre em Q, isto é, não há modificação nos tipos possíveis para z.
2. $f_Q(t)(v) = p$.
3. $f_Q(t)(x_i) = q_i$.

onde tanto p quanto os q_i são determinados a partir de $t(x_i)$, $R_\otimes$ e $R_{i\otimes}$.

Por exemplo, consideremos que existem apenas os tipos $\{integer, real, string\}$ e que a tabela de R_+ é a seguinte:

		R_+
integer	*integer*	*integer*
real	*real*	*real*
string	*string*	*string*

e seja o comando

$$Q : a \leftarrow b + c$$

Supõe-se também que o mapeamento t anterior a Q é dado por:

$$t(a) = t(c) = \{integer, real, string\}$$
$$t(b) = \{integer, real\}$$

ou seja, não há informação sobre os tipos de a e c. Pode-se concluir então, que o mapeamento $f_Q(t)$ após o comando Q é dado por:

$f_Q(t)(a) = \{integer, real\}$ (regra 2).

$f_Q(t)(b) = \{integer, real\}$ (regra 3).

$f_Q(t)(c) = \{integer, real\}$ (regra 3).

Para todas as outras variáveis z, $f_Q(t)(z) = t(z)$ (regra 1).

Esta definição da função f_Q corresponde à determinação generalizada das constantes h_b e g_b do capítulo 3 (figura 5.2).

[3] Iremos trabalhar com uma notação mais simples que a tratada no artigo, o qual trata múltiplas atribuições em um único comando.

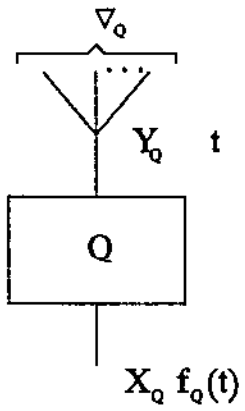


Figura 5.2: Bloco básico com suas equações

A solução para o problema progressivo é obtida adotando-se valores iniciais (normalmente $\emptyset$) para os mapeamentos que precedem os comandos do programa ou da rotina. A seguir, entra-se num processo iterativo em que todos os mapeamentos são recalculados aplicando-se as funções f_Q acima. Sempre que um comando tem mais que um predecessor, aplica-se a operação de união a cada conjunto de tipos (note que o problema é disjuntivo). Assim, em um comando como o da Figura 5.3 teremos:

$$t(z) = f_{Q_1}(t_1)(z) \cup f_{Q_2}(t_2)(z)$$

para toda variável z.

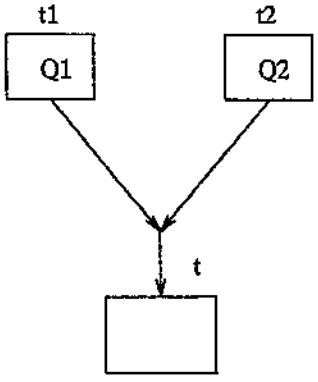


Figura 5.3: Comando com dois predecessores

A fim de simplificar a discussão adotaremos uma notação matricial. Seja $\{Q_1, Q_2, \ldots, Q_n\}$ o conjunto de comandos de atribuição (blocos) do programa ou rotina em questão e seja F a matriz quadrada de ordem n de funções f_{ij} onde:

$$f_{ij} = \begin{cases} f_{Q_j} & \text{se} \quad Q_j \in \nabla_{Q_i} \\ f_\emptyset & \text{caso contrário} \end{cases}$$

($f_\emptyset$ é o mapeamento que atribui o conjunto vazio de tipos a todas as variáveis.)

Seja $s = (t_1, t_2, \ldots, t_n)$ o vetor de mapeamentos nos pontos que precedem imediatamente os respectivos comandos Q_i e seja

$$s' = (t'_1, t'_2, \ldots, t'_n)$$

onde

$$t'_i = \bigcup_{j=1}^{n} f_{ij}(t_j)$$

Escrevemos, então, $s' = F(s)$.

Utilizando esta notação, a solução do sistema de equações será dada por $F^*(\emptyset)$.

Por outro lado, o seguinte lema é demonstrado em [KU78]: se s é um mapeamento (geral) seguro, e se $x \supseteq F(x) \cap s$, então x também é um mapeamento seguro. Nesta notação os símbolos $\supseteq$ e $\cap$ são estendidos para mapeamentos gerais, isto é, vetores de mapeamentos. Entende-se por mapeamento seguro o mapeamento onde para cada variável é associado um conjunto de tipos suficientemente grande para conter todas as possibilidades ao longo de todas as execuções do programa.

Define-se a função $F_s(x)$ como

$$F_s(x) = F(x) \cap s$$

De acordo com o lema acima, para procurar o menor x tal que $x = F(x) \cap s$, faz-se a aplicação de $F_s^*(\emptyset)$, isto é, o ponto onde $v = F_s(v)$ (o ponto fixo de F_s). $F_s^*(\emptyset)$ pode ser definido como:

$$F_s^*(\emptyset) = \lim_{i \to \infty} F_s^i(\emptyset)$$

onde $\emptyset$ denota o mapeamento onde todas as variáveis em todos os pontos do programa não estão associadas a nenhum tipo[4].

Pode-se demonstrar que este processo converge para um ponto fixo depois de um número finito de iterações. O resultado deste processo iterativo já seria uma solução parcial para o problema de inferência de tipos. Ou seja, se uma variável v pode, em alguma execução, ser do tipo t no ponto p do programa, o mapeamento da variável v para este ponto conterá o tipo t. Porém, se o processo de inferência for encerrado por aqui, o mapeamento poderá conter também muitos tipos que a variável com certeza nunca assumirá.

Pode-se verificar facilmente que o número máximo de iterações necessárias para convergir é da ordem de $n_p n_v n_t$ onde

n_p = tamanho de programa (número de comandos de atribuição)
n_v = número de variáveis
n_t = número de tipos

[4] O mapeamento s é um limitante para a solução obtida. Por sua vez, esta solução será um novo limitante para uma aplicação de análise regressiva, conforme será visto adiante.

5.3 Problema regressivo

Sua definição é semelhante à anterior: dado um certo comando $Q : v \leftarrow \otimes(x_1, x_2, \cdots, x_n)$ do programa e um mapeamento t das variáveis e tipos no ponto imediatamente posterior a Q, define-se uma função $b_Q : (V \rightarrow S) \rightarrow (V \rightarrow S)$ que retorna o mapeamento para os tipo das variáveis no ponto imediatamente anterior a Q.

Segue definição:

1. $b_Q(t)(z) = t(z)$ se z não ocorre em Q, isto é, não há modificação nos tipos possíveis de z.
2. $b_Q(t)(v) = T$ se v ocorre somente no lado esquerdo da atribuição; isto significa que se for atribuído a v algum valor no ponto Q, do ponto de vista do problema regressivo, antes de Q, v poderia assumir valores de qualquer tipo.
3. $b_Q(t)(x_i) = q_i$, onde os q_i podem ser determinados a partir dos $t(x_j)$ e do $t(v)$.

Considerando o mesmo exemplo da seção anterior, com a mesma tabela para R_+ e com o mapeamento t após o comando Q:

$$t(a) = \{integer, real\}$$
$$t(b) = \{integer, real, string\}$$
$$t(c) = \{integer, string\}$$

pode-se concluir o seguinte:

1. $b_Q(t)(a) = T$ (regra 2).
2. $b_Q(t)(b) = b_Q(t)(c) = \{integer\}$, isto é, em função dos tipos de a, b, c e do operador $+$ depois do comando Q, pode-se concluir que b e c poderiam ser somente $\{integer\}$ antes de Q (regra 3).

A solução, considerando somente o problema regressivo, é obtida adotando-se os valores iniciais (normalmente $\emptyset$) como mapeamento para os comandos do programa. A seguir, como no problema progressivo, entra-se em um processo iterativo em que os mapeamentos são recalculados aplicando-se as funções b_Q. Sempre que um comando tem mais que um sucessor, aplica-se a operação de união a cada conjunto de tipos.

Analogamente ao problema progressivo, demonstra-se o lema: se $x \supseteq B(x) \cap s$, onde s é um mapeamento seguro, então x também é um mapeamento seguro.

Também são válidas as seguintes funções:

$B(x)$ aplicação de b_Q em cada ponto do programa, perfazendo as uniões necessárias quando há mais de um sucessor ou predecessor;

$B^*(x)$ ponto fixo de $B(x)$;

$B_s(x) = B(x) \cap s$;

$B_s^*(x)$ ponto fixo de $B_s(x)$, mais preciso que $B^*(x)$.

Como no problema progressivo, a solução obtida com $B_s^*(\emptyset)$ é uma solução parcial para o problema regressivo, ou seja, em cada ponto do programa, os mapeamentos das variáveis contém todos os tipos que a variável poderá vir a ter.

Podemos também demonstrar que este processo converge para um ponto fixo após um número finito de iterações, semelhante ao problema progressivo.

5.4 Integração das análises progressiva e regressiva

As soluções obtidas por $F_s^*(\emptyset)$ não podem ser melhoradas por uma nova aplicação de F_s, visto que tais soluções são um ponto fixo. O mesmo ocorre na abordagem regressiva com a função B_s. A inferência de tipos para um programa é feita alternando-se aplicações dos dois processos, progressivo e regressivo, até que se convirja para um valor fixo.

Em cada iteração do método, progressivo ou regressivo, necessita-se de um mapeamento seguro para se tomar como limitante superior para o crescimento das funções F e B. Quanto menor for tal mapeamento, mais preciso será o resultado da aplicação.

Um mapeamento inicial seguro óbvio pode ser o mapeamento onde todas as variáveis podem ser de todos os tipos em todos os pontos do programa (denotado por 1). Este é o mapeamento para a primeira aplicação da análise (que pode ser tanto a progressiva F_s^* quanto a regressiva B_s^*). A segunda aplicação pode ser realizada tendo como limitante superior o resultado da anterior, que é um mapeamento seguro e menor que o inicial. Assim, sucessivamente, segue-se nesta iteração até que se chegue a um ponto fixo.

Algoritmicamente, o problema progressivo pode ser resolvido definindo-se a função $\mathcal{F}$ em função de um mapeamento s seguro.

$$\mathcal{F}(s) = F_s^*(\emptyset)$$

ou

$$\begin{aligned}
\mathcal{F}(s) \;=\; \{\;& v \leftarrow \emptyset \\
& \textbf{repeat} \\
& \quad v' \leftarrow v; \\
& \quad v \leftarrow s \cap F(v); \\
& \textbf{until } v' = v; \\
& \textbf{return } v; \\
\}\;&
\end{aligned}$$

A função $\mathcal{B}$ pode ser definida analogamente. Define-se então a função K como

$$K(s) = \mathcal{B}(\mathcal{F}(s))$$

A solução do problema é dada por

$$K^*(1)$$

ou numa versão algorítmica

$s = 1;$
repeat
    $s' = s;$
    $y = F_s^*(\emptyset);$
    $s = B_y^*(\emptyset);$
until $s' = s;$

Kaplan e Ullman mostram que este algoritmo obtém o melhor resultado possível com as informações que dispõe.

Pode-se verificar que o número total de iterações das funções $\mathcal{F}$ e $\mathcal{B}$ é limitado por $n_p^2 n_v^2 n_t^2$. Considerando como operação elementar a inclusão ou exclusão de um tipo em um mapeamento de uma variável, o número total de operações é limitado por $n_p^3 n_v^3 n_t^3$. Na prática este limite dificilmente seria atingido.

Uma melhoria sugerida por Kaplan e Ullman e introduzida no algoritmo é fazer com que os mapeamentos utilizados sejam os produzidos na iteração atual e não os da iteração anterior. Pode-se provar que desta forma se converge, em geral, de maneira mais rápida. Neste caso a interseção com o mapeamento limitante precisa ser feita antecipadamente, isto é, ao invés de se fazer na função $F_s(x) = F(x) \cap s$, em cada função $f(x)$ a interseção já precisa ser feita.

5.5 Um exemplo

Mostramos nesta seção um exemplo mais completo para ilustrar os conceitos tratadas pelo trabalho de Kaplan e Ullman. Trata-se da inferência de tipos para o trecho de programa em $\mathcal{LL}$ mostrado na Figura 5.4. Neste caso, o conjunto T de tipos é $\{integer, real, string, range, boolean\}$.

```
repeat
    a ← a + b;
    b ← −c;
    p(−a);
    a ← 100;
until b > a
```

Figura 5.4: Trecho de programa a ser otimizado

Consideremos as tabelas R_+, R_- e $R_>$ abaixo:

		R_+
integer	*integer*	*integer*
real	*real*	*real*
string	*string*	*string*

	R_-
integer	*integer*
real	*real*
range	*range*

		$R_>$
integer	*integer*	*boolean*
real	*real*	*boolean*
string	*string*	*boolean*

Além disso, supõe-se que o compilador associa o tipo *integer* com a constante 100.

Na Figura 5.5 são mostrados os estados de cada um dos mapeamentos em cada ponto do programa. As duas colunas ao lado do programa mostram os resultados da análise progressiva e regressiva respectivamente.

1. Os itens 1.i da coluna da esquerda mostram os resultados da i-ésima iteração da primeira aplicação da análise progressiva em cada comando.
2. Os itens 1.i da coluna da direita mostram os resultados da i-ésima iteração da primeira aplicação da análise regressiva em cada comando.
3. Os itens 2.i mostram os resultados da i-ésima iteração da segunda aplicação da análise progressiva em cada comando.

onde

I, R, S, P representam os tipos *integer*, *real*, *string* e *range* respectivamente.

x, em $x \leftarrow p(-a)$, significa uma variável fictícia (lembrar que o método trata somente atribuições, por isso a chamada de procedimento precisa ser convertida).

Inicia-se com todas as variáveis associadas ao conjunto vazio de tipos (não há informação a respeito dos tipos das variáveis). Será iterado, inicialmente o problema progressivo.

Após 3 iterações do método progressivo note que chega-se a um valor fixo. Inicia-se então o método regressivo, tendo como mapeamento inicial o mapeamento vazio e como limitante superior os mapeamentos resultantes da análise progressiva. Este também converge em 3 iterações. A nova aplicação do método progressivo produz uma modificação; itera-se mais uma vez, quando então converge. Como uma nova aplicação do método regressivo (não mostrado na figura) não produz modificações, termina-se o processo. Os mapeamentos finais para os pontos que seguem cada comando simples são dados pelos itens indicados com 2.2.

Diversas conclusões podem ser obtidas neste exemplo:

- Todas as variáveis, no final do trecho do programa, devem ser do tipo *integer*.
- A variável c é do tipo *integer* em todos os pontos do programa.
- Antes do início do comando **repeat** requer-se que as variáveis sejam do tipo especificado no item 1.3 da coluna da análise regressiva.

	f_Q	b_Q
	$t(a) = t(b) = t(c) = T$	(1.3) $t(a) = t(b) = \{I, R\}; t(c) = \{I\}$ (1.2) $t(a) = t(b) = \{I, R\}; t(c) = \{I\}$ (1.1) $t(a) = t(b) = \{I, R\}; t(c) = \{I\}$
repeat		(1.3) $t(a) = t(b) = \{I, R\}; t(c) = \{I\}$ (1.2) $t(a) = t(b) = \{I, R\}; t(c) = \{I\}$ (1.1) $t(a) = t(b) = \{I, R\}; t(c) = \{I\}$
$a \leftarrow a + b$		
	(1.1) $t(a) = t(b) = \{I, R, S\}; t(c) = T$ (1.2) $t(a) = t(b) = \{I, R, S\}; t(c) = \{I, R, P\}$ (1.3) $t(a) = t(b) = \{I, R, S\}; t(c) = \{I, R, P\}$ (2.1) $t(a) = t(b) = \{I, R\}; t(c) = \{I\}$ (2.2) $t(a) = t(b) = \{I, R\}; t(c) = \{I\}$	(1.3) $t(a) = \{I, R\}; t(b) = \{I, R, S\}; t(c) = \{I\}$ (1.2) $t(a) = \{I, R\}; t(b) = \{I, R, S\}; t(c) = \{I\}$ (1.1) $t(a) = \{I, R\}; t(b) = \{I, R, S\}; t(c) = \{I\}$
$b \leftarrow -c$		
	(1.1) $t(a) = \{I, R, S\}; t(b) = t(c) = \{I, R, P\}$ (1.2) $t(a) = \{I, R, S\}; t(b) = t(c) = \{I, R, P\}$ (1.3) $t(a) = \{I, R, S\}; t(b) = t(c) = \{I, R, P\}$ (2.1) $t(a) = \{I, R\}; t(b) = t(c) = \{I\}$ (2.2) $t(a) = \{I, R\}; t(b) = t(c) = \{I\}$	(1.3) $t(a) = \{I, R\}; t(b) = t(c) = \{I\}$ (1.2) $t(a) = \{I, R\}; t(b) = t(c) = \{I\}$ (1.1) $t(a) = \{I, R\}; t(b) = \{I\}; t(c) = \{I, R, P\}$
$x \leftarrow p(-a)$		
	(1.1) $t(a) = \{I, R\}; t(b) = t(c) = \{I, R, P\}$ (1.2) $t(a) = \{I, R\}; t(b) = t(c) = \{I, R, P\}$ (1.3) $t(a) = \{I, R\}; t(b) = t(c) = \{I, R, P\}$ (2.1) $t(a) = \{I, R\}; t(b) = t(c) = \{I\}$ (2.2) $t(a) = \{I, R\}; t(b) = t(c) = \{I\}$	(1.3) $t(a) = \{I, R\}; t(b) = t(c) = \{I\}$ (1.2) $t(a) = \{I, R\}; t(b) = t(c) = \{I\}$ (1.1) $t(a) = \{I, R\}; t(b) = \{I\}; t(c) = \{I, R, P\}$
$a \leftarrow 100$		
	(1.1) $t(a) = \{I\}; t(b) = t(c) = \{I, R, P\}$ (1.2) $t(a) = \{I\}; t(b) = t(c) = \{I, R, P\}$ (1.3) $t(a) = \{I\}; t(b) = t(c) = \{I, R, P\}$ (2.1) $t(a) = t(b) = t(c) = \{I\}$ (2.2) $t(a) = t(b) = t(c) = \{I\}$	(1.3) $t(a) = t(b) = t(c) = \{I\}$ (1.2) $t(a) = t(b) = t(c) = \{I\}$ (1.1) $t(a) = t(b) = \{I\}; t(c) = \{I, R, P\}$
$x \leftarrow b > a$		
until	(1.1) $t(a) = t(b) = \{I\}; t(c) = \{I, R, P\}$ (1.2) $t(a) = t(b) = \{I\}; t(c) = \{I, R, P\}$ (1.3) $t(a) = t(b) = \{I\}; t(c) = \{I, R, P\}$ (2.1) $t(a) = t(b) = t(c) = \{I\}$ (2.2) $t(a) = t(b) = t(c) = \{I\}$	

Figura 5.5: Exemplo com resultado das funções f e b

Capítulo 6

Inferência de tipos em $\mathcal{LL}$

Como visto anteriormente, a análise de fluxo de dados para a inferência de tipos precisa ser feita tanto no sentido progressivo, seguindo o fluxo de execução do início para o fim, quanto no sentido regressivo. Viu-se também a abordagem de Kaplan e Ullman, descrita para comandos de atribuição. Neste capítulo são formuladas descrições para comandos estruturados da linguagem.

Para cada uma das construções da linguagem $\mathcal{LL}$ foi criada uma função para análise progressiva, outra para a análise regressiva. Estas funções trabalham sobre a estrutura do comando modificando os seus mapeamentos anteriores e posteriores.

Os mapeamentos refletem o estado das variáveis, antes e depois da execução do comando. Mapeamentos são associações entre cada uma das variáveis e os conjuntos de tipos que a variável pode assumir. Como mapeamentos estão associados a pontos do programa, a cada comando está associado um mapeamento posterior e um anterior. Estes conjuntos variam ao longo do programa (lembre que em $\mathcal{LL}$ variáveis não precisam ser declaradas e, portanto, podem não ter tipo fixo).

De forma geral, estes procedimentos trabalham sobre os seguintes parâmetros:

- *c*: estrutura do comando a ser otimizado. Esta estrutura aponta também para os mapeamentos anterior e posterior;
- *map_exit*: usada em comandos não repetitivos; retorna o mapeamento combinado dos comandos **exit** contidos em *c* para ser tratado por comandos repetitivos;
- *map_return*: retorna o mapeamento combinado dos pontos onde foram encontrados comandos do tipo **return**. Este parâmetro será utilizado pelo procedimento que trata funções e procedimentos;
- *tipos_result*: retorna os possíveis tipos para uma função; isto é feito analisando as expressões encontradas com o **return**.

Outros objetos das estruturas de dados usadas neste capítulo são os seguintes:

- *SC*: seqüência de comandos a ser executada dentro de um comando condicional ou repetitivo ou comandos de um corpo de uma rotina;

- *m_a(Q)*: mapeamento anterior a Q, isto é, estado das variáveis antes da execução de Q;
- *m_p(Q)*: mapeamento posterior a Q, isto é, estado das variáveis após a execução de Q;
- *m_aux(Q)*: mapeamentos auxiliares de controle (usados mais tarde);
- $t(expr)$: tipos da expressão *expr*.

Descreveremos de uma forma geral estes procedimentos, usando como exemplo as construções mais importantes de $\mathcal{LL}$. A descrição é feita em alto nível, usando notação semelhante à própria linguagem $\mathcal{LL}$[1]. Estes procedimentos correspondem às funções f_Q e b_Q descritas anteriormente no Capítulo 5. No próximo capítulo está indicada a implementação efetiva em C++.

Na notação utilizada:

"$\leftarrow$" significa atribuição de apontadores (para objetos);

"$\Leftarrow$" denota o operador de cópia;

"*seja*" explicita os componentes da estrutura de um comando.

O processo básico para a inferência de tipos é a análise de expressões. É nesta análise que as mudanças no estado das variáveis especificamente acontecem, pois no caso de comandos as modificações são conseqüência de sua estrutura, em particular das expressões que o compõem.

6.1 Análise de expressões

Expressões são representadas por uma árvore, onde cada subárvore é, por sua vez, uma expressão; variáveis, constantes e denotações são as folhas desta árvore; os nós internos são os operadores ou chamadas de funções.

A análise progressiva (procedimento *ProgressExpr*, figura 6.1) associa com cada expressão *expr* um conjunto de tipos $t(expr)$. Este conjunto de tipos é deduzido a partir das subexpressões que compõem a expressão maior. Para isso, existe a função R, a qual toma como argumentos, a operação a ser executada e uma lista de conjuntos de tipos (associados aos operandos) e retorna o conjunto de tipos possíveis para a expressão.

Por exemplo, em uma expressão simples como $a + b$, tendo como conjunto de tipos possíveis $T_a = \{inteiros, reais\}$ e $T_b = \{inteiros, caracteres\}$ seria de se esperar que o resultado da função $R(+, T_a, T_b)$ fosse $\{inteiros\}$ (supondo que inteiros só podem ser somados com inteiros).

Este processo é repetido para todas as subexpressões. Para isso, é feito um percurso em "pós-ordem" da árvore da expressão.

Em termos do método de Kaplan e Ullman, isto corresponde a supor que a expressão fará parte de uma atribuição (efetiva ou fictícia), e à determinação dos tipos possíveis para o seu resultado. Neste caso, o parâmetro m é o mapeamento que reflete os tipos das variáveis antes de executar a expressão. Como argumentos são objetos modificáveis, no final do procedimento, m

[1] É importante lembrar que nesta "pseudo" $\mathcal{LL}$ os parâmetros são objetos modificáveis.

```
procedure ProgressExpr( expr, m ) is
(* O mapeamento m é modificado refletindo os tipos deduzidos *)
    case expr of
      folha(a) ⇒ t(expr) ← m(a);
      ⊗(e1, e2, ..., en) ⇒
        forall i in 1..n do
          ProgressExpr( ei, m );
        t(expr) ← R⊗(t(e1), t(e2), ..., t(en));
        forall i in 1..n do
          t(ei) ← R⊗,(t(e1), t(e2), ..., t(en))
        endforall
    endcase
endprocedure
```

Figura 6.1: Análise progressiva de expressões

passa a refletir os tipos das variáveis após a execução da expressão. A rotina tem efeito lateral atribuindo tipos às expressões, o que foi aqui denotado por "$t(e) \leftarrow \ldots$".

Note-se que esta maneira de tratar a expressão da forma $\otimes(e_1, e_2, \ldots, e_n)$ corresponde a um trecho de programa da forma

$$\begin{array}{ll} v_1 & \leftarrow e_1; \\ v_2 & \leftarrow e_2; \\ \ldots & \\ v_n & \leftarrow e_n; \\ \otimes(v_1, v_2, \ldots, v_n) & \end{array}$$

onde v_i são variáveis fictícias (temporárias). Esta observação é necessária, pois Kaplan e Ullman consideram somente variáveis como componentes de uma expressão, não tratando subexpressões.

Na análise progressiva têm-se os tipos dos operandos e deseja-se saber os tipos possíveis para a expressão. Temos exatamente o contrário na análise regressiva: em função dos tipos das expressões (além do operador e dos tipos dos operandos) faz-se a inferência dos tipos para cada uma das subexpressões que as compõem.

No seguinte exemplo temos que $c + d$ é uma expressão do tipo *integer* (conhecido por algum processo). Sabe-se que os tipos dos operandos são $T_c = \{inteiros, reais\}$ e $T_d = T$ (ou seja, não se sabe nada sobre os tipos de d). Podemos concluir que d também só pode ser dos tipos *inteiro* e *real*.

O processo também é repetido recursivamente, fazendo um percurso em "pré-ordem" invertido da árvore (ver figura 6.2).

A função $T_{\otimes_i}(t_e, t_1, t_2, \ldots, t_n)$ retorna, em função do tipo esperado para a expressão e dos tipos dos operandos, o tipo do i-ésimo operando.

```
procedure RegressExpr( expr, m ) is
(* O mapeamento m é modificado refletindo os tipos deduzidos *)
   case expr of
        folha(a) ⇒ t(e) ← m(a);
        ⊗(e1, e2, ..., en) ⇒
           forall i in n..1 do
              t(ei)←T⊗i(t(expr), t(e1), t(e2), ..., t(en));
           endforall
           forall i in n..1 do
              RegressExpr( ei, m );
           endforall
     endcase
endprocedure
```

Figura 6.2: Análise regressiva de expressões

6.2 Análise do comando de atribuição

O comando de atribuição possui dois componentes: a variável que terá seu conteúdo alterado e uma expressão que contém o novo valor para a variável.

Para o tratamento deste comando, são desenvolvidas duas rotinas: *ProgressAtrib* e *RegressAtrib* (figura 6.3 e figura 6.4). A primeira faz a análise progressiva do comando, isto é, verifica a influência de uma atribuição nos estados das variáveis após a sua execução. Para isso, toma o mapeamento anterior associado ao comando e, tendo analisado as variáveis e operadores envolvidos na expressão, produz alterações sobre o mapeamento posterior ao comando. Uma alteração é fazer com que a variável atribuída tenha o mesmo tipo da expressão.

```
procedure ProgressAtrib( c ) is
   seja c ≡ v ← expr
      m_p(c) ⇐ m_a(c);
      ProgressExpr(expr, m_p(c));
      m_p(c)(v) ← t(expr);
endprocedure
```

Figura 6.3: Análise progressiva de atribuição

A segunda rotina mencionada faz a análise regressiva, isto é, em função do estado das variáveis após a execução da atribuição, infere os tipos das variáveis antes da execução. Por exemplo, se sabemos que a variável a qual se está atribuindo uma expressão é do tipo t após o comando, podemos deduzir que a expressão também é do tipo t. *RegressAtrib* toma o mapeamento posterior ao comando e altera o mapeamento anterior.

```
procedure RegressAtrib( c ) is
   seja c ≡ v ← expr

   m ⇐ m_p(c);
   t(expr) ← m(v);
   m(v) ← T;
   RegressExpr(expr, m);
   m_a(c) ⇐ m;
endprocedure
```

Figura 6.4: Análise regressiva de atribuição

6.3 Análise do comando repetitivo loop

As funções para a análise de tipos do comando **loop** são *ProgressLoop* e *RegressLoop* para as abordagens progressiva e regressiva. Como nos outros comandos, a análise é feita de acordo com o fluxo de execução, o qual está exemplificado na figura 6.5.

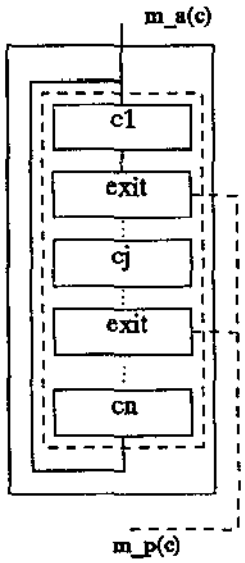


Figura 6.5: Diagrama da construção **loop**

Durante a análise progressiva, o mapeamento anterior do comando, que é atualizado pelo procedimento de análise do comando que o precede, é utilizado para a alteração da seqüência de comandos SC. Como existe um outro caminho que leva a SC, o mapeamento deste caminho deve ser levado em consideração para a atualização de seu mapeamento. Como este caminho vem do próprio SC, o mapeamento é calculado como a união do mapeamento anterior de c e o mapeamento posterior de SC. Note-se que o mapeamento posterior de uma seqüência de comandos SC não é o mesmo do comando repetitivo (ver figura 6.5).

A única saída de um comando **loop** é através de um comando **exit** ou **return**. O mapeamento posterior de c é calculado então, como a união dos mapeamentos de cada um dos comandos **exit** encontrados na seqüência SC. Comandos **return** têm tratamento semelhante e serão tratados mais adiante.

Um esboço de *ProgressLoop* pode ser visto na figura figura 6.6.

```
procedure ProgressLoop( c, map_return, tipo_result ) is
   seja c ≡ loop SC endloop

   map_exit ← ∅;
   m_a(SC) ← m_a(c) ∪ m_p(SC);
   ProgressSeq(SC, map_exit, map_return, tipos_result);
   m_p(c) ⇐ map_exit;

endprocedure
```

Figura 6.6: Análise progressiva do comando **loop**

Para a análise regressiva simplesmente atualiza-se o mapeamento posterior de SC com o mapeamento anterior da própria SC, pois como a saída de um comando **loop** se dá somente com um comando **exit**, o único sucessor de sua seqüência de comandos é ela própria. Faz-se em seguida a análise recursiva da SC e a atualização do mapeamento anterior de c com o mapeamento anterior de SC.

```
procedure RegressLoop( c, map_return, tipos_result ) is
   seja c ≡ loop SC endloop

   m_p(SC) ⇐ m_a(SC);
   RegressSeq( SC, m_p(c), map_return, tipos_result );
   m_a(c) ⇐ m_a(SC);

endprocedure
```

Figura 6.7: Análise regressiva do comando **loop**

6.4 Análise do comando repetitivo repeat

Para a análise de tipos de uma construção lingüística como o comando **repeat** constrói-se as funções *ProgressRepeat* e *RegressRepeat* para a análise progressiva e regressiva respectivamente.

ProgressRepeat toma como argumento a estrutura do comando **repeat** a ser otimizado. Esta estrutura contém o mapeamento anterior, o mapeamento posterior, a estrutura da seqüência de comandos a ser repetida e a expressão condicional da repetição. A análise é feita seguindo o fluxo de processamento.

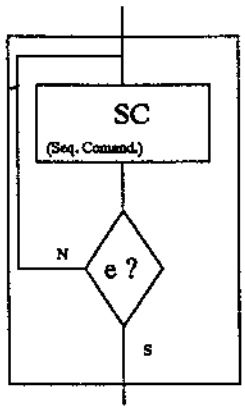


Figura 6.8: Diagrama da construção **repeat**

Primeiramente é feita a análise da seqüência de comandos SC. Esta SC terá seu mapeamento anterior atualizado de forma a refletir as alterações dos tipos. Isto é feito tomando-se os mapeamentos dos dois pontos que podem preceder SC: o início do comando e a expressão condicional (como pode ser visto na figura 6.8, existem dois caminhos para se chegar até SC). Recursivamente é feita a chamada da função que trata seqüência de comandos, passando SC como parâmetro. Esta função atualiza o mapeamento posterior a SC, o qual será usado como entrada para o tratamento da expressão.

Seguindo, é feita a chamada da função para a análise progressiva da expressão condicional. Esta função atualiza o mapeamento que recebeu como parâmetro, em função das variáveis e operadores envolvidos, devolvendo-o para a atualização do mapeamento posterior do comando.

Um esboço de *ProgressRepeat* pode ser visto na figura 6.9.

```
procedure ProgressRepeat( c, map_return, tipos_result ) is
   seja c ≡ repeat SC until expr

   map_exit ← ∅;
   m_a(SC) ← m_a(c) ∪ m_p(SCE);
   ProgressSeq( SC, map_exit, map_return, tipos_result );
   m ⇐ m_p(SC);
   ProgressExpr(expr, m);
   m_p(SCE) ⇐ m;
   m_p(c) ← (m ∪ map_exit);
endprocedure
```

Figura 6.9: Análise progressiva do comando **repeat**

O mapeamento anterior do comando c é atualizado a priori. O mapeamento final é calculado em função de eventuais comandos **exit** encontrados, além da expressão condicional, como foi visto acima.

O problema regressivo (figura 6.10) faz a análise seguindo o fluxo no sentido contrário ao da execução. Cada comando é analisado do final para o seu início. Inicia-se fazendo a atualização do mapeamento final do comando. Em seguida, usa-se este mapeamento como entrada para a análise regressiva da expressão condicional. O resultado desta análise é usado para a atualização do mapeamento posterior de SC antes da chamada do procedimento de sua análise regressiva. Por fim, atualiza-se o mapeamento do comando com o mapeamento de SC[2].

```
procedure RegressRepeat( c, map_return, tipos_result ) is
   seja c ≡ repeat SC until expr

   m ← m_p(c) ∪ m_a(SC);
   t(expr) ← "boolean";
   RegressExpr( expr, m );
   m_p(SC) ⇐ m;
   RegressSeq( SC, m_p(c), map_return, tipos_result );
   m_a(c) ⇐ m_a(SC);

endprocedure
```

Figura 6.10: Análise regressiva do comando **repeat**

O comando **repeat** poderia ter sido transformado em um **loop** equivalente e só então proceder a análise. O tratamento é feito separado porque o compilador $\mathcal{LL}$ constrói representações distintas para cada estrutura da linguagem.

Outros comandos repetitivos são tratados de forma análoga.

6.5 Tratamento dos comandos exit e return

O resultado do cálculo dos mapeamentos entre variáveis e tipos é influenciado pela existência de comandos **exit** e pelo comando **return**.

O comando **exit**, faz com que o fluxo de execução seja desviado para o comando seguinte ao comando repetitivo onde o comando **exit** está encaixado (no caso de aninhamento de comandos repetitivos, o comando **exit** faz o desvio para o fim do comando mais interno). Este desvio cria um novo caminho no grafo de fluxo de controle.

Como pode ser visto na figura 6.11 a única coisa que a função que trata o **exit** faz é unir o mapeamento anterior ao comando com o mapeamento acumulado de outros comandos **exit** já analisados.

Na abordagem regressiva, sempre que se encontra um comando **exit**, sabe-se que o fluxo de execução foi alterado, e que o seu comando posterior não é o que o sucede no fonte, mas sim o comando seguinte ao comando repetitivo c. Os mapeamentos de variáveis e tipos deve

[2] A expressão "boolean" na verdade tem tratamento mais complexo, ou seja, "*expr* = *true*" deve produzir o tipo "boolean" em tempo de execução. A simplificação tem por objetivo somente facilitar a explicação.

```
procedure ProgressExit( c, map_exit ) is
   seja c ≡ exit

   map_exit ← map_exit ∪ m_a(c);
endprocedure
```

Figura 6.11: Análise progressiva do comando exit

ser inicializado de acordo, isto é, o mapeamento posterior do comando **exit** é atualizado com o mapeamento posterior a c (figura 6.12).

```
procedure RegressExit( c, map_exit ) is
   seja c ≡ exit

   m_a(c) ⇐ map_exit;
endprocedure
```

Figura 6.12: Análise regressiva do comando **exit**

Em comandos não repetitivos, como condicionais, atribuições etc, o comando **exit** não tem influência sobre seus mapeamentos, mas sim sobre o comando repetitivo onde estes comandos estão encaixados. O procedimento a ser tomado é então receber das seqüências de comandos SC o mapeamento proveniente dos comandos **exit** existentes e repassá-las ao comando repetitivo que contém estes comandos. No problema regressivo tem-se o oposto: toma-se o mapeamento final do comando repetitivo e repassa-se às seqüências de comandos para que sejam atualizados os mapeamentos de algum comando **exit** existente.

A análise do comando **return** é análogo. Além do tratamento feito no comando **exit** deve fazer o tratamento de eventuais expressões.

6.6 Análise do comando condicional

A estrutura de um comando **if** (**if** e_1 **then** SC_1 **elsif** e_2 **then** SC_2 **elsif** $\cdots$ **elsif** e_n **then** SC_n **else** SC_{else}) pode ser vista na figura 6.14. A existência de várias expressões condicionais deve-se ao fato de $\mathcal{LL}$ permitir a cláusula **elsif**.

Existe um procedimento *ProgressIf* para a análise progressiva e um *RegressIf* para a análise regressiva, os quais podem ser vistos nas figuras 6.13 e 6.15.

Novamente, no problema progressivo, a análise é feita seguindo o fluxo de execução (ou os diferentes fluxos de execução). Avalia-se inicialmente a expressão condicional. Usa-se como parâmetro, o mapeamento inicial do comando em questão. A avaliação produz um novo mapeamento, possivelmente alterado em função das operações sobre as variáveis da expressão. Este

```
procedure ProgressIf( c, map_ant, map_acumul, map_exit, map_return,
tipos_result ) is
   seja c ≡ < <e1,sc1>, <e2,sc2> ... <Else,SCelse> >

   map_expr ⇐ m_a(c);
   ProgressExpr(e1, map_epxr)
   m_a(sc1) ⇐ map_expr;
   Progress(sc1, map_exit, map_return, result_typess);
   map_acumul ⇐ m_p(sc1);

   if (ha sequencia de clausulas Elsif)
      i = 2;
      while (i < n)
         ProgressExpr(ei, map_expr);
         m_a(sci) ⇐ map_expr;
         ProgressSeq(sci, map_exit, map_return, result_types);
         map_acumul ← map_acumul ∪ m_p(sci);
         i ← i + 1;
      endwhile
   endif
   if (ha clausula Else)
      m_a(SCelse) ⇐ map_expr;
      ProgressSeq(SCelse, map_exit, map_return, result_types);
      map_acumul ← map_acumul ∪ m_p(SCelse);
   endif

   m_p(c) ⇐ map_acumul;
endprocedure
```

Figura 6.13: Análise progressiva do comando **if**

mapeamento, como pode ser visto no diagrama (figura 6.14), é válido nos pontos 1 e 2.

Para o ponto 2 o processo é quase o mesmo do ponto 0. A diferença entre o ponto 0 e os pontos iniciais de cada cláusula **elsif** é o mapeamento usado como parâmetro: no primeiro caso, o mapeamento usado é o inicial do comando **if**; nos seguintes, é usado o resultado da análise expressão condicional anterior.

Para as seqüência de comandos (SC_i) o processo se passa de forma diferente. Os pontos que precedem a cada uma das seqüências de comandos são os correspondentes às expressões condicionais de cada seqüência. Assim, cada uma das seqüências de comandos é avaliada tendo como mapeamento de entrada o mapeamento da sua expressão condicional.

Como pode ser visto no diagrama, cada SC (inclusive SC_{else}) precede o mesmo ponto do comando **if**: o ponto de saída do comando. O mapeamento deste ponto é calculado como a união de cada um dos mapeamentos de todos os SCi, isto é, o tipo que uma variável pode assumir neste ponto será qualquer tipo que ela possa ter no final de cada SCi.

A análise regressiva é feita no sentido contrário da execução. Inicia-se com o mapeamento posterior do comando em questão. Este mapeamento é a única entrada para cada uma das seqüências SC. Para a análise de cada uma das expressões condicionais os parâmetros são os

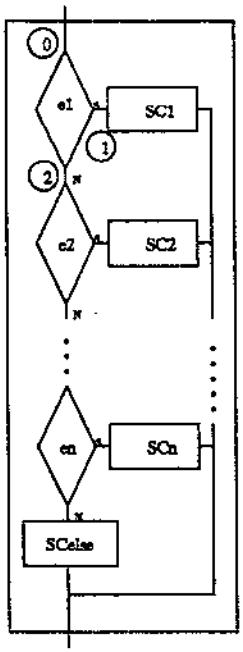


Figura 6.14: Diagrama da construção **if**

seguintes: o mapeamento resultante da análise regressiva das SC e o mapeamento da expressão condicional abaixo da expressão em questão.

O comando de seleção **case** recebe um tratamento análogo.

6.7 Análise de uma rotina e de um módulo

A análise do corpo de um método, procedimento ou função (o que corresponde á função $\mathcal{K}$ descrita anteriormente) é feita simplesmente iterando alternadamente as rotinas de análise progressiva e regressiva para a seqüência de comandos relativa à rotina (função $\mathcal{F}$ e $\mathcal{B}$ respectivamente) até que se obtenha a convergência. Antes de cada iteração os valores de cada mapeamento são iniciados corretamente.

A convergência global de um módulo (ou seja, o final da fase de otimização) se dá quando não há mais alteração no conjunto de tipos de retorno[3] encontrado para cada uma das funções.

[3] Variável *map_return*

```
procedure RegressIf( c, map_e, map_exit, map_return, tipos_result ) is
   seja c ≡ < <e1,sc1>, <e2,sc2> ... <en,scn>, <Else,SCelse> >

   if (ha' clausula Else)

      m_p(SC_else) ⇐ m_p(c)
      RegressSeq(SCelse, map_exit, map_return, result_types);
      map_expr ⇐ m_a(SC_else);
   else
      map_expr ⇐ m_p(c);

   if (ha sequencia de clausulas Elsif)
      i = n;
      while (ha clausula Elsif) (i ≠ 1)
         m_p(SCi) ⇐ m_p(c);
         RegressSeq(SCi, map_exit, map_return, result_types);
         map_expr ← map_expr ∪ m_a(SCi);
         RegressExpr(ei, map_expr);
         i ← i − 1;
      endwhile
   endif

   m_p(SC1) ⇐ m_p(c);
   RegressSeq(SC1, map_exit, map_return, result_types);
   map_expr ← map_expr ∪ m_a(SC1);
   RegressExpr(e1, map_expr);

   m_a(c) ⇐ map_expr;
endprocedure
```

Figura 6.15: Análise regressiva do comando **if**

Capítulo 7

Implementação

O otimizador de código é invocado após a construção da árvore de programa. Para entender melhor sua implementação, a próxima seção contém uma breve descrição do funcionamento do compilador llc.

7.1 O compilador

O compilador **llc** é composto de um analisador léxico, um analisador sintático, um analisador semântico, um otimizador de tipos e um gerador de código os quais interagem através de uma árvore de programa e de uma tabela de símbolos [BK94, Bac95]. O analisador léxico e o analisador sintático (baseado no método *LR*) foram gerados pelas ferramentas **flex** [fle] e **bison** [bis], respectivamente.

O analisador semântico tem por finalidade a construção da árvore de programa relativa ao módulo que está sendo compilado. Algumas "maquilagens" são feitas na árvore de forma a facilitar a tarefa de otimização e geração de código.

O gerador de código faz a geração, a partir da árvore de programa, de código para uma linguagem hospedeira que será posteriormente traduzido pelo respectivo compilador[1]. Como já foi dito, é na fase entre a análise semântica e a geração de código (após a construção da árvore) que o otimizador é invocado.

A estrutura (simplificada) de um nó desta árvore pode ser vista na figura 7.1. O método `Get_code` retorna a função sintática do nó em questão. `first` retorna um apontador para o primeiro "filho" de um nó. `next` retorna um apontador para o próximo "irmão" do nó. O último "filho" retornará um apontador para o nó "pai". `last` indica quando é o último nó de uma seqüência. Na figura 7.2 há um exemplo de uma subárvore relativa a um comando de atribuição.

O campo de otimização (`pOpt`) contém as informações necessárias para a transformação do *binding* dinâmico em estático. Do ponto de vista do gerador de código, a única informação relevante para esta transformação é a que diz respeito ao tipo de uma expressão (além da

[1] Fazendo o gerador de código à parte, pode-se criá-lo diferentemente para diferentes linguagens hospedeiras. Atualmente a única linguagem para qual existe um gerador é C++.

```
class TreeNode
{
  public:
    TreeNode ();
    ~TreeNode ();

    NodeCode Get_code();

    // informacao de otimizacaoo
    void Set_type (TreeNode *p, Restr r = sameTYPE);
    TreeNode *Get_type ();
    void Set_restriction (Restr);
    Restr Get_restriction ();

    // percorrendo a arvore
    TreeNode *first();
    TreeNode *next();
    int      last();

    OptimizationInfo  *pOpt;
    ...
}
```

Figura 7.1: Nó da árvore de programa

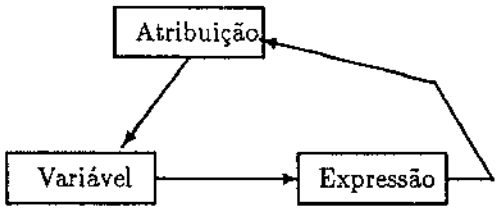


Figura 7.2: Árvore de programa para comando de atribuição

informação sobre a restrição sobre este tipo, isto é, se é um subtipo ou tipo estrito). Nos outros nós nenhuma informação de otimização é requerida. Os métodos `Set_type`, `Get_type`, `Set_restriction` e `Get_restriction` tratam destas atualizações (assim como os métodos relativos na classe `OptimizationInfo` abaixo).

A classe `OptimizationInfo`, usada pelo gerador de código, pode ser vista na figura 7.3.

Do ponto de vista do otimizador, muitas outras informações são necessárias.

7.2 Comandos

Para cada comando do programa, `pOpt` mantém dois mapeamentos: um contém o estado das variáveis antes do comando e o outro o estado depois do comando — com as alterações nos estados dos tipos que podem ser provocadas por este comando (vide figura 7.4). Note que o

```
class OptimizationInfo
{
 // private:

    TreeNode *pType;
    Restr R;

  public:

    OptimizationInfo ();
    OptimizationInfo (TreeNode *p, Restr r);

    virtual void set_type (TreeNode *p, Restr r);
    virtual TreeNode *get_type ();
    virtual Restr get_restriction ();

    virtual void set_strict_type (TreeNode *p);
    virtual TreeNode *get_strict_type();

    virtual void set_subtype (TreeNode *p);
    virtual TreeNode *get_subtype ();
};
```

Figura 7.3: Informações de otimização para o gerador de código

estado das variáveis posterior a um comando é igual ao estado anterior do próximo comando[2], então não são necessárias duas estruturas por comando: a cada dois comandos, os mapeamentos posterior do primeiro e anterior do segundo podem ser compartilhados. Apenas o mapeamento anterior do primeiro e o mapeamento posterior do último comando de uma seqüência não são compartilhados. Isso foi implementado da seguinte forma: cada comando possui apontadores para o mapeamento anterior e posterior; quando possível estes apontadores apontam para uma estrutura já apontada por outro comando. Um exemplo, utilizando a estrutura do **if**, pode ser vista na figura 7.5.

Mapeamentos são simplesmente conjuntos de vetores de bits. Para cada variável existe um vetor contendo dois bits para cada tipo: o primeiro indica se o tipo está presente (1) ou não (0), isto é, se a variável pode ou não ser deste tipo; o segundo indica a restrição do tipo, isto é, se a variável é exatamente daquele tipo ou pode ser um subtipo.

No exemplo da figura 7.6 vê-se que a variável **var 1** pode ser de um subtipo do tipo 1 (ou o próprio) ou estritamente dos tipos 3 e 4. Note que a cada variável e tipo é associado um número, índice deste vetores.

Kaplan e Ullman utilizam-se de um mapeamento limitante do crescimento das funções (lembrando que apesar da função $\mathcal{F}$ ser decrescente, F pode crescer: $F_s(x) = F(x) \cap s$). São necessários então, dois mapeamentos para cada comando: um de trabalho e outro como limitante superior, o que corresponde aos mapeamentos x e s respectivamente.

Também foi mostrado que os mapeamentos resultantes da função $\mathcal{F}$ são os melhores limi-

[2]Comandos, da forma que estão na árvore, possuem somente uma entrada e uma saída.

```
class myOptimizationInfo : public OptimizationInfo
{
   private:
     // there are two maps: one for forward and another to
     // backward analysis. When in forward analysis the first map is
     // modified and the second is used as an upper bound and
     // vice-versa when in backward analysis.
     map *m_a[2],    // previous mapping
     *m_p[2],    // next mapping
     *m_aux[2];  // used in some cases, like in Forall analysis

   public:
     myOptimizationInfo();

     // overloades the Bacarin definition; for expr node
     TreeNode *get_type();
     Restr get_restriction();
};  // myOptimizationInfo
```

Figura 7.4: Classe de otimização na "visão" do otimizador

tantes para a função $\mathcal{B}$. Optou-se então por criar dois mapeamentos anteriores (e posteriores) para cada comando, chamados progressivo e regressivo; usa-se o mapeamento progressivo como sendo x (de trabalho) e o regressivo como s (limitante) na análise progressiva e vice-versa na análise regressiva.

Como a solução do problema é dada por $k = (\mathcal{BF})^*(s)$, onde s é uma solução segura (por exemplo 1), inicializamos o mapeamento posterior em todos os pontos do programa como sendo U (todas as variáveis podem ser de todos os tipos).

7.3 Expressões

A estrutura de otimização (`pOpt`) para expressões precisa manter somente um apontador para um conjunto de tipos (dois bits por tipo, como na figura 7.6). Este conjunto também precisa ser duplo: o de trabalho e o limitante. No momento em que este conjunto for consultado qualquer um dos conjuntos pode ser usado, já que se sabe que o algoritmo convergiu, isto é, o algoritmo regressivo não produziu modificações no progressivo — os conjuntos são iguais.

Este é o único ponto que interessa, a princípio, para o compilador. Quando vai decidir entre gerar código com *binding* estático ou dinâmico são nós de expressões da árvore de programa que são consultados. O gerador de código pode produzir código otimizado somente quando se determinou o tipo inequivocamente, isto é, se só um tipo foi associado ao nó em questão. A função `Get_type` só retornará o tipo se o conjunto de tipos de uma expressão for unitário. Além disso, para geração de código com chamadas estáticas o tipo precisa ser estrito, isto é, o conjunto de tipos não deve permitir subtipos (a função `Get_restriction` é responsável por esta verificação)[3].

[3] Há a chance de se otimizar uma chamada cuja expressão permite subtipos se o método em questão não foi

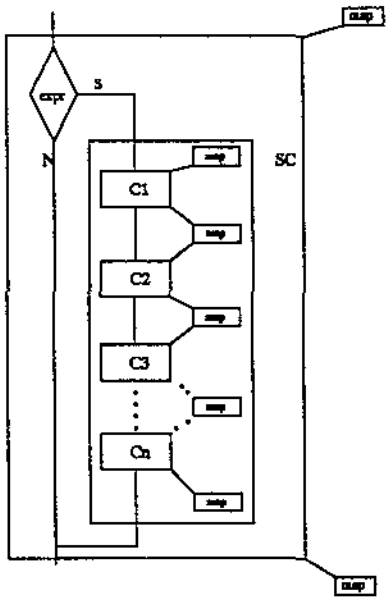


Figura 7.5: Compartilhamento de mapeamentos

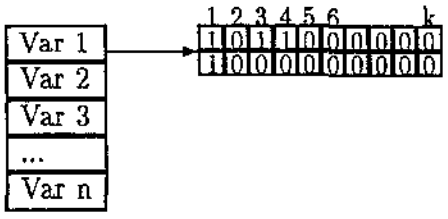


Figura 7.6: Esquema de um mapeamento

7.4 Outros nós

Ainda outros tipos de nós da árvore de programa contêm informações para otimização.

Nos nós relativos a módulo há uma tabela dos tipos conhecidos no momento.

Nos nós relativos a funções, procedimentos e métodos é necessário manter uma tabela com todas as variáveis, argumentos e constantes visíveis.

Declarações de variáveis precisam manter somente informações sobre seu número (índice do vetor de bits).

Declarações de constantes contêm um identificador e uma expressão. O identificador é tratado como uma variável[4] e a expressão de inicialização tratada como uma expressão normal.

redefinido para nenhum subtipo.

[4]Constantes funcionam exatamente como variáveis, a não ser pelo fato de só receberem atribuição uma vez.

Declarações de tipos precisam ter o seu número de ordem e a informação de todos os seus subtipos.

7.5 Mapeamentos

As operações sobre mapeamentos, implementados como vetores de bits, são óbvias: são obtidas por meio de operações binárias *and*, *or* e *not*.

Uma das operações apresenta uma particularidade: a operação que dado um conjunto de tipos, um tipo (representado pelo seu número) e a restrição ("mesmo tipo" ou "subtipo") acrescenta o tipo no conjunto. Além de assinalar o bit do tipo e da restrição, caso esta seja "subtipo" todos os bits correspondentes a subtipos do tipo em questão precisam ser assinalados. Como o nó relativo a declaração do tipo mantém a informação de todos os seus subtipos, esta operação é facilmente executada. Tal informação está representada como um conjunto de tipos, isto é, um vetor de bits também; é suficiente fazer a união do conjunto de subtipos com o conjunto em questão.

Ilustrando com um exemplo:

C = conjunto de tipos, representado por um vetor de bits;

S_{T_i} = conjunto de subtipos do tipo T_i;

t_i = tipos a serem acrescentados no conjunto C;

r_i = restrições a serem acrescentadas no conjunto C;

inicialmente com os valores:

$C = \{< 0,0 >, < 0,0 >, < 1,0 >, < 0,0 >, < 0,0 >\}$, ou seja este conjunto contém somente o tipo T_3;

$S_{T_1} = \{T_4\}$ e $S_{T_2} = \{T_5\}$;

$S_{T_i} = \emptyset$ para $i = 3,4,5$.

Acrescentado-se o tipo T_1, com a restrição "mesmo tipo" no conjunto C, teríamos

$$C+ < T_1, \text{ mesmo tipo } >= \{< 1,0 >, < 0,0 >, < 1,0 >, < 0,0 >, < 0,0 >\}$$

ou seja, simplesmente assinalou-se os bits correspondentes a este tipo. Agora se desejássemos acrescentar o tipo T_2 com a restrição "subtipo", teríamos

$$C+ < T_2, \text{ subtipo } >= \{< 0,0 >, < 1,1 >, < 1,0 >, < 0,0 >, < 1,1 >\}$$

tanto os bits correspondentes ao tipo T_2 como os correspondentes a todos os seus subtipos precisam ser assinalados.

Deve-se notar também que quando um módulo é compilado, tem-se somente os tipos conhecidos até o momento da compilação. Não é possível garantir que o módulo sendo compilado não será ligado mais tarde a um outro, o qual tenha mais declarações de tipos. Neste caso, rotinas podem ser invocadas passando como argumento objetos de qualquer tipo: conhecido até o momento ou não. Por isso foi necessário criar o "tipo desconhecido", usado para iniciar o conjunto de tipos de todos os argumentos de uma rotina. Isso se deve ao fato de que não está se fazendo a análise inter módulos. Esta abordagem garante a segurança da solução gerada mas faz com que muitas chamadas de procedimentos permaneçam dinâmicas.

Quando for possível garantir ao compilador que nenhum outro tipo será usado além dos já conhecidos (o que acontece quando se compila o módulo principal) pode-se obter resultados melhores (ver seção 8.1). Optou-se por criar uma chave de compilação para informar ao compilador quando se trata do módulo principal.

7.6 Procedimentos

Como indicado nos capítulos anteriores, para cada uma das construções da linguagem, existe uma função em C++ responsável pela sua análise progressiva e outra pela análise regressiva. Estas funções trabalham, como já foi visto, sobre a estrutura intermediária gerada pelo compilador (uma árvore de programa "decorada").

Um exemplo de tal tradução pode ser visto na figura 7.7.

```
/*****************************************************************/
/* ProgressLoop                                                  */
/*****************************************************************/
void ProgressLoop(TreeNode *c, map *map_return,
  type_set *result_types)
{
#define SC (c->first())
   map *m, *map_exit = new map(map_size,N_types);

   m = new map(m_p(SC,PROGRES), map_size, N_types);
   m->uniao(m_a(c,PROGRES));
   m_a(SC,PROGRES)->copy(m);
   ProgressSeq(SC, map_exit, map_return, result_types);
   m_p(c,PROGRES)->copy(map_exit);
   m_p(c,PROGRES)->inter(m_p(c,REGRES));
   delete(map_exit);
   delete(m);
#undef SC
} // ProgressLoop
```

Figura 7.7: Exemplo de procedimento em C++

Uma diferença substancial encontrada entre esta implementação e a especificação da figura 6.6 é a interseção feita com o mapeamento limitante superior. Isto se deve à utilização imediata dos resultados da mesma iteração (vide observação no fim da seção 5.4).

7.7 Convergência

Uma dificuldade encontrada na implementação do método Kaplan e Ullman foi o controle da convergência. Como foi visto no capítulo 5, a convergência local (dentro de cada fase: progressiva ou regressiva) se dá quando não houve modificação desde a última iteração. Uma forma eficiente de fazer tal verificação é observar se houve alguma alteração em cada parte do algoritmo.

Porém, a melhoria sugerida neste mesmo capítulo 5 foi implementada, isto é, são sempre usados os mapeamentos da mesma iteração e não os da anterior, como sugere o método originalmente. Decorre disto que a interseção com o limitante superior precisa ser feita imediatamente e não mais no final de cada fase.

Para se verificar a convergência, neste caso, não é mais suficiente observar em cada parte do algoritmo, pois algumas operações podem produzir modificações intermediárias. A solução adotada foi considerar algumas operações conjugadas com a operação de interseção com o mapeamento limitante superior como uma operação indivisível, ignorando as modificações intermediárias.

Capítulo 8

Conclusão

O objetivo do presente trabalho foi a introdução de uma fase de otimização no compilador da linguagem $\mathcal{LL}$ e com ela minimizar o efeito de *late binding* em chamadas de rotinas, eliminando sempre que possível o *binding* dinâmico. Isso foi obtido por meio da inferência de tipos de expressões em cada ponto do programa sendo analisado. Utilizou-se o método de Kaplan e Ullman que faz a propagação das informações dos tipos das variáveis tanto no sentido progressivo quanto no sentido regressivo.

Procurou-se unificar a forma de descrição de cada um dos métodos de inferência de tipos de forma a facilitar a comparação entre eles e a demonstração que o método Kaplan e Ullman é o que melhor se aplica ao problema de otimização de tipos em $\mathcal{LL}$.

Outra contribuição gerada como conseqüência e "subproduto" deste trabalho, foi um gerador de advertências. O otimizador tem condições de descobrir quando um conjunto de tipos possíveis para uma certa expressão é vazio e gerar uma advertência durante a compilação.

Comentamos a seguir os resultados obtidos e algumas extensões possíveis.

8.1 Resultados obtidos

Exemplo 1

O exemplo das figuras 8.1, 8.2 e 8.3 ilustra um programa $\mathcal{LL}$ e o resultado da compilação em suas versões não otimizada e otimizada.

Nos pontos onde o *binder* é chamado na figura 8.2, a versão otimizada chama diretamente a função que implementa a operação para os tipos descobertos das expressões. Neste exemplo todas as rotinas são chamadas estaticamente pelo compilador: todas as chamadas ao *binder* foram substituídas pela invocação estática.

Em grande parte dos casos, os programas escritos em linguagens similares a $\mathcal{LL}$ não terão necessidade de usar uma mesma variável para tipos diferentes ao longo de um programa, principalmente variáveis simples como contadores, índices e acumuladores. São justamente os casos onde o otimizador obtém os maiores ganhos: otimizando variáveis usadas como inteiros, reais e lógicas. Tais variáveis geralmente são usadas com denotações.

```
module fib is
function Fib(limite) : integer is
   i ← 1;
   j ← 0;
   s ← 1;
   n ← 0;
   while n < limite do
      i ← i+j;
      j ← i−j;
      s ← s+i;
      n ← n+1;
   endwhile;
   return s;
endfunction;
begin
   soma ← Fib(10);
endmodule
```

Figura 8.1: Somatório de termos de uma série de Fibonacci em *LL*

```
...
  for ( ; ; ) {
    RTS_push(_System_true_c);
    RTS_push(n);
    RTS_push(limite);
    BINDER(_OperatorLess_f2c,2);
    _System__boolean_OperatorEqual_f2();
    if (RTS_pop() == _System_false_c) break;
    RTS_push(i);
    RTS_push(j);
    BINDER(_OperatorPlus_f2c,2);
    i = RTS_pop();
    RTS_push(i);
    RTS_push(j);
    BINDER(_OperatorMinus_f2c,2);
    j = RTS_pop();
    RTS_push(s);
    RTS_push(i);
    BINDER(_OperatorPlus_f2c,2);
    s = RTS_pop();
    RTS_push(n);
    IntegerDenotation("1");
    BINDER(_OperatorPlus_f2c,2);
    n = RTS_pop();
  }
```

Figura 8.2: Trecho do resultado da compilação (não otimizado)

```
...
  for ( ; ; ) {
    RTS_push(_System_true_c);
    RTS_push(n);
    RTS_push(limite);
    _System__integer_OperatorLess_f2();
    _System__boolean_OperatorEqual_f2();
    if (RTS_pop() == _System_false_c) break;
    RTS_push(i);
    RTS_push(j);
    _System__integer_OperatorPlus_f2();
    i = RTS_pop();
    RTS_push(i);
    RTS_push(j);
    _System__integer_OperatorMinus_f2();
    j = RTS_pop();
    RTS_push(s);
    RTS_push(i);
    _System__integer_OperatorPlus_f2();
    s = RTS_pop();
    RTS_push(n);
    IntegerDenotation("1");
    _System__integer_OperatorPlus_f2();
    n = RTS_pop();
  }
```

Figura 8.3: Trecho resultado da compilação (otimizado)

Exemplo 2

Nas figuras que se seguem serão mostrados exemplos de resultados obtidos na otimização de um programa mais complexo que o anterior. Trata-se de um programa para visibilidade de polígonos usado como exemplo na dissertação de Evandro Bacarin [Bac95]. Este programa recebe como argumento um polígono e um ponto interior ao mesmo e produz como resultado o polígono visível por este ponto. O programa utiliza a biblioteca de estruturas e de algoritmos LEDA [Näh].

Na figura 8.4 pode ser vista o trecho final do programa $\mathcal{LL}$ para visibilidade de polígonos. Nele vemos a geração do resultado no arquivo de saída.

```
while (not Empty(S)) do
    p←Pop(S);
    Write(Xcoord(p),llfile);
    Write(" ",llfile);
    Write(Ycoord(p),llfile);
    Write("\n",llfile);
endwhile;
Close(llfile);
```

Figura 8.4: Trecho de programa em $\mathcal{LL}$

As diferenças entre as duas versões da compilação, vistas nas figuras 8.5 e 8.6, novamente é na invocação de rotinas. A versão não otimizada chama o *binder* em todos os pontos enquanto o programa otimizado elimina algumas das indireções. O *binder* ainda é chamado porque as funções `Ycoord` e `Xcoord` podem ser usadas tanto para pontos como para segmentos. A variável p, por ser resultado de um pop, pode ser de qualquer tipo, não possibilitando a determinação correta da implementação de `Ycoord` e `Xcoord`.

Verificamos que ganhos maiores poderiam ser obtidos se, na compilação do módulo principal, não fosse usado o "tipo desconhecido" (ver seção 7.5). Nas figuras seguintes se vê um exemplo de uso do otimizador com e sem o "tipo desconhecido".

Exemplo 3

A figura 8.7 mostra um exemplo da função *Colinear*, escrita em $\mathcal{LL}$, que recebe três parâmetros sem declarar seus tipos. Nas figuras 8.8 e 8.9 são mostrados os trechos compilados com e sem o "tipo desconhecido" (trechos referentes à primeira linha do if).

Como no exemplo anterior, não é possível otimizar a chamada das funções `Xcoord` e `Ycoord` pois não se conhece o tipo de $p1$, $p2$ e $p3$ e elas estão declaradas tanto para pontos como para polígonos. No entanto, em ambos os casos, o resultado é sempre do tipo *real*. Isso possibilita a chamada estática das funções que implementam o os operadores "+" e "*".

Esta opção, embora produza bons resultados, faz com que o tempo de execução do otimizador cresça bastante (esse tempo é desprezível em uma compilação normal). Isso se deve ao fato de se substituir o "tipo desconhecido" pelo conjunto de todos os tipos na inicialização dos

```
for ( ; ; ) {
  RTS_push(_System_true_c);
  RTS_push(_Pto_Vis_S_c);
  BINDER(_Empty_f1c,1);
  BINDER(_OperatorNot_f1c,1);
  _System__boolean_OperatorEqual_f2();
  if (RTS_pop() == _System_false_c) break;
  RTS_push(_Pto_Vis_S_c);
  BINDER(_Pop_f1c,1);
  p = RTS_pop();
  RTS_push(p);
  BINDER(_Xcoord_f1c,1);
  RTS_push(llfile);
  BINDER(_Write_p2c,2);
  StringDenotation("   ");
  RTS_push(llfile);
  BINDER(_Write_p2c,2);
  RTS_push(p);
  BINDER(_Ycoord_f1c,1);
  RTS_push(llfile);
  BINDER(_Write_p2c,2);
  StringDenotation("\n");
  RTS_push(llfile);
  BINDER(_Write_p2c,2);
}
RTS_push(llfile);
BINDER(_Close_p1c,1);
```

Figura 8.5: Versão não otimizada da compilação

argumentos das rotinas. O fato de a dimensão deste conjunto ser muito grande faz com que as consultas às tabelas de métodos e tipos seja muito demorada. Este aspecto poderia ser melhorado futuramente, através de uma estrutura de dados e algoritmos mais eficientes.

Embora a versão otimizada ainda não seja tão eficiente quanto um programa escrito diretamente em C++, foram eliminados dois níveis de indireção: o *binder* geral e o específico.

Em geral, não foram necessárias mais que três iterações em cada fase (progressiva ou regressiva) e três iterações da composição das fases para cada rotina. Em programas semelhantes ao da figura 8.1 têm-se ganhos, em termos de tempo de execução, da ordem de cinqüenta por cento. Em termos de espaço ocupado, programa objeto fica ligeiramente menor que a versão não otimizada.

8.2 Extensões futuras

Com as informações disponíveis

Uma melhoria a ser introduzida na geração de código, aproveitando as informações já disponíveis, é gerar o *binding* estático mesmo quando o conjunto de tipos de uma expressão não é unitário mas permite somente subtipos de um tipo. Isso pode ser obtido se a operação a ser gerada não

```
for ( ; ; ) {
  RTS_push(_System_true_c);
  RTS_push(_Pto_Vis_S_c);
  _System__stack_Empty_f1();
  _System__boolean_OperatorNot_f1();
  _System__boolean_OperatorEqual_f2();
  if (RTS_pop() == _System_false_c) break;
  RTS_push(_Pto_Vis_S_c);
  _System__stack_Pop_f1();
  p = RTS_pop();
  RTS_push(p);
  BINDER(_Xcoord_f1c,1);
  RTS_push(llfile);
  _System__real_Write_p2();
  StringDenotation("   ");
  RTS_push(llfile);
  _System__string_Write_p2();
  RTS_push(p);
  BINDER(_Ycoord_f1c,1);
  RTS_push(llfile);
  _System__real_Write_p2();
  StringDenotation("\n");
  RTS_push(llfile);
  _System__string_Write_p2();
}
RTS_push(llfile);
_System__outfile_Close_p1();
```

Figura 8.6: Versão otimizada da compilação

foi redefinida para nenhum subtipo. Por exemplo, se na atribuição abaixo:

$$c : X \leftarrow Y + Z$$

descobriu-se que Y pode ser $\{inteiro\}$ ou qualquer subtipo mas a operação "+" não foi sobrecarregada em nenhum subtipo, então pode-se fazer a geração de código com chamada estática do método que implementa esta operação.

Outra melhoria que não exige modificações na estrutura atual pode ser obtida quando sabe-se que um tipo utilizado em $\mathcal{LL}$ tem um correspondente na linguagem hospedeira, por exemplo, *integer* do módulo *System* padrão e `int` em C++. Se uma variável mantém o mesmo tipo ao longo de todo o programa ou rotina ela pode ser declarada no programa objeto com seu tipo correspondente. Isso pode ser obtido por meio de um percurso simples na árvore de programa. Durante este percurso é criado um mapeamento que conterá a união de todos os mapeamentos ao longo do corpo de uma rotina. No final, as variáveis para as quais o conjunto de tipos obtido foi unitário podem ser geradas diretamente no tipo correspondente.

Com exigência de mais informações

A convergência entre rotinas é acusada quando o conjunto de tipos de retorno de uma função não modifica entre duas iterações. Esse critério pode ser expandido para acompanhar mudanças

```
function Colinear(p1;p2;p3):boolean is
(* Dados tres pontos -- p1,p2,p3 – esta funcao verifica se eles sao
    colineares, atraves do calculo do seguinte determinante:
                                        1 x1 y1
                                        1 x2 y2
                                        1 x3 y3
*)
 if ( (Xcoord(p2)*Ycoord(p3) + Xcoord(p1)*Ycoord(p2) +
       Xcoord(p3)*Ycoord(p1) – Xcoord(p2)*Ycoord(p1) –
       Xcoord(p3)*Ycoord(p2) – Xcoord(p1)*Ycoord(p3)) =
     0.0 )
        then return true;
        elsif (true)
            then return false;
    endif;
endfunction;
```

Figura 8.7: Função em $\mathcal{LL}$

no conjunto de tipos de entrada da rotina, ou seja, os argumentos. Quando sabe-se que os argumentos de uma rotina são sempre de um mesmo tipo, essa informação pode ser usada por quem utiliza esta rotina. Neste caso, o controle de convergência deve indicar a necessidade de uma nova iteração sempre que houver alguma alteração também no conjunto de tipos da interface de algum método.

Análise entre módulos: para que a informação de otimização possa fluir de um módulo para outro é necessária a existência de alguns arquivos gerados e consultados pelo otimizador. Tais arquivos deveriam conter a informação a respeito dos tipos requeridos para os argumentos e os tipos resultantes de cada rotina. Essa análise eliminaria a necessidade do "tipo desconhecido" (vide seções 7.5 e 8.1).

```
void _Pto_Vis_Colinear_f3 () {
  ...
  RTS_push(p2);
  BINDER(_Xcoord_f1c,1);
  RTS_push(p3);
  BINDER(_Ycoord_f1c,1);
  BINDER(_OperatorStar_f2c,2);
  RTS_push(p1);
  BINDER(_Xcoord_f1c,1);
  RTS_push(p2);
  BINDER(_Ycoord_f1c,1);
  BINDER(_OperatorStar_f2c,2);
  BINDER(_OperatorPlus_f2c,2);
  ...
}   // end of function
```

Figura 8.8: Compilação com a presença do "tipo desconhecido"

```
void _Pto_Vis_Colinear_f3 () {
  ...
  RTS_push(p2);
  BINDER(_Xcoord_f1c,1);
  RTS_push(p3);
  BINDER(_Ycoord_f1c,1);
  _System__real_OperatorStar_f2();
  RTS_push(p1);
  BINDER(_Xcoord_f1c,1);
  RTS_push(p2);
  BINDER(_Ycoord_f1c,1);
  _System__real_OperatorStar_f2();
  _System__real_OperatorPlus_f2();
  ...
}   // end of function
```

Figura 8.9: Compilação com opção para módulo principal

Bibliografia

[ASU86] Alfred Aho, Ravi Sethi, e Jeffrey Ullman. *Compilers: Principles, Techinics and Tools.* Addison-Wesley Publishing Company, 1986.

[Bac95] Evandro Bacarin. Projeto e implementação de $\mathcal{LL}$: Uma linguagem de bibliotecas baseada em objetos. Dissertação de mestrado, DCC - UNICAMP, Março de 1995.

[bis] Bison – gerador de analisador sintático. Free Software Fundation.

[BK94] Evandro Bacarin e Tomasz Kowaltowski. Relatório de implementação do compilador $\mathcal{LL}$. DCC - IMECC - UNICAMP, Janeiro de 1994.

[CU89] Craig Chambers e David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object oriented programming language. In *Conference on Program Language Design and Implementation*, volume 24 de *SIGPLAN Notices*, pp. 146–160, Portland, OR, Junho de 1989. ACM.

[CU90] Craig Chambers e David Ungar. Type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Conference on Program Language Design and Implementation 90*, SIGPLAN Notices, pp. 150–164, White Plains, NY, Junho de 1990. ACM.

[fle] Flex – gerador de analisador léxico. Free Software Fundation.

[GR84] A. Golgberg e D. Robson. SMALLTALK *80, the language and its implementation.* Addison Wesley, Massachussets, 1984.

[GW75] Susan Graham e Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Conf. Record 2nd ACM Symp. on Princ. of Prog. Lang*, pp. 22–34, Janeiro de 1975.

[JM76] Neil D. Jones e Steven S. Muchnick. Binding time optimization in programming languages: Some thoughts toward the design of a ideal language. In *3o. Symp. on Principles of Programming Languages*, pp. 77–94, Atlanta / Ga, 1976.

[KB94a] Tomasz Kowaltowski e Evandro Bacarin. $\mathcal{LL}$ - an object-based library language. DCC - IMECC - UNICAMP, Fevereiro de 1994.

[KB94b] Tomasz Kowaltowski e Evandro Bacarin. *LC- An Object-Based Library Language - Reference Manual (version 1.2).* Departamento de Ciência da Computação - Universidade Estadual de Campinas, Campinas, Brasil, Março de 1994.

[KD94] Uday P. Khedker e Dhananjay M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM - TOPLAS*, 16(5):1472–1511, Setembro de 1994.

[Kow86] Tomasz Kowaltowski. Implementation of structured data flow analysis. *Revista Brasileira de Computação*, 4(2):95–114, 1985/86.

[KU77] J. B. Kam e J. D. Ullman. Monotone data flow analysis framework. *Acta Informatica*, 7:305–317, Janeiro de 1977.

[KU78] Marc A. Kaplan e Jeffrey D. Ullman. A general scheme for the automatic inference of variable types. In ACM Press, editor, *Proceedings of* 5^{th} *Symposium on Principles of Programming Languages*, SIGPLAN Notices, pp. 60–75, Janeiro de 1978.

[KU80] Marc A. Kaplan e Jeffrey D. Ullman. A scheme for the automatic inference of variable types. *Jornal of the Association for Computing Machinery*, 27(1):128–145, Janeiro de 1980.

[Näh] Stephan Näher. *LEDA User Manual.* Max-Planck-Institut für Informatik, Universität des Saarlandes.

[PS94] Jens Palsberg e Michael I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, Aarhus University, Denmark, 1994.

[Sil84] Katia Luckwü de Santana Silva. Estudo de alguns algoritmos para análise global de fluxo de dados. Dissertação de mestrado, DCC - UNICAMP, 1984.

[US87] David Ungar e Randall Smith. Self: The power of simplicity. In *Proceedings of OOPSLA'87*, SIGPLAN Notices, pp. 227–242. ACM, Outubro de 1987.

[VHU92] Jan Vitek, R. Nigel Horspool, e James S. Uhl. Compile-time analyse of object oriented programs. In *4th International Conference on Compiler Construction*, volume 641, pp. 236–250. Spring Verlag - LNCS, 1992.