OPTIMIZING EQUATIONAL PROGRAMS

Robert Strandh

Department of Computer Science, The Johns Hopkins University
Baltimore, Maryland 21218

ABSTRACT

Equational programming [HO82b] involves replacing subterms in a term according to a set of *equations* or *rewrite rules.* Each time an equation is applied to the term, the subterm that matches the left hand side of the equation is replaced by the corresponding right hand side. In that process several *nodes* of the term tree are created. Some of these nodes may later turn out to be useless, and will be reclaimed.

This paper discusses important relationships between two equational programs. In particular we define the term *mutual confluence* and show that two equational programs with the mutual confluence property have the same *output behavior* with very general assumptions about the reduction strategy. As an application of our result, we discuss *source-to-source* transformations of an equational program E to an equational program F. Our transformations are used as a part of a compiler to improve execution time of E by avoiding the creation of too many nodes in the reduction process. We show that our transformations indeed give E and F the mutual confluence property, thus preserving the output behavior of E when transformed to F.

Preserving the output behavior is more general than preserving just normal forms, in that we allow for infinite computations where we output stable parts of the term, *i.e.*, parts that can never change as a result of further reductions.

1. Introduction

The equational programming language [HO82b] was created as an experiment in lazy evaluation. The creators emphasized simple semantics over execution speed. Lazy evaluation was chosen because it allows for a bigger class of programs to halt whenever a reasonable output of the program can be deduced. A minimum of builtin functions is provided, and all of them are entirely equivalent to the inclusion of a large, possibly infinite, set of equations. This is true in particular for the arithmetic functions, such as *add*, and the arithmetic comparison functions, such as *less*. No builtin conditional function is provided. The function *if* can easily be specified by the user by the two equations $if(true, x, y) = x$ and $if(false, x, y) = y$. Such a definition would not work in a language that uses innermost evaluation (e.g. Lisp),

whenever one of the arguments to *if* involves an infinite or erroneous computation.

The input to an equational program is a *term*, such as *reverse* [(*a b c*)] or *add* [*factorial*[4];*factorial*[3]].

The input term is *reduced* in zero or more steps to an *output term*. The output term (if any) is *equal* to the input, as a logical consequence of the equations. The output is in *normal form*, *i.e.*, it cannot be further reduced. A program may fail to produce a term in normal form, as a result of an *infinite computation*. Notice that in that case, we may still get some output from the program, namely parts of the term that cannot change as a result of further reductions. In the case of an infinite computation the output either stops after some finite time, or goes on forever, displaying an increasingly large part of an infinite output term, or the program runs out of memory and produces a failure. The output of the examples shown above will be (*c b a*) and 30 respectively, provided that we have defined *reverse*, *factorial*, and *add* in the natural way.

In a program, each equation is interpreted as a *rewrite rule*. The *left hand side* of the equation is a *pattern* that may *match* a subterm of the term that is currently reduced. A *tree pattern matcher* [HO82a] is a table driven automaton. In each state, the automaton looks at the root symbol of the current subterm and decides what state to go to next and how to move in the term tree. The decisions are based on the root symbol and the current state. If a match occurs, the subterm that matched, is replaced by the corresponding right hand side of the equation. This process is repeated until no more matches can be made, at which point the resulting term is written as output. In practice, the program does not wait until a term is in normal form, but rather starts writing part of the term to the output as soon as it can be shown that some part of the term will never change.

In programming with equations [HOS] the major consumer of CPU time is the creation nodes in right hand sides. There are two ways to improve performance. One is to optimize the time taken to create such a node. The other is to avoid creating such nodes if possible. This paper discusses the second method.

As a motivation for the current work, consider the following term, representing the right hand side of some equation. We assume that the function *if* has the standard definition:

$$if\ [equ\ [g\ [h\ [x\ ]];0];f\ [g\ [h\ [x\ ]]];i\ [g\ [h\ [x\ ]]]]$$

We notice two things with this term. The first is that, if we generate this term tree in the straightforward way, then we end up creating the subterm $g\ [h\ [x\ ]]$ three times. The problem is that $g\ [h\ [x\ ]]$ is a *common subterm* in the term. The second is that once we have evaluated the subterm $equ\ [g\ [h\ [x\ ]];0]$, the value of which is usually either *true* or *false*, then the equations for *if* will keep exactly one of its arguments and throw away the other. This means that no matter what the value of the predicate is, we end up throwing away either $f\ [g\ [h\ [x\ ]]]$ or $i\ [g\ [h\ [x\ ]]]$. Here, the problem is that the term has *deep subterms* that are costly to build, and may be useless.

The reader may argue that these cases are rare. However, deep subterms occur naturally in may programs. Common subterms (or *common subexpressions*) may be rare, but when such subexpressions do occur, it is potentially disastrous not to take care of them. Consider the following definition. It takes a list of pairs (the list and the pairs are both constructed using *cons*) and returns a structure (constructed with the symbol *struct*) of two elements: a list of the first element of each pair in the original list and a list of the second element of each pair in the original list:

$$split\ [()] = struct\ [();()];$$
$$split\ [((f\ .\ s)\ .\ rest)] = struct\ [(f\ .\ firsts\ [split\ [rest]]);(t\ .\ seconds\ [split\ [rest]])];$$

Notice that it is here quite natural to have the expression $split\ [rest]$ occur twice in the right hand side of the second equation. If we create the right hand side in the straightforward manner, without taking into consideration the common subexpressions, we will evaluate the expression $split\ [rest]$ twice. The recurrence relation for the cpu time taken to compute the function *split* becomes $t(n) = 2t(n-1) + c$, were $t(n)$ is the time it takes to compute the function for an input size of n, and c is a constant. With $t(0) = 1$, the solution is exponential. If we recognize the common subexpression and compute it only once, we get $t(n) = t(n-1) + c$ which is linear. In one example, we managed to improve performance of an equational program (a type checker for the language ML) by a factor of more than 100 for some inputs by recognizing and eliminating common subexpressions.

In order to remedy the two problems discussed above, we will make a source-to-source transformation of the set of equations. The transformation will preserve the input-output behavior of the original set of equations, but intermediate steps may differ. In real life, our transformations may not be made as explicit source-to-source transformations, but rather may be a part of the compilation process. It is, however, conceptually easier to view the transformations as operating directly on the source code, and giving a different source program as a result.

The implementation of the equational language requires the set of equations to have the *strong left sequentiality* property. This property is defined and treated in more detail in [OD85]. Strong sequentiality is also discussed in detail in [HL79]. For the purpose of this paper it suffices to note that the property is stronger than the *non-overlapping* property required for the system of equations to have the *Church-Rosser* property (or equivalently the *confluence* property), and that the property is trivially preserved if an equation is added, where the root of the left hand side is a symbol that has no other occurrence in any left hand side of the system, including the left hand side of the new equation. In this paper, we will not elaborate on *strong left sequentiality*, but simply show that our transformations are of the kind that trivially preserves the property. The interested reader is referred to [OD85] for details.

2. Related work

Backus [Ba78] realized early the advantages of functional style. Friedman and Wise [FW76] and Henderson and Morris [HM76] pioneered lazy evaluation. Equational programming was created by Hoffmann and O'Donnell [HO82b] [HOS85] [OD85]. Strong sequentiality was defined by Huet and Lévy [HL79], and is discussed in [OD85] together with strong left sequentiality. Important results on confluence appear in [OD77]. Thatte [Th85] shows how to transform a regular equational program to a program where the usage of symbols in equations is more restricted. Our results apply in particular to such programs.

3. Mutual Properties of Equational Programs

In what follows, we use capital letters T, U, V, ... to denote arbitrary terms. The notation $T[\vec{x}]$ means a term that contains references to the variables in $\vec{x} \equiv x_1, x_2, \ldots, x_n$. We use lower case letters f, g, h, ... to denote literal symbols in terms. Lower case s, t, u are used to denote *symbol variables*. E and F are equational programs. Atomic symbols are shown as lower case letters a, b, c, ... We use standard Arabic digits to denote numerals. Lower case letters v, w, x, ... denote variables. For instance:

$$T[x, y, z] \equiv f[x; g[1; y; z]; a]$$

We use the arrow $\rightarrow$ to mean reduction in one step, and $\rightarrow^*$ to mean reduction in zero or more steps. A subscript on an arrow as in $\rightarrow_E$ and $\rightarrow_E^*$ means reduction with respect to the set E of equations. It is left out whenever E is understood. We write $L(E)$ to refer to the *language* of E, *i.e.*, set of terms made up of symbols used in E.

For the remainder of this paper we assume that all equational programs are *regular* [OD77], which guarantees that our programs have the *Church-Rosser property*, or equivalently, the *confluence property*. Recall that a system E of equations has the *confluence property* iff $W \rightarrow_E^* X$ and $W \rightarrow_E^* Y$ implies that $\exists Z$ such that $X \rightarrow_E^* Z$ and $Y \rightarrow_E^* Z$. We now introduce the concept of *mutual confluence*. The parallel to the *confluence property* should be obvious.

Definition 1:

Two systems, E and F, of equations are said to have the *mutual confluence property*, iff

for every $W \in L(E) \cap L(F)$, $W \rightarrow_E^* X$ and $W \rightarrow_F^* Y$ implies that $\exists Z$ such that $X \rightarrow_E^* Z$ and $Y \rightarrow_F^* Z$, and

for every $W \in L(E) \cap L(F)$, W is in normal form with respect to E iff W is in normal form with respect to F.

Lemma 1:

The mutual confluence property guarantees *mutual uniqueness of normal forms, i.e.*, if $X \in L(E) \cap L(F)$ is in normal form then for every $W \in L(E) \cap L(F)$, $W \rightarrow_E^* X$ if and only if $W \rightarrow_F^* X$.

Proof:

Only if: Since $W \rightarrow_E^* X$ and (trivially) $W \rightarrow_F^* W$, it follows from the mutual confluence property that $\exists Z$ such that $X \rightarrow_E^* Z$ and $W \rightarrow_F^* Z$. Since X is in normal form, the length of the reduction path $X \rightarrow_E^* Z$ must be zero, or equivalently, $X \equiv Z$. It follows that $W \rightarrow_F^* X$. The *if* part is symmetrical.

We now switch our attention to the output of a reduction system. We shall define a *node set* of a term T to be a set of pairs, (p, s) where p is a *path* from the root of the term. A path is a sequence of positive integers $i_1, i_2, \ldots, i_k$. A path uniquely defines a position in a term by giving a sequence of child numbers in a left to right order to follow, starting at the root. We represent a path with numbers separated by periods, e.g. 2.3.1 for the first child of the third child of the second child of the root. The path to the root node is represented by the symbol ϵ. The second element of a pair is the *symbol* of the node specified by the path. A *stable node set* of a term T is a node set of T in which all the nodes are stable, *i.e.*, they cannot change as a result of any reductions in E. An *output set* of a term T is a stable node set, O, of T such that if $(p.i, s) \in O$ for some path p, some integer i and some symbol s, then $(p, t) \in O$ for some symbol t. Intuitively, an output set is a set of stable nodes such that all ancestors of that node are also stable and members of the set. An output set of T is also an output set of every term U such that $T \rightarrow_E^* U$ in any program E. To justify calling such a set an *output set*, we consider a general output scenario, where some process issues a sequence of demands for symbols, located at certain positions of a term. Presumably such a process must know the symbols of all the ancestors of a node, in order to demand the symbol of that node. Otherwise there is a risk that the node does not exist. Furthermore, a node given as the response to a demand from such a process, has to be stable, to insure that it will not change as a result of further reductions. The set of all demands for output together with the responses, up to some time t after start of reduction thus forms an *output set*. All reasonable output strategies (e.g. leftmost outermost) can be thought of as special cases of our general scenario. Finally we define the maximal output set of a term T (written $O_{\max}(T)$) to be the output set of T such that every other output set of T is a subset of $O_{\max}(T)$. The statement "O is an output set of T" will be written $O \subseteq O_{\max}(T)$

Now suppose that some output process P generates a sequence D of demands for output from equational program E working on input term T. We shall assume that P generates a demand, waits for the response, generates a new demand, etc, until either P decides to stop, or until the program enters an infinite computation trying to satisfy a demand, in which case the last demand will remain unsatisfied. P may stop either because the demands and responses represent a term in normal form, or simply because no more demands are needed by P. The

sequence D, called a *demand sequence*, will consist of pairs of demands and responses, where a demand is a *path* and a response is a *symbol*. Elements of a demand sequence are similar to the elements of an output set, but possibly augmented by (p, ϵ) which denotes an unsatisfied demand. If (p, ϵ) is an element of D then it is always the last element. Initially, the only restriction we have on P is that at any time, the elements of D make up an output set, again possibly augmented by (p, ϵ). We shall call such an output process a *general* output process. The set of possible demand sequences may vary depending on *reduction strategy*. Initially, we shall consider *parallel* reduction as it is the most general strategy. By a parallel strategy we mean a strategy where we first find all possible redexes in the term and then reduce them all in some order, e.g. innermost first. The process is then repeated for the new term.

Before we state our main theorem, we state some intermediate results

Lemma 2:

$T \rightarrow^* U$ implies $O_{\max}(T) \subseteq O_{\max}(U)$.

Proof:

This is an immediate consequence of the fact that $O_{\max}(T)$ contains all nodes that are both stable in T, and whose every ancestor is also stable in T.

Lemma 3:

If for some term T, $T \rightarrow^* U$ then a parallel reduction strategy will reduce T in finite time to some term U', such that $O_{\max}(U) \subseteq O_{\max}(U')$.

Proof:

We use Theorem 8 in [OD77]. From that theorem, it follows that a parallel strategy will find a term U' such that $U \rightarrow^* U'$. Using lemma 2 it follows that $O_{\max}(U) \subseteq O_{\max}(U')$

Definition 2:

An equational program F is said to *mimic* an equational program E for a reduction and output strategy S if for every term $T \in L(E) \cap L(F)$, every possible demand sequence D of E on input T is also a possible demand sequence of F on input T.

We shall consider two reduction systems E and F to have the same *output behavior* if E mimics F and F mimics E.

Theorem 1:

Two regular reduction systems E and F that have the *mutual confluence property* have the same output behavior using a general output process and a parallel reduction strategy.

Proof:

We prove that F mimics E. Consider any possible demand sequence, D of E and an input term T. Case 1: the last element of D is a satisfied demand. Then D represents an output set of some term V such that $T \rightarrow_E^* V$. Since E and F have the mutual

confluence property, $\exists U$ such that $V \rightarrow_E^* U$ and $T \rightarrow_F^* U$. By lemma 2, $O_{\max}(V) \subseteq O_{\max}(U)$ By lemma 3 a parallel reduction strategy in F will, in finite time, find a term U' such that $O_{\max}(U) \subseteq O_{\max}(U')$, and therefore $O_{\max}(V) \subseteq O_{\max}(U')$. It follows that D is a possible demand sequence of F on input T. Case 2: the last element of D is an unsatisfied demand. Call the last element l. Case 1 applies to the sequence $D - l$. A term U' will be created by F as above. We must show that no term created by F will satisfy l. Suppose such a term exists. Call it W. Then $U' \rightarrow_F^* W$. Now $T \rightarrow_F^* W$ and $T \rightarrow_E^* V$. By the mutual confluence property it follows that $\exists Z$ such that $V \rightarrow_E^* Z$ and $W \rightarrow_F^* Z$. By lemma 3 a parallel reduction strategy in E will eventually produce a term X such that $O_{\max}(Z) \subseteq O_{\max}(X)$. Since l would be satisfied by W it would also be satisfied by Z and X. In other words, E would eventually satisfy l which contradicts our assumptions. By a similar symmetric argument, E also mimics F. It follows that E and F have the same output behavior.

The results stated above are valid even if we use other output strategies that the general one, and other reductions strategies than the parallel one. We must be careful, however, that the output strategy is not more general than the reduction strategy. Imagine for instance a system where we use the general output strategy, but a leftmost-outermost reduction strategy. In one system, say E we may get a term like $c\ 1[f\ [x];c\ 2[x]]$ where $c\ 1$ and $c\ 2$ are stable, and $f\ [x]$ generates an infinite computation. In a system F we may get $c\ 1[f\ [x];h\ [x]]$ where $h\ [x] \rightarrow c\ 2[x]$. Even if E and F have the mutual confluence property, F does not mimic E. A demand sequence $D \equiv (\epsilon,c\ 1),(2,c\ 2)$ is valid for E, whereas $D \equiv (\epsilon,c\ 1),(2,\epsilon)$ is valid for F, if a leftmost outermost reduction strategy is used. The reason is, of course, that when the demand for the second child of the root is processed by F, it goes off in a leftmost reduction, trying to reduce $f\ [x]$, thus ending up in an infinite computation.

In our implementation of the equational language, the reduction process uses a strategy similar to *leftmost-outermost.* The output strategy is leftmost outermost, but the reduction order tries to make nodes in the term stable in a leftmost-outermost order. This means that reduction process may sometimes deviate from leftmost-outermost, if necessary to make the node requested by the output process stable. We shall call this strategy *leftmost-outermost-stablizing.* Where a leftmost-outermost output strategy is used, demand sequences are restricted. If two elements $(p.i.q\ ,s)$ and $(p.j.r\ ,t)$ appear in a demand sequence D for some paths p, q and r, some integers i and j and some symbols s and t where $i < j$, then $(p.i.q\ ,s)$ always appears before $(p.j.r\ ,t)$ in D. We state here, without proof, a similar result to the one above but for the leftmost-outermost-stablizing strategy.

Theorem 2:

Two reduction systems, E and F that have the mutual confluence property, have the same output behavior using a leftmost-outermost-stablizing reduction strategy.

Proof:

Omitted.

4. Transformations that Preserve Mutual Confluence

We mentioned in the introduction that we are interested in source-to-source transformations from E to F. In particular, we are interested in programs that *simulate* other programs.

Definition 3:

An equational program F is said to *simulate* an equational program E iff

for all terms $W \in L(E) \cap L(F)$, $W \rightarrow_E X$ implies $W \rightarrow_F^* X$,

for all terms $W \in L(E) \cap L(F)$, $W \rightarrow_F^* Y$ implies $\exists U$ such that $Y \rightarrow_F^* U$ and $W \rightarrow_E^* U$, and

for all terms $W \in L(E) \cap L(F)$, W is in normal form with respect to E iff W is in normal form with respect to F.

Theorem 3:

If an equational program F *simulates* an equational program E, then E and F have the mutual confluence property.

Proof:

The mutual confluence property requires that if $W \rightarrow_E^* X$ and $W \rightarrow_F^* Y$ then there must exist a Z such that $X \rightarrow_E^* Z$ and $Y \rightarrow_F^* Z$ From definition 3, we know that there is a U such that $Y \rightarrow_F^* U$ and $W \rightarrow_E^* U$. Clearly $U \in L(E) \cap L(F)$. Since E has the confluence property, there is a Z such that $W \rightarrow_E^* Z$ and $U \rightarrow_E^* Z$. By definition 3, it must also be the case that $U \rightarrow_F^* Z$, and therefore $Y \rightarrow_F^* Z$.

The second part of definition 1 follows immediately from definition 3.

5. Making a term shallow

Our first transformation is used where a term is so big that there is a risk that we end up throwing away a substantial part of the term, depending on the sequence of reductions performed.

We define the *depth* of a term to be the longest path from the root of the term to a leaf of the term. In this context, we will treat as *leaves* of a term, any *variable* or *constant*. The justification for the above is that these categories of symbols do not need a node allocated for them in the term. Constants that are explicitly mentioned in the program are pre-allocated at compile time, and variables pre-exist at runtime. Pre-allocating constants at compile time

gives us the automatic advantage of sharing.

We will say a term is *deep* if it has a depth of at least three. A term that is not deep is *shallow.*

Let E be a set of equations containing the equation $U[\vec{x}] = T_0$, where $T_0 \equiv t_0[T_1[\vec{x}_1];T_2[\vec{x}_2];\ldots;T_n[\vec{x}_n]]$ is a deep term of arity n. Then a nonempty subset $S \subseteq \{T_1, T_2, \ldots, T_n\}$ of terms have a depth of at least 2. Otherwise T_0 would have a depth of no more than 2, and would thus not be deep.

We create a modified set of equations F from E by applying the following transformation:

Transformation 1:

F is created from E by replacing T_0 in E by a new term T_0'. T_0' is created from T_0 by replacing exactly one $T_i \in S$ in T_0 by the term $t_i[\vec{x}_i]$ where t_i is a new literal symbol of arity $|\vec{x}_i|$. In addition, we add a new equation $t_i[\vec{x}_i] = T_i[\vec{x}_i]$ to F.

The term T_0' may or may not be shallow. In addition, we have created a new set of equations that may have deep right hand sides. These newly created right hand sides will eventually be processed recursively so that in the end, all right hand sides are shallow.

Theorem 4:

If applied to a program that has the *strong left sequentiality* property, transformation 1 preserves that property, and thereby the *Church-Rosser* property.

Proof:

The left hand side of the new equation is $t_i[\vec{x}_i]$, where t_i is a new symbol, *i.e.*, it does not occur in any place other than at the root of the added equation. As mentioned in the introduction, adding an equation with such a left hand side, trivially preserves the *strong left sequentiality* property. The *strong left sequentiality* is sufficient for the *Church-Rosser* property to hold. It follows that the *Church-Rosser* is preserved.

Theorem 5:

Transformation 1 implies that F simulates E.

Proof Sketch:

Each reduction in E has a counterpart in F except where $U[\vec{x}]$ is used. In E a reduction $U[\vec{x}] \rightarrow t_0[T_1[\vec{x}_1];T_2[\vec{x}_2];\ldots;T_i[x_i]\ldots;T_n[\vec{x}_n]]$ is simulated in F by the two reduction steps

$U[\vec{x}] \rightarrow t_0[T_1[\vec{x}_1];T_2[\vec{x}_2];\ldots;t_i[x_i]\ldots;T_n[\vec{x}_n]] \rightarrow t_0[T_1[\vec{x}_1];T_2[\vec{x}_2];\ldots;T_i[x_i]\ldots;T_n[\vec{x}_n]]$.

To prove the second half of the definition 3 requires some additional terminology that is outside the scope of this paper. It should be clear, however, that we can construct the required term U by reducing all nodes in Y of the form $t_i[\vec{x}]$ to $T_i[\vec{x}]$.

6. Finding common subexpressions

This topic has been discussed by many different authors in the context of code generation. In combination with making a term shallow, we have some additional considerations that normally do not exist in the context of the present literature. To illustrate that, consider the following right hand side:

$$f\ [g\ [h\ [x\ ];h\ [x\ ]];i\ [y\ ]]$$

It is tempting to consider h[x] a common subexpression of this term. However, consider the case where the term above is the *then* clause of some *if* expression. If we start to compute the common subexpressions, only to find somewhat later that the *else* clause will be used, and the *then* clause will be thrown away, then we wasted the effort. Consequently, we shall consider a subexpression common only if it appears in more than one child of the root symbol. We call such a common subexpression *root common.*

Let E be an equational program containing the equation $U[\vec{x}] = T_0[T_1[\vec{x}];\vec{x}]$ where $T_1[\vec{x}]$ is a root common subexpression in T_0.

Transformation 2:

F is created from E by replacing T_0 by the term $t_0[T_1[\vec{x}];\vec{x}]$, where t_0 is a new symbol of arity $|\vec{x}| + 1$. In addition we add a new equation $t_0[y;\vec{x}] = T_0[y;\vec{x}]$

Theorem 6:

If applied to a program that has the *strong left sequentiality* property, transformation 2 preserves that property, and thereby the *Church-Rosser* property.

Proof:

The left hand side of the new equation is $t_i[y;\vec{x}_i]$, where t_i is a new symbol, *i.e.*, it does not occur in any place other than at the root of the added equation. For the same reason as in Theorem 4, it follows that the *Church-Rosser* is preserved.

Theorem 7:

Transformation 2 implies that F simulates E.

Proof Sketch:

Similar to the proof of Theorem 5.

Transformation 2 is useful, especially if the right hand side of an equation contains a root common subexpression that will take a substantial computation to reduce. Such expressions occur naturally in many programs. In our implementation, transformation 2 interacts with transformation 1 in the following way: we apply transformation 2 where possible, also to new equations generated by the transformation. When no more instances can be found, apply transformation 1 to some equation. If no such instance can be found, stop. Otherwise apply

transformation 1 once, and then repeat the entire process by trying to apply transformation 2 again. Notice that transformation 2 may apply to one of the new equations created by transformation 1 because of the way we define *root common subexpression* above.

We conclude with an example of the two transformations applied to Ackermann's function. The original equation is:

$$a\,(m\,,n\,) = if\ (equ\,(m\,,0),add\,(n\,,1),\\ if\ (equ\,(n\,,0),a\,(subtract\,(m\,,1),1),\\ a\,(subtract\,(m\,,1),a\,(m\,,subtract\,(n\,,1)))))$$

The only common subexpression, $subtract\,(m\,,1)$, is not root common, so transformation 2 does not apply. The third child of the right hand side is deep, so transformation 1 applies. We get:

$$a\,(m\,,n\,) = if\ (equ\,(m\,,0),add\,(n\,,1),p\ 1(m\,,n\,));\\ p\ 1(m\,,n\,) = if\ (equ\,(n\,,0),a\,(subtract\,(m\,,1),1),\\ a\,(subtract\,(m\,,1),a\,(m\,,subtract\,(n\,,1))))$$

where $p\ 1$ is a new symbol. The right hand side of the first equation is shallow. The right hand side of the second equation has a root common subexpression, $subtract\,(m\,,1)$, that appears in both the second and the third child. Applying transformation 2 gives:

$$a\,(m\,,n\,) = if\ (equ\,(m\,,0),add\,(n\,,1),p\ 1(m\,,n\,));\\ p\ 1(m\,,n\,) = p\ 2(subtract\,(m\,,1),m\,,n\,);\\ p\ 2(x\,,m\,,n\,) = if\ (equ\,(n\,,0),a\,(x\,,1),a\,(x\,,a\,(m\,,subtract\,(n\,,1))))$$

where p2 is a new symbol, and x is a new variable. The right hand side of the third equation is deep, so we apply transformation 1 to the third child:

$$a\,(m\,,n\,) = if\ (equ\,(m\,,0),add\,(n\,,1),p\ 1(m\,,n\,));\\ p\ 1(m\,,n\,) = p\ 2(subtract\,(m\,,1),m\,,n\,);\\ p\ 2(x\,,m\,,n\,) = if\ (equ\,(n\,,0),a\,(x\,,1),p\ 3(x\,,m\,,n\,));\\ p\ 3(x\,,m\,,n\,) = a\,(x\,,a\,(m\,,subtract\,(n\,,1)))$$

where p3 is a new symbol. Finally, the right hand side of the last equation is deep, so transformation 1 applies to the second child:

$$a\,(m\,,n\,) = if\ (equ\,(m\,,0),add\,(n\,,1),p\ 1(m\,,n\,));\\ p\ 1(m\,,n\,) = p\ 2(subtract\,(m\,,1),m\,,n\,);\\ p\ 2(x\,,m\,,n\,) = if\ (equ\,(n\,,0),a\,(x\,,1),p\ 3(x\,,m\,,n\,));\\ p\ 3(x\,,m\,,n\,) = a\,(x\,,p\ 4(m\,,n\,));\\ p\ 4(m\,,n\,) = a\,(m\,,subtract\,(n\,,1))$$

In the final program, all the right hand sides are shallow, and there are no common subexpressions in any one right hand side. The two programs have the same output behavior in our implementation, and the transformations described are routinely applied by the compiler.

7. Bibliography

AC75 Aho, A. V. and Corasick, M. J. "Efficient String Matching: An Aid to Bibliographic Search", *Communications of the ACM* 18:6 pp. 333-340 (1975).

Ba78 Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", *Communications of the ACM* 21:8 pp. 613-641 (1978).

FW76 Friedman, D. and Wise, D. "Cons should not evaluate its arguments", *3rd International Colloquium on Automata, Languages and Programming.*, Edinburgh University Press pp. 257-284 (1976).

Ga85 Gabriel, R. P. *Performance and Evaluation of Lisp Systems*, MIT Press, Cambridge, Mass. (1985).

HL79 Huet, G. and Lévy, J.-J. "Computations in Non-ambiguous Linear Term Rewriting Systems", INRIA Technical Report #359 (1979).

HM76 Henderson, P. and Morris, J. H. "A Lazy Evaluator", *3rd ACM Symposium on Principles of Programming Languages* pp. 95-103 (1976).

HO82a Hoffmann, C., O'Donnell, M. "Pattern Matching in Trees", *Journal of the ACM*, pp. 68-95 (1982).

HO82b Hoffmann, C., O'Donnell, M. "Programming with Equations", *ACM Transactions on Programming Languages and Systems* pp. 83-112 (1982).

HOS85 Hoffmann, C., O'Donnell, M. and Strandh, R., "Programming with Equations", *Software, Practice and Experience*, (December 1985).

Jo84 Johnsson, T. "Efficient Compilation of Lazy Evaluation", *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*, Montreal (1984).

KMP77 Knuth, D. E., Morris, J. and Pratt, V. "Fast Pattern Matching in Strings" *SIAM Journal on Computing* 6:2 pp. 323-350 (1977).

OD77 O'Donnell, M. J. *Computing in Systems Described by Equations, Lecture Notes in Computer Science* v. 58, Springer-Verlag (1977).

OD85 O'Donnell, M. J. *Equational Logic as a Programming Language*, MIT Press, Cambridge, Mass. (1985).

Th85 Thatte, S. "On the Correspondence Between Two Classes of Reduction Systems" *Information Processing Letters* 20, pp. 83-85, (1985).