Properties of Unfolding-based Meta-level Systems

Torben Amtoft
Computer Science Department
Aarhus University
Ny Munkegade, building 540
DK-8000 Århus C, Denmark
`e-mail: tamtoft@daimi.aau.dk`

Abstract

It is well known that the performance of a program can often be improved by means of program transformation. Several program transformation techniques, eg. partial evaluation, work as follows: it is recognized that the original program often, when executed, enters states with common components. From these components alone it may be possible to do a lot of computations once and for all, which otherwise would have to be done again and again.

The evaluation of the common components mentioned above may itself benefit from identifying common components and evaluating them separately once and for all. Even this evaluation process may possess common components, etc... – an arbitrarily high level of "nesting" can be achieved, at least in theory.

The purpose of this paper is threefold:

1. A multilevel transition semantics for a logic language will be set up, expressing the ideas above. When restricted to two levels (the number of levels employed by most program transformation systems) the semantics gives a framework general enough to incorporate many program transformation tactics. The framework also includes "run time" - of the original program, of the transformed one and the transformation itself. So one can reason (in a limited way, of course) about efficiency improvements.

2. It has long been suspected that certain kinds of program transformations are able to speed up execution by (at most) a constant factor only: when an interpreter is partially evaluated a constant corresponding to the "interpreter overhead" disappears, when two loops are combined a factor 2 is typically saved, etc. On the other hand, it is easy to come up with examples where execution time is reduced by an order of magnitude. The reason for this can be identified as being either "strong" transformation techniques or a non-optimal execution strategy for the original program. Under certain conditions, reflecting the absence of these factors, it can be rigorously shown that at most a constant factor is achieved. As a simple corollary, it can be shown that by the use of (two-level) program transformation total execution time (ie. transformation time plus execution of the transformed program) can not be smaller than the square root of the execution time of the original program. More generally, by the use of n-level program transformation total execution time can not be reduced to less than the nth root of the original execution time.

3. After a program transformation based on the unfold/fold framework has been performed, it may easily happen that the definition domain of the transformed program is strictly smaller than the definition domain of the original program. We will show that this - when a certain rather weak condition is met - cannot happen within the meta-level framework.

Keywords: Partial evaluation, program transformation, transition semantics, reduction of complexity.

1 Introduction

Consider the *append* program with the *basic rules*

append([A|X],Y,[A|Z]) $\leftarrow$ *append(X,Y,Z).*
append([],Y,Y).

Evaluation of a query *append*(X,Y,*Z*) (where X and Y are ground lists, *Z* a variable) requires $n{+}1$ logical inference

[0]

steps, where n is the length of the list X. On the other hand, using the basic rules it is possible to deduce eg. the *meta rule* (written in the same language as the basic rules)

append([A,B,C|X],Y,[A,B,C|Z]) $\leftarrow$ *append(X,Y,Z)*

in 3 inference steps. Exploiting this rule, the query *append*(X,Y,*Z*), X of lenght n, can be evaluated using about $n/3$ inference steps (on the other hand, the unification performed during each inference step will be more time-consuming than the unification done when the basic rules are applied). So the addition of the meta rule above speeds up execution by a factor 3, corresponding to the meta rule representing a *shortcut* in the computation of length 3.

Of course, it is possible to do arbitrarily long shortcuts. If a shortcut is made of length p, query evaluation can be done in time n/p, the total number of inference steps being $p + n/p$. It is easy to see that for all p, this number is larger than (twice) the square root of n.

On the other hand, in the process of making the meta rule representing a shortcut of length p a *meta-meta rule* representing intermediate shortcuts of length q may be useful, thus increasing the number of levels present to 3. Doing that, the total number of inference steps becomes $n/p + p/q + q$. Now it is easy to see that this number cannot be less than thrice the cube root of n.

Trivial and useless as they may seem, the observations above exhibit some very general properties for transformations based on *unfolding* and *folding* only. Firstly, these kind of transformations are - when performed with a fixed finite set of meta rules (and meta-meta etc, with a generic name simply called meta rules) - able to reduce execution time with at most a constant (depending on this fixed set); secondly, no matter how one chooses this fixed set, the total number of inference steps cannot be less than the nth root of the number of inference steps made when using the basic rules only, where n is the number of levels employed. These claims will be made precise and stated as provable theorems in the subsequent sections.

1.1 Meta-level systems

We now make preparations for formalizing the ideas expressed above. As we want to reason about complexity, the meaning of programs will be expressed in terms of transitions (between various program configurations). The following entities are involved in a meta-level system:

- A sequence of sets of transitions: $\mathrm{T}= \mathrm{T}_1\ \mathrm{T}_2 \ldots \mathrm{T}_i$ where $t \in \mathrm{T}_i$ means that t is a valid transition at level i.
- A sequence of set of meta-rules (where a meta-rule takes the same form as a transition): $\mathrm{M}= \mathrm{M}_0\ \mathrm{M}_1\ \mathrm{M}_2 \ldots \mathrm{M}_i$. M_0 is a representation of the original program, while M_i is the set of meta-rules at level i. We demand that M_i is finite for each $i > 0$. Furthermore, we require that for all $t \in \mathrm{M}_i, i > 0$, it holds that $t \in \mathrm{T}_i$ (ie. that all meta-rules are valid)

A specification of a meta-level system (M,T) can naturally be broken into two parts:

1. Rules defining T_i as function of $\mathrm{M}_0 \ldots \mathrm{M}_{i-1}$ (when working at level i only meta-rules at lower levels can be exploited). These rules will naturally be given as a Plotkin-style transition semantics.
2. A specification of $\mathrm{M}_0\mathrm{M}_1\ldots$ (first the basic rules in M_0 are given, thereby T_1 is implicitly defined via (1), then M_1 must be chosen as a subset of T_1 etc.)

(1) is supposed to be done once and for all (thus specifying the semantics of the system).

1.2 Specifying and Implementing a Meta-level system

In order to specify and implement a meta-level system a number of issues, some of which are listed below, must be settled. These decisions will then implicitly define the M_i sequence. We will not go into details, as the results in this paper do not depend on the form of the meta-level system.

- Given that a transition at level i with "source" s is wanted, it remains to find the corresponding "target", ie. a t such that $s \to t$ is a valid transition at level i. Many such t will exist, representing various degrees of "reduction". Often one will reduce s until some sort of "normal form" is reached; of course care has to be taken to ensure termination.
- For any $i \geq 1$, it must be settled *which* configurations will be sources of transitions in M_i, and *when* these meta-rules are to be generated. Roughly speaking, there are two ways to proceed:
  - To compute the meta-rules bottom up, ie. start to compute all the rules wanted as members of M_1, then (if any) the rules in M_2, etc. When all meta-rules are stored, the system is able to solve queries (now working at the top level). This is the tabulation method and is the one used in most program transformation systems.
  - To use a top-down (or call-by-need) approach: meta-rules are generated only when needed to solve a given query (in an efficient way). This is the technique corresponding to classical memoization.

1.3 Related work

At least when restricted to two levels, meta-level systems as defined in this paper bear a strong resemblance to the *expression procedures* invented by Scherlis [Sch80] in a functional setting (he, however, made no attempt to measure the efficiency improvements). One of his reasons for preferring the framework of expression procedures to the unfold/fold framework of [BurDar77] was that transformation based on expression procedures are guaranteed to preserve termination properties. We will show something analogous in the context of meta-level systems in section 6.

1.4 Overview of the paper

The rest of this paper is organized as follows: In section 2 a couple of known transformation/evaluation strategies are shown to be expressible in terms of meta-level systems. In section 3 an transition semantics is set up, assigning cost measures to each transition. In section 4 some theorems concerning an upper bound on how much can be gained by working on a higher level are given. In section 5 we discuss the applicability of those theorems and the limitations of the framework. Section 6 is devoted to the problem of whether one when working on a higher level risks to enter an infinite computation not present at level 1. Section 7 briefly discusses how to extend the framework to deal with an unrestricted use of folding, as well as how to translate it to a functional setting. Section 8 concludes.

For a full version of this paper, including all proofs omitted here, see [Amt91].

2 Examples

Loop Combining

Consider the set of Horn clauses (which adds one to each element of a list of unary numbers)

a1([],[]).
a1([N|R],[s(N)|R1]) ← a1(R,R1).

a suitable representation of which being M_0. Since the conjunction *a1(X,Y)*, *a1(Y,Z)* *either* reduces to the empty conjunction (denoted □) via the substitution {X/[], Y/[], Z/[]} *or* to the conjunction *a1(X1,Y1)*, *a1(Y1,Z1)* via the substitution {X/[N|X1], Y/[s(N)|Y1], Z/[s(s(N))|Z1]}, (a representation of) the clauses

a1([],[]),a1([],[]).
a1([N|X1],[s(N)|Y1]),a1([s(N)|Y1],[s(s(N))|Z1])
    ← a1(X1,Y1),a1(Y1,Z1).

will qualify as constituting M_1. We can now expect resolution of a query *a1([0,s(0)],Y)*, *a1(Y,Z)* to be faster at level 2, ie. when the clauses in M_1 can be employed, than at level 1.

What really happens in practice is of course that an "eureka"-definition

a2(X,Y,Z) ≡ *a1(X,Y),a1(Y,Z)*

is made, and then the meta-rule is expressed in terms of *a2* (the introduction of *a2* on the right sides representing a *folding* step). Then the meta-rule takes the following form

a2([],[],[]).
a2([N|X1],[s(N)|Y1],[s(s(N))|Z1]) ← a2(X1,Y1,Z1).

and the query above now will be *a2([0,s(0)],Y,Z)*. (Furthermore, if one is not interested in the Y parameter of *a2* one could remove it).

We have thus seen a classical program transformation based on the fold/unfold framework *without "where-abstractions"* [BurDar77] that can be expressed in terms of meta-level systems. We conjecture that almost all examples of applications of the fold/unfold framework can be made to fit into this scheme - see [Sch80] for a further discussion on the subject.

A sufficient condition for a transformation in general to be expressible is that the predicates in question are divided into two classes, A (the "eureka predicates") and P, where all predicates (in A as well as in P) are defined solely in terms of predicates belonging to P - the intuition being that P is the "real" <u>p</u>rogram, while a member of A is just an <u>a</u>bbreviation. Folding is allowed against predicates in A only.

Partial Evaluation

Let *f(X,Y,Z)* be a predicate with input parameters X and Y and output parameter Z, and let *f(a,b,Z)* be a query to be solved. Instead of directly working on this query at level 1, one can instead work on the query *f(a,Y,Z)* at level 1, thus *partially evaluating f* with respect to its first parameter being *a*. The result of this partial evaluation is then stored as a *residual function* in M_1, and finally the original query can be solved at level 2 using this residual function.

The behavior of a "naive" polyvariant partial evaluator (ie. one employing no stronger techniques than first doing some unfolding and then folding back into residual functions) thus fits into the meta-level framework.

The Fibonacci function

Now consider the fibonacci program defined by the following basic rules (where the input is given as unary numbers, and where *add* is left unspecified):

fib(0,1).
fib(s(0),1).
fib(s(s(N)),R) ← fib(s(N),R1),fib(N,R2),add(R1,R2,R).

It is well known that evaluation using this program directly (ie. working at level 1) suffers from exponential time behavior. However, evaluation of *fib*(N,*R)* can be done in *constant time* at level N, provided the M_i's are as follows:

- M_1 contains (a representation of) *fib(s(s(0)),2)*
- M_2 contains (a representation of) *fib(s(s(s(0))),3)*

etc. On the other hand one must also take account of the time used to *derive* those meta-rules. Each of those can be derived in constant time (using "lower" rules), so we end up by stating that evaluation of *fib*(N,*R)* at level N can be done in *linear time.*

As discussed in section 1.2, meta-rules can basically be computed either bottom-up or top-down. If the top-down method is chosen, we in effect simulate the well-known *memoization* technique for computing the fibonacci function.

3 Defining the semantics

We will now define T_i as function of $M_0, \ldots, M_{i-1}$. To do so, we first need (as we work within a logic language) to discuss the properties of substitutions a bit. The treatment is mainly borrowed from [Pal89].

3.1 Substitutions

A substitution is a mapping α from variables into terms such that $x\alpha = x$ for all but a finite number of x, those x where $x\alpha \neq x$ called the *domain* of α. $E(\alpha)$ is defined to be the equation system $(x_1 = t_1, \ldots, x_n = t_n)$ where $t_i = x_i\alpha$ and where $\{x_1 \ldots x_n\}$ is the domain of α. Substitutions can be composed in the natural way. Substitutions are ordered by saying that $\alpha \preceq \beta$ iff there exists a γ such that $\alpha\gamma = \beta$; this makes a preorder.

In our treatment we only need to consider the class of *idempotent* substitutions, where α is idempotent iff $\alpha\alpha = \alpha$. By

- identifying substitutions α and β such that $\alpha \preceq \beta$ and $\beta \preceq \alpha$ (ie. converting the preorder into a partial order)
- adding an extra element $\top$ and stipulating $\alpha \preceq \top$ for all α

we get a complete lattice I (where the empty substitution ε is the bottom element). In fact, the least upper bound (of two elements $\neq \top$) is given by

$$\alpha \uparrow \beta = \texttt{mgu}(E(\alpha) \cup E(\beta)) \qquad (1)$$

If an equation E is not unifiable, $\texttt{mgu}(E)$ is taken to be $\top$.

Example 3.1
Let α be the substitution $\{x/f(y,a), z/g(b)\}$ and let β be $\{x/f(b,w), z/g(y)\}$. Then

$$\begin{aligned} & E(\alpha) \cup E(\beta) \\ = \;& \{x = f(y,a), z = g(b), x = f(b,w), z = g(y)\} \end{aligned}$$

so $\alpha \uparrow \beta = \{y/b, w/a, x/f(b,a), z/g(b)\}$.
On the other hand, $\{x/a\} \uparrow \{x/b\} = \top$.

It is in fact possible (as done in [BKPR89] and implicitly in [Fra85]) to build a theory of logic programming in terms of the least-upper-bound operation. The advantage of this is that then all the nice properties of a complete lattice are inherited, and the use of composition - having bad algebraic properties, as almost nothing but associativity and the existence of a neutral element holds - can be avoided.

3.2 The transition system

A transition in T_i takes the form

$$(\vec{G}, \phi) \overset{i:c}{\to} (\vec{H}_1, \psi_1); \ldots; (\vec{H}_n, \psi_n)$$

Here $\vec{G}$ is a *conjunction* of atomic goals, and $\phi \in I \neq \top$ can be thought of as "the substitutions made before the transition". A pair of the form $(\vec{G}, \phi)$ will be called a *configuration.* After the transition is made, the "search-tree" is split into n branches, each branch again consisting of a conjunction of atomic goals $(\vec{H_i})$ and a substitution ψ_i (which may be $\top$). c is intended to denote the number of inference steps performed during the transition. We also say that c is the *cost* of the transition. Notice that we - in contrast to most operational semantics for logic languages where a transition represents one branch only - keep track of the entire search tree (c can be interpreted as the size of this tree). This is done in order to be able to measure the complexity of the computation process as a whole. For technical reasons, in a configuration $(\vec{G}, \phi)$ we will demand the different atomic goals in $\vec{G}$ to have disjoint variable sets, variables really identical must be glued together by ϕ.

We feel free to drop right hand side configurations containing a $\top$, thus eg.

$$(\vec{G}, \phi) \overset{i:c}{\to} (\vec{H}_1, \psi_1); (\vec{H}_2, \top); (\vec{H}_3, \psi_3) \text{ and}$$

$$(\vec{G}, \phi) \overset{i:c}{\to} (\vec{H}_1, \psi_1); (\vec{H}_3, \psi_3)$$

are identified.

To indicate how programs - ie. collection of Horn Clauses - should be stored as members of M_0, it is best to consider an example: the program

p(0,A,s(A)).
p(s(N),A,R) ← p(N,A,E),p(N,E,R).

(computing 2^n with unary numbers) will be represented as

$$\begin{array}{l}(p(N,A,R),\varepsilon) \overset{0:0}{\rightarrow} \\ (\Box,\phi_1);((p(N1,A,E1),p(N2,E2,R)),\phi_2)\end{array}$$

where
$\phi_1 = \{N/0, R/s(A)\}$, $\phi_2 = \{N/s(N2), N1/N2, E1/E2\}$ (the choice of the cost c for "transitions" in M_0 is immaterial).

We now present the 4 inference rules:

The "compose-rule"

$$\frac{\begin{array}{l}(\vec{G},\phi) \overset{i:c_0}{\rightarrow} (\vec{H}_1,\psi_1);\ldots;(\vec{H}_n,\psi_n), \forall j \in \{1\ldots n\}: \\ (\vec{H}_j,\psi_j) \overset{i:c_j}{\rightarrow} (\vec{K}_{j1},\theta_{j1});\ldots;(\vec{K}_{jm_j},\theta_{jm_j})\end{array}}{(\vec{G},\phi) \overset{i:c}{\rightarrow} (\vec{K}_{11},\theta_{11});\ldots;(\vec{K}_{nm_n},\theta_{nm_n})}$$

where $c = c_0 + c_1 + \ldots + c_n$. Each branch in the search tree can thus be developed further on. Notice that the cost assigned to the transition in the conclusion mirrors that the different branches are supposed to be searched sequentially; to model or-parallism the definition of c should be changed into $c = c_0 +\ \max(c_1,\ldots,c_n)$.

The "and-rule"

$$\frac{\begin{array}{l}(\vec{G_1},\phi) \overset{i:c_1}{\rightarrow} (\vec{H}_{11},\psi_{11});\ldots;(\vec{H}_{1n},\psi_{1n}) \\ (\vec{G_2},\phi) \overset{i:c_2}{\rightarrow} (\vec{H}_{21},\psi_{21});\ldots;(\vec{H}_{2m},\psi_{2m})\end{array}}{(\vec{G_1}\vec{G_2},\phi) \overset{i:c}{\rightarrow} \ldots;(\vec{H_{1j}}\vec{H_{2k}},\theta_{jk});\ldots}$$

where $c = c_1 + c_2$, $\theta_{jk} = \psi_{1j} \uparrow \psi_{2k}$ for $1 \le j \le n$, $1 \le k \le m$. Notice that reduction of a conjunction not necessarily takes place from left to right, in contrary to what is the case in most current logic languages. This is done to model the fact that eg. a partial evaluator may work on the second goal in a conjunction before having solved the first goal completely. The significance of this discrepancy will be discussed in section 5. Again the model does not cater for and-parallism, to do so the definition of c should be changed into $c = \max\ (c_1, c_2)$.

The "unfold-rule"

$$\frac{\begin{array}{c}(\vec{G},\phi) \overset{i':c'}{\rightarrow} (\vec{G}_1,\phi_1);\ldots;(\vec{G}_n,\phi_n) \text{ is} \\ \text{(a renamed version of) a rule in } \mathrm{M}_{i'}, i' < i, \phi \preceq \psi\end{array}}{(\vec{G},\psi) \overset{i:c}{\rightarrow} (\vec{G}_1,\psi_1);\ldots;(\vec{G}_n,\psi_n) \text{ where } \psi_j = \psi \uparrow \phi_j}$$

Here c is the number of j such that $\psi_j \ne \top$, however $c = 1$ if this number is zero, and the renaming is done in order to avoid name clashes.

Our model does not attempt to estimate the complexity of a given unification process, but only counts the number of branches that are created.

Finally a rule, expressing that just nothing can happen:

$$\frac{}{(\vec{G},\phi) \overset{i:0}{\rightarrow} (\vec{G},\phi)}$$

Observation 3.2 *It can easily be proven by induction in the derivation tree that in a transition*

$$(\vec{G},\phi) \overset{i:c}{\rightarrow} (\vec{H}_1,\psi_1);\ldots;(\vec{H}_n,\psi_n)$$

it for all i holds that $\phi \preceq \psi_i$.

3.3 "Loop Combining" revisited

We will now go through the *a1* example from section 2 in details. As the substitutions involved can become quite large, involving many intermediate (later on superfluous) items, we will cheat a bit and only show the relevant items present in a given substitution.

M_0 contains the *a1* procedure represented as the transition

$$(a1(X,Y),\varepsilon) \overset{0:0}{\rightarrow} (\Box,\phi_1);(a1(X1,Y1),\phi_2)$$

where
$\phi_1 = \{X/[], Y/[]\}$, $\phi_2 = \{X/[N1|X1], Y/[s(N1)|Y1]\}$. By the unfold-rule, we have

$$(a1(X,Y),\{Y/U\}) \overset{1:2}{\rightarrow} (\Box,\alpha_1);(a1(X1,Y1),\alpha_2) \qquad (2)$$

where $\alpha_i = \phi_i \uparrow \{Y/U\}$ for $i = 1,2$. Applying the unfold-rule to the configuration $a1(U,Z),\{Y/U\}$ and the following renamed version of *a1*:

$$(a1(U,Z),\varepsilon) \overset{0:0}{\rightarrow} (\Box,\psi_1);(a1(U1,Z1),\psi_2)$$

(where
$\psi_1 = \{U/[], Z/[]\}$, $\beta_2 = \{U/[M1|U1], Z/[s(M1)|Z1]\}$) we get

$$(a1(U,Z),\{Y/U\}) \overset{1:2}{\rightarrow} (\Box,\beta_1);(a1(U1,Z1),\beta_2) \qquad (3)$$

where $\beta_i = \psi_i \uparrow \{Y/U\}$ for $i = 1,2$. Combining (2) and (3) via the and-rule, we arrive at

$$\begin{array}{l}((a1(X,Y),a1(U,Z)),\{Y/U\}) \overset{1:4}{\rightarrow} \\ (\Box,\gamma_{11});(a1(U1,Z1),\gamma_{12});(a1(X1,Y1),\gamma_{21}); \\ ((a1(X1,Y1),a1(U1,Z1)),\gamma_{22})\end{array}$$

where $\gamma_{ij} = \alpha_i \uparrow \beta_j$ for $i,j = 1,2$. Since $\gamma_{12} = \gamma_{21} = \top$, this can (using our convention about throwing away $\top$) be written as the following meta-rule, to be stored in M_1:

$$\begin{array}{l}((a1(X,Y),a1(U,Z)),\{Y/U\}) \overset{1:4}{\rightarrow} \\ (\Box,\gamma_{11});((a1(X1,Y1),a1(U1,Z1)),)\gamma_{22})\end{array} \qquad (4)$$

where $\gamma_{11} = \{X/[], Y/[], U/[], Z/[]\}$, $\gamma_{22} = \{X/[N1|X1], Y/[s(N1)|Y1], Z/[s(s(N1))|Z1], U1/Y1, U/[s(N1)|Y1]\}$.

Now consider the query *(a1([s(0)],Y),a1(Y,Z))*. We must write this as the configuration

$$(a1(X,Y), a1(U,Z), \{Y/U, X/[s(0)]\}) \tag{5}$$

Using the M_1-rule (4) we by the unfold-rule obtain

$$\begin{aligned}((a1(X,Y), a1(U,Z))\{Y/U, X/[s(0)]\}) &\overset{2:1}{\rightarrow} \\ (\Box, \delta_1); ((a1(X1,Y1), a1(U1,Z1)), \delta_2)\end{aligned} \tag{6}$$

$$\begin{aligned}\delta_1 = {} & \gamma_{11} \uparrow \{X/[s(0)]\} = \top, \\ \delta_2 = {} & \gamma_{22} \uparrow \{X/[s(0)]\} \\ = {} & \{X/[s(0)], Y/[s(s(0))|Y1], U/[s(s(0))|Y1], \\ & U1/Y1, Z/[s(s(s(0)))|Z1], X1/[]\}\end{aligned}$$

After renaming (4) (by exchanging all variable suffices from 1 to 2 and giving variable without suffices the suffix 1), we by the unfold-rule get the transition

$$\begin{aligned}((a1(X1,Y1), a1(U1,Z1)), \delta_2) \overset{2:1}{\rightarrow} \\ (\Box, \sigma_1); ((a1(X2,Y2), a1(U2,Z2)), \top)\end{aligned} \tag{7}$$

where $\sigma_1 = \delta_2 \uparrow \{X1/[], Y1/[], U1/[], Z1/[]\}$. By combining (6) and (7) using the "compose-rule", we arrive at

$$\begin{aligned}((a1(X,Y), a1(U,Z))\{Y/U, X/[s(0)]\}) \overset{2:2}{\rightarrow} \\ (\Box, \{X/[s(0)], Y/[s(s(0))], U/[s(s(0))], Z/[s(s(s(0)))]\})\end{aligned}$$

ie. we can solve (5) at level 2 at cost 2. On the other hand, it is easily seen that to solve it at level 1 requires cost 4. So in this case (which can be generalized to lists of arbitrary length), working at one higher meta-level makes the execution time (in our model) decrease by a factor 2.

4 Upper bounds on time-improvement

We now formulate and prove some basic properties of meta-level systems. First a lemma stating that "by making the query more specific the evaluation tree gets pruned":

Lemma 4.1
Suppose $(\vec{G}, \phi) \overset{i:c}{\rightarrow} (\vec{G}_1, \phi_1); \ldots; (\vec{G}_n, \phi_n)$
Let $\phi \preceq \beta$. *Then there exists a* $c' \leq c$ *such that*

$$(\vec{G}, \beta) \overset{i:c'}{\rightarrow} (\vec{G}_1, \psi_1); \ldots; (\vec{G}_n, \psi_n)$$

where $\psi_j = \phi_j \uparrow \beta$ *for* $j \in \{1 \ldots n\}$.

Proof: A straight-forward induction in the derivation tree. □

Definition 4.2 (Maximal Short Cut) *Let (M,T) be a meta-level system. For all* $i > 0$ *we define* S_i *by*

$$S_i = max(max\{c | c \textit{ is the cost of a rule in } M_i\}, 1)$$

(this is well-defined since there are finitely many rules in M_i only). S_i can be thought of as the maximal short cut possible by applying transitions at level i.

Definition 4.3 (Time to create M_i) *Let (M,T) be a meta-level system. For all* $i > 0$ *we define* C_i *by*

$$C_i = max(\sum_{t \in M_i} \textit{the cost of } t, 1)$$

C_i can be thought of as the time required to calculate the meta-rules in M_i.

Definition 4.4 (Total Meta-time) *Let (M,T) be a meta-level system. For all* $i > 0$ *we define* TC_i *by*

$$TC_i = C_1 + \ldots + C_i$$

TC_i *can be thought of as the time required to calculate all the meta-rules (of level not greater than i).*

We are now in position to prove that we by "going up one level" are able to reduce evaluation time by at most a constant:

Theorem 4.5 (At most constant speed-up) *Let (M,T) be a meta-level system. Suppose*

$$(\vec{G}, \phi) \overset{i+1:c}{\rightarrow} (\vec{H}_1, \psi_1); \ldots; (\vec{H}_n, \psi_n)$$

for some $i > 0$. *Then there exists a* c' *such that*

$$(\vec{G}, \phi) \overset{i:c'}{\rightarrow} (\vec{H}_1, \psi_1); \ldots; (\vec{H}_n, \psi_n)$$

and $c' \leq S_i c$

Proof: A straight-forward induction in the derivation tree, the interesting case being when the "unfold-rule" is applied: Suppose we have

$$(\vec{G}, \psi) \overset{i+1:c}{\rightarrow} (\vec{G}_1, \psi_1); \ldots; (\vec{G}_n, \psi_n)$$

because (a renamed version of) the following rule (where $\phi \preceq \psi$) belongs to $\mathrm{M}_{i'}$, $i' < i + 1$:

$$(\vec{G}, \phi) \overset{i':c'}{\rightarrow} (\vec{G}_1, \phi_1); \ldots; (\vec{G}_n, \phi_n)$$

Here $\psi_j = \phi_j \uparrow \psi$ for $i \in \{1 \ldots n\}$, $c =$ the number of j such that $\psi_j \neq \top$, however at least 1. There are two cases:

$i' < i$**:**
Now $(\vec{G}, \psi) \overset{i:c}{\rightarrow} (\vec{G}_1, \psi_1); \ldots; (\vec{G}_n, \psi_n)$
and since by definition $\mathrm{S}_i \geq 1$ we have $c \leq \mathrm{S}_i c$.

$i' = i$**:** Since a rule in M_i represents a transition in T_i, we have

$$(\vec{G},\phi) \overset{i:c'}{\rightarrow} (\vec{G}_1,\phi_1);\ldots;(\vec{G}_n,\phi_n)$$

Lemma 4.1 now tells us that there exists a $c'' \leq c'$ such that

$$(\vec{G},\psi) \overset{i:c''}{\rightarrow} (\vec{G}_1,\psi_1);\ldots;(\vec{G}_n,\psi_n)$$

This is as desired, since $c'' \leq c' \leq \mathrm{S}_i \leq \mathrm{S}_i c$ (exploiting $c \geq 1$).

□

An immediate corollary of the above is

Corollary 4.6 *Let (M,T) be a meta-level system. Suppose*

$$(\vec{G},\phi) \overset{i:c_i}{\rightarrow} (\vec{G}_1,\phi_1);\ldots;(\vec{G}_n,\phi_n)$$

for some $i > 0$. Then there exists a c_1 such that

$$(\vec{G},\phi) \overset{1:c_1}{\rightarrow} (\vec{G}_1,\phi_1);\ldots;(\vec{G}_n,\phi_n) \text{ and}$$

$$c_1 \leq S_1 \ldots S_{i-1} c_i \tag{8}$$

Exploiting that the inequality

$$n^n(a_1 \ldots a_n) \leq (a_1 + \ldots + a_n)^n \tag{9}$$

holds for arbitrary positive $a_1 \ldots a_n$, we from (8) can deduce that

$$\begin{aligned} c_1 \leq\ & \mathrm{S}_1 \ldots \mathrm{S}_{i-1} c_i \leq \mathrm{C}_1 \ldots \mathrm{C}_{i-1} c_i \\ \leq\ & \frac{(\mathrm{C}_1 + \ldots + \mathrm{C}_{i-1} + c_i)^i}{i^i} = (\frac{\mathrm{TC}_{i-1} + c_i}{i})^i \end{aligned}$$

We have thus proved the following interesting theorem:

Theorem 4.7 (At most polynomial speed-up) *Let (M,T) be a meta-level system. Suppose*

$$(\vec{G},\phi) \overset{i:c_i}{\rightarrow} (\vec{G}_1,\phi_1);\ldots;(\vec{G}_n,\phi_n)$$

Then there exists a constant c_1 such that

$$(\vec{G},\phi) \overset{1:c_1}{\rightarrow} (\vec{G}_1,\phi_1);\ldots;(\vec{G}_n,\phi_n) \text{ and}$$

$$c_i + TC_{i-1} \geq i\sqrt[i]{c_1} \tag{10}$$

5 Discussion

The topic of the first part of this section will be how the results above apply to the examples from section 2. In the second part, discrepancies between our model and the "real world" of computations are treated in some detail.

Loop Combining

As $\mathrm{S}_1 = 4$ in this case (as the cost of (4) is 4), theorem 4.5 tells us that evaluation at level 2 cannot be more than 4 times as fast as evaluation at level 1. As mentioned in section 3.3, we actually gain a factor 2.

Partial Evaluation

Theorem 4.5 says that (in our model) we cannot gain more than a constant by using a residual program instead of the original one, this constant being independent of the dynamic input. However, the constant may depend on the value of the *static input*, since different values of the static input give rise to different residual functions and hence different meta-level systems. Thus - in the case of an interpreter being partially evaluated - the familiar notion of an "interpretation overhead" independent of the source program is *not* supported by our theorems. There are good reasons for this, as it is easy to construct "interpreters" where the corresponding interpretation overhead can become arbitrarily large. However, for most "realistic" interpreters the interpretation overhead in fact is independent of the source program, loosely speaking due to the fact that each construct in the language being interpreted gives rise to a run-time action.

The Fibonacci function

Evaluation of *fib(n)* at level 1 requires an exponential number of inference steps, say (about) a^n. Theorem 4.7 then tells us that we - when using n levels - cannot hope for a number of inference steps smaller than n times the nth root of a^n, ie. an. As we saw in section 2 it is in fact possible to compute *fib(n)* in time proportional to n using n meta-levels. So in this case we (apart from possibly a constant factor) get as much speed up as we could expect.

Notice that even though theorem 4.7 apparently states that we at most get a polynomial improvement when using a meta-level system, the fibonacci function is reduced from having exponential time-behavior into being linear. This is due to the fact that there is no bound on the number of levels employed; in fact the number of levels equals the input.

5.1 Limitations of the model

In the "real world" it is possible to come up with examples of programs being sped up with an order of magnitude when executed at "one higher level". On the other hand, theorem 4.5 says that in our model this is impossible. There are two features not accounted for by our model:

- The use of "strong" transformation techniques. The model only caters for simple use of folding/unfolding.
- The model implicitly assumes that when solving a conjunction of goals, the order in which the subgoals are solved is chosen in an optimal way (eg. theorem 4.5 says that there *exists* a transition at level 1 with certain properties). In practice, however, one has to choose a simple strategy, eg. to evaluate the goals from left to right. It is well known from the theory of logic programming that such a strategy may be suboptimal.

We will now embark upon each of those two features:

Strong transformation techniques

In the model, T_1 and T_2 in some sense are isomorphic: the latter has access to more rules than the former, but as those rules are derivable in T_1, the transitions at level 2 can (as expressed in theorem 4.5) be simulated at level 1. On the other hand, suppose M_1 contained a rule of the form

$$((P, Q), \alpha) \dot{\rightarrow} (P, \alpha) \tag{11}$$

where α is such that $P\alpha = Q\alpha$. (11) states that identical conjuncts need to be evaluated only once (in the world of functional programming, the natural counterpart to (11) is the "where-abstraction": eg. an expression *f(g(x),g(x))* is transformed into *f(v,v)* **where** $v =$ *g(x)*). (11) cannot be simulated at level 1, and our intuition about meta-rules representing a finite shortcut thus does not hold: the shortcut gained by applying (11) can be arbitrary big, in fact equal the cost needed to solve Q the second time. Using rules like (11) eg. the fibonacci program can be transformed into a linear version.

Another way to improve efficiency by (potentially) more than a constant is to prove two expressions to be equivalent (typically by some kind of induction) and then replace the second by the first. To see an example of this (in a functional setting), consider the list append operator $\circ$. We want to show that it is associative, ie. that $x \circ (y \circ z) = (x \circ y) \circ z$ for all x, y, z. This is done by induction in x, where the basic step comes from the unfoldings

$$\begin{aligned} [] \circ (y \circ z) &\rightarrow y \circ z, \\ ([] \circ y) \circ z &\rightarrow y \circ z \end{aligned}$$

and where the induction step comes from the unfoldings

$$(x :: w) \circ (y \circ z) \quad \rightarrow \quad (x :: (w \circ (y \circ z))), \tag{12}$$

$$\begin{aligned} ((x :: w) \circ y) \circ z \quad &\rightarrow \quad (x :: (w \circ y)) \circ z \\ &\rightarrow \quad (x :: ((w \circ y) \circ z)) \end{aligned} \tag{13}$$

Since two unfoldings are made in (13) and only one in (12), we see that the expression $x \circ (y \circ z)$ is more efficient than its semantic equivalent $(x \circ y) \circ z$: by replacing the latter by the former one may save arbitrarily many (namely the length of x) unfoldings. Using the equivalence above (and also exploiting that $x \circ [] = x$ for all x), one can (using an "accumulating parameter") transform a program doing list reversal from being quadratic into being linear (in the length of the list). Another application of the "rewriting" technique is given in [PetBur82]: by a sequence of steps, where function definitions are replaced by equivalent but more efficient counterparts, the fibonacci program is transformed into a logarithmic version.

[BurDar77, p. 48,64] gives a brief informal account of the efficiency (measured in terms of aritmetic operations performed) of the new program versus the efficiency of the old. Here **where**-abstractions and rewritings are claimed to be the only sources of efficiency improvement; unfolding only preserves efficiency. As our model measures efficiency in terms of the number of inference steps made, the latter claim does not hold in our framework.

Evaluating a Conjunct

We now give an example of a program P which can be transformed into a program R such that R, when solved using a left-to-right strategy, is faster than P, when solved using a left-to-right strategy, by an order of magnitude. This is due to the fact that the transformation process itself carries out some "not-left-to-right" steps. The program is

switch([],[]).
switch([a|X],[b|Y]) $\leftarrow$ *switch(X,Y).*
switch([b|X],[a|Y]) $\leftarrow$ *switch(X,Y).*
onlya([]).
onlya([a|X]) $\leftarrow$ *onlya(X).*

Consider queries of the form

switch([a,s_1*,. . . ,*s_n*],Y),onlya(Y)*

where each s_i is either an "a" or a "b". It is easily seen that to evaluate these at level 1 using the left-to-right strategy at least n inference steps are needed. On the other hand, since (with *p(X,Y),*α being an abbreviation for *(switch(X,Y),onlya(Z)),*$\alpha \uparrow \{Y/Z\}$)

$$(p(X,Y), \varepsilon) \overset{1:6}{\rightarrow}$$
$$(\Box, \{X/[], Y/[]\}); (p(X1, Y1), \{X/[b|X1], Y/[a|Y1]\})$$

we can include this as a rule in M_1. Then we can evaluate (to fail) such queries in 1 step at level 2 (using a left-to-right strategy!), regardless of n. In a functional setting the above phenomenon has the counterpart: for a strict language, the call-by-name nature of program transformation may improve evaluation time by more than a constant factor.

6 Termination properties

It may happen that the program resulting from a program transformation terminates less often than the original one. To illustrate this within the framework of meta-level-systems, consider the program

p(a).
q(X).
q(b) ← q(b).

represented as the M_0 rules

$$(p(X),\varepsilon) \overset{0:0}{\to} (\square,\{X/a\})$$
$$(q(Y),\varepsilon) \overset{0:0}{\to} (\square,\varepsilon);(q(Y),\{Y/b\})$$

When the query (*p(X),q(X)*) (represented as the configuration ((p(X), q(Y)), {X/Y})) is evaluated at level 1 using the left-to-right strategy, termination is guaranteed (and fast). On the other hand, suppose the metarule

$$((p(X),q(Y)),\{X/Y\}) \overset{1:2}{\to}$$
$$(p(X),\{X/Y\});((p(X),q(Y)),\{X/b,Y/b\})$$

(which is a valid transition in T_1) is member of M_1. This corresponds to that the original program is transformed into

r(X) ← p(X).
r(b) ← r(b).
p(a).

and the query above can be formulated as *r(X)*. Here at level 2, left-to-right evaluation of this query can go on forever, since the the second rule for r will cause the search tree to be infinite.

What goes wrong in the example above is essentially that the metarule constructed does not represent any *leftmost* transition, but only represents a (useless) rightmost transition. Below we will see that if meta rules *do* carry out some "leftmost" actions, then left-to-right evaluation at level 2 will terminate when left-to-right evaluation at level 1 does.

What we are interested in is at first glance a theorem stating something like "if a configuration can give rise to an infinite computation at level 2, then it also can give rise to an infinite computation at level 1". On the other hand, interpreted within the model used so far such a theorem (though pretty easy to prove under some very weak conditions) will not be very interesting, since almost any logic programs can give rise to infinite computation if the evaluation order is chosen sufficiently "stupid". Therefore we need to modify the model a bit, in a way such that the complexity is divided into two parts: one measuring the "number of left-most inference steps made" and the second measuring the "number of other inference steps". For ease of presentation, we also change the model so each transition represents a single branch of the search tree only. A transition thus now takes the form

$$(\vec{G},\phi) \overset{i:(l,r)}{\longrightarrow} (\vec{G_1},\phi_1)$$

denoting that the configuration ($\vec{G}$,ϕ) at level i reduces to the configuration $(\vec{G_1},\phi_1)$, where $\phi_1 \neq \top$, using l (called the *left-cost*) left-to-right steps and r (called the *right-cost*) other steps. Left-to-right evaluation is thus modelled by the right-cost being 0. What we are after is something like the following two theorems:

Hoped-for-Theorem 6.1 *Let a meta-level system be given, where all rules have a left cost greater than 0 (cf. the discussion earlier). Suppose*

$$(\vec{G},\phi) \overset{i+1:(l,r)}{\longrightarrow} (\vec{G_1},\phi_1)$$

Then there exists $l' \geq l$ and r' such that

$$(\vec{G},\phi) \overset{i:(l',r')}{\longrightarrow} (\vec{G_1},\phi_1)$$

Hoped-for-Theorem 6.2
Suppose $(\vec{G},\phi) \overset{1:(l,-)}{\longrightarrow} (\vec{H},\psi)$
(where the symbol "-" denotes that we do not care about the actual value of the cost). Then there exists $l' \geq l$, $\vec{J}$,θ such that

$$(\vec{G},\phi) \overset{1:(l',0)}{\longrightarrow} (\vec{J},\theta), (\vec{J},\theta) \overset{1:(-,-)}{\longrightarrow} (\vec{H},\psi)$$

These theorems will enable us to reason as follows: suppose a query ($\vec{G}$,ϕ) at level i can be evaluated using a left-to-right strategy infinitely far, ie. for all n there exists a configuration $(\vec{H_n},\psi_n)$ such that

$$(\vec{G},\phi) \overset{i:(n',0)}{\longrightarrow} (\vec{H_n},\psi_n)$$

where $n' \geq n$. Then repeated application of theorem 6.1 tells us that there exists a $n'' \geq n'$ such that

$$(\vec{G},\phi) \overset{1:(n'',-)}{\longrightarrow} (\vec{H_n},\psi_n)$$

and theorem 6.2 then tells us that there exists a configuration $(\vec{J_n},\theta_n)$ and a $n''' \geq n''$ such that

$$(\vec{G},\phi) \overset{1:(n''',0)}{\longrightarrow} (\vec{J_n},\theta_n)$$

ie. that left-to-right evaluation of the configuration $(\vec{G},\phi)$ can also at level 1 go on infinitely far.

Now we give the new definition of T_i. In order to give transitions a bit more structure, we actually use two kinds of transitions, denoted $\longrightarrow$ and $\Longrightarrow$ respectively. The latter can be thought of as the transitive closure of the former.

The (new) compose-rule

$$\frac{(\vec{G},\phi) \overset{i:(l_1,r_1)}{\longrightarrow} (\vec{G_1},\phi_1), (\vec{G_1},\phi_1) \overset{i:(l_2,r_2)}{\Longrightarrow} (\vec{G_2},\phi_2)}{(\vec{G},\phi) \overset{i:(l,r)}{\Longrightarrow} (\vec{G_2},\phi_2)} \quad (14)$$

where $l = l_1 + l_2, r = r_1 + r_2$.

The (new) and-rule

$$\frac{\begin{array}{l}(\vec{G_1},\phi) \overset{i:(l_1,r_1)}{\longrightarrow} (\vec{H_1},\psi_1),\\ (\vec{G_2},\phi) \overset{i:(l_2,r_2)}{\longrightarrow} (\vec{H_2},\psi_2), \psi_1 \uparrow \psi_2 \neq \top\end{array}}{(\vec{G_1}\vec{G_2},\phi) \overset{i:(l,r)}{\longrightarrow} (\vec{H_1}\vec{H_2},\psi)} \quad (15)$$

where $l = l_1, r = l_2 + r_1 + r_2, \psi = \psi_1 \uparrow \psi_2$.

The (new) unfold-rule

$$\frac{\begin{array}{l}(\vec{G},\phi) \overset{j:(l,r)}{\Longrightarrow} (\vec{G_1},\phi_1) \text{ is a rule in } \mathrm{M}_j,\\ j < i, \phi \preceq \beta, \beta \uparrow \phi_1 \neq \top\end{array}}{(\vec{G},\beta) \overset{i:(1,0)}{\longrightarrow} (\vec{G_1},\beta_1) \text{ where } \beta_1 = \beta \uparrow \phi_1} \quad (16)$$

As before, renaming is done in order to avoid name-clashes. Finally two "simple rules":

$$\frac{}{(\vec{G},\phi) \overset{i:(0,0)}{\longrightarrow} (\vec{G},\phi)}$$

and

$$\frac{(\vec{G},\phi) \overset{i:(l,r)}{\longrightarrow} (\vec{G_1},\phi_1)}{(\vec{G},\phi) \overset{i:(l,r)}{\Longrightarrow} (\vec{G_1},\phi_1)}$$

Lemma 6.3 *Let a meta-level system be given. Suppose*

$$(\vec{G},\phi) \overset{i:(l,r)}{\longrightarrow} (\vec{G_1},\phi_1)$$

and suppose that $\phi \preceq \beta$, $\beta \uparrow \phi_1 \neq \top$. Then

$$(\vec{G},\beta) \overset{i:(l,r)}{\longrightarrow} (\vec{G_1},\beta_1)$$

where $\beta_1 = \beta \uparrow \phi_1$.

The above also holds when $\longrightarrow$ is replaced by $\Longrightarrow$ in the premise as well as in the conclusion.

The proof is similar to the proof of lemma 4.1. Notice that $=$ instead of $\leq$ holds between the costs involved, due to the fact that the time spent on unsuccessful branches is not included.

Lemma 6.4 *(14) and (15) are still valid, even if one replaces* all *occurrences of $\longrightarrow$ by $\Longrightarrow$ (in the premises as well as in the conclusion).*

Proof: Concerning (14), one proceeds by induction in the derivation tree for the first transition in the premise. Concerning (15), one first shows that it is still valid when $\longrightarrow$ is replaced by $\Longrightarrow$ in the first premise and in the conclusion; then one shows that (15) is valid even if the replacement is done also for the second premise. □

Definition 6.5 *A meta-level system is* left-progressing *if it for all rules $\in M_i$ $(i > 0)$:*

$$(\vec{G},\phi) \overset{i:(l,r)}{\Longrightarrow} (\vec{G_1},\phi_1)$$

holds that $l > 0$

Now we are in position to formulate and prove our first "hoped-for-theorem" (6.1).

Theorem 6.6 *Let a left-progressing meta-level system be given. Suppose*

$$(\vec{G},\phi) \overset{i+1:(l,r)}{\longrightarrow} (\vec{G_1},\phi_1)$$

$(i > 0)$. Then there exists $l' \geq l$ and r' such that

$$(\vec{G},\phi) \overset{i:(l',r')}{\Longrightarrow} (\vec{G_1},\phi_1)$$

The above also holds when $\longrightarrow$ is replaced by $\Longrightarrow$ in the premise.

Proof: An easy induction in the derivation tree. To cope with the compose-rule and the and-rule we need lemma 6.4. The only interesting case is the unfold-rule (16): suppose that

$$(\vec{G},\beta) \overset{i+1:(1,0)}{\longrightarrow} (\vec{G_1},\beta_1)$$

because there is a rule in M_i

$$(\vec{G},\phi) \overset{i:(l_1,r_1)}{\Longrightarrow} (\vec{G_1},\phi_1)$$

where $\phi \preceq \beta$ and $\beta_1 = \phi_1 \uparrow \beta$. Then lemma 6.3 tells us that

$$(\vec{G},\beta) \overset{i:(l_1,r_1)}{\Longrightarrow} (\vec{G_1},\beta_1)$$

which is as desired since $l_1 \geq 1$ according to our assumption about the meta-level system being left-progressing. □

Next we turn to the second "hoped-for-theorem" (6.2). It is in order to facilitate the proof of this we have distinguished between $\longrightarrow$ and $\Longrightarrow$. It turns out that in order for the theorem to hold, we must require the meta-level system to be "standard":

Definition 6.7 *A meta-level system is* standard *iff all the basic rules in M_0 are of the form*

$$(G,\varepsilon) \overset{0:(-,-)}{\longrightarrow} (\vec{H}_1,\psi_1)$$

ie. the left side consists of a single atomic goal only, and there are no "already existing substitutions".

To require this is no serious restriction, as this will be the case whenever M_0 consists of "translations" of Horn Clauses.

Lemma 6.8 *Let a meta-level system be given. Suppose*

$$(\vec{G},\phi) \overset{i:(l,-)}{\longrightarrow} (\vec{H},\psi)$$

If $\vec{H} = \Box$ then there exists $l' \geq l$ such that

$$(\vec{G},\phi) \overset{i:(l',0)}{\Longrightarrow} (\Box,\psi)$$

If $\vec{H}$ is non-empty, ie. of the form $H_h\vec{H}_r$, then there exists $\vec{J}$,θ and $l' \geq l$ such that

$$(\vec{G},\phi) \overset{i:(l',0)}{\Longrightarrow} (H_h\vec{J},\theta), (\vec{J},\theta) \overset{i:(-,-)}{\longrightarrow} (\vec{H}_r,\psi)$$

Proof: Omitted. □

Lemma 6.9 *Let a standard meta-level system be given. Suppose*

$$(\vec{G},\phi) \overset{1:(-,-)}{\longrightarrow} (\vec{H},\psi), (\vec{H},\psi) \overset{1:(l,0)}{\longrightarrow} (\vec{H}_1,\psi_1)$$

Then there exists $l' \geq l$, $\vec{J}$,θ such that

$$(\vec{G},\phi) \overset{1:(l',0)}{\Longrightarrow} (\vec{J},\theta), (\vec{J},\theta) \overset{1:(-,-)}{\longrightarrow} (\vec{H}_1,\psi_1)$$

Proof: Omitted. It is in order for this lemma to be valid that the meta-level system must be standard. □
By a straight forward induction we hence obtain

Corollary 6.10
Let $(\vec{G},\phi) \overset{1:(-,-)}{\longrightarrow} (\vec{H},\psi), (\vec{H},\psi) \overset{1:(l,0)}{\Longrightarrow} (\vec{H}_1,\psi_1)$
Then there exists $l' \geq l$,$\vec{J}$,θ such that

$$(\vec{G},\phi) \overset{1:(l',0)}{\Longrightarrow} (\vec{J},\theta), (\vec{J},\theta) \overset{1:(-,-)}{\longrightarrow} (\vec{H}_1,\psi_1)$$

Now we can show our second "hoped-for" theorem:

Theorem 6.11 *Let a standard meta-level system be given. Suppose*

$$(\vec{G},\phi) \overset{1:(l,-)}{\longrightarrow} (\vec{H},\psi)$$

Then there exists $l' \geq l$,$\vec{J}$,θ such that

$$(\vec{G},\phi) \overset{1:(l',0)}{\Longrightarrow} (\vec{J},\theta)(\vec{J},\theta) \overset{1:(-,-)}{\longrightarrow} (\vec{H},\psi)$$

The above also holds when $\longrightarrow$ is replaced by $\Longrightarrow$ in the premise, in which case it also is necessary to replace $\longrightarrow$ by $\Longrightarrow$ in the conclusion.

Proof: Induction in the derivation tree. The case where the and-rule (15) is applied is covered by lemma 6.8. □

7 Work in progress

The work described so far is currently being extended along two lines:

1. As mentioned in section 2, only a restricted use of folding can be modeled by the (current) meta-level framework; and though most examples of applications of the fold/unfold framework fit into this scheme, it is certainly desirable to incorporate folding in its full generality. To do so, roughly the following steps are needed:

   - A "fold-rule" (applicable at all levels) must be defined. As we aim to ensure that "everything going on at level $i+1$ can happen at level i as well", the notion of folding has to be generalized a bit.
   - As "real evaluation" at level 1 only does unfolding, it must be shown that an evaluation tree at level 1 containing foldings can be replaced by an evaluation tree without folding steps and with "suitably related" costs. This is essentially some variant of the Church-Rosser property.

   We certainly can expect theorem 4.5 to hold still. Concerning termination properties, it may of course now very well happen that these are violated, due to "too much folding". Intuitively, to avoid this any meta-rule must contain at least one unfolding against a "progress rule", a progress rule being a rule not folded against.

2. A similar framework should certainly be set up for a functional language. It is of course always possible to translate a functional program into a logic equivalent, but laziness will *not* be modeled: to see this, suppose that f and g are two (unary) functions, and that the value of g does not depend on its argument. Then the composite function application *g(f(x))* will be expressed as the goal conjunction *f(X,Y),g(Y,Z)*. According to a *lazy* semantics for the functional language, *f* will not be evaluated. On the other hand, a logic evaluator *a priori* has to evaluate the goal *f(X,Y)* - even if it has been specified that the only interesting variable is Z - since this goal might fail, thus making the whole conjunction fail. In order to refrain from evaluating *f*, the logic evaluator must ensure that *f* has one (and only one) solution, requiring some quite sophisticated analysis.

   3 different kinds of semantics seem natural for "functional meta-level systems":

   - (Standard) evaluation is strict, transformation is strict: Then the system fits into our logic framework, and the same results (eg. at most constant speed up) hold. An example of such a system is SIMILIX[BonDan90] where **let**-expressions are inserted during transformation to guarantee that argument expressions are neither thrown away nor duplicated by unfolding.

- Evaluation is strict, transformation is (partially) lazy: Then transformation is able to reduce execution time by an order of magnitude, as sketched in section 5.1 (but also to augment it, if transformation is not fully lazy but only call-by-name, since expressions may be duplicated).
- Evaluation is lazy, transformation is lazy: then we can expect to show that transformation only speeds up by at most a constant, but as mentioned this requires a substantially new framework.

8 Conclusion

- A framework has been set up, general enough to incorporate concepts like partial evaluation, eureka-definition based program transformation, memoization etc. - in short, all sort of things built on unfolding, (limited use of) folding and memoization only. The basic deficiency is that it does not account for "strong" transformation techniques, due to the fact that all levels in some sense are isomorphic.
- As already mentioned, an analogue to the result proven in section 6 is presented in [Sch80] - he is also able to prove preservation of termination properties under some very weak conditions. We think it simplifies matters considerably (though the reader may disagree!) to work in a logical setting.
- A complexity measure for a given transition was given. A realistic measure should also attempt to estimate the complexity of a given unification; this, however, seems very hard to do.
- The framework could be extended to cope with parallelism, as mentioned in section 3. On the other hand, as long as the same kind of parallelism is allowed at all levels, essentially the same theorems can be obtained.
- In practice it is often a rather slow business to produce meta-rules. This is due to the interpretation overhead typically present; the amount of bookkeeping necessary; the choices between various things to do (eg. to unfold or not to unfold), etc. However, as our results concerning meta-rule generation time (eg. theorem 4.7) are of negative character they will still hold even if this is taken into account.

References

[Amt91]
Torben Amtoft Hansen: *Properties of Unfolding-based Meta-level Systems.* DAIMI-PB 348, March 1991, Computer Science Department, Aarhus University.

[BKPR89] F.S. de Boer, J.N. Kok, C. Palamidessi and J.J.M.M. Rutten: *From Failure to Success: Comparing a Denotational and a Declarative Semantics for Horn Clause Logic.* Technical Report CS-R89, Centre for Mathematics and Computer Science, Amsterdam, 1989

[BonDan90] Anders Bondorf, Olivier Danvy: *Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data types.* Technical Report no. 90-4, DIKU, University of Copenhagen, Denmark

[BurDar77] R.M.Burstall, J. Darlington: *A Transformation System for Developing Recursive Programs.* Jour. of the ACM, January 1977, vol. 24, no. 1, pp. 44-67.

[Fra85] Gudmund Frandsen: *A Denotational Semantics for Logic Programming.* DAIMI-PB 201, November 1985, Computer Science Department, Aarhus University

[Pal89] Catuscia Palamidessi: *Algebraic Properties of Idempotent Substitutions.* Technical Report TR-33/89, University of Pisa, 1989.

[PetBur82] Alberto Pettorossi, R.M. Burstall: *Deriving very Efficient Algorithms for Evaluating Linear Recurrence Relations Using the Program Transformation Technique.* Acta Informatica, vol. 18, 1982, pp. 181-206.

[Sch80] W.L. Scherlis: *Expression Procedures and Program Derivation.* PhD thesis, Stanford University, August 1980. Computer Science Report STAN-CS-80-818