Compilação Eficiente de Linguagens Funcionais Através de Transformação de Programas

André Luís de Medeiros Santos

alms@di.ufpe.br

Departamento de Informática – UFPE

Caixa Postal 7851, Cidade Universitária

Recife PE 50732-970

Resumo

Neste artigo apresentamos como o uso das técnicas de transformação automática de programas podem ser utilizadas em um compilador de linguagens funcionais para se obter resultados expressivos de melhora na qualidade do código gerado. As transformações utilizadas são relativamente simples se olhadas isoladamente, porém atuando em conjunto elas obtém resultados que, frequentemente, são obtidos de uma forma ad-hoc ou através de transformações específicas.

Abstract

In this article we present how automatic program transformation techniques can be used in a functional language compiler to improve the quality and performance of the code generated. The transformations we use are relatively simple if analysed on their own, but when combined they achieve results that are often obtained only in an ad-hoc way or through specific transformations.

1 Introdução

A compilação de linguagens através da técnica de transformação de programas já é conhecida e vem sendo utilizada há bastante tempo [Kra88, Kel89, FLM91]. O estudo de técnicas de transformação de programas especificamente para melhoria de qualidade do código tem, entretanto, frequentemente se concentrado em estudar transformações caras do ponto de vista computacional e cujo efeito em programas é verificado em programas de teste especificamente voltados para refletir casos ideais para aquela transformação.

Neste artigo apresentamos como a técnica de transformação de programas vem sendo utilizada para melhorar o desempenho do código gerado pelo *Glasgow Haskell Compiler* (GHC) [PHH+93], um compilador otimizador para a linguagem funcional Haskell [HPW+92] desenvolvido na Universidade de Glasgow (Escócia).

Devido à necessidade de serem acopladas a um compilador a ser utilizado por um grande número de usuários, as transformações não devem ser muito caras computacionalmente e ao mesmo tempo ter um efeito significativo em grande número de programas genéricos, e não em programas específicos para testar estas otimizações.

Com isto em mente foram estudadas um grande número de transformações relativamente simples e não muito caras computacionalmente, e que, quando combinadas, frequentemente substituem transformações muito mais complexas ou que são frequentemente implementadas de forma ad-hoc. Neste artigo descrevemos as principais transformações deste conjunto, através de exemplos de sua utilização para melhorar a qualidade do código de alguns fragmentos de código que ocorrem em programas compilados pelo GHC.

Inicialmente apresentamos a notação a ser utilizada neste artigo, chamada *Core*, que é a linguagem intermediária utilizada pelo compilador. Em seguida descrevemos três classes de transformações, várias delas bastante simples, que obtiveram resultados significativos na performance de um extenso conjunto de programas testados. Finalmente apresentamos as conclusões do artigo.

2 A linguagem Intermediária *Core*

A linguagem intermediária *Core* é ela mesma uma linguagem funcional, porém sem nenhum açucaramento sintático, isto é, com todas as características de alto nível da linguagem Haskell (como casamento de padrões, sobrecarga e compreensões de listas) já *transformadas* em construções mais elementares, utilizando técnicas bastante conhecidas [Pey87].

A linguagem é apresentada na Figura 1. Como pode ser observado ela possui abstrações lambda, construtores de tipos, tipos de dados básicos (*unboxed*), aplicações de funções e construtores, `case`s e `let`s. A linguagem possui ainda abstração e aplicação de tipos, baseada no lambda cálculo de segunda ordem, de forma que todas as transformações devem preservar informação sobre os tipos do programa, informações estas que são usadas tanto em algumas transformações como para auxiliar o gerador de código posteriormente. Neste artigo, para simplicidade, iremos ignorar esta característica da linguagem.

`let`s e `case`s desempenham um papel importante para viabilizar o trabalho utilizando transformação de programas, pois possuem uma interpretação operacional associada:

- toda a avaliação é feita através de `case`s, portanto `case`s representam avaliação;
- toda alocação dinâmica de objetos (em uma *heap*) é feita através de `let`s. Como a linguagem é preguiçosa (*lazy*) a avaliação do lado direito do `let` só ocorre se (e quando) o seu valor for demandado pelo programa, e uma única vez, já que, na primeira vez em que é avaliado, um `let` é atualizado com seu resultado (*update*).

Note que o fato destas construções terem esse significado operacional implica na simplicidade do formato da aplicação de função (e de construtores), pois todos os

Programa	Prog	$\rightarrow$	$\mathit{Binding}_1$; $\ldots$; $\mathit{Binding}_n$	$n \geq 1$
Bindings	$\mathit{Binding}$	$\rightarrow$	Bind	
		$\mid$	`rec` $\mathit{Bind}_1 \ldots \mathit{Bind}_n$	
	Bind	$\rightarrow$	var `=` Expr	
Expressões	Expr	$\rightarrow$	$\mathit{Expr}\ \mathit{Atom}$	Aplicação
		$\mid$	$\mathit{Expr}\ \mathit{ty}$	Aplicação de tipos
		$\mid$	$\lambda\ \mathit{var}_1 \ldots \mathit{var}_n$ `->` Expr	Abstração Lambda
		$\mid$	$\Lambda\ \mathit{ty}$ `->` Expr	Abstração de Tipos
		$\mid$	`case` Expr `of` Alts	Expressão case
		$\mid$	`let` $\mathit{Binding}$ `in` Expr	Definição local
		$\mid$	$\mathit{con}\ \mathit{Atom}_1 \ldots \mathit{Atom}_n$	Construtor $n \geq 0$
		$\mid$	$\mathit{prim}\ \mathit{Atom}_1 \ldots \mathit{Atom}_n$	Op. Primitiva $n \geq 0$
		$\mid$	Atom	
Átomos	Atom	$\rightarrow$	var	Variável
		$\mid$	$\mathit{Literal}$	Objeto *unboxed*
Literais	$\mathit{Literal}$	$\rightarrow$	$\mathit{integer} \mid \mathit{float} \mid \ldots$	
Alternativas	Alts	$\rightarrow$	$\mathit{Calt}_1; \ldots ; \mathit{Calt}_n;\ [\mathit{Default}]$	$n \geq 0$
		$\mid$	$\mathit{Lalt}_1; \ldots ;\ \mathit{Lalt}_n;\ [\mathit{Default}]$	$n \geq 0$
Alt.de Construtores	Calt	$\rightarrow$	$\mathit{con}\ \mathit{var}_1 \ldots \mathit{var}_n$ `->` Expr	$n \geq 0$
Alt.de Literais	Lalt	$\rightarrow$	$\mathit{Literal}$ `->` Expr	
Alt. default	$\mathit{Default}$	$\rightarrow$	var `->` Expr	

Figura 1: Sintaxe da Linguagem *Core*

seus argumentos passam a ser átomos. Isso também simplifica (reduz) bastante o número de transformações de que precisamos, pois reduz o número de posições onde podemos ter expressões.

O uso desta notação, que torna explícitas as características de avaliação e alocação de uma função ou mesmo de uma expressão é fundamental para o sucesso da técnica de transformação de programas que desenvolvemos a seguir, pois:

- ela é concreta o suficiente para permitir que mostremos exatamente como estamos reduzindo ou afetando o custo de avaliação ou alocação de uma expressão, ao mesmo tempo em que

- ela é abstrata o suficiente para que as transformações sejam aplicáveis a qualquer implementação concreta de uma linguagem funcional, e não apenas à maquina abstrata utilizada pelo *Glasgow Haskell Compiler*.

3 Transformações

Em [San95] é apresentada uma lista completa das transformações efetuadas pelo compilador. Neste artigo nos concentraremos em apenas algumas delas, e as apresentaremos através de exemplos de seus efeitos, que frequentemente ocorrem em "cascata", ou seja, uma transformação leva a outra e assim sucessivamente.

3.1 Evitando avaliação repetida

A expressão `x+x` (onde `x` tem tipo `Int`) em um programa fonte gera o seguinte código intermediário em *Core*:

```
(\a -> \b -> case a of
               MkInt a# -> case b of
                             MkInt b# -> case a# +# b# of
                                           r# -> MkInt r#) x x
```

Isto ocorre devido à expansão do código do operador sobre inteiros (`+`), que avalia seus dois argumentos e só então executa a operação primitiva de soma (`+#`) (que retorna e opera sobre inteiros *unboxed*, isto é, em sua representação mais simples, sem o construtor de tipo `MkInt`). Finalmente é retornado um inteiro em sua forma *boxed*, o que é feito utilizando-se o construtor de tipo `MkInt`. Neste caso em particular os dois argumentos são a mesma variável `x`. Note que distinguindo as versões *boxed* e *unboxed* dos inteiros temos mais uma vez um controle maior sobre a característica operacional do operador de adição sobre inteiros (`+`), e sabemos extamente quando estamos realmente utilizando o operador de adição de inteiros do nível de máquina (`+#`). Esta notação explícita de tipos de dados *unboxed* é descrita e discutida em [PL91].

Note que neste ponto já estamos usando uma transformação: a expansão de código, ou *inlining*, que substitui uma ou mais ocorrências de uma expressão ligada por um `let` por seu lado direito:

$$\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ \ldots x \ldots \Longrightarrow \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ \ldots e \ldots$$

neste caso a definição do operador (`+`) que estamos expandindo é implícita.

As vantagens da expansão de código de uma forma geral são:

- pode permitir a eliminação da definição (`let`), se todas as suas ocorrências forem expandidas, economizando sua alocação;
- a definição se torna disponível no local em que é utilizada, permitindo transformações como beta-redução;
- expõe mais informação contextual (i.e. do local onde é utilizado) à expressão do lado direito do `let`.

Obviamente a expansão de código deve ser feita seletivamente, de forma a evitarmos uma explosão de código, devido à uma duplicação excessiva de código. Esta seleção é também um assunto bastante estudado por si só, e não o discutiremos aqui. Uma análise específica sobre o que expandir ou não é feita em [San95].

A segunda transformação que utilizamos é exatamente a beta-redução[1]:

$$(\lambda x \texttt{ -> } e)\ y \quad \Longrightarrow \quad e[y/x]$$

Na beta-redução estamos na realidade fazendo um pouco da avaliação do programa em tempo de compilação, já que a beta-redução é o mecanismo básico de avaliação em linguagens funcionais. A própria sintaxe da linguagem *Core* garante que não teremos problemas de perda de compartilhamento de expressões ao fazermos a substituição, já que os argumentos são sempre variáveis.

Ao efetuarmos uma beta-redução durante a compilação movemos a avaliação do tempo de execução para o tempo de compilação, e também expomos informação contextual (local) à variável (ou constante) passada como argumento.

A nossa expressão agora se torna:

```
case x of
  MkInt x# -> case x of
                MkInt y# -> case x# +# y# of
                              r# -> MkInt r#
```

Podemos então utilizar a transformação que chamamos de redução (ou avaliação) do `case`, que possui três formas básicas:

$$\begin{array}{l} \texttt{case } C\ v_1 \ldots v_n \texttt{ of} \\ \quad \ldots \\ \quad C\ x_1 \ldots x_n \texttt{ -> } e \\ \quad \ldots \end{array} \quad \Longrightarrow \quad e[v_i/x_i]_{i=1}^{n}$$

em que o `case` avalia uma expressão que explicitamente tem o construtor (aqui representado por C) de uma das alternativas,

$$\begin{array}{l} \texttt{case } v \texttt{ of} \\ \ldots \\ C\ x_1 \ldots x_n \texttt{ -> } \ldots \left(\begin{array}{l} \texttt{case } v \texttt{ of} \\ \ldots \\ C\ y_1 \ldots y_n \texttt{ -> } e \\ \ldots \end{array} \right) \\ \ldots \end{array} \Longrightarrow \begin{array}{l} \texttt{case } v \texttt{ of} \\ \quad \ldots \\ \quad C\ x_1 \ldots x_n \texttt{ -> } \ldots e[x_i/y_i]_{i=1}^{n} \\ \quad \ldots \end{array}$$

em que temos como saber o construtor ao qual uma variável está ligada pelo fato de ela já ter sido avaliada antes por um outro `case` e

[1] A notação $e[x/y]$ denota a expressão e em que *todas* as ocorrências livres de x são substituidas por y.

$$\begin{array}{l}\texttt{let } x \texttt{ = } C\ x_1 \ldots x_n \\ \texttt{in } \ldots \left(\begin{array}{l} \texttt{case } x \texttt{ of} \\ \quad \ldots \\ \quad C\ y_1 \ldots y_n \texttt{ -> } e \\ \quad \ldots \end{array} \right) \ldots \end{array} \implies \begin{array}{l} \texttt{let } x \texttt{ = } C\ x_1 \ldots x_n \\ \texttt{in } \ldots e[x_i/y_i]_{i=1}^{n} \ldots \end{array}$$

em que temos como saber o construtor de uma variável por ela estar ligada explicitamente a um construtor.

No nosso exemplo essa transformação (em sua segunda versão) nos leva ao código desejado, em que avaliamos `x` uma única vez:

```
case x of
  MkInt x# -> case x# +# x# of
                r# -> MkInt r#
```

3.2 Casamento de padrão preguiçoso

A transformação de um casamento de padrão é normalmente feito de forma preguiçosa, isto é, a expressão do lado direito do padrão só deve ser avaliada se e quando necessário, e é portanto bastante ineficiente:

```
let (x,y) = expr in h x y
```

é transformado para

```
let t = expr
in let x = case t of (a,b) -> a
   in let y = case t of (a,b) -> b
      in h x y
```

em *Core*. Desta forma alocamos três objetos na *heap*, que poderão ser eventualmente avaliados e atualizados (o que também é uma operação relativamente cara). Entretanto se o analisador de demanda (*strictness*) identificar que (`h x y`) é estrito em `x` *ou* `y` isso também implica que ele é estrito também em `t`, isto é, temos certeza que `t` será avaliado durante a avaliação de (`h x y`). Utilizamos então a transformação que chamamos de `let-to-case`:

$$\begin{array}{l}\texttt{let } v \texttt{ = } e_v \\ \texttt{in } e \end{array} \implies \begin{array}{l} \texttt{case } e_v \texttt{ of} \\ \quad C_k\ v_{k1} \ldots v_{kl} \texttt{ -> let } v \texttt{ = } C_k\ v_{k1} \ldots v_{kl} \texttt{ in } e \end{array}$$

Onde passamos de uma alocação de um objeto na *heap* que possivelmente será avaliado e atualizado para uma expressão que imediatamente avalia a expressão do lado direito do `let` original e já nos retorna os seus subcomponentes explicitamente decompostos. Essa transformação só pode ser feita se a variável do `let` for estrita (i.e. se temos certeza que ela será avaliada em algum momento) e, para evitar duplicação excessiva de código, apenas quando o tipo da expressão possuir um único construtor, como é o caso de tuplas, inteiros etc., mas não de listas, por exemplo.

Temos então no nosso exemplo:

```
case expr of (w,z) -> let t = (w,z)
                      in let x = case t of (a,b) -> a
                         in let y = case t of (a,b) -> b
                            in h x y
```

A redução de `case` (que vimos na seção anterior) elimina então as expressões `case` do lado direito de `x` e de `y`, e então temos:

```
case expr of (w,z) -> let t = (w,z)
                      in let x = w
                         in let y = z
                            in h x y
```

`t` é portanto código morto, podendo ser eliminado, e as variáveis `x` e `y` podem trivialmente ser expandidas e então removidas, já que estão ligadas a outras variáveis:

```
case expr of (w,z) -> h w z
```

Que é uma versão eficiente de casamento de padrão, desta vez estrito.

3.3 Simplificando `cases` aninhados

A transformação que analisaremos agora simplifica expressões em que um `case` avalia uma outra expressão `case`:

$$
\begin{array}{l}
\texttt{case}\left(\begin{array}{l}\texttt{case } e_c \texttt{ of}\\ alt_{c1}\ \texttt{->}\ e_{c1}\\ \ldots\\ alt_{cm}\ \texttt{->}\ e_{cm}\end{array}\right)\texttt{of}\\
alt_1\ \texttt{->}\ e_1\\
\ldots\\
alt_n\ \texttt{->}\ e_n
\end{array}
\quad\Longrightarrow\quad
\begin{array}{ll}
\multicolumn{2}{l}{\texttt{case } e_c \texttt{ of}}\\
alt_{c1}\ \texttt{->} & \texttt{case } e_{c1} \texttt{ of}\\
 & \quad alt_1\ \texttt{->}\ e_1\\
 & \quad \ldots\\
 & \quad alt_n\ \texttt{->}\ e_n\\
\ldots & \\
alt_{cm}\ \texttt{->} & \texttt{case } e_{cm} \texttt{ of}\\
 & \quad alt_1\ \texttt{->}\ e_1\\
 & \quad \ldots\\
 & \quad alt_n\ \texttt{->}\ e_n
\end{array}
$$

Uma instância específica desta transformação, que chamamos de "`case-of-case`" é descrita em [Aug87] e em [Kel89]. Eles estavam preocupados, entre outras coisas, em fazer o "curto-circuito" de `ifs`, que também ocorre frequentemente em linguagens imperativas. Por exemplo:

```
if (b1 && b2) then e1 else e2
```

onde `b1` e `b2` são expressões booleanas, e `&&` é o operador de conjunção. Se `b1` for falso, não há necessidade de avaliarmos `b2`, pois o resultado será `e2` de qualquer forma. A definição de `&&` já possui esta característica:

```
(&&) b1 b2 = case b1 of
             True  -> b2
             False -> False
```

Vamos agora transformar a expressão original para a nossa notação utilizando `cases`:

```
case (b1 && b2) of True  -> e1; False -> e2
```

Ao expandirmos a definição de `&&` obtemos:

```
case (case b1 of True -> b2; False -> False) of
True  -> e1
False -> e2
```

Aplicamos então a transformação `case-of-case`:

```
case b1 of
True  -> case b2 of
           True  -> e1
           False -> e2
False -> case False of
           True  -> e1
           False -> e2
```

O segundo `case` interno agora avalia um construtor explicitamente, podemos portanto aplicar a transformação de redução do `case`, obtendo:

```
case b1 of
True  -> case b2 of
           True  -> e1
           False -> e2
False -> e2
```

Conseguimos desta forma o efeito desejado, evitando a chamada à função `&&` e com código bastante simples.

O exemplo acima mostra uma limitação da transformação: `e2` ocorre duas vezes na expressão transformada, apesar de que será avaliado no máximo uma vez, já que as duas ocorrências são em alternativas diferentes da expressão `case`. Isso poderia gerar explosão de código, caso `e2` seja uma expressão grande.

Este risco de duplicação de código pode ser facilmente eliminado, definindo-se, através de um `let`, uma variável cuja função é basicamente permitir o compartilhamento das duas cópias de `e2` em uma só. Desta forma obtemos as vantagens de uma eliminação de subexpressões comuns, porém sem o custo da análise de subexpressões comuns (neste caso). O resultado final seria:

```
let v = e2
in case b1 of
     True  -> case b2 of
                True  -> e1
                False -> v
     False -> v
```

Este `let` pode, ainda, ser anotado e implementado de forma bastante eficiente como um simples ponteiro à código, e não como um `let` comum, que é alocado na *heap*.

Este mesmo método se ajusta, através da abstração de variáveis livres, à casos de `case`-of-`case` com quaisquer tipos de dados, e não apenas em casos de tipos de dados com construtores sem argumentos, como é o caso dos booleanos. Desta forma a transformação é extendida para inúmeros casos em que ela não seria utilizada antes, seja devido ao risco de duplicação de código, seja devido ao tipo das expressões sendo avaliadas.

3.4 Movendo `let`s localmente

Um conjunto de transformações cujo efeito causou bastante surpresa foram as transformações que implicam simplesmente na mudança de posição de `let`s, já que não há referências na literatura de tal tipo de transformação.

Ao mudarmos a posição de um `let` estamos, à priori, apenas modificando *quando* aquele `let` vai ser alocado, e não quando ele vai ser avaliado, já que a linguagem que estamos trabalhando é uma linguagem *lazy*.

Desta forma nós temos bastante liberdade de antecipar ou atrasar esta alocação, limitados apenas pelo fato de termos que manter as variaáveis do `let` em escopo.

Um exemplo interessante ocorre quando temos a definição de uma constante, como:

```
let x = [1,2,3]
in ...
```

Uma possível tradução para *Core* seria

```
let x = let v1 = let v2 = 3:[]
                 in 2:v2
        in 1:v1
in ...
```

Neste caso alocamos apenas um objeto (*closure*) `x` antes de avaliarmos o corpo do `let`. Somente se (e quando) a avaliação de `x` for feita pelo corpo do `let` é que avaliamos o `x`, o que implica na alocação de `v1` e no retorno da expressão `(1:v1)`, que por sua vez também implica na atualização (*update*) de `x`. Um processo semelhante ocorrerá se (e quando) o valor de `v1` for demandado.

Esta alocação é portanto feita de forma extremamente preguiçosa, o que como vimos implica em uma única alocação no caso em que `x` não seja avaliado, mas implicará em várias alocações se toda a lista `x` for necessária. Pior ainda, tanto o `let` do `x` quanto o de `v1`, se avaliados, implicam em atualizações (*updates*), já que os seus lados direitos não se encontram em forma normal (i.e. não são expressões lambda, constantes ou construtores).

Por outro lado, poderíamos ter uma outra forma de traduzir a expressão original, desta vez para:

```
let v2 = 3:[]
in let v1 = 2:v2
   in let x =  1:v1
      in ...
```

Neste caso alocamos imediatamente três objetos na *heap*, mas eles já estão em forma normal, e portanto se eles realmente forem avaliados em algum momento, o serão de forma mais barata que na primeira versão, pois estamos evitando as atualizações. Além disso, como vimos na seção 3.1, `let`s diretamente ligados a construtores nos dão oportunidade às vezes de fazer uma redução de `case`, o que também não ocorria na primeira tradução, já que apenas `v2` estava diretamente ligada a um construtor.

A transformação da primeira forma na segunda pode ser feita através do que chamamos `let` *floating* para fora, já que os `let`s estão saindo de dentro de outros[2]:

`let` x `= let (rec)` *bind* `in` e_x `in` b	$\Longrightarrow$	`let (rec)` *bind* `in let` x `=` e_x `in` b

Pode-se perguntar porque então não geramos imediatamente o código na segunda forma, ao invés de termos uma transformação para isso. A resposta é que existem casos em que desejamos fazer exatamente a transformação oposta, isto é, mover `let`s para dentro de outros `let`s!

Uma razão para isso pode ser exemplificada se voltarmos ao nosso exemplo de compilação de casamento de padrão preguiçoso, em que contamos com a possiblidade de descobrir que pelo menos uma das partes da tupla é estrita, ou seja, será com certeza avaliada:

```
let t = expr
in let x = case t of (a,b) -> a
   in let y = case t of (a,b) -> b
      in h x y
```

Vamos supor que a função `h` seja

```
h x y = let v = x + y
        in (v*3,v+3)
```

e que façamos a expansão da chamada à função `h`. Temos então:

```
let t = expr
in let x = case t of (a,b) -> a
   in let y = case t of (a,b) -> b
      in let v = x + y
         in (v*3,v+3)
```

Até este ponto podemos ver que a expressão não é estrita nem em `x` nem em `y`, pois como o resultado é uma tupla não temos como saber (pela expressão dada) se os

[2]existe uma transformação semelhante para `let`s recursivos.

seus argumentos serão ou não avaliados, i.e. se `v` (e consequentemente `x` e `y`) serão avaliados.

Entretanto, se fizermos uma transformação de `let` *floating* para *dentro* do lado direito de `v` dos outros `lets`[3], obtemos:

```
let v = let t = expr
        in let x = case t of (a,b) -> a
           in let y = case t of (a,b) -> b
              in x + y
in (v*3,v+3)
```

Desta forma nós levamos os `lets` para o mais próximo possível de onde eles são utilizados (referenciados), e portanto passamos a ter *mais informação contextual!* Podemos ver que *nesta nova posição* `x` e `y` são utilizados de forma estrita em seu corpo `(x + y)` e portanto podemos fazer a transformação descrita na seção 3.2 neste novo contexto, obtendo:

```
let v = case expr of (w,z) -> w + z
in (v*3,v+3)
```

Através de uma análise extensa de casos em que uma ou outra versão desta transformação foi utilizada pudemos definir uma série de heurísticas para decidir os casos em que utilizamos a transformação em uma ou outra direção. Os exemplos acima caracterizam os principais fatores envolvidos.

Em [San95, PPS96] são discutidas em detalhe essas e outras transformações envolvendo a mudança de posição de `lets`, inclusive com análise detalhada de seus efeitos, vantagens e desvantagens. É mostrado ainda, informalmente, que a heurística utilizada para escolher a direção em que a transformação é aplicada não corre o risco de gerar não terminação do compilador, que poderia ocorrer se uma transformação sempre deixasse margem para que a outra "desfizesse" o seu trabalho.

4 Conclusão

Como vimos através dos exemplos mostrados no artigo, o efeito combinado de transformações relativamente simples são bem maiores do que se suporia observando as transformações isoladamente.

À primeira vista as transformações são bastante simples e o código em que elas são aplicáveis jamais seria escrito por um programador, mas durante o processo de compilação dos programas elas ocorrem com grande frequência e em situações inesperadas.

O conjunto de transformações apresentado é composto por transformações simples, locais, e que não dependem de análises caras para sua utilização.

De fato, em [San95] são mostrados os resultados obtidos utilizando as transformações acima e outras semelhantes em 46 programas do *nofib Benchmark Suite* [Par92], escritos por inúmeras pessoas diferentes e muitos deles com centenas ou

[3] isto só é possível devido ao fato destes `lets` não serem referenciados no corpo de `v`.

mesmo milhares de linhas de código. Outra característica importante desses programas é que eles foram escritos cada um para uma aplicação diferente, e não para serem usados especificamente como programas de teste.

Estes experimentos mostram que instâncias das transformações que apresentamos ocorrem com muita frequência, e que seu efeito combinado reduz, em média, cerca de 19% no tempo de execução e 18% no total de memória (*heap*) utilizada, chegando a 51% e 79%, respectivamente, em casos extremos. Apenas dois exemplos apresentaram resultados negativos no uso das transformações, porém esses casos são explicados e resolvidos utilizando-se outras transformações, também apresentadas em [San95].

Referências

[Aug87] Lennart Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Science, Chalmers University of Technology, S–412 96 Göteborg, November 1987.

[FLM91] M. Fradet and Daniel Le Metayer. Compilation of functional languages by program transformation. *ACM Trans. on Programming Languages and Systems*, 13(1), January 1991.

[HPW+92] Paul Hudak, Simon Peyton Jones, Philip Wadler, et al. Report on the functional programming language haskell. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[Kel89] R. A. Kelsey. *Compilation by Program Transformation.* PhD thesis, Yale University, Department of Computer Science, May 1989. YALEU/DCS/RR-702.

[Kra88] D. A. Kranz. *ORBIT - an optimising compiler for Scheme.* PhD thesis, Yale University, Department of Computer Science, May 1988.

[Par92] William Partain. The nofib benchmarking suite. In John Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow*, Scotland, 1992. Springer Verlag, Workshops in Computing.

[Pey87] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[PHH+93] Simon Peyton Jones, C. Hall, Kevin Hammond, William Partain, and Philip Wadler. The Glasgow Haskell Compiler: a technical overview. In *UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, March 1993.

[PL91] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens. In *Functional Programming Languages and Computer Architecture*, pages 636–666, September 1991.

[PPS96] Simon Peyton Jones, William Partain, and André Santos. Let-floating: moving bindings to give faster programs. In *ACM International Conference on Functional Programming*. ACM Press, 1996.

[San95] André Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, July 1995.