

Simple Driving Techniques

Mads Rosendahl

University of Roskilde, Dept. of Computer Science
Building 42.1, P.O. Box 260, DK-4000 Roskilde, Denmark

* madsr@ruc.dk

Abstract. Driving was introduced as a program transformation technique by Valentin Turchin in some papers around 1980. It was intended for the programming language REFAL and used in metasystem transitions based on super compilation. In this paper we present one version of driving for a more conventional lisp-like language. Our aim is to extract a simple notion of driving and show that even in this tamed form it has much of the power of more general notions of driving. Our driving technique may be used to simplify functional programs which use function composition and will often be able to remove intermediate data structures used in computations.

A supercompiler (= supervised compiler) is a special program transformation system which, while evaluating the program, supervises its operation and uses the information to reconstruct the program in a more efficient way. A supercompiler may work on expressions with free variables and will then drive it through the interpretation process and produce a tree of states and transitions. The tree is made into a finite graph by generalizing patterns or configurations of expressions. A configuration is an expression without free variables but with "holes" in which one may insert other expressions. A supercompiler will have a strategy for generalizing configurations (*i.e.* make the holes bigger or make new holes). In order to guarantee termination the driving algorithm should generalize configurations into a finite set of simpler configurations called *basic configurations*.

The concept of super compilation is due to Valentin Turchin [10], [11]. The work and concepts are closely tied to the language Refal and not easily rephrased for other programming languages.

The strategy for generalizing configurations may be viewed as an abstraction function on expressions, or as an upper bound operation when several configurations should be described by a more general configuration. Different strategies will define different sets of possible configurations. To ensure that the driving terminates we need a strategy that does not produce infinitely many configurations for a program. As in abstract interpretation this may be achieved if the set of configurations is finite or has some finite-height property.

Our aim is to use driving to optimize functional programs where composition of user-defined functions plays a central role. We want to specialize functions

* This research was partly sponsored by the IT-University in Copenhagen

when we know that the argument values are produced by certain other functions. We will use the driving technique and define a simple set of basic configurations. The result is a transformation system which is easy to implement and easy to use.

1 Background

The bulk of the work reported here was done in 1986 as part of the authors MSc. Thesis with Neil Jones as supervisor [4]. The abstract interpretation part of the thesis was later published in FPCA'89 [5] but much to the supervisor's regret the transformation part did not appear in journal form at that time.

The aim of the thesis was to construct time bound functions for programs - *i.e.* functions that return an upper bound of the execution time of a program given the size of input. The time bound function is constructed automatically in two stages. Using an abstract interpretation the program is transformed to a new program that computes the time bound function. The abstract interpretation is viewed as a different (abstract) semantics, thus defining a new language and we constructed a compiler from the language of the abstract semantics to the language of the standard semantics. This is reexamined in the final example of this paper.

In the second stage this generated program is simplified as much as possible, hopefully into a simple closed-form expression. The main challenge in the second stage is to remove various intermediate data structures used in the generated program. The driving technique described here is able to remove many such structures and more generally to optimize programs containing function compositions.

2 Language

The driving principle is here presented for a small lisp-like, first-order, eager, functional language. A program in the language consists of n -functions in a recursion equation system

$$\begin{aligned} f_1(x_1, \dots, x_{m_1}) &= exp_1 \\ \dots & \\ f_n(x_1, \dots, x_{m_n}) &= exp_n \end{aligned}$$

Expressions are constructed from constants, parameter names, basic operations, conditional expressions and calls to functions in the program.

$exp ::= c_i$	numbers, strings, true , false , nil
x_i	parameters
$op_i(exp_1, \dots, exp_{i_n})$	basic operations
if exp_1 then exp_2 else exp_3	conditional
$f_i(exp_1, \dots, exp_{m_i})$	function calls

Operations include the functions `car`, `cdr`, `cons`, `atom`, `null`, `eq` on dotted pairs and various arithmetic operations on numbers.

Running a program consists of calling the first function in the program with the input to the program. As examples of programs in the language consider

```
ff(x) = flip(flip(x))
flip(x) = if atom(x) then x else cons(flip(cdr(x)),flip(car(x)))
```

and

```
sumn(n) = sum(fromn(n))
sum(x) = if null(x) then 0 else add(car(x),sum(cdr(x)))
fromn(n) = if eq(n,0) then nil else cons(n,fromn(sub(n,1)))
```

The first program flips a binary tree twice, thus returning a copy of the input as output. The second program computes the sum of the numbers from 1 to n , where n is the input to the program.

3 Example

The example here is taken from Wegbreit [15].

Consider a program that appends three lists by the function

```
f(x,y,z)      = append(append(x,y),z)
append(x,y)   = if null(x) then y
               else cons(car(x),append(cdr(x),y))
```

We try to improve this small program by analyzing the right hand side of the first function:

```
append(append(x,y),z)
```

Here we immediately get the first function composition, and this is viewed as a *basic configuration* (or goal or procedure-expression). By functions we here mean the user defined functions, everything else is considered as basic expressions, since they cannot be unfolded. The configuration is represented using the notation `[append(append(.,.),.)]` where `."` denotes a hole which may be substituted for an arbitrary expression. The configuration is now viewed as a function and analyzed without its original context. This is in contrast to Turchin's work [11] where the configurations are recognized as being basic at their second appearance, hence the first reduction is done in the original context. The present method is chosen because of its simpler termination properties.

The configuration contains three free variables, and as with Turchin [11] we drive the free variables forcefully through the expressions of the program until the expression represents a *progressive* method for computing the configuration it defines (cp. [6]):

```
[append(append(.,.),.)] (a,b,c) =
  if null(a) then append(b,c)
  else cons(car(a),append(append(cdr(a),b),c))
```

The driving is done by constructing a conditional expression with the first condition to be performed in a normal order evaluation of the expression. This method resembles the idea of outside-in reductions in driving REFAL [11].

This expression contains two configurations: The trivial one:

```
[append(.,.)]
```

and the original configuration:

```
[append(append(.,.),.)]
```

Hence we loop back and get the new function definition:

```
append3(a,b,c) = if null(a) then append(b,c)
                  else cons(car(a),append3(cdr(a),b,c))
```

Inserting in the original context the program is improved to

```
f(x,y,z)        = append3(x,y,z)
append(x,y)     = if null(x) then y
                  else cons(car(x),append(cdr(x),y))
append3(a,b,c) = if null(a) then append(b,c)
                  else cons(car(a),append3(cdr(a),b,c))
```

4 The driving algorithm

When unfolding function calls in program transformation systems certain criteria are needed to ensure termination. The most restrictive rule (after no unfolding) would be only to allow one unfolding per function in each branch. Here we use a somewhat extended principle by only allowing one unfolding per *basic configuration* in each branch. By only defining a finite number of basic configurations termination is easily secured. In this section we define the basic configurations and show how progressive computation schemes can be established. Afterwards the driving algorithm is described.

Unlike Turchin we define a fixed set of basic configurations independently of the program we transform. Our basic configurations are expressions with function compositions - *i.e.* where arguments to function calls are function calls. We will show that with this limited set of configurations we are able to make quite powerful transformations.

The set of basic configurations is defined as equivalence classes in the set of expressions, and two expressions are equivalent if they are function calls with call to the same functions as arguments. The configuration can be characterized by the first function name and a list of arguments with either a function name or an indication that the argument can be anything. We write basic configurations in square brackets $[f_i(u_1, \dots, u_{m_i})]$ where u_k is either a hole: " _ " or a function call where all arguments are unknown: $f_k(-, \dots, -)$. As an example $[append(append(.,.),.)]$ is a basic configuration but $[append(append(cdr(.),.),.)]$ is not and will be generalized to the former.

A basic configuration where all arguments are holes (-) is called a trivial basic configuration. There is a natural ordering of basic configurations with the trivial configurations being more general than configurations with function names as arguments. Our aim is to locate as many expressions as possible based on non-trivial basic configurations in the program and construct better computational schemes for these configurations.

Given a (non-trivial) basic configuration we define a new function to compute expressions of this form. We use the most general expression described by a basic configuration as the right hand side and the free variables in the expression as parameter names.

1. Search the expression in normal order (outside-in) for the first basic condition to be evaluated. A condition (a test in an if-expression) is basic if it does not contain any calls to user defined functions.
2. In the context of this condition being respectively true and false the arguments to the outer function call are reduced (and unfolded) as much as possible without producing any conditional expressions.
3. If the two new argument lists are reduced to progressive forms, *i.e.* contain calls where arguments are smaller, then insert them in the function call resulting in two new expressions. They are reduced as much as possible without getting any conditional expressions, and *without removing any function compositions*. A conditional expression is now constructed by the condition found above and these two expressions. The last requirement is stronger than normally in driving, but it is due to our restricted set of basic configurations. The first requirement is equivalent to reduction of *transient* configurations in super compilation.
4. If the two argument sets could not use the information from the condition, then unfold the outer function call and insert the arguments. The expression is then reduced as much as possible as in (3).

Reductions.

“Reduce as much as possible” means the application of a number of standard reduction schemes (the names are taken from [15]):

Local simplification rules. This is also called symbolic evaluation (in [7]). For examples `null(cons(a,b))` is changed to `false` and `car(cons(a,b))` is changed to `a`. Such transformations are safe provided the sub-expressions do not contain non-terminating computations.

Distributing conditionals. This is also called reduction to normal form (in [14]). The conditionals are taken out of arguments to basic operations and functions, so they appear outermost in expressions. This makes the next step easier. For example

```
append(if null(a) then nil else cons(car(a),append(cdr(a),b)),c)
```

is change into

```
if null(a) then append(nil,c) else
append(cons(car(a),append(cdr(a),b)),c)
```

Evaluating in context. In the simplest form this consists in removing duplicate conditions in an expression. When a function call is expanded we know the result of all predicates in conditionals leading to the expression.

Expanding function bodies. This is also called application (in [6]) or unfolding. The method must be used with care, and we adopt a criteria similar to Wegbreit's [15]: Functions are only expanded if they give condition free-expressions and if they do not remove function compositions in the program.

5 Collecting basic configurations

The collection of basic configurations and simplification of their defining expressions is essentially a fixpoint problem. We only want to analyze the basic configurations that can be reached (called) from the first function in the program, and simplifying a right hand side for a basic configuration may result in other expressions being simplified. Our approach uses a depth-first solution to the problem. We start by analyzing the first basic configuration in the program. The analysis is suspended whenever we locate a new basic configuration to be analyzed. A single depth-first analysis is sufficient in the cases we have considered and we have not encountered examples where further recursion would improve the program.

The driving algorithm starts at the right hand side of the first function in the program. All basic configurations are examined from outside in, and progressive definition of the configurations (found in the last section) are analyzed in the same way recursively. To secure termination of the driving algorithm a basic configuration is only analyzed once in each branch, and to improve the residual program, three operations are performed: generalization of basic configurations, recognition of earlier defined function and solving difference equations. The algorithm is described in two parts: a local one dealing with the analysis of a single basic configuration, and the global one controlling generalization.

Local part. The driving algorithm treats basic configuration in the following five steps.

Given a basic configuration:

1. Construct a progressive computation scheme for it (see previous section).
2. Locate and drive basic configurations in sub-expressions by the global part of the driving algorithm.
3. Locate calls to the original basic configuration. If there are such recursive calls then the expression is a recursive definition of the basic configuration viewed as a function in its free variables. This definition is then examined by:
4. (a) Matching it with function definitions in the original program. If it is defined in advance then substitute the expression with a call to this function.
(b) Solve difference equations by matching the definition with known functions with simpler computational form.

- (c) Otherwise construct a new recursive function and substitute the expression with a call to this function. (Folding).
- 5. If there are no recursive calls in the expression then do nothing.

Generalization.

Assume that, while driving a configuration $[f(g(-, \dots, -), h(-, \dots, -))]$ we reach the configuration $[f(g(-, \dots, -), k(-, \dots, -))]$ and we do not reach other calls to the configuration $[f(g(-, \dots, -), h(-, \dots, -))]$. In this situation the original configuration should be generalized to $[f(g(-, \dots, -), -)]$. When analyzing the defining expression for this configuration all appearances of more specific configurations (*i.e.* where the second argument is a function call) should also be generalized to this configuration. This idea is well known in the literature [6] [11] [15].

Global part. The global part of the driving algorithm controls the program transformation system.

1. Start with the right hand side of the first function in the program.
2. Search the expression from outside in for a basic configuration.
3. If the configuration is a generalization of an identified, but suspended configuration, then backtrack and generalize the first identified (outermost) configuration. If the configuration is either trivial or identical with a suspended configuration, then loop back by returning the expression.
4. Otherwise suspend the analysis of the current configuration while we analyze this new configuration with the local part of the driving algorithm.
5. If the resulting expression is a conditional expression then the basic configuration was not successful; continue driving with the arguments of the basic configuration (the original call).
6. Otherwise reduce the expression in its context and continue driving with this expression.

6 Finite difference equations

Certain recursive functions on integers may be simplified into well-known closed-form expressions. A quite general rule is the following:

$$\begin{aligned}
 F(x) &= \text{if } x = \ell \text{ then } c_1 \text{ else } f(x) + c_2 \cdot F(x - 1) \\
 &= c_1 \cdot c_2^{x-\ell} + \sum_{i=\ell+1}^x f(i) \cdot c_2^{x-i}
 \end{aligned}$$

When viewed as a program transformation it may not, however, produce a simpler program since the summation and power operation will be done using a recursive function. The original program uses $(x - \ell)$ recursive calls whereas the transformed program will use in the order of $(x - \ell)^2$ calls. There are some

special cases, however, where the transformation will produce simpler programs. We will consider four such cases.

$$\begin{aligned} F(x) &= \text{if } x = \ell \text{ then } c_1 \text{ else } c_2 + F(x - 1) \\ &= x \cdot c_2 + (c_1 - c_2 \cdot \ell) \end{aligned}$$

$$\begin{aligned} F(x) &= \text{if } x = \ell \text{ then } c_1 \text{ else } c_2 + c_3 \cdot x + F(x - 1) \\ &= \frac{c_3}{2} \cdot x^2 + \left(c_2 + \frac{c_3}{2}\right) \cdot x + \left(c_1 - \frac{c_3}{2} \cdot \ell(\ell - 1)\right) \end{aligned}$$

$$\begin{aligned} F(x) &= \text{if } x = \ell \text{ then } c_1 \text{ else } c_2 + c_3 \cdot F(x - 1) && \text{where } c_3 \neq 1 \\ &= G(x) \cdot \left(c_1 - \frac{c_2}{1 - c_3}\right) + \frac{c_2}{1 - c_3} \end{aligned}$$

where $G(x) = \text{if } x = \ell \text{ then } 1 \text{ else } c_3 \cdot G(x - 1)$

$$\begin{aligned} F(x) &= \text{if } x = \ell \text{ then } c_1 \text{ else } c_2 + c_3 \cdot x + c_4 \cdot F(x - 1) && \text{where } c_4 \neq 1 \\ &= G(x) \cdot \left(\frac{c_2 + \ell \cdot c_3}{c_4 - 1} + c_1 + \frac{c_3 \cdot c_4}{(c_4 - 1)^2}\right) - x \cdot \frac{c_3}{c_4 - 1} - \left(\frac{c_2}{c_4 - 1} + \frac{c_3 \cdot c_4}{(c_4 - 1)^2}\right) \end{aligned}$$

where $G(x) = \text{if } x = \ell \text{ then } 1 \text{ else } c_4 \cdot G(x - 1)$

General decrement.

There are some similar cases for numeric function on data structures. One simple example is the following which may be useful as a simplification rule when driving numeric programs.

$$\begin{aligned} F(x) &= \text{if null}(x) \text{ then } c_1 \text{ else add}(c_2, F(\text{cdr}(x))) \\ &= \text{add}(c_1, \text{mul}(c_2, \text{length}(x))) \end{aligned}$$

7 Example: flip

Consider the program

```
ff(x) = flip(flip(x))
flip(x) = if atomp(x) then x else cons(flip(cdr(x)), flip(car(x)))
```

When we use the driving technique on this program the initial basic configuration is `[flip(flip(.))]`. The first basic condition to be evaluated in the expression `flip(flip(x))` is `atomp(x)`. If this condition is true then the expression `flip(flip(x))` is easily simplified to `x`. If the condition is false we attempt to expand the inner function call:

```
flip(if atomp(x) then x else cons(flip(cdr(x)), flip(car(x))))
```

which in this context may be simplified to

```
flip(cons(flip(cdr(x)), flip(car(x))))
```

Since the expression is condition-free we may proceed and expand the outer function call:

```

if atom(cons(flip(cdr(x)),flip(car(x))))
then cons(flip(cdr(x)),flip(car(x)))
else cons(flip(cdr(cons(flip(cdr(x)),flip(car(x))))),
flip(car(cons(flip(cdr(x)),flip(car(x))))))

```

This may be reduced to

```

cons(flip(flip(car(x)),flip(flip(cdr(x))))

```

This is recognised as a successful reduction since no function compositions are removed and since it constitutes a progressive evaluation scheme. The resulting program is now

```

ff(x) = if atomp(x) then x else cons(ff(car(x)),ff(cdr(x)))

```

Although this is the identity function we cannot prove this using the driving technique. We have, however, removed the binary tree constructed as an intermediate data structure in the program.

8 Example: sumn

Let us also consider the program

```

sumn(n) = sum(fromn(n))
sum(x) = if null(x) then 0 else add(car(x),sum(cdr(x)))
fromn(n) = if eq(n,0) then nil else cons(n,fromn(sub(n,1)))

```

The initial basic configuration is [sum(fromn(.))]. The first basic condition to be evaluated in the expression sum(fromn(n)) is eq(n,0). If this condition is true then the expression is easily reduced to 0. Otherwise we get the expression

```

sum(cons(n,fromn(sub(n,1))))

```

and unfolded to

```

add(car(cons(n,fromn(sub(n,1))),sum(cdr(cons(n,fromn(sub(n,1))))))

```

The reduced definition for the basic configuration is now

```

[sum(fromn(.))] (n) = if eq(n,0) then 0 else
add(n,sum(fromn(sub(n,1)))

```

This is one of the finite difference equations listed above, and we obtain the result

```

sumn(x) = add(mul(div(1,2),mul(x,x)),mul(div(1,2),x))

```

This could also be written as

$$\text{sumn}(x) = \frac{1}{2}x^2 + \frac{1}{2}x$$

9 Time bound programs

A time bound program is a program which from information about the size of input computes an upper bound to the execution time of the program. As execution time we will here use the number of sub-expressions being evaluated. We will here outline how time bound programs are constructed. A fuller account in a semantic framework may be found in [5]. The motivation for our work on driving was to transform time bound programs into closed form (non-recursive) expressions.

Consider the reverse function in the language

```
reverse (x) =
  if null(x) then nil
  else append(reverse(cdr(x)),cons(car(x),nil))
append (x,y) =
  if null(x) then y
  else cons(car(x),append(cdr(x),y))
```

Step counting version The first part is to extend the program with profiling information so that the output of the program is the number of sub-expressions being evaluated. We call this the step counting version of the program

```
t-reverse (x) =
  if null(x) then 4
  else add(11,add(t-reverse(cdr(x))
                 t-append(reverse(cdr(x))) ))
t-append (x) =
  if null(x) then 4 else add(10,t-append(cdr(x)))
reverse (x) =
  if null(x) then nil
  else append(reverse(cdr(x)),cons(car(x),nil))
append (x,y) =
  if null(x) then y
  else cons(car(x),append(cdr(x),y))
```

This program is constructed fairly easily from the original program using a transformation function \mathbf{T} on expressions

$$\begin{aligned} \mathbf{T}[c_i] &= 1 \\ \mathbf{T}[x_i] &= 1 \\ \mathbf{T}[op_i(e_1, \dots, e_n)] &= \text{add}(1, \mathbf{T}[e_1], \dots, \mathbf{T}[e_n]) \\ \mathbf{T}[\text{if } e_1 \text{ then } e_2 \text{ else } e_3] &= \text{if } e_1 \text{ then } \text{add}(1, \mathbf{T}[e_1], \mathbf{T}[e_2]) \text{ else } \text{add}(1, \mathbf{T}[e_1], \mathbf{T}[e_3]) \\ \mathbf{T}[f_i(e_1, \dots, e_n)] &= \text{add}(1, \text{t-}f_i(e_1, \dots, e_n), \mathbf{T}[e_1], \dots, \mathbf{T}[e_n]) \end{aligned}$$

Abstract semantics. The arguments to a step counting version of a program are ordinary values in our lisp-like language: numbers, strings and dotted pairs of values and output is numbers. The next part is to extend the value domain with a special atom `all` that denotes any possible atom. The `all` may also appear in dotted pairs. We then construct an abstract interpretation which gives an upper

bound of what the standard semantics would have given when `all` values are substituted with other atoms. The abstract interpretation is an extension of the standard semantics. The main difference is that conditions in if-expressions may have the value `all`. If the condition is `all` then the result should be an upper bound of the results in the two branches.

The usual approach in abstract interpretation is to implement it directly - typically using some fixpoint iteration strategy. We, however, view it as a new semantics. It defines a new language with the same syntax but different semantics. The construction now proceeds by translating the program in this special semantics into a new program in our usual semantics. The abstract semantics and the translation scheme is described in [5]. The translation changes basic operations and conditional expressions but leaves constants, parameters and function calls unchanged. For example the equality test `eq(a,b)` is translated into

```
if or(eq(a,"all"),eq(b,"all")) then "all" else eq(a,b)
```

The conditional expression `if a then b else c` in a step counting version of a function is translated into

```
if eq(a,all) then max(b,c) else if a then b else c
```

For other functions the translation is

```
if eq(a,all) then lub(b,c) else if a then b else c
```

where the `lub` function is defined as

```
lub(a,b) = if eq(a,b) then a else
           if and(consp(a),consp(b))
             then cons(lub(car(a),car(b)),lub(cdr(a),cdr(b)))
             else "all"
```

We refer to [5] for examples of programs after this translation.

Inverted size measures. When we want to construct a time bound program we need a size measure. For some programs it is the length of input lists but for other programs it the maximal depth of binary trees or something else. In the example here the length function is the natural choice. Our approach may, however, be used for other size measures.

We may use values with `"all"` atoms to represent sets of values. For example `("all".("all".("all".nil)))` represents all lists of length 3. This leads us to define the function

```
length-1 (x) =
  if eq(x,0) then nil
  else cons("all",length-1(sub(x,1)))
```

which has the property that `length(length-1(n)) = n` for any non-negative number. We call the function `length-1` an inverted size measure. The other composition `length(length-1(x))` is not, however, an identity map. It may therefore be more useful to think of the size measure as an abstraction function and the inverted size measure as a concretization function in a Galois connection.

Time bound program. We will now compose the translated step counting version of the program with the inverted size measure. After simplifications based on constant propagation the result is the following program.

```

time (x) = t-reverse(length-1(x))
length-1 (x) =
  if eq(x,0) then nil
  else cons("all",length-1(sub(x,1)))
t-reverse (x) =
  if null(x) then 4
  else add(11,add(t-reverse(cdr(x))
                  t-append(reverse(cdr(x))) ))
t-append (x) =
  if null(x) then 4 else add(10,t-append(cdr(x)))
reverse (x) =
  if null(x) then nil
  else append(reverse(cdr(x)),cons(car(x),nil))
append (x,y) =
  if null(x) then y
  else cons(car(x),append(cdr(x),y))

```

This program computes an upper bound to the execution time of the reverse program based on the length of its input. We can always construct such time bound programs but the time bound program may fail to terminate for some input sizes even if the original program terminates for all input of the given size. We cannot in general expect to be able to transform time bound programs into simple closed form expressions. The driving technique may, however, for a number of programs simplify the time bound programs to closed form.

10 Driving time bound programs

In the following example we will use the driving technique on the `time` function from the previous section.

To start with we try to solve difference equations in the program. It is convenient to do this before using the driving algorithm because we might otherwise solve the same difference equation many times in the algorithm. There is only one function in the program to be simplified in this way:

```
t-append (x) = add(4,mul(10,length(x)))
```

where `length` is the usual length function from Lisp.

```
length(x) = if null(x) then 0 else add(1,length(cdr(x)))
```

The initial basic configuration is `[t-reverse(length-1(.))]` with the expression

```

[t-reverse(length-1(.))] (x) =
  if eq(x,0) then 4
  else add(15,
    add(t-reverse(length-1(sub(x,1)))
      mul(10,length(reverse(length-1(sub(x,1)))) ))

```

Here we may notice that `[length(reverse(length-1(.)))]` is not a basic configuration in our approach. We analyze the expression for outside in so the next basic configuration to be analyzed is `[length(reverse(.))]`.

```
[length(reverse(.))] (x) =
  if null(x) then 0
  else length(append(reverse(cdr(x)),cons(car(x),nil) ))
```

giving the new configuration

```
[length(append(.))] (x,y) =
  if null(x) then length(y)
  else add(1,length(append(cdr(x),y)))
```

This difference equation can be solved giving

```
[length(append(.))] (x,y) =
  add(length(x),length(y))
```

Inserted in the context from the previous configuration we get

```
[length(reverse(.))] (x) =
  if null(x) then 0
  else add(1,length(reverse(cdr(x))))
```

This function is matched against previously defined functions in the program thus giving

```
[length(reverse(.))] (x) = length(x)
```

Inserted in the original context we get

```
[t-reverse(length-1(.))] (x) =
  if eq(x,0) then 4
  else add(15,
    add(t-reverse(length-1(sub(x,1)))
      mul(10,(length(length-1(sub(x,1)))))) )
```

where we get the new configuration

```
[length(length-1(.))] (x) =
  if eq(x,0) then 0
  else add(1,length(length-1(sub(x,1))))
```

This difference equation is easily solved to

```
[length(length-1(.))] (x) = x
```

The original configuration is now

```
[t-reverse(length-1(.))] (x) =
  if eq(x,0) then 4
  else add(15
    add(t-reverse(length-1(sub(x,1)))
      mul(10,sub(x,1)) )
```

This difference equation can be solved to

```
[t-reverse(length-1(_))] (x) =  
  add(4,add(mul(10,x),mul(5,mul(x,x)) ))
```

Finally the program can be reduced to

```
time (x) = add(4,add(mul(10,x),mul(5,mul(x,x)) ))
```

This could also be written as

$$\text{time}(x) = 5x^2 + 10x + 4$$

11 Related works

The idea of specializing functions with functions as arguments can be found as procedure-expressions in Scherlis [6] and in super-compilation and driving in Turchin [11]. Also Wegbreit [15] describes a way to improve function composition by matching sub-goals in unfolded expressions from more general goals. The works by Scherlis and Wegbreit describe transformation rules, but they cannot answer the question about in which order the rules should be applied. Super compilation gives a general answer to this question although the set of basic configurations is difficult to characterize.

Several authors have considered similar transformation systems which may remove intermediate data structures [12], [13]. Wadler's work on listlessness and deforestation may perform many of the same simplifications of programs as reported here. Our work is not, however, restricted to trees or lists and does not require branching to be done using pattern matching.

More recently Sørensen, Glück and Jones [9],[3],[8] have re-examined Turchin's work and defined the notion of positive super compilation. Their aim is more general than ours and they will capture most of the power of Turchin's supercompiler.

Driving and super compilation may be seen as a generalization of partial evaluation. If we used configurations of the form $[f_i(u_1, \dots, u_{m_i})]$ where u_k are either constants or the special symbol *var* then we would produce specialized version of programs when some arguments were known. In partial evaluation, however, the problem of when to generalize is not trivial and it would have to be implemented in the generalization strategy.

12 Conclusion

The present program transformation system is fairly easy to implement. The original version was implemented in Lisp, but we have since rewritten it in ML. The system is especially useful as a tool to simplify automatically generated programs. Such programs are not always just intended to be run by computer. If the intended recipient is a human then we may appreciate receiving it in a simplified version where unnecessary computations are removed.

The transformation system is guaranteed to terminate since there is only a finite number of basic configurations for any given program. There is a risk of code explosion since the number of basic configurations in theory is quite large but we have not been able to construct examples where that would occur. Non-trivial basic configurations are sparsely distributed in real programs.

Some of the reductions we use may remove non-terminating sub-expressions. Expressions of the form `car(cons(a,b))` may be reduced to `a` even though the expression `b` may contain non-terminating sub-expressions. Due to this the resulting program may terminate with input for which the original program would not terminate. It is not a very likely situation and will not be an issue if the original programs are total.

The driving technique does not impose any special requirements on the programs. If there are no non-trivial basic configurations in the program or if we cannot produce optimized versions for the configurations then no transformation will take place and we will just return the original program.

Acknowledgement. To Neil D. Jones for inspiration and encouragement through the years.

References

1. R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *J.ACM.* vol. 24, no 1, pp. 44 - 67, Jan., 1977.
2. J. Cohen and J. Katcoff. Symbolic Solution of Finite-Difference Equations. *Transactions on Mathematical Software.* vol. 3, no. 3, pp. 261 - 271, 1977.
3. N. D. Jones. The Essence of Program Transformation by Partial Evaluation and Driving. *Logic, Language, and Computation.* LNCS vol 792, pp. 206-224, Springer-Verlag, 1994
4. M. Rosendahl. Automatic Program Analysis. Master's thesis, Department of Computer Science, University of Copenhagen, 1986
5. M. Rosendahl. Automatic Complexity Analysis. *FPCA '89, London, England,* ACM Press, pp. 144-156, 1989.
6. W. L. Scherlis. Program Improvement by Internal Specialization. *8'th Symposium on Principles of Programming Language.* ACM, pp. 41-49, Jan., 1981.
7. Peter Sestoft. The structure of a Self-applicable Partial Evaluator *Programs as Data Objects* LNCS, vol 217, pp. 236 - 256, 1986
8. M. H. Sørensen. Turchin's supercompiler revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1990
9. M. H. Sørensen, R. Glück. Introduction to Supercompilation *Partial Evaluation: Practice and Theory,* LNCS, vol. 1706, pp. 246-270, 1999
10. V. F. Turchin, R. M. Nirenburg, D. V. Turchin. Experiments with a Supercompiler. *ACM Symposium on Lisp and Functional Programming.* Pittsburgh PA, August, 1982.
11. V. F. Turchin. The Concept of a Supercompiler. *ACM TOPLAS.* vol. 8, no. 3, pp. 292-325, July, 1986.
12. P. Wadler. Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection on Compile-time. *ACM Symposium on Lisp and Functional Programming.* Austin Texas, August, 1984.

13. P. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, vol. 73, pp. 231-248, 1990
14. B. Wegbreit. Mechanical Program Analysis. *C.ACM.* vol. 18, pp. 528 - 539, Sept., 1975.
15. B. Wegbreit. Goal-Directed Program Transformation. *IEEE Transaction on Software Engineering*, vol SE-2, no 2, pp. 69 - 80, June, 1976
16. B. Wegbreit. Verifying Program Performance. *J.ACM.* Vol. 23, No. 4, pp. 691 - 699, October, 1976