ARITY ANALYSIS
(Working Notes Only, Not Ready for Publication Yet)

Dana N. Xu and Simon Peyton Jones

September 1, 2006

Abstract

Virtually every compiler performs transformations on the program it is compiling in an attempt to improve efficiency. However, a transformation known as "lambda floating" has not received much attention in the past. In this paper we describe an analysis on the arity of a function which determines the number of lambdas that can be floated out. We give detailed measurements of the effect in an optimising compiler for the higher-order polymorphic functional language Haskell. Our results show that it is an important transformation offering an average reduction of 5% in execution time.

1 Introduction

Consider a summation function `h1` adopted from [7] which has an implicit accumulating parameter `y`.

```
h1 = \n -> \x -> if x<n then let v = h1 n (x+1)
                             in \y -> v (x+y)
                        else \y -> y
```

A semantically-equivalent function which differs only in the positioning of the `\y` is this:

```
h2 = \n->\x->\y-> if x<n then let v = h2 n (x+1)
                               in v (x+y)
                          else y
```

Given same arguments, the two functions produce the same value but `h2` is much more efficient than `h1` to evaluate. In this case, the time and space consumption is significantly reduced by 60% and 50% respectively. There are two reasons for this improvement:

1. a great number of lambda-closure-builds are saved during recursions.
2. many optimization techniques [**?**, **?**, **?**] can be applied to `h2` during compilation whereas few or no optimizations can be done for `h1`.

A question one may ask is "how often functions are defined with inner lambdas such as `h1`?". While programmers may not define functions in this way, however, compiled code which goes through many transformation stages often ends up in this form. For example, the `h1` definition results from the fusion of `fold (+) 0 [1..(n-1)]` by the short-cut deforestation technique proposed in [7]. Thus, a good compiler should transform `h1` into `h2` whenever possible. We call this transformation "lambda floating" as it floats the inner lambda(s) to the top level of its function definition.

The Glasgow Haskell Compiler (GHC) is an optimising compiler for the higher-order polymorphic functional language Haskell [18]. The compilation process is expressed as a series of correctness-preserving program transformations. To determine how many inner lambdas should be floated to the top level, it is equivalent to saying what arity we should give to a function where arity is defined as the number of arguments a function expects. GHC-6.4 (and earlier) takes the syntactic number of top level lambdas in a function definition as its arity without considering inner lambda(s) because it is a non-trivial task to find the ideal arity of each function. This challenge is discussed in §2.3. In this paper, we make the following contributions:

- We use a type system to analyse a function's arity independent of its callers. This analysis is effectively a forward arity analysis (§4.1).
- We propose a complementary backward arity analysis which derives the arity information of each (higher order) function's parameter based on the usage of each function (§4.2). This also helps in improving the precision of the arity information obtained by the forward analysis.
- We conduct experiments on the effectiveness of selective lambda floating guided by the two analyses (§5).

Our analysis results are not lazy-language-specific as we determine that funtion `f` has arity n if it is called at all. Thus, analogous transformations should be equally successful in other compilers for strict languages.

2 Background and key ideas

In this section, we explain with examples *why* we need an arity analysis, *how* the lambda floating can be done and *which* lambda should be floated out.

2.1 The cost of building a lambda

Given a function `f` of the form $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.e$ where e is not a lambda abstraction and `f` has no free variables, if `f` is given n arguments, most implementation will perform multi-reductions [17] whereby each call is viewed as a vector application, rather than a series of binary applications. For example, we have

```
ff1 = \x -> \y -> let {n = x+2} in n + y
```

Evaluation of `(ff1 1 5)` is as follows. (We use `|->` to represent β-reduction)

```
    (ff1 1 5)
|-> let n = 1 + 2 in n + 5
|-> let n = 3 in n + 5
|-> 3 + 5
```

In contrast, for a semantically-equivalent definition like

```
f1 = \x -> let {n = x + 1} in \y -> n + y
```

after taking the first argument, we need to build a lambda closure before taking the second argument.

```
    (f1 1 5)
|-> (let {n = 1 + 2} in \y -> n + y) 5
|-> (let {n = 3} in \y -> n + y) 5
|-> (\y -> 3 + y) 5   {- extra lambda closure -}
|-> 3 + 5
```

This shows that if we float the lambda `y` to the top level, we save building lambda once. Similarly, for the case-expression,

```
f2 = \x -> case x >0 of
                True -> \y -> x+y
                False -> \y -> x-y
ff2 = \x -> \y -> case x>0 of
                    True -> x+y
                    False -> x-y
```

Calls to execute `ff2` is faster than the corresponding one for `f2` as one lambda closure is saved as shown below.

```
(ff2 2 3)
-> case 2>0 of
      True -> 2 + 3
      False -> 2 - 3
-> 2+3

(f2 2 3)
-> (case 2>0 of
       True -> \y -> 2+y
       False -> \y -> 2-y) 3
-> (\y->2+y) 3       {- extra lambda closure -}
-> 2+3
```

Thus, the purpose of floating lambdas to the top level of a function's definition is to save building lambda closures during execution, where possible. The gain can best be seen in a recursive function definition. Consider the example `h1` in Secton 1, many lambda closures are built during recursive invocations. However, for the definition of `h2`, no lambda closure is built.

Arity of a function is informally defined as follows. Given `let f = \`x_1`,...,`x_m `-> e1 in e2`, `f` has arity n iff `let f = \`x_1`,...,`x_m`,`x_{m+1}`,...,`x_n`-> (e1 `x_{m+1}`,...,`x_n`) in e2` takes no more steps to evaluate. For example, functions `f1` and `f2` have arity 2 and `h1` has arity 3.

Arity analysis may also be used to provide guidance for a better fully-lazy lambda lifting technique and cheap deforestation. These two applications are discussed in Appendix A.

2.2 Floating lambda out by η expansion

The easiest way to float lambda(s) out to the top level of a function's definition is to apply η-expansion. The number of η-expansions that should be applied depends solely on the arity information obtained from our analyses.

Definition 1 (η-expansion) *In lambda-calculus, the η expansion rule states* `f ==> \x -> f x` *provided* `x` *does not occur free in* `f` *and* `f` *is a function.*

There are three rules of transformation (named beta-reduction, let-apply and if-apply shown in Figure 1) which preserve the operational semantics in a lazy language setting. We use `==>` to represent a one step transformation and `==>`* to represent multiple steps of transformation.

With these three rules, inner lambdas can be easily floated to the top level after applying a suitable number of η-expansions. For example, for a function of arity 2, we can apply η-expansion twice as follows.

```
  \x -> if x>0 then let m = x+1
                    in \y -> m + y
                else \z -> z
==>* [apply eta-expansion twice]
  \a1 -> \a2 -> ((\x -> if x>0 then let m = x+1
                                    in \y -> m + y
                               else \z -> z
                 ) a1 a2)
==> [apply beta-reduction]
  \a1 -> \a2 -> ((if a1>0 then let m = a1 + 1
                               in \y -> m + y
                          else \z -> z) a2
==> [apply if-apply law]
  \a1 -> \a2 -> (if a1>0 then (let m = a1+1
                               in \y -> m + y) a2
                         else (\z -> z) a2
==> [apply let-apply law]
  \a1 -> \a2 -> if a1>0 then let m = a1 + 1
                             in ((\y -> m + y) a2)
                        else (\z -> z) a2
==>* [apply beta reduction for each branch]
  \a1 -> \a2 -> if a1>0 then let m = a1 + 1
                             in m + a2
                        else a2
```

2.3 Analysing for lambda floating

The arity of a function indicates the ideal number of lambdas that should appear at the top level of a function's definition. Both forward and backward analyses are used in deriving such information. We give an overview of these two mechanims in this section. Technical details of how each step is implemented and what the actual gain achieved are discussed in later sections.

2.3.1 Forward arity analysis

We need to bear in mind that the purpose of floating lambda out is to improve efficiency of the code. Generally speaking, floating lambda out can be a disaster under some situations, whereby it *may* introduce high run-time overhead. For example,

```
  let f3 = \n -> let m = fib n
                 in \x -> x + m
```

$$(\lambda x \to e_1)\ e_2 \quad \texttt{==>} \quad (e_1[e_2/x]) \qquad \text{(beta-reduction)}$$
$$(\texttt{let}\ x = e_1\ \texttt{in}\ \lambda y \to e_2)\ e_3) \quad \texttt{==>} \quad \texttt{let}\ x = e_1\ \texttt{in}\ ((\lambda y \to e_2)\ e_3) \qquad \text{(let-apply)}$$
$$((\texttt{if}\ e_1\ \texttt{then}\ \lambda x \to e_2\ \texttt{else}\ \lambda y \to e_3)\ e_4) \quad \texttt{==>} \quad \texttt{if}\ e_1\ \texttt{then}\ ((\lambda x \to e_2)\ e_4)\ \texttt{else}\ ((\lambda y \to e_3)\ e_4) \qquad \text{(if-apply)}$$

Figure 1: Transformation Rules

```
   in map (f3 100) [1..1000]
==>*
   let f3 = \n -> \x -> let m = fib n
                        in x+ m
   in map (f3 100) [1..1000]
```

The transformed code will compute `(fib 100)` one thousand times while the original code only computes it once. This tells us that given an expression of the form `let x = e1 in e2`, if `e1` is an expensive expression, we may not be able to afford losing the sharing by floating out any lambda in `e2`. Thus, when we encounter such `e1`, we should assign arity 0 to the whole expression `let x = e1 in e2` to mean no lambda should be floated from this expression.

Consider another situation,

```
  let f4 = \n -> if (fib n > 1234) then \y -> y+1
                                   else \y -> y-1
  in map (f4 100) [1..1000]
==>*
  let f4 = \n -> \y -> if (fib n > 1234) then y+1
                                        else y-1
  in map (f4 100) [1..1000]
```

the expensive computation `(fib n)` in the original code is only computed once while the `(fib n)` in the transformed code will be computed 1000 times. An if-expression is subsumed by case-expression:

```
if e0 then e1 else e2  <==>  case e0 of True -> e1
                                        False -> e2
```

To generalise it, for expression `case e0 of {pi -> ei}`, if e0 is expensive, we should assign arity 0 to the whole case-expression regardless what arity each branch has.

We can see that some expressions have arity 0 although they contain some lambdas. This leads to another concern: for the different branches of the case alternatives, they may have different arity. To play safe, we need to choose the minimum arity among these branches. For example,

```
f5 = \n -> if n > 0 then let m = fib n
                         in \x -> x + m
                    else \y -> n - y
```

The then-branch gives arity 0 and the else-branch gives arity 1 so the whole if-expression has arity 0, that is to say function `f5` has arity 1.

To find the arity of a recursive function, we need to do a fixpoint computation to get its arity. Consider the `h1` in the §1,

```
h1 = \n -> \x -> if x<n then let v = h1 n (x+1)
                             in \y -> v (x+y)
                        else \y -> y
```

where both LHS and RHS has the same arity, we say a fixpoint is reached. Suppose we start from 0, that means we assume `h1` has arity 0, then we look at the RHS. As `h1` has arity 0, the expression `(h1 n (x+1))` is not cheap and has arity 0 (as arity cannot go negative). This leads the whole let-expression to have arity 0. Since one branch has arity 0, the other branch has arity 1, the if-expression has the minimum of these two which is 0. Thus, the RHS has arity 2. As the arity at LHS is 0 and RHS gives arity 2, a fixpoint is not reached. Let us increase the arity at LHS to 2 directly. After going through the similar procedure, we get arity 2 for the RHS. We reach a fixpoint. However, this is not the end of the story. If we continue increasing the arity at LHS to 3, we will get arity 3 for the RHS - another fixpoint. This means if we start fixpoint computation from 0, we will get the least fixpoint. Unfortunately, this is not what we want, we want the greatest fixpoint as we want to float as many lambda(s) out as possible under safe conditions. Thus, we propose to start computing fixpoint from ∞. In § 4.1.1, explanantion on the computation of greatest fixpoint is given.

For higher-order function like `f = \g -> \x -> g x 1`, as we do not have information about `g`, we say `f` has arity 2. In order to derive the arity of a function's parameter, we need to analyse the usage of a function. A mechanism that does this is explained in the next section.

2.3.2 Backward arity analysis

Forward analysis finds only the arity for each function, but not arity for each of its parameter if the parameter is a function itself, while backward analysis can do the job. Backward analysis of a function aims to answer the following question:"given arity of the function's result, what are the arities of the function's parameters?". The more precise the arity we have for the function's result, the more precise the arities we have for the parameters. For this purpose, a function may be seen as a monotonic *arity transformer*: it transforms an arity of the function's result into arities of the function's parameters. For example:

```
let g = ...
    s f  = f 3
in ...(s g)...
```

An arity transformer for `f` can produce the following table:

`(s g)`	`f`
0	1
1	2
2	3
:	:
n	$n+1$

It says if `(s g)` applies to n arguments, then `f` applies to $n+1$ arguments. Suppose, we see an application `(s g 4 5 6)` which implies `(s g)` has arity 3, by looking up the table above, we know `f` has arity 4.

The backward analysis strategy is essentially built for each let-expression because the toplevel function definition can be modeled in the same way as a let-binding. In the rest of this section, we describe three analysis plans to compute an arity transformer for expression of this format.

$$\texttt{let}\ \{f = rhs\}\ \texttt{in}\ body.$$

Plan A. Intuition tells us that we need to analyse *body* before *rhs*. For example:

```
let f10 = \g ->\x -> g x
in f10 h 2 3
```

From the *body*, we can get {`f10::3`}. If we feed this usage information to the *rhs*, we can get {`(\g->\x->g x)::3`} which gives {`g::2`}. However, if we have this:

```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f [] ys = []
zipWith f xs [] = []
zipWith f (x:xs) (y:ys) = (f x y):zipWith xs ys

f9 foo xs ys = let goo = foo 6 7 8
               in zipWith goo xs ys
```

Without analysing the definition of `zipWith`, its usage (i.e. `zipWith goo xs ys`) can only give information {`zipWith::3`} which says nothing about `goo`. So **P**lan A is not desirable.

There are more things to consider before we introduce **P**lan B, for example:

```
let f7 = \x -> if (fib 100) then \y-> x+y
                            else \y-> x*y
in zipWith f7 [1..1000] [1..1000]
```

in this case, it is safe to float `\y->` out (i.e. assigning `f7` arity 2) as `f7` cannot do any work without taking two arguments, as this is enforced by the definition of `zipWith`. (Note: with or without floating `\y->` out, `(fib 100)` will be computed for 1000 times.) However, for definition like

```
let f8 = if (fib 100) then \x->\y->x+y
                      else \x->\y->x*y
in zipWith f8 [1..1000] [1..1000]
```

it is not safe to float any of the `\x->` and `\y->` out (i.e. should assign `f8` arity 0) as the expensive computation `(fib 100)` will be evaluated 1000 times. Thus, the backward analysis is not only to count the number of times a function is used, but also should capture how the function is used. This leads to the idea of deriving a usage type which captures the usage of each free variable in an expression. For example, in the application `zipWith f8 [1..1000] [1..1000]`, the usage type for `f8` is $\langle 1, 2\rangle$ which says if `f` applies to 1 argument, it will definitely apply to 2 arguments.

Definition 2 (Usage Type) *$\langle a, b\rangle$ is a usage type of a function f if no partial application of f to i arguments $\forall i.a \leq i < b$ is shared.*

Plan B. Analyse the *rhs*, then the *body*, then the *rhs* again, and the *body* again. This iteration continues until it reaches a fix point. In the case of `f9`, from the definition of `zipWith`, we know its first parameter `f` has usage type $\langle 1, 2\rangle$. From the definition of `goo`, we know `foo`'s usage type is $\langle 1, 3\rangle$ which states if `foo` applies to either 1 or 2 arguments, it will definitely apply to 3 arguments. This gives the first row of Table 1 where "?" means the arity of `goo` is unknown. From the *body*, we know `goo` has usage type $\langle 1, 2\rangle$. Thus, `foo 6 7 8` shares the same usage type which means if `foo 6 7 8` applies to 1 argument, it will definitely apply to 2 argument. This means if `foo` applies to 4 arugments, it will definitely apply to 5 arguments. Thus, the usage of `foo` is $\langle 4, 5\rangle$. This gives the second row of Table 1. Note that the $\langle 1, 3\rangle$ and $\langle 4, 5\rangle$ cannot be combined to form $\langle 1, 5\rangle$ as explained in the example with `f8` which is used *many* times by its caller. Linearity does influence the usage type of a variable, for example, we have:

Table 1: Arity Transformer

goo	foo
?	$\langle 1, 3\rangle$
$\langle 1, 2\rangle$	$\langle 4, 5\rangle$

```
t1 = \x -> (x 1 2, True)
t2 = \y -> (y 1 2, y 3 4)
```

As `x` is used *once*, the usage type of `x` is $\langle 0, 2\rangle$ while that of `y` is $\langle 1, 2\rangle$ as it is used *more than once*. An operator $\sqcap$ will take care of this, specifically both `y 1 2` and `y 3 4` give $\langle 0, 2\rangle$ and $\langle 0, 2\rangle \sqcap \langle 0, 2\rangle = \langle 1, 2\rangle$. Formal definition of $\sqcap$ can be found in §4.2. Although this sophisticated analyse could capture the full glory of an arity transformer, the complexity of this analysis is exponential for heavily nested let-expressions. So our analyser instead uses a brutal, yet effective, approximation: **Plan C**.

Plan C. We choose to use a lighter version of **Plan B** by deriving its parameter's usage solely from the function's definition. In the case of `f9`, only the first row of the Table 1 is computed.

2.4 seq and error

Two special operators `seq` and `error` in Haskell make the analyses in a lazy language setting invalid. These operators are special in that they perform strict evaluation on their argument(s). The η-expansion is only valid if $\perp$ and `\x->`$\perp$ are equivalent in all contexts. They are certainly equivalent when applied to some argument - they both fail to terminate. If we are allowed to force the evaluation of an expression in any other way, e.g. using `seq` in Haskell then $\perp$ and `\x->`$\perp$ will not be equivalent. The function `seq :: a -> b -> b` evaluates its first argument before returning its second. For example, given

```
f11 = \x -> if x then \y -> 1
                 else \y -> 2

ff11 = \x -> \y -> if x then 1 else 2
```

`(f11` $\perp$ `` `seq` True)`` diverges whereas `(ff11` $\perp$ `` `seq` True)`` converges. This means if we apply η-expansion to `f11`, the program may terminate more often. Another function we need to take into consideration is `error :: String -> a` which takes a `String`, prints the string, and brings execution to a halt. From the semantics point of view, `error` is considered as $\perp$ or divergence. If we η-expand `f12` with:

```
f12 = \x -> if x < 0 then error ``...''
                     else \y -> \z -> x+y+z
```

we will have:

```
ff12 = \x -> \y -> \z -> if x < 0 then error ``...''
                                  else x + y + z
```

which converges more often then `f12`.

Although these two operators violate the semantic equivalence of η-expansion, in common cases, the transformed code is more efficient. Moreover, making programs converge more often is not a bad thing at all. Thus, we choose to perform η-expansion on them, by default. However, users intend to adhere strictly to the semantics of `seq`/`error`, we can analyse where they are used and then force them to have arity 0 to prevent lambda-floating.

3 Language and Cost Framework

Before we give a formal arity analysis and the "lambda-floating" transformation themselves, we must first introduce the language we use. Figure 2 shows the abstract syntax of the Core, the intermediate language of the GHC [22] with omission of some features that obstruct the clarity of the presention of the key ideas. (Our implementation supports the full Core language.)

$$
\begin{array}{rcll}
n & \in & \texttt{Const} & \textbf{Constants} \\
c & \in & \texttt{Constr} & \textbf{Constructors} \\
v, x & \in & \texttt{Var} & \textbf{Variables} \\
p & \in & \texttt{Pat} & \textbf{Patterns} \\
e & \in & \texttt{Exp} & \textbf{Expressions} \\
val & \in & \texttt{Val} & \textbf{Values} \\
e & ::= & n \mid v \mid \lambda x.e \mid e_1\ v & \\
 & & \mid \texttt{case}\ e_0\ \texttt{of}\ alts & \\
 & & \mid \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 & \\
 & & \mid \texttt{letrec}\ [v_i\ =\ e_i]_{i=1}^{n}\ \texttt{in}\ e_0 & \\
alts & ::= & \{\overrightarrow{p} \rightarrow \overrightarrow{e}\} & \\
p & ::= & c\ x_1 \ldots x_n & \\
val & ::= & n \mid v \mid \lambda x.e &
\end{array}
$$

Figure 2: Syntax of the source language.

Only a value denoted by *val* is considered a cheap expression where *val* is defined in Figure 2.

Definition 3 (Expression is Cheap) *Expression e is cheap iff e is a val.*

Throughout the paper we take a few liberties with the syntax: we allow infix operators and `if`-expressions. We use letter e and e_i (where $i \geq 0$) to represent arbitrary expression and use $\overrightarrow{x}$ to represent $x_1\ \ldots\ x_n$. We use `\x-> e` and $\lambda x.e$ interchangeablly to represent lambda abstraction. We allow $e_1\ e_2$ as it can easily be converted to $\texttt{let}\ v = e_1\ \texttt{in}\ e_2$. We also allow multiple definitions in a single let-expression to abbreviate a sequence of nested let-expressions. We write $\texttt{let}\ \{\overrightarrow{x} = \overrightarrow{e_1}\}\ \texttt{in}\ e_2$ as short hand for $\texttt{let}\ \{x_1 = e_1, \ldots, x_n = e_n\}\ \texttt{in}\ e_0$. A list of mutual recursive functions are grouped together with the construct `letrec`. A singleton in the `lecrec` construct means a self-recursive function.

3.1 Operational Semantics

In order to reason about the usefulness and correctness of a transformation we must have a model that calculates costs to execute it. Figure 3 shows the call-by-need semantics adopted from [15] in which a tick-algebra is described to count the costs of execution. One transition $\rightarrow$ is counted as one tick.

Transitions are over configurations consisting of a heap Γ (which contains bindings), the expression currently being evaluated e, and a stack S:

$$
\begin{array}{rcl}
\Gamma & ::= & \{x = e\} \\
S & ::= & (v \mid alts \mid \#x)^*
\end{array}
$$

The heap is a partial function from variable to terms whereas the stack may contain variables (the arguments to applications), case alternatives, or *update markers* denoted by $\#x$ for some variable x. Update markers ensure that a binding to x will be recreated in the heap with the result of the current evaluation; this allows sharing to be captured by the semantics.

Formal convergence of a closed configuration is defined as follows.

Definition 4 (Convergence) *For closed configurations $\langle \Gamma, e, S \rangle$,*

$$
\begin{array}{rcl}
\langle \Gamma, e, S \rangle \Downarrow^n & =_{def} & \exists \Gamma', val. \langle \Gamma, e, S \rangle \rightarrow^n \langle \Gamma', val, \epsilon \rangle, \\
\langle \Gamma, e, S \rangle \Downarrow & =_{def} & \exists n. \langle \Gamma, e, S \rangle \Downarrow^n, \\
\langle \Gamma, e, S \rangle \Downarrow^{\leq n} & =_{def} & \exists m. \langle \Gamma, e, S \rangle \Downarrow^m \wedge\ m \leq n.
\end{array}
$$

3.2 Improvement

The η-expansion and rules in Figure 1 are known semantics-preserving transformations. Thus, the lambda-floating transformation is semantics-preserving. In order to show the significance of the arity analysis, we need to show that the transformed code *improves* the orginal one with respect to execution time. We adopt the terminologies and results in [15]. Contexts are in the following form:

$$
\begin{array}{rcl}
\mathcal{C}, \mathcal{D} & ::= & [.] \mid x \mid \lambda x.\mathcal{C} \mid \mathcal{C}x \mid c\ \overrightarrow{x} \\
 & & \mid \texttt{let}\ \{x = \mathcal{C}\}\ \texttt{in}\ \mathcal{D} \\
 & & \mid \texttt{case}\ \mathcal{C}\ \texttt{of}\ \{c_i\ \overrightarrow{x_i} \rightarrow \mathcal{D}_i\}
\end{array}
$$

Definition 5 (Improvement) *We say that e is improved by e', written $e \trianglerighteq e'$, if for all $\mathcal{C}$ such that $\mathcal{C}[e]$ and $\mathcal{C}[e']$ are closed,*

$$\mathcal{C}[e]\Downarrow^n \Rightarrow \mathcal{C}[e']\Downarrow^{\leq n}$$

The notation $\mathcal{C}[e]\Downarrow^n$ and $\mathcal{C}[e]\Downarrow^{\leq n}$ identify closed expression $\mathcal{C}[e]$ with initial configuration $\langle \emptyset, \mathcal{C}[e], \epsilon \rangle$.

4 Type Systems

The two analyses mentioned in §2.3 are modelled using two type systems which are described in §4.1 and §4.2 respectively. The soundness of these two type systems can be proven with a proof technique that is similar to the improvement theory in [15]. Details is still being worked out and omitted in this paper.

4.1 Forward Arity Analysis

An intuitive approach of finding the arity of a function is to create a counter, initiate it to 0, and whenever we encounter a lambda, we increase it by one and whenever we encounter an application, we decrease it by one. Let us treat such arity counter as the type of an expression. For example, given the arity of (+) is 2, `((+) x)` has arity $2 - 1 = 1$, `((+) x y)` has arity $1 - 1 = 0$, `(\y -> (+) x y)` has arity $0 + 1 = 1$ and `(\x->\y->(+) x y)` has arity $1 + 1 = 2$.

$$
\begin{array}{rcll}
\langle \Gamma\{x = e\}, x, S\rangle & \rightarrow & \langle \Gamma, e, \#x : S\rangle & (Lookup)\\
\langle \Gamma, val, \#x : S\rangle & \rightarrow & \langle \Gamma\{x = val\}, val, S\rangle & (Update)\\
\langle \Gamma, e\ x, S\rangle & \rightarrow & \langle \Gamma, e, x : S\rangle & (Unwind)\\
\langle \Gamma, \lambda x_1 \ldots \lambda x_n.e, y_1 : \cdots : y_n : S\rangle & \rightarrow & \langle \Gamma, e[\overrightarrow{y_i}/\overrightarrow{x_i}], S\rangle & (Subst)\\
\langle \Gamma, \texttt{case}\ e\ \texttt{of}\ alts, S\rangle & \rightarrow & \langle \Gamma, e, alts : S\rangle & (Case)\\
\langle \Gamma, c_i\ \overrightarrow{y}, \{c_i\ \overrightarrow{x_i} \rightarrow e_i\} : S\rangle & \rightarrow & \langle \Gamma, e_j[\overrightarrow{y}/\overrightarrow{x_j}], S\rangle & (Branch)\\
\langle \Gamma, \texttt{let}\ \{\overrightarrow{x} = \overrightarrow{e}\}\ \texttt{in}\ e_0, S\rangle & \rightarrow & \langle \Gamma\{\overrightarrow{x} = \overrightarrow{e}\}, e_0, S\rangle \quad \overrightarrow{x} \notin dom(\Gamma, S) & (Letrec)
\end{array}
$$

Figure 3: The abstract machine semantics for call-by-need.

Following this idea, we are now ready to define the full type system. A type assumption Δ binds program variables to their arity. A judgement for forward analysis has the form

$$\Delta \vdash e :: \eta$$

This states that given Δ, `e` has arity η assuming that any free variable in it has arity given by Δ. The type expression is defined as follows.

$$\eta \quad ::= \quad \infty \mid 0 \mid 1 \mid 2 \mid \ldots$$

Type η ranges over non-negative integers from `0` to ∞ which forms a lattice whose bottom value is `0` and top value is ∞. It is always safe to assign arity `0` to an expression as it means no lambda can be floated out from the expression. We write $[\![\eta]\!]$ to denote the semantics of the type η. Thus,

$$[\![\eta]\!] = \{e \mid \exists e'.\lambda x_1..x_\eta.(e\ x_1..x_\eta) \texttt{==>}^* e'\ \&\ \mathcal{C}[e]\Downarrow^n \Rightarrow \mathcal{C}[e']\Downarrow^{\leq \eta}\}$$

4.1.1 Typing Rules

Figure 4 lists a set of typing rules. For a constant, it is obvious that it should have arity `0`. A variable `v` has type η if the type assumed for `v` in Δ is η. In the rule `(ABS)`, we assume the bound variable `x` has arity `0` when finding the arity of e. This is because in the forward arity analysis we are only interested in finding a function's arity and we do not find arity information of its parameters even though its parameter may be a function itself. Since the expression `\x -> e` has one more parameter than the expression `e` which has type η, its type should be $\eta + 1$. For example, we may have the following derivation

$$\frac{\{\texttt{double::1,x::0}\} \vdash \texttt{(double x) :: 0}}{\{\texttt{double::1}\} \vdash \texttt{(\x -> double x) :: 1}}$$

For function application rule `(APP)`, as the number of arities cannot be negative, if e_1 has arity 0, the application $(e_1\ v)$ has arity 0. On the other hand, if e_1 has arity bigger than 0, after it is being applied to an actual parameter v, the resulting arity should be $\eta - 1$. For example, we assume `h1` is a function having arity 3 and $\Delta = \{\texttt{h1:3,n::0,x::0}\}$, we may have this typing.

$$\frac{\dfrac{\Delta \vdash \texttt{h1::3}}{\Delta \vdash \texttt{(h1 n) :: 2}} \quad \Delta \vdash \texttt{(x + 1) :: 0}}{\Delta \vdash \texttt{(h1 n (x + 1)) :: 1}}$$

As explained in §2.3.1, we need to check expensiveness when we encounter case-expression or let-expression. For the case-expression, if e_0 is a val, we analyse expressions at each branch and find the minimum of their arities. The operator $\sqcap$ is defined as $\eta_1 \sqcap \eta_2 = min(\eta_1, \eta_2)$. However, if e_0 is an expensive expression, we do not bother to check the expressions at branches and simply return 0. This means it is not worth floating any lambda out of this expression. Similar ideas apply to `(LET)` and `(LETREC)`.

One thing that must be mentioned is that in `(LETREC)`, it involves an iteration to find fixpoint of η_1. We choose to find the greatest fixpoint. In our inference algorithm, we assign ∞ to η_1 initially. Consider the recursive function `h1` in the §1, it is obvious that the whole let-expression has arity ∞ if `h1` has arity ∞. As `(\y -> y)` has arity 1 and we choose the minimum arity of the two branches, the whole if-expression has arity 1 which means the RHS has arity 3. We can decrease the arity of `h1` from ∞ to 3 directly. After going through the same procedure, we find that RHS has arity 3. We reach a fixpoint - a greatest fixpoint! Termination of this iteration is proven in Appendix B. The reason we want to find greatest fixpoint is that we want to float as many lambdas out as possible.

4.2 Backward Arity Analysis

In the forward analysis, we focus on finding the arity of a function whereas in the the backward analysis, we focus on finding the arity information of the function's each parameter as well as a more precise arity of the function by analysing its usage at its call site(s).

4.2.1 Type Expression

An extended set of type expressions are defined as follows.

$$
\begin{array}{lcl}
v & \in & \textbf{Var}\\
\eta & ::= & \infty \mid 0 \mid 1 \mid 2 \mid \ldots\\
\psi & ::= & \langle \eta_1, \eta_2\rangle\\
\Phi & ::= & \emptyset \mid \{v :: \psi\}\\
\delta & ::= & \bullet \mid \psi \rightarrow \delta \mid \top
\end{array}
$$

The η still captures the arity of a variable like the one in the forward analysis. We introduce Φ which contains a set of bindings of variables to their usage type, ψ. As we need to capture arity information of each parameter of a function, we introduce δ. The symbol $\bullet$ says "don't care" whereas the symbol $\top$ indicates the arity is unknown. The $\psi \rightarrow \delta$ gives arity of each parameter of a function. For example, we have function definition `f h x = h x 0`, the expression `(\h -> \x -> (h x 0))` has type $\langle 0, 2\rangle \rightarrow \langle 0, 0\rangle \rightarrow \bullet$. During the backward analysis, we do not derive arity for a function itself when we analyse its definition, thus, we give it $\bullet$. A function's usage type is usually captured in Φ. The operations on usage types are defined in Figure 5.

Sometimes, we use shorthand 0 to represent $\langle 0, 0\rangle$, ∞ to represent $\langle \infty, \infty\rangle$ and $\bullet$ to denote a function type `0 -> ... ->` `0 ->` $\bullet$ which has type 0 for each parameter and $\top$ to denote a function type ∞ `->` $\ldots$ `->` ∞ `->` $\top$ whose each parameter's

$$\frac{}{\Delta \vdash n :: 0} \quad (\texttt{CONST}) \qquad \frac{x :: \eta \in \Delta}{\Delta \vdash x :: \eta} \quad (\texttt{VAR}) \qquad \frac{\Delta \cup \{x :: 0\} \vdash e :: \eta}{\Delta \vdash (\lambda x.e) :: \eta + 1} \quad (\texttt{ABS})$$

$$\frac{\Delta \vdash e_1 :: \eta_1 \qquad \Delta \vdash v :: \eta_2}{\Delta \vdash (e_1\ v) :: max(0, \eta_1 - 1)} \quad (\texttt{APP})$$

$$\frac{\Delta \vdash e_i :: \eta_i}{\Delta \vdash (\texttt{case}\ val\ \texttt{of}\ \{\overrightarrow{p_i} \rightarrow \overrightarrow{e_i}\}) :: \sqcap \eta_i} \quad (\texttt{CASE1}) \qquad \frac{\Delta \vdash e_i :: \eta_i \qquad e_0 \neq val}{\Delta \vdash (\texttt{case}\ e_0\ \texttt{of}\ \{\overrightarrow{p_i} \rightarrow \overrightarrow{e_i}\}) :: 0} \quad (\texttt{CASE2})$$

$$\frac{\Delta \vdash val :: \eta_1 \qquad \Delta, v :: \eta_1 \vdash e_2 :: \eta_2}{\Delta \vdash (\texttt{let}\ v\ =\ val\ \texttt{in}\ e_2) :: \eta_2} \quad (\texttt{LET1}) \qquad \frac{\Delta \vdash e_1 :: \eta_1 \qquad \Delta, v :: \eta_1 \vdash e_2 :: \eta_2 \qquad e_1 \neq val}{\Delta \vdash (\texttt{let}\ v\ =\ e_1\ \texttt{in}\ e_2) :: 0} \quad (\texttt{LET2})$$

$$\frac{\Delta \cup \{v_i :: \eta_i\}_{i=1}^n \vdash val_i :: \eta_i \qquad \Delta \cup \{v_i :: \eta_i\}_{i=1}^n \vdash body :: \eta_b}{\Delta \vdash (\texttt{letrec}\ [v_i\ =\ val_i]_{i=1}^n\ \texttt{in}\ e_2) : \eta_b} \quad (\texttt{LETREC1})$$

$$\frac{\Delta \cup \{v_i :: \eta_i\}_{i=1}^n \vdash e_i :: \eta_i \qquad \Delta \cup \{v_i :: \eta_i\}_{i=1}^n \vdash body :: \eta_b \qquad e_i \neq val_i}{\Delta \vdash (\texttt{letrec}\ [v_i\ =\ e_i]_{i=1}^n\ \texttt{in}\ e_2) : 0} \quad (\texttt{LETREC2})$$

Figure 4: Forward Arity Analysis Typing Rules

& and $\sqcup$ operations

$$\begin{array}{rcll} \langle 0, \eta_1 \rangle & \& & \langle 0, \eta_2 \rangle & = \langle 1, min(\eta_1, \eta_2) \rangle \\ \langle \eta_1, \eta_2 \rangle & \& & \langle \eta_3, \eta_4 \rangle & = \langle min(\eta_1, \eta_3), min(\eta_2, \eta_4) \rangle \end{array}$$

$$\begin{array}{rcll} \langle 0, \eta_1 \rangle & \sqcup & \langle 0, \eta_2 \rangle & = \langle 0, min(\eta_1, \eta_2) \rangle \\ \langle \eta_1, \eta_2 \rangle & \sqcup & \langle \eta_3, \eta_4 \rangle & = \langle min(\eta_1, \eta_3), min(\eta_2, \eta_4) \rangle \end{array}$$

$$\begin{array}{rcll} \bullet & \sqcup & \delta & = \bullet \\ \top & \sqcup & \delta & = \delta \\ \psi_1 \rightarrow \delta_1 & \sqcup & \psi_2 \rightarrow \delta_2 & = (\psi_1 \sqcup \psi_2) \rightarrow (\delta_1 \sqcup \delta_2) \end{array}$$

Ordering on usage types

$$\bullet \sqsubseteq \delta \qquad \delta \sqsubseteq \top \qquad \frac{\eta_1 \leq \eta_3 \qquad \eta_2 \leq \eta_4}{\langle \eta_1, \eta_2 \rangle \sqsubseteq \langle \eta_3, \eta_4 \rangle}$$

$$\frac{\psi_2 \sqsubseteq \psi_1 \qquad \delta_1 \sqsubseteq \delta_2}{\psi_1 \rightarrow \delta_1 \sqsubseteq \psi_2 \rightarrow \delta_2}$$

Figure 5: Operations on usage types

type is ∞. A function $splitParaFun$ is used in splitting a function's first parameter and its result. For example, $splitParaFun(\langle 0,3\rangle\texttt{->}\langle 1,2\rangle\texttt{->}\bullet)$ gives $(\langle 0,3\rangle, \langle 1,2\rangle\texttt{->}\bullet)$.

$$\begin{array}{rcl} splitParaFun(\bullet) & = & (0, \bullet) \\ splitParaFun(\rightarrow) & = & (\infty, \top) \\ splitParaFun(\psi \rightarrow \delta) & = & (\psi, \delta) \end{array}$$

4.2.2 Typing Rules

A type assumption Π binds program variables to their usage type δ and a usage type ψ tells the usage of `e`. A judgement for backward analysis has the form

$$\Pi; \psi \vdash e :: (\delta, \Phi)$$

This judgement states that if e is a lambda abstraction and the result of e is used in a way stated by ψ, then δ gives usage type of each parameter of e and Φ gives the usage type of each free variable in e.

Figure 6 lists the backward arity analysis typing rules. The rule `(CONST)` and `(VAR1)` are rather obvious. Let us look at the rule `(ABS)` before the rule `(VAR2)`. In rule `(ABS)`, if $\lambda x.e$ has usage type $\langle 0, 2\rangle$, then e will have usage type $\langle 0, 1\rangle$. If $\lambda x.e$ has usage type $\langle 1, 2\rangle$, then e will have usage type $\langle 0, 1\rangle$ as e takes one less argument. As x is bound in the expression $\lambda x.e$, $\{x :: \psi_x\}$ is removed from Φ which only captures *free* variable's usage type. The expression e may contain a free variable that is not bound in Π and hence the rule `(VAR2)` is needed.

Operators $\otimes$ and $\oplus$ in the rest of the rules are used in merging two sets. They operate differently when $(v :: \psi)$ appears in both sets. One apply operator & and and the other applies $\sqcup$ to the two usage types. The operators $\otimes$ and $\oplus$ are defined as follows.

$$\begin{array}{l} \Phi_1 \otimes \Phi_2 = \forall (v_i :: \psi_i) \in \Phi_1, (v_j :: \psi_j) \in \Phi_2. \\ \qquad\qquad if\ v_i == v_j \\ \qquad\qquad then\ \{v_i :: (\psi_i\ \&\ \psi_j)\} \cup \\ \qquad\qquad\quad ((\Phi_1 \cup \Phi_2) - \{v_i :: \psi_i, v_j :: \psi_j\}) \\ \qquad\qquad else\ \Phi_1 \cup \Phi_2 \end{array}$$

$$\begin{array}{l} \Phi_1 \oplus \Phi_2 = \forall (v_i :: \psi_i) \in \Phi_1, (v_j :: \psi_j) \in \Phi_2. \\ \qquad\qquad if\ v_i == v_j \\ \qquad\qquad then\ \{v_i :: (\psi_i \sqcup \psi_j)\} \cup \\ \qquad\qquad\quad ((\Phi_1 \cup \Phi_2) - \{v_i :: \psi_i, v_j :: \psi_j\}) \\ \qquad\qquad else\ \Phi_1 \cup \Phi_2 \end{array}$$

In the rule `(APP)`, function $splitParaFun$ gives ψ_2 for type checking e_2. It is obvious that e_1's arity increases by one before being applied to e_2. For case-expression, since forward analysis already handles the case when e_0 is expensive, we do not need to repeat it in the backward analysis. As we want to find usages of each function, we analyse e_0 as well as e_i at each branch and pairwise finding the least upper bound of δ_i with the operator $\sqcup$ defined in Figure 5.

For let-expression, as explained earlier, we choose **Plan C**. So we give usage 0 when type checking e_1. Similar reasoning holds for the rule `(LETREC)`. You may notice that $\{v_i :: \psi_i\}$ does not appear in the resulting Φ. Our inference algorithm returns an new expression with these arity

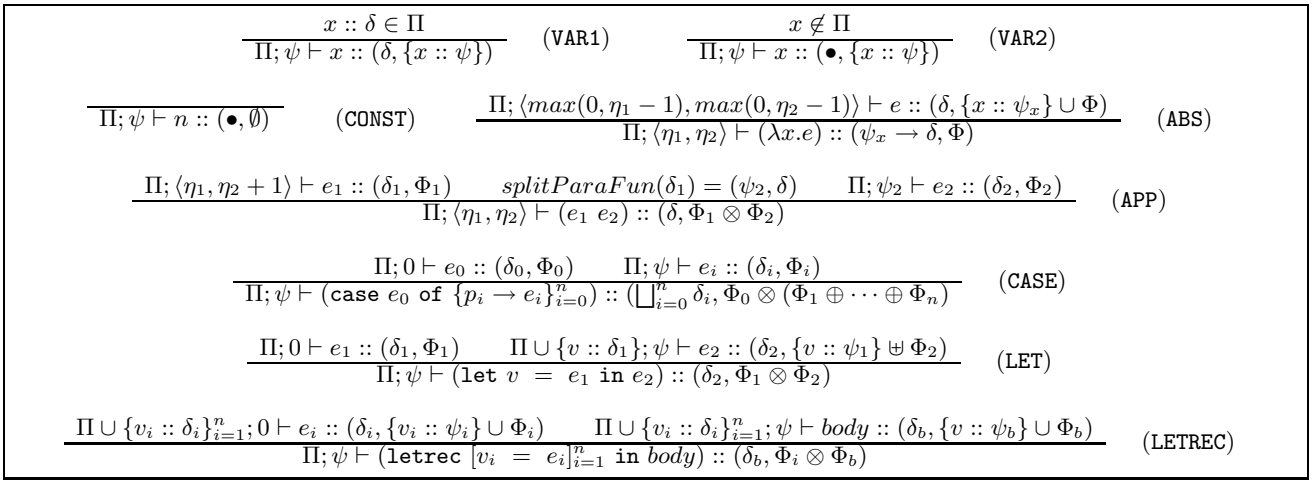

Figure 6: Backward Arity Analysis Typing Rules

information attached to each bound variable besides returning (δ, Φ). In the case of `(LET)`, expression `let` $v :: \psi_1 = e_1$ `in` e_2 is returned and in the case of `(LETREC)`, expression `letrec` $v :: (\psi_i \otimes \psi_b) = e_1$ `in` e_2 is returned.

4.3 Combining FW and BW Arity Analysis

The forward arity analysis and the backward arity analysis can be applied independently which means from each analysis, we can get the arity of a function and can apply the corresponding number of η-expansions to the function. For example, we have:

```
let f13 = \x -> if expensive_exp
                then \y -> e1
                else \y -> e2
in f13 1 2
```

In the forward arity analysis, from the definition of `f13`, we see that the if-test is an expensive expression, so `\y->` will not be floated out. Thus, no η-expansion is done which means the analysis gives safe result. On the other hand, in the backward analysis, from the usage of `f13`, the application `(f13 1 2)`, we have $\Phi = \{$`f13` $:: \langle 0, 2\rangle\}$. This tells us that `f13` is called *once* and `f13` is always applied to two arguments. According to this information, it is safe to float `\y->` out.

However, backward analysis cannot replace forward analysis because if we have:

```
let f14 = if val
             then \x->\y -> e1
             else \y -> e2
in zipWith f14 [1..1000] [1..1000]
```

the backward arity analysis cannot tell it is safe to float `\x->\y->` out so it chooses to tell the compiler not to float them out as explained in § 2.3.2. However, the forward arity analysis can tell it is safe to float them out as `val` is cheap and we can afford duplicate its computation.

Both analyses have their own duty and strength. As backward arity analysis also captures the arity information of each function's parameter which can improve the precision of the forward analysis, we choose to apply the backward arity analysis before the forward arity analysis in our implementation. Detailed measurement for the performance of each analysis as well as their combination is presented in the next section.

5 Measurement

We measure the effect of our transformations on a sample consisting of XX "spectral" programs and XX "real" programs from our NoFib test suite [16]. The programs range in size from a few hundread to a few thousand lines of Haskell. Programs are run on a PC with Intel Pentium M processor 1.80GHz and 1.00GB of RAM.

The table 5 summarises the effect of the forward arity analysis, the backward arity analysis and the total effect of the two analyses compiled with `ghc-6.5 -O -farityfw`, `ghc-6.5 -O -faritybw` and `ghc-6.5 -O -farity` respectively where the option `-farity` indicates "compiling with arity analysis". The number of inner lambdas being floated out is counted and the -X% shows the time reduction (in percentage) after lambda floating transformation is done based on the result of the arity analysis. The last three lines give the mininum, maximum and geometric mean of the last column.

6 Related Work

Using correctness-preserving transformation as a compiler optimization is certainly a well established technique [1, 3]. In the functional programming area especially the idea of compilation by transformation has received quite a lot of attention [14, 12, 13, 6, 2, 20].

Generally speaking, it is dangerous to float lambda(s) out as it raises the potential of duplicating computation of whatever expression used to be above the lambda. Particularly, in [20], it shows in some cases, floating `let` out of `lambda` (i.e. the reverse transformation of the lambda-floating) helps improve performance. Perhaps this is why there are few papers about how to float lambda out safely. In the chapter 4.4 of [7], Gill first pointed out (in 2 pages) the need of having an arity analysis which determines the number of inner lambdas that should be floated out as the resulting transformation saves building lambda closures as

Table 2: How "lambda floating" compares with -O

Program	Forward		Backward		Backward+Forward	
	No. of λs floated	-farityfw	No. of λs floated	ghc -O -faritybw	No. of λs floated	ghc -O -farity
sum	X	X	X	X	X	-60%
gen_regexps	X	X	X	X	X	-50%
maillist	X	X	X	X	X	-33%
multiplier	X	X	X	X	X	-11.11%
typecheck	X	X	X	X	X	-5.17%
compress	X	X	X	X	X	-4.35%
nucleic2	X	X	X	X	X	-4.35%
sphere	X	X	X	X	X	-4.17%
listcompr	X	X	X	X	X	-3.57%
simple	X	X	X	X	X	-3.45%
event	X	X	X	X	X	-3.23%
wang	X	X	X	X	X	-3.13%
solid	X	X	X	X	X	-2.53%
treejoin	X	X	X	X	X	-2.44%
para	X	X	X	X	X	-2.34%
life	X	X	X	X	X	-2.00%
primetest	X	X	X	X	X	-1.45%
tak	X	X	X	X	X	-1.26%
wave4main	X	X	X	X	X	-1.11%
lcss	X	X	X	X	X	-1.05%
Min	X	X	X	X	X	-1.05%
Max	X	X	X	X	X	-60%
Geo. Mean	X	X	X	X	X	X%

well as exposes more opportunities for further optimization. This inspired us to make a deeper exploration of the effectiveness of the arity analysis.

Besides our work, *arity raising* [21, 8] and *lambda lifting* [11] also increase a function's arity, however, these three pieces of work are orthogonal though their names are similar. Arity raising transforms a function of one parameter into a function of several parameters by decomposing the structure of the original one parameter into individual components in that structure Lambda lifting is a program transformation to remove free variables. An expression containing a free variable is replaced by a function applied to that variable.

Backward analysis was first introduced in [9] and the difficulty of analysing higher-order function in a backward fashion is discussed in the Section 6 of [9] by giving an example `(apply f x)` in which the information about `x` is unknown from the definition of `apply`. Fortunately, in this paper we are only interested in finding arity of a function. In this case, `x` is not a function, thus the precision of its arity information is not crucial and we can simply assign `x` arity 0 as an approximation.

Computable backward analysis for higher-order functional programs is discussed in [4], but the technique of reversing abstract interpretation has not been successfully applied to perform absence analysis. The arity analysis shares the same model as absence analysis, so the technique cannot be applied here either.

Eta-expansion is used in improving binding-time analysis that makes a partial evaluator yields better results [5]. In this paper, we use eta-expansion to float appropriate number of inner lambdas to the top level a function. These are two orthogonal applications of eta-expansion.

7 Conclusion

We have presented two mechanisms to find a function's ideal arity. Each mechanism is modelled by a type system which handles a language equivalent to the Core intermediate language of the Glasgow Haskell Compiler. We have proven each type system sound with respect to the operational semantics (involving calculation of costs) of the language. In addition, we have implemented a inference algorithm for each type system and measured the effects of the lambda floating transformations. The improvements we obtain are modest but significant.

References

[1] AV Aho, R Sethi, and JD Ullman. *Compilers - principles, techniques and tools*. Addison Wesley, 1986.

[2] AW Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[3] DF Bacon, SL Graham, and OJ Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[4] Tyng-Ruey Chuang and Benjamin Goldberg. Backwards analysis for higher-order functions using inverse images. Technical report tr1992-620 dept comp sci, new york university, November 1992.

[5] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 18(6):730–751, 1996.

[6] P Fradet and D Le Metayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13(1):21–51, January 1991.

[7] Andrew John Gill. Cheap deforestation for non-strict functional languages. *PhD Thesis*, (TR-1996-5), January 1996.

[8] John Hannan and Patrick Hicks. Higher-order arity raising. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*. Philadelphia, May 1998.

[9] John Hughes. Backwards analysis of functional programs. Dept comp sci, univ of glasgow, 1988.

[10] RJM Hughes. *The design and implementation of programming languages*. Ph.D. thesis, July 1983.

[11] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In Jouannaud, editor, *Proc IFIP Conference on Functional Programming and Computer Architecture*, pages 190–205. LNCS 201, Springer Verlag, 1985.

[12] R Kelsey. Compilation by program transformation. Yaleu/—dcs/—rr-702, phd thesis, department of computer science, yale university, May 1989.

[13] R Kelsey and P Hudak. Realistic compilation by program transformation. In *Proc ACM Conference on Principles of Programming Languages*, pages 281–292. ACM, January 1989.

[14] David A. Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. *Proceedings of the SIGPLAN'86 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 21(7):219–233, July 1986.

[15] A. K. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Symposium on Principles of Programming Languages*, pages 43–56.

[16] WD Partain. The `nofib` benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202.

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

[18] SL Peyton Jones, RJM Hughes, L Augustsson, D Barton, B Boutel, W Burton, J Fasel, K Hammond, R Hinze, P Hudak, T Johnsson, MP Jones, J Launchbury, E Meijer, J Peterson, A Reid, C Runciman, and PL Wadler. Report on the programming language Haskell 98. `http:/haskell.org`, February 1999.

[19] SL Peyton Jones and D Lester. A modular fully-lazy lambda lifter in HASKELL. *Software Practice and Experience*, 21(5):479–506, May 1991.

[20] SL Peyton Jones, WD Partain, and A Santos. Let-floating: moving bindings to give faster programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*. Philadelphia, May 1996.

[21] S. A. Romanenko. Arity raiser and its use in program specialization. *3rd European Symposium on Programming*, 432:341–360, 1990.

[22] The GHC Team. *The Glasgow Haskell Compiler User's Guide*. www.haskell.org/ghc/documentation.html, 1998.

A Other Applications of Arity Analysis

In this section, we give two more applications of arity analysis. We believe there are some more applications yet to be discovered.

A.1 Fully-Lazy Lambda Lifting

Lambda Lifting [11], an orthogonal transformation to lambda-floating, is a program transformation to remove free variables. An expression containing a free variable is replaced by a function applied to that variable. For example,

```
f x = g 5  where g y = y + x
```

here, `x` is a free variable of `g` so it is added as an extra argument:

```
f x = g 5 x  where g y x = y + x
```

A fully lazy lambda lifter [10, 19] makes each *maximal free sub-expression* of a lambda abstraction into a function applied to those expressions. For example,

```
f = \x -> (\y -> (+) (sqrt x) y)
```

here, `((+) (sqrt x))` is a maximal free expression in `(\ y . (+) (sqrt x) y)` so this inner abstraction is replaced with

```
(\g -> \y -> g y) ((+) (sqrt x))
```

Now, if a partial application of f is shared, the result of evaluating `(sqrt x)` will also be shared rather than re-evaluated on each application of f. This is similar to the code motion optimisation in procedural languages where constant expressions are moved outside a loop or procedure. However, a clever fully lazy lambda lifter who knows the arity of `(+)` is 2 will do the following transformation instead:

```
(\z -> \ y -> (+) z y) (sqrt x)
```

If we replace `(+)` by a arbitrary function `h` of arity 1, the clever lifter should obtain

```
(\g -> \y -> g y) (h (sqrt x))
```

This example shows the arity information of a function plays a pivot role in selecting ideal free sub-expressions for fully lazy lambda lifting.

A.2 Cheap Deforestation

In [7], Gill introduced a technique for eliminating intermediate lists from programs. The gist of the method is a rewrit rule "`foldr/build`":

```
foldr k z [] = z
foldr k z (x:xs) = k x (foldr k z xs)

build g = g (:) []

-- rule ``foldr/build''
foldr k z (build g) = g k z
```

The function `map` can be defined in terms of `foldr` and `build`:

```
map f xs = build (\c n -> foldr (c . f) n xs)
```

Suppose we find an application `(map f (build g))`, we want to transform the call like this:

```
map f (build g)
= {- inline map (DANGER!) -}
  build (\c -> \n -> foldr (c . f) n (build g))
= {- apply foldr/build rule -}
  build (\c -> \n -> g (c . f) n)
```

The difficulty is in the step marked DANGER!. Here we substitute `(build g)` for `xs` in the body of `map`, but this occurrence of `xs` is under a lambda abstraction. In general, one can make a program run arbitrarily more slowly by substituting a redex inside a lambda abstraction. However, if we know the parameter of `build` has arity 2, we can safely inline `(build g)`. This example shows that the arity information of a parameter of a function plays crucial role in inlining which is a pivot in the cheap deforestation technique.

B Forward Arity Inference

Theorem 1 (Termination of Forward Inference) *If a recursive function's definition is well-typed, then fixpoint iteration for finding its arity terminates.*

There are two cases to consider. Case1: The recursive function does not have terminating case. Its arity is ∞. Only one iteration occurs and it terminates. Case2: The recursive function have terminating case(s). According to the typing rules (i.e. minimum arity value is chosen when having more than one branches), after one iteration, the arity will be lowered from ∞ to a constant (say `n`). Since the bottom of the lattice is 0 (i.e. there is no infinite-decreasing chain), it will take at most `n` steps to terminate.