The Meta Transformation Tool for Skeleton-Based Languages

Marco Aldinucci

Computer Science Dept. – University of Pisa – Corso Italia, 40 – I-56125 Pisa – Italy
Email: aldinuc@di.unipi.it

Abstract. Academic and commercial experience with skeleton-based systems has demonstrated the benefits of the approach but also the lack of methods and tools for algorithm design and performance prediction. We propose a (graphical) transformation tool based on a novel internal representation of programs that enables the user to effectively deal with program transformation. Given a skeleton-based language and a set of semantic-preserving transformation rules, the tool locates applicable transformations and provides performance estimates, thereby helping the programmer in navigating through the program refinement space.

1 Introduction

Structured parallel programming systems allow a parallel application to be constructed by composing a set of basic parallel patterns called *skeletons*. A skeleton is formally an higher order function taking one or more other skeletons or portions of sequential code as parameters, and modelling a parallel computation out of them.

Cole introduces the skeleton concept in the late 80's [8]. Cole's skeletons represent parallelism exploitation patterns that can be used (instantiated) to model common parallel applications. Later, different authors acknowledge that skeletons can be used as constructs of an explicitly parallel programming language, actually as the only way to express parallel computations in these languages [9, 5]. Recently, the skeleton concept evolved, and became the coordination layer of structured parallel programming environments [4, 6, 17].

Usually, the skeletons' set includes both data parallel and task parallel patterns. Data parallel skeletons model computations in which different processes cooperate to compute a single data item, whereas task parallel skeletons model computations whose parallel activities come from the computation of different and independent data items.

In most cases, in order to implement skeletons on parallel architectures efficiently, compiling tools based on the concept of *implementation template* (actually parametric process networks) have been developed [8, 5].

Furthermore, due to the fact that the skeletons have a clear functional and parallel semantics, different rewriting techniques have been developed that allow skeleton programs to be transformed/rewritten into equivalent ones achieving different performances when implemented on the target architecture [7, 11]. These transformations can also be driven by some kind of analytical performance models, associated with the implementation templates of the skeletons, in such a way that only those rewritings leading to efficient implementations of the skeleton code are considered.

The research community has been proposing several development frameworks based on the refinement of skeletons [9, 10, 18]. In such frameworks, the user starts by writing an initial skeletal program/specification. Afterwards, the initial specification may be subjected to a cost-driven transformation process with the aim of improving the performance of the parallel program. Such transformation is done by means of semantic preserving rewriting rules. A rich set of rewriting rules [1, 2, 3, 11] and cost models [18, 20] for various skeletons have been developed recently.

In this paper we present Meta, an interactive transformation tool for skeleton-based programs. The tool basically implements a term rewriting system that may be instantiated with a broad class of skeleton-based languages and skeleton rewriting rules. Basic features of the tool include the identification of applicable rules and the transformation of a subject program by application of a rule chosen by the user, or accordingly with some performance-driven heuristics. Meta is based on a novel program representation (called *dependence tree*) that allows to effectively implement a rewriting system via pattern-matching.

The Meta tool can be used as a building block in general transformational refinement environments for skeleton languages. Meta has already been used as transformation engine of the FAN skeleton framework [3, 10], that is a pure data parallel skeleton framework. Actually, Meta is more general and may be also used in a broad class of mixed task/data parallel skeleton languages [6, 17, 20].

The paper is organised as follows. Section 2 frames the kind of languages and transformations Meta can deal with. The Skel-BSP language, used as a test-bed for Meta, is presented. Section 3 describes the Meta transformation tool and its architecture. Then, Section 4 discusses a case study. Section 5 assesses some related work and concludes.

2 Skeletons and transformations

We consider a generic structured coordination language TL (for *target language*) where parallel programs are constructed by composing procedures in a conventional base language using a set of high-level pre-defined skeletons. We also assume that the skeletons set has three kinds of skeletons: *data parallel*, *task parallel* and *sequential* skeletons. Sequential skeletons encapsulate functions written in any sequential base language and are not considered for parallel execution. The others provide typical task and data parallel patterns. Finally, we constrain data parallel skeletons to call only sequential skeletons. This is usually the case in real applications and it is satisfied by the existing skeleton languages [9, 4, 6, 3, 20, 17]. Applications written in this way have a (up to) three-tier structure shown in Fig. 1.

In order to preserve generality, Meta can be specialised with the TL syntax and its three skeleton sets. The only requirement we ask is that the above constraint on skeleton calls holds. This makes our work applicable to a variety of existing languages.

Besides a skeleton-based TL, the other ingredient of program refinement by transformation is a set of semantic preserving rewriting rules. A rule for TL is a pair $L \rightarrow R$, where L and R are fragments of TL programs with variables $\nu_0, \nu_1 \ldots$ ranging over TL types, acting as placeholder for any piece of program. We require that every variable occurring in R must occur also in L and that L is not a variable. Moreover, a variable may be constrained to assume a specified type or satisfy a specific property (e.g., we

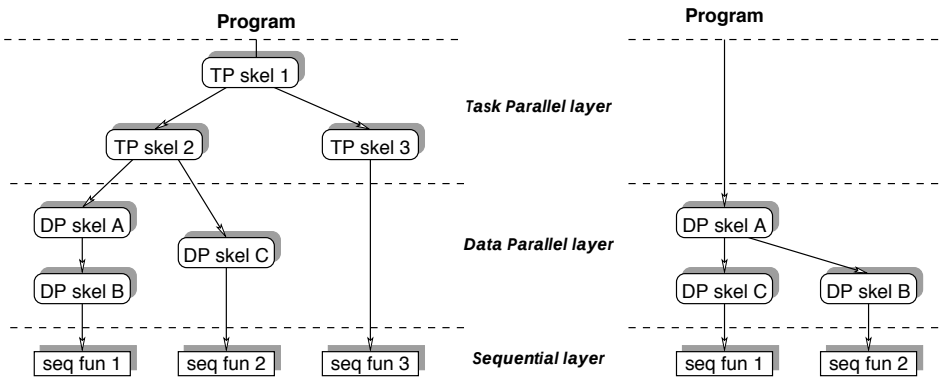


Fig. 1. Three-tier applications: two correct skeleton calling schemes.

may require an operator to distribute over another operator). The left-hand side L of a rule is called a *pattern*.

In the rest of the paper, we consider a simple concrete target language as a test-bed for the Meta tranformation tool: Skel-BSP[20]. Skel-BSP has been defined as a subset of P3L [5] on top of BSP [19] and it can express both data and task parallelism. The following defines a simplified Skel-BSP syntax which is particularly suitable for expressing rules and programs in a compact way:

TL_prog ::= *TP* | *DP*
TP ::= farm "(" *TP* ")" | pipe "{" *TPlist* "}" | *DP*
TPlist ::= *TP* | *TP*, *TPlist*
DP ::= map *Seq* | map2 *Seq* | scanL *Seq* | reduce *Seq* | *Seq* |
comp "(" out *Var*, in *Varlist* ")" "{" *DPlist* "}"
DPlist ::= *Var* "=" *DP Varlist* | *Var* "=" *DP Varlist*, *DPlist*
Var ::= ⟨ *a string* ⟩
Varlist ::= *Var* | *Var*, *Varlist*
Seq ::= ⟨ *a sequential C function* ⟩

TL_prog can be formed with skeleton applications, constants, variables or functions applications. Each skeleton instance may be further specified by its name just adding a dotted string after the keywords (e.g. comp.*mss*). Variables are specified by a name and by a type ranging over (all or some of) the base language types (e.g. all C types except pointers). Variable types may suppressed where no confusion can arise.

The pipe skeleton denotes functional composition where each function (stage) is executed in pipeline on a stream of data. Each stage of the pipe runs on different (sets of) processors. The farm skeleton denotes "stateless" functional replication on a stream of data. The map, scanL and reduce skeletons denote the namesake data parallel functions [7] and do not need any further comment. map2 is an extended version of map, which works on two arrays (of the same lengths) as follows: map2 $f\ [x_0, \ldots, x_n]\ [y_0, \ldots, y_n] = [f\ x_0\ y_0, \ldots, f\ x_n\ y_n]$. The comp skeleton expresses the sequential composition of data parallel skeletons. The body of the comp skeleton is a sequence of equations defining variables via expressions. Such definitions follows the *single-assignment* rule: there is at most one equation defining each variable.

comp.*name* (out *outvar*, in *invars*){
  $outvar_1$ = dp.*1* Op_1 $invars_1$
    ⋮
  $outvar_n$ = dp.*n* Op_n $invars_n$}

where: $\forall k = 1..n, invars_k \subseteq (\bigcup_{i<k} outvar_i \cup invars)$, $outvar \in \bigcup_{i \leq n} outvar_i$

The skeletons into the comp are executed in sequence on a single set of processors in a lock-step fashion, possibly with some (all-to-all) data re-distributions among steps. The cost estimate of Skel-BSP is based on the Valiant's Bulk-Syncronous Parallel model [19, 20].

2.1 Examples

In this section, we consider a couple of simple Skel-BSP programs: the maximum segment sum and the polynomial evaluation. Both programs are the Skel-BSP presentation of parallel algorithms appeared in [3, 10].

Maximum segment sum Given a one-dimensional array of integers v, the maximum segment sum (MSS) is a contiguous array segment whose members have the largest sum among all segments in v. Suppose we would like to compute the MSS of a stream of arrays. The following code is a first parallel program for computing MSS following a simple strategy [3, 10]:

```
pipe.mss {
  map pair,                  /* : int [n]    → int [n][2] */
  scanL Op+,                 /* : int [n][2] → int [n][2] */
  map P1,                    /* : int [n][2] → int [n]    */
  reduce max}                /* : int [n]    → int        */
```

The comments on the right hand side state the type of each skeleton instance; types are expressed using a C-like notation. The operator Op_+ is defined as follows:

$$[x_{i,1}, x_{i,2}]Op_+[x_{j,1}, x_{j,2}] = [max\{x_{i,1} + x_{j,2}, x_{j,1}\}, x_{i,2} + x_{j,2}]$$

while $pair\ x = [x, x]$ and $P_1\,[x_1, x_2] = x_1$. Intuitively, the purpose of scanL is to produce an array s whose ith element is the maximum sum of the segments of x ending at position i. Using a sequential program, this task can be accomplished simply by using scanL with operator $Op_1(a, b) = max(a + b, b)$. Unfortunately, such operator is not associative, thus this simple scanL cannot be parallelised. Op_+ uses an auxiliary variable to preserve the associativity. This variable is thrown away at the end of the scanL computation by the P_1 operator. Finally, reduce sorts out the maximum element of array s yielding to the desired maximum segment sum r.

Polynomial evaluation Let us consider the problem of evaluating in parallel a polynomial $a_1x + a_2x^2 + \ldots a_nx^n$ at m points $y_1, \ldots y_m$. The most intuitive solution consists in parallelising each basic step of the straightforward evaluation algorithm, i.e. first compute the vector of powers $ys^i = [y_1^i, \ldots, y_m^i], i = 1 \ldots n$, then multiply by

the coefficients, and, finally, sum up the intermediate results. The algorithm can be coded in Skel-BSP as follows.

```
comp.pol_eval (out zs, in ys, as) {
  ts = scanL * ys,                                  /* ts[i] = ys^i : float [n][m] */
  ds = map2 (*_sa) as, ts,                          /* ds[i] = [a_i * y_1^i, ..., a_i * y_m^i] : float [n][m] */
  zs = reduce + ds}                                 /* zs[i] = [sum_{i=1}^n a_i * y_1^i, ..., sum_{i=1}^n a_i * y_m^i] : float [m] */
```

where $*_{sa}$ multiplies each element of a vector by a scalar value, $*$ and $+$ are overloaded to work both on scalars and (element-wise) on vectors. On the right side (in comments) we describe the variable values and types.

2.2 Transformation rules

When we design a transformation system a foremost step is the choice of the rewriting rules to be included and the definition of their costs. The goal of the system is to derive a skeletal program with the best performance estimate by successive (semantic preserving) transformations (rewrites). Each transformation/rewrite correspond to the application of a rewriting rule. Here, we only collect the transformations needed to demonstrate the use of Meta on an example. We refer back to the literature for the proofs of the soundness of the rules [1, 2, 3, 7, 11]. For the sake of brevity, we use $L \rightleftarrows R$ to denote the couple of rules $L \to R$ and $R \to L$.

In the following, TSk_i can be any skeleton (task or data parallel, sequential), DSk_i can be any data parallel or sequential skeleton. $Op_1, Op_2, \ldots$ denote variables ranging over sequential functions. *pair* and P_1 are sequential auxiliary functions, defined in the previous section. The labelled elision $< \cdots >_n$ represents an unspecified chunk of code that appears (unchanged) in both sides of the rules.

farm insertion/elimination These rules state that farms can be removed or introduced on top of a TSk skeleton [2]. The rule preserves the constraint on layers since TSk cannot appear into a data parallel skeleton. A farm replicates TSk without changing the function it computes. Thus, it just increases task parallelism among different copies during execution.

$$\mathsf{TSk} \quad \underset{\leftarrow}{\overset{\rightarrow}{}} \quad \mathsf{farm}\ (\mathsf{TSk})$$

pipe → comp The pipe skeleton represents the functional composition for both task and data parallel skeletons. The comp models a (possibly) more complex interaction among data parallel skeletons. If all the stages DSk_1, $\mathsf{DSk}_2 \ldots$ of the pipe are data parallel (or sequential) skeletons, then the pipe can be rewritten as a comp in which each DSk_i gets its input from DSk_{i-1} and outputs towards DSk_{i+1} only. Also in this case the two formulations differ primarily in the parallel execution model. When arranged in a pipe, the DSk_1, $\mathsf{DSk}_2 \ldots$ are supposed to run on different sets of processors, while arranged in a comp, they are supposed to run (in sequence) on a single set of processors.

```
pipe {                                          comp (out z, in a) {
  DSk_1 Op_1,                                     b = DSk_1 Op_1 a,
  DSk_2 Op_2,                       ->            c = DSk_2 Op_2 b,
  < ... >_1                                       < ... >_1
  DSk_n Op_n}                                     z = DSk_n Op_n y}
```

map fusion/fission This rule denotes the map (backwards) distribution through functional composition [7]. Notice that when we apply form left-to-right we do not require the two maps in the left hand side to be adjacent in the program code. We just require that the input to the second one (q) is the output from the first one.

comp (out *outvar*, in *invars*) {
$< \cdots >_1$
q = map Op_1 p,
$< \cdots >_2$
r = map Op_2 q,
$< \cdots >_3$ }

$\rightarrow$
$\leftarrow$

comp (out *outvar*, in *invars*) {
$< \cdots >_1$
q = map Op_1 p,
r = map $(Op_2 \circ Op_1)$ p,
$< \cdots >_2$
$< \cdots >_3$ }

It is important to notice that, while rules are required to be *locally* correct, Meta ensures the *global* correctness of programs. For instance, using the rule from left-to-right (map fusion) the assignment in the grey box is not required to appear. Meta provides the program with the additional assignment (in the grey box) only if the intermediate result q is referenced in some expressions into $< \cdots >_2$ or $< \cdots >_3$.

SAR-ARA This rule (applied from left-to-right) aims to reduce the number of communications using the very complex operator Op_3. In general, the left-hand side is more communication intensive and less computation intensive than the right-hand side. The exact tradeoff for an advantageous application heavily depends on the cost calculus chosen (see [3, 10]).

comp (out *outvar*, in *invars*){
$< \cdots >_1$
q = scanL Op_1 p,
r = map P_1 q,
s = reduce Op_2 r,
$< \cdots >_2$ }

$\rightarrow$
$\leftarrow$

comp (out *outvar*, in *invars*){
$< \cdots >_1$
t = map $pair$ p,
u = reduce Op_3 t,
v = map P_1 u,
x = map P_1 v,
$< \cdots >_2$ }

Op_1 must distribute forward over Op_{aux}. Op_3 is defined as follows:

$$[x_{i,1}, x_{i,2}]Op_3[x_{j,1}, x_{j,2}] = [x_{i,1}Op_{\text{aux}}(x_{i,2}Op_1x_{j,1}), x_{i,2}Op_{\text{aux}}x_{j,2}]$$

$$[x_{i,1}, x_{i,2}]Op_{\text{aux}}[x_{j,1}, x_{j,2}] = [x_{i,1}Op_2x_{j,1}, x_{i,2}Op_2x_{j,2}]$$

Notice that, whereas operator Op_2 works on single elements, operators Op_1 and Op_{aux} are defined for pairs (arrays of length 2), and Op_3 works on pairs of pairs.

3 The transformation tool

In this section, we describe a transformation tool which allows the user to write, evaluate and transform TL programs, preserving their functional semantics, and possibly improving their performance. The tool is interactive. Given an initial TL algorithm, it proposes a set of transformation rules along with their expected performance impact. The programmer chooses a rule to be applied and successively (after the application) the tool looks for new matches. This process is iterated until the programmer deems the resulting program satisfactory, or there are no more applicable rules.

The strategy of program transformation is in charge of the programmer since, in general, the rewriting calculus of TL is not confluent: applying the same rules in a different order may lead to programs with different performance. The best transformation sequence may require a (potentially exponential) exhaustive search.

In the following, we define an abstract representation of TL programs and transformation rules, we describe the algorithm used for rule matching, and finally we sketch the structure of the tool.

3.1 Representing programs and rules

The Meta transformation system is basically a term-rewriting system. Both TL programs and transformation rules are represented by means of a novel data structure, so-called *dependence tree*. Dependence trees are basically labelled trees, thus the search for applicable rules reduces to the well established theory of subtree matching [12]. The tool attempts to annotate as many nodes of the tree representation as possible with a *matching rule instance*, i.e., a structure describing which rule can be used to transform the subtree rooted at the node, together with the information describing how the rule has been instantiated, the performance improvement expected and the applicability conditions to be checked (e.g., the distributivity of one operator over another).

The dependence tree is essentially an abstract syntax tree in which each non-leaf node represents a skeleton, with sons representing the skeleton parameters that may in turn be skeletons or sequential functions. The leaves must be sequential functions, constants or the special node *Arg( )*. Unlike a parse tree, a dependence tree directly represents the data dependence among skeletons: if the skeleton Sk_1 directly uses data produced by another skeleton Sk_2, then they will compare as adjacent nodes in the dependence tree, irrespectively of their position in the parse tree. Each edge in the dependence tree represents the dependence of the head node from the data produced by the tail node. The dependence tree of a program is defined constructively, combining information held in the parse tree (PT) and in the data flow graph (DFG) of the program. The algorithm to build dependence trees is shown in Table 1. The algorithm is illustrated in Fig. 2, which shows the parse tree, the data flow graph and the correspondent dependence tree of the polynomial evaluation example (see Sect. 2.1). The nodes labelled with *DPblock* mark the minimum subtrees containing at least one data parallel skeleton, nodes *Arg(as)* and *Arg(ys)* represent the input data of a *DPblock*. In other words, *DPblock* nodes delimit the border between the task parallel and the data parallel layers.

It is important to understand why we need to introduce a new data structure instead of using the parse tree directly. The main reason lies in the nature of the class of languages we aim to deal with, i.e. mixed task/data parallel languages. Nested skeleton calls find a very natural representation as trees. On the contrary, data parallel blocks based on the single-assignment rule (e.g. Skel-BSP comp) need a richer representation in order to catch the dependences among the skeletons (for example a data flow graph). The dependence tree enables us to compact all the information we need in a single tree, i.e. in a data structure on which we can do pattern-matching very efficiently.

There is one more point to address. The dependences shown in Fig. 2 are rather simple. In general, as shown in Fig. 3, a data structure produced by a single TL statement may be used by more than one statement in the rest of the program. We have

Table 1. Building up the dependence tree.

Input: PT and DFG for a correct TL program. The starting node x is the root of PT.
No nested *DPblock* are allowed (which can be easily flattened).
Output: The dependence tree DT.

Method:

1. Let x denote the current node, starting from the root of PT;
2. Copy x from PT on DT along with the arc joining it with its parent (if any),
   the arc is undirected as it comes from PT;
3. **if** not(x = *DPblock*)
4. **then** Recursively apply the algorithm to all sons of x in PT (in any order);
5. **else** Apply Procedure dpb(*DPblock*).

Procedure dpb($Node$):
a. From $Node$ follow backward the incoming edges in DFG;
b. **for each** node C_i reached in this way, **do**
c. Copy C_i from DFG to DT along with its out-coming edges;
d. Recursively apply dpb(C_i) until the starting node *DPblock* or a sink is reached;
In the former case add a node *Arg* to represent the formal parameter name.

two choices: (1) to keep a shared reference to the expression (tree), or (2) to replicate it. In option (1), the data flow can no longer be fully described by a tree. In addition, sharing the subtrees rules out the possibility of applying different transformations at the shared expression (tree) for different contexts. The Meta transformational engine adopts the second option, allowing us to map the data flow graph into a tree-shaped dependence structure. The drawback of replicating expressions is a possible explosion of the code size when we rebuild a TL program from the internal representation. To avoid this, the engine keeps track of all the replications made. This ensures a single copy of all replicated subtrees that have not been subject to an independent transformation.

Figure 4 depicts the internal representation of rule map fusion from Section 2.2. We represent the two sides of the rule as dependence trees, some leaves of which are variables represented by circled numbers. During the rule application, the instantiations of the left-hand side variables are substituted against their counterparts on the right-hand side. We call the set of circled figures a *rule interface*. Since in all our rules, the left-hand and the right-hand sides have the same variables occurring the same number of times, the interfaces of both rule sides are the same. Figure 4 demonstrates how the conditions of applicability and the performance of the two sides of a rule are reported to the programmer. Notice in Fig. 4 the "functional" fcomp, i.e. a special node used to specify rules in which two (or more) variables of the pattern are rewritten in the functional composition of them. Since variables have no sons, Meta first rewrites variables as sons of fcomp, then it makes the contractum $\{\nu_0 = f_0, \ldots \nu_n = f_n\}$ and, afterwards the result is equated using $\mathsf{fcomp}(f_0, \ldots f_n) = f_n \circ \cdots \circ f_0$.

3.2 Rule matching

Since programs and rules are represented by trees, we can state the problem of finding a candidate rule for transforming an expression as the well-known *subtree matching problem* [14, 16, 12]. In the most general case, given a pattern tree P and a subject tree

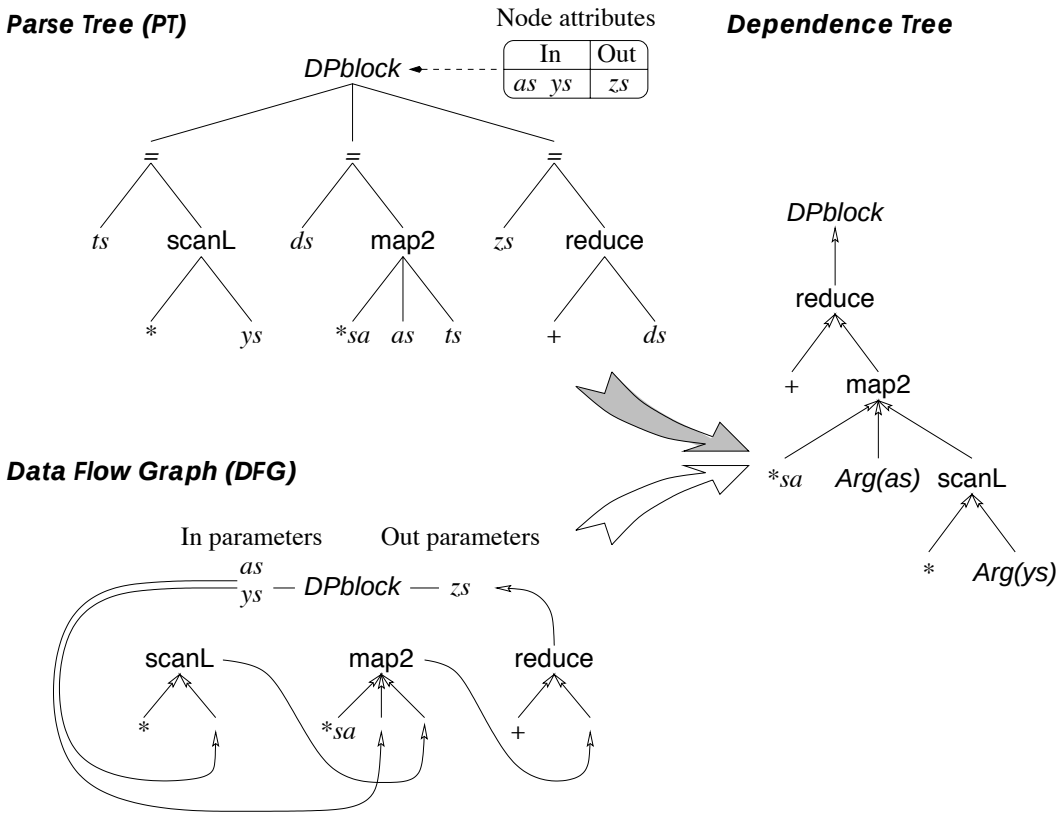


Fig. 2. The parse tree, the data flow graph and the dependence tree of polynomial evaluation. Skel-BSP skeletons are in serif font. Special nodes are in *slanted serif* font. Sequential functions are in *italic* font.

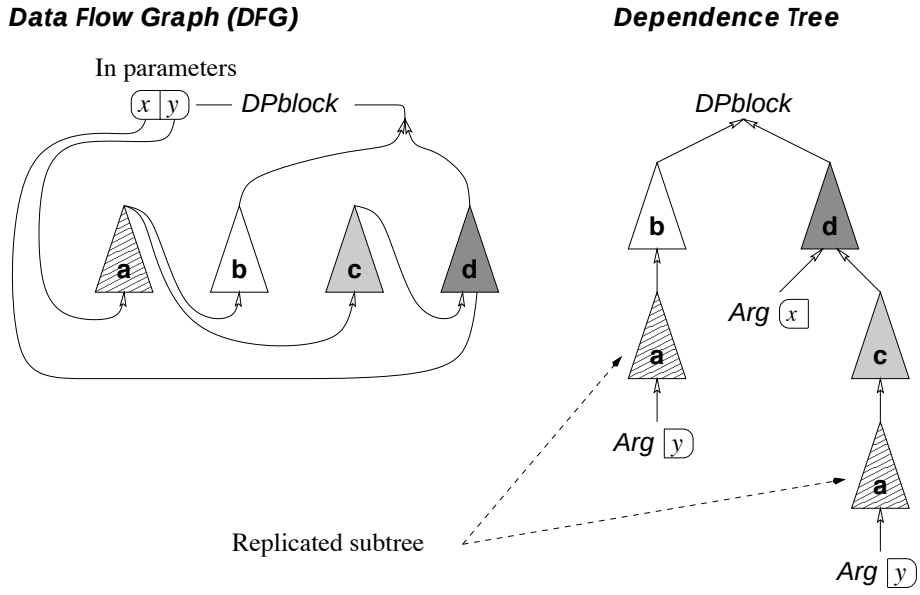


Fig. 3. Replicating shared trees. Each triangle stands for a tree representing a TL expression.

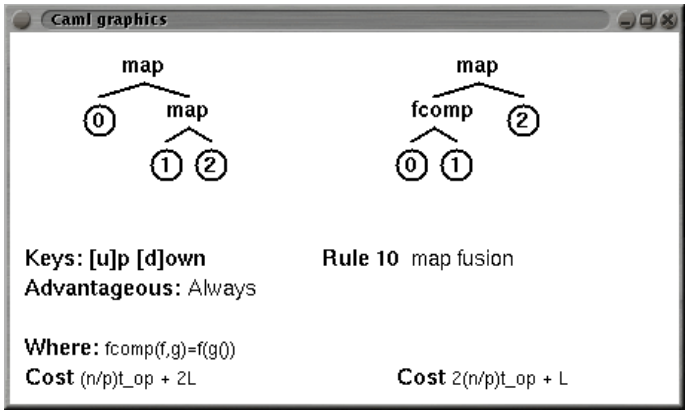


Fig. 4. Internal representation of rule map fusion, conditions of its applicability and performance of the two sides of the rule.

T, all occurrences of P as a subtree of T can be determined in time $\mathcal{O}(|P| + |T|)$ by applying a fast string matching algorithm to a proper string representation [16]. Our problem is a bit more specific: the same patterns are matched against many subjects and the subject may be modified incrementally by the sequence of rule applications. Therefore, we distinguish a *preprocessing phase*, involving operations on patterns independent of any subject tree, and a *matching phase*, involving all operations dependent on some subject tree. Minimising the matching time is our first priority.

The Hoffmann-O'Donnell bottom-up algorithm [12] fits our problem better than the string matching algorithm. With it, we can find all occurrences of a forest of patterns F as subtrees of T in time $\mathcal{O}(|T|)$, after suitable preprocessing of the pattern set. Moreover, the algorithm is efficient in practice: after the preprocessing, all the occurrences of elements in F can be found with a single traversal of T. The algorithm works in two steps: it constructs a *driving table*, which contains the patterns and their interrelations; then, the table is used to drive the matching algorithm.

The bottom-up matching algorithm We textually represent labelled trees as Σ-terms over a given alphabet Σ. Formally, all symbols in Σ are Σ-terms and if a is a q-ary symbol in Σ then $a(t_1, \ldots, t_q)$ is a Σ-term provided each of t_i is. Nothing else is a Σ-term. Let S_ν denote the set of $(\Sigma \cup \{\nu\})$-terms.

In addition, let $F = \{P_1, P_2, \ldots\}$ be a pattern set, where each pattern P_i is a tree. The set of all subtrees of the P_i is called a *pattern forest (PF)*. A subset M of PF is a match set for F if there exists a tree $t \in S_\nu$ such that every pattern in M matches t at the root and every pattern in $PF \setminus M$ does not match t at the root.

The key idea of the Hoffmann-O'Donnell bottom-up matching algorithm is to find, at each point (node) n in the subject tree, the set of all patterns and all parts of patterns which match at this point. Suppose n is a node labelled with the symbol b, and suppose also we have already computed such sets for each of the sons of n. Call these sets, from left to right $M_1, \ldots, M_q$. Then the set M of all pattern subtrees that match at n contains ν (that match anywhere), plus those patterns subtrees $b(t_1, \ldots, t_q)$ such that t_i is in M_i, $1 \leq i \leq q$. Therefore, we could compute M by forming $b(t_1, \ldots, t_q)$ for all combinations $(t_1, \ldots, t_q)$, $t_i \in M_i$. Once we have assigned these sets to each

node, we have essentially solved the matching problem, since each match is triggered by the presence of a complete pattern in some set.

Notice that there can be only finitely many such sets M, because both Σ and the set of sub-patterns are finite. Thus we could precompute these sets, and code them by some enumeration to build driving tables. Given such tables, the matching algorithm becomes straightforward: traverse the subject tree in postorder and assign to each node n the code of the set of partial matches as n. However, for certain pattern forest the number of such sets M (thus the complexity of the generation of driving tables) grows exponentially with the cardinality of the pattern set.

Nevertheless, there is a broad class of pattern sets which can be preprocessed in polynomial time/space in the size of the set: all sets yielding *simple pattern forests*. For an extensive treatment we refer back to Hoffmann-O'Donnell paper [12] and provide only a brief explanation here.

Let $a, b, c \ldots \in \Sigma$. Now, let P and P' be pattern trees. P *subsumes* P' if, for all subject trees T, P has a match in T implies that P' has a match in T. Then P is *inconsistent* with P' if there is no subject tree T matched by both P and P'. P and P' are *independent* if there exist T_1, T_2, and T_3 such that T_1 is matched by P but not by P', T_2 is matched by P' but not by P, and T_3 is matched by both P and P'. Given distinct patterns P and P', exactly one of the three previous relations must hold between them. A pattern forest is called *simple* if it contains no independent subtrees. For instance, the pattern forest including the pattern trees $P = a(b, \nu)$ and $P' = a(\nu, c)$ is not simple, since P and P' are independent with respect to $T_1 = a(b, b), T_2 = a(c, c), T_3 = a(b, c)$. On the contrary, the pattern set $F_s = \{a(a(\nu, \nu), b),\ a(b, \nu)\}$ lead to a simple pattern forest. The current set of Skel-BSP rules [1, 2, 20] and FAN rules [3] can be fully described by simple pattern forests. Simple pattern forests suffice even for the implementation of much more complex languages like LISP and the combinator calculus are based on simple pattern forests [13]. In addition, since the driving table depends only on the language and on the list of rules, it can be generated once and for all for a given set of rules and permanently stored for several subsequent match searches.

3.3 Tool architecture and implementation

The transformation engine applies the matching algorithm in an interactive cycle as follows:

1. Use the matching algorithm to annotate the dependence tree with the matching rules.
2. Check whether the rules found satisfy the type constraints and whether the side conditions hold (possibly interacting with the user).
3. Apply the performance estimates to establish the effect of each rule.
4. Ask the programmer to select one rule for application. In case no rule is applied, terminate; otherwise start again from Step 1.

We envision the Meta tool as a part of a general tool implementing a transformational refinement framework for a given target language TL. The global tool structure is depicted in Fig. 5 (the part already implemented is highlighted with a dotted box). The whole system has two main capabilities: the conversion between TL programs

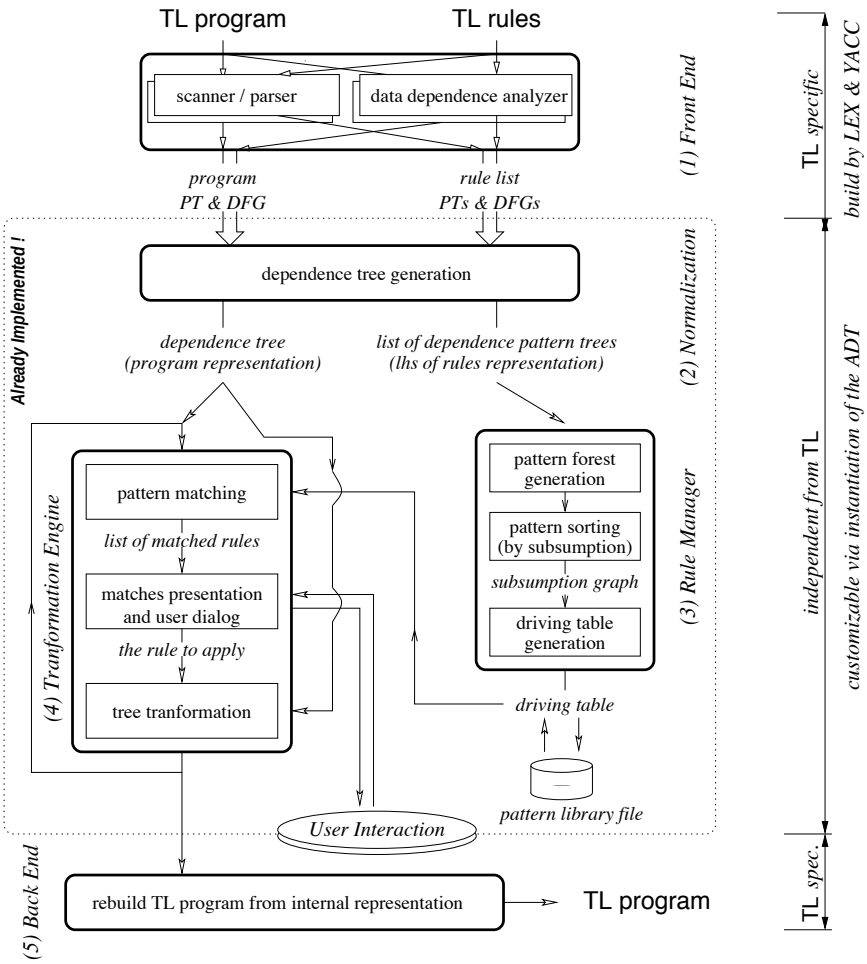


Fig. 5. Global structure of the Meta transformation system.

and their internal representation (dependence tree) and the transformation engine working on dependence trees.

The system architecture is divided into five basic blocks:

1. The *Front End* converts a TL program into a parse tree and a data flow graph.
2. The *Normalisation* uses the PT and DFG to build the dependence tree both for the TL program and for the set of transformation rules.
3. The *Rule Manager* implements the matching preprocessing; it delivers a matching table to drive the transformation engine. The driving table may be stored in a file.
4. The *Transformation Engine* interacts with the user and governs the transformation cycle.
5. The *Back End* generates a new TL program from the internal representation.

A prototype of the system kernel (highlighted in Fig. 5 with a dotted box) has been implemented in Objective Caml 2.02. Our implementation is based on an abstract data type (ADT) which describes the internal representation (dependence tree) and the functions working on it. The implementation is very general and can handle, via instantiation of the ADT, different languages with the requirement that rules and programs are written in the same language. Moreover, since several execution

models and many cost calculi may be associated with the same language, any compositional way of describing program performance may be embedded in the tool by just instantiating the performance formulae of every construct. We call a cost calculus *compositional* if the performance of a language expression is either described by a function of its components or by a constant.

The Meta tranformation tool prototype is currently working under both Linux and Microsoft Windows. A graphical interface is implemented using the embedded OCaml graphics library.

4 A case study: design by transformation

We discuss how Meta can be used in the program design process for algorithm MSS, introduced in Section 2.1 and reported in the top-left corner of Table 2.

First, the tool displays the internal representation of the program (Fig. 6 (a)) and proposes 5 rules (Fig. 6 (b)). The first one is pipe→comp rule, the others are instances of the farm introduction rule. The four stages of the pipe use exactly the same data distribution, but since each stage use a different set of processors each stage has to scatter and gather each data item. Transforming the pipe in a comp (that use just one set of processors) would get rid of many unnecessary data re-distributions. Let us suppose the user chooses to apply pipe→comp, the resulting version is shown in Fig. 6 (c). Next, Meta proposes a couple of rules (Fig. 6 (d)): SAR-ARA to further reduce the number of communications into the comp, thus to optimise the program behaviour on a single data item, and farm introduction to enhance the parallelism among different data items of the stream. Both rules improve the performance of the program, let us suppose to choose the SAR-ARA (Fig. 6 (e)).

Then, the transformation process continues choosing (in sequence) map fusion rule (2 times) and farm introduction rule. The resulting program is only one of the more than twenty different formulations Meta is able to find applying the transformation rules to the initial program. Table 2 shows some of the semantic-equivalent formulations derivable.

5 Related work and conclusions

In this paper, we have discussed the design and the implementation of an interactive, graphical transformation tool for skeleton-based languages. The Meta tool is (indeed) language-independent and is easily customisable with a broad class of languages, rewriting rules and cost calculi.

The design of our transformation engine Meta was influenced by the PARAMAT system [15]. However, our approach differs in many aspects. First, our goal is the optimisation of high-level parallelism, rather than the parallelisation of low-level sequential codes. Second, we do not define (as PARAMAT does) any a priori "good" parallel structure, we rather try to facilitate the exploration of the solution space toward the best parallel structure.

Beyond the described features, Meta may be instantiated with a set pre-defined heuristics to work as semi-automatic optimisation tool. As an example Meta recognises Skel-BSP data-parallel-free programs and optimises them with a standard sequence of rewriting rules. Such program formulation (called normal form) is proved to be, under

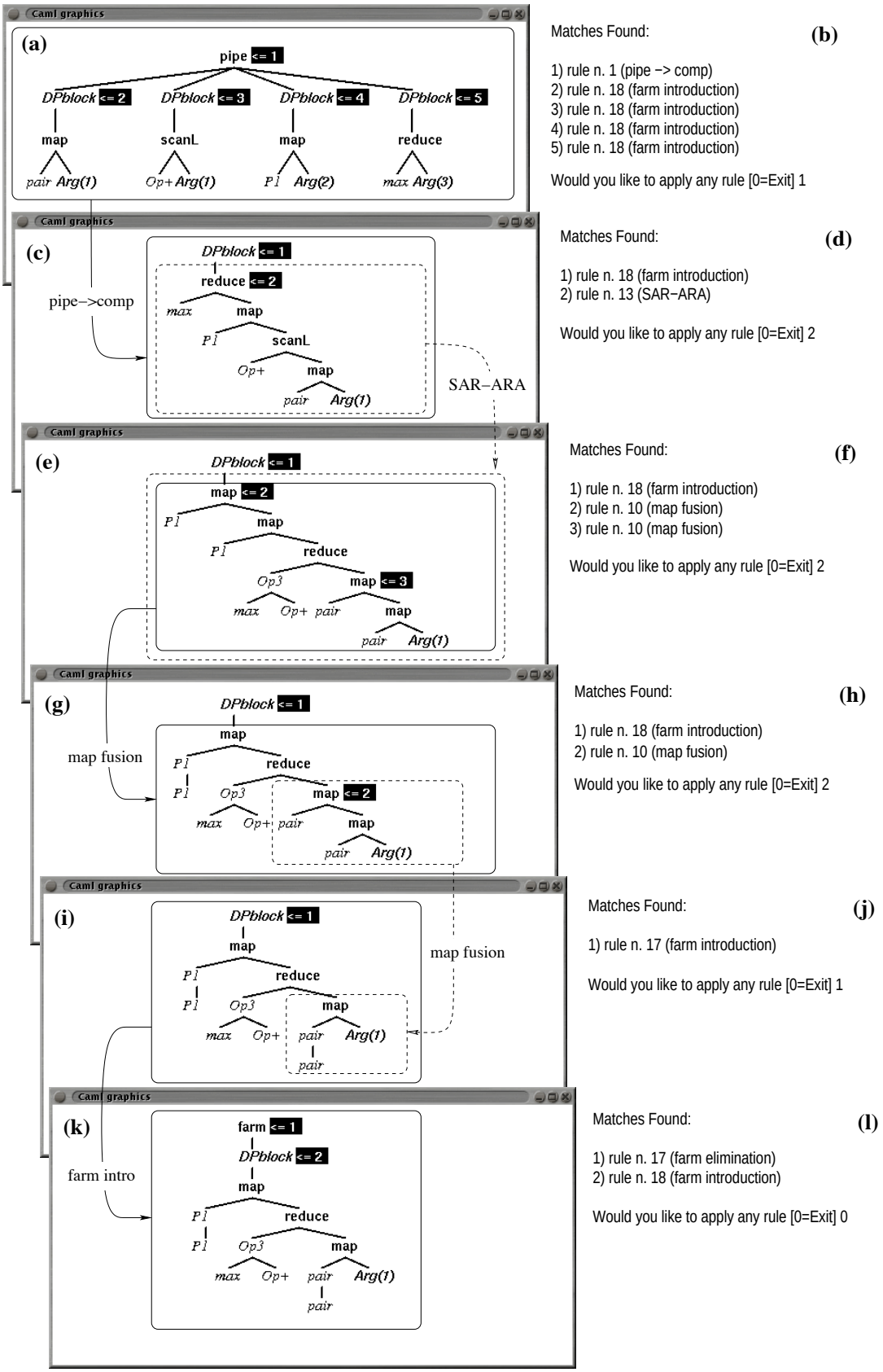
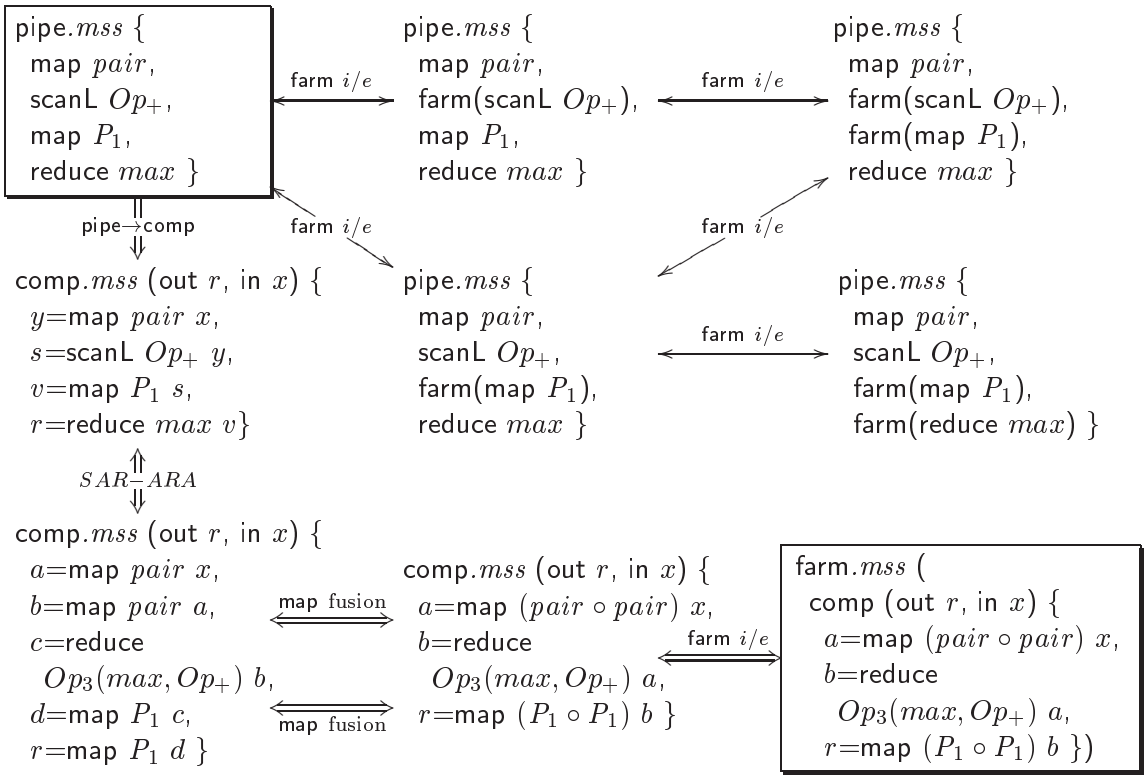


Fig. 6. Transformation of the MSS program using the Meta tool. Skel-BSP skeletons are in serif font. Special nodes are in *slanted serif* font. Sequential functions are in *italic* font.

Table 2. Some of the transformations proposed by Meta for the MSS example. The double-arrow path denotes the derivation path followed in Fig. 6.



mild requirements, the fastest among the semantic-equivalent formulations that can be reached (using the rewriting rules) [2].

Space limitation has prevented us to deal with performance prediction capabilities of Meta. It is clear that the accurateness of performance prediction made by Meta primarily depends on the accurateness of the target language cost calculus. The use of Meta with FAN has proved that in many cases good parallel programs can be obtained via transformations [3]. We are currently completing the integration of Meta with FAN and we plan to experiment it in the transformation of large real world application structures.

Acknowledgements I am very grateful to Sergei Gorlatch, Christian Lengauer and Susanna Pelagatti for many fruitful discussions. This work has been partially supported by a travel grant from the German-Italian exchange project VIGONI.

References

[1] M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In S. Gorlatch, editor, *CMPP'98: First International Workshop on Constructive Methods for Parallel Programming*, number MIP-9805 in University of Passau technical report, May 1998.

[2] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimization. In *proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems*, MIT, Boston, USA, November 1999. IASTED/ACTA press.

[3] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Towards parallel programming by transformation: The FAN skeleton framework. *Parallel Algorithms and Applications*, 2000. To appear.

[4] P. Au, J. Darlington, M. Ghanem, Y. Guo, H.W. To, and J. Yang. Co-ordinating heterogeneous parallel computation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proc. of Europar '96*, pages 601–614. Springer-Verlag, 1996.

[5] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[6] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: an heterogeneous HPC environment. *Parallel Computing*, 25(13–14):1827–1852, December 1999.

[7] R. S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computing Science*. NATO ASI Series, 1988.

[8] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[9] J. Darlington, A. J. Field, P.G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel Programming Using Skeleton Functions. In M. Reeve A. Bode and G. Wolf, editors, *PARLE'93 Parallel Architectures and Langauges Europe*. Springer Verlag, June 1993. LNCS No. 694.

[10] S. Gorlatch and S. Pelagatti. A transformational framework for skeletal programs: Overview and case study. In Jose Rohlim, editor, *Proc. of Parallel and Distributed Processing. Workshops held in Conjunction with IPPS/SPDP'99*, volume 1586 of *LNCS*, pages 123–137, Berlin, 1999. Springer.

[11] S. Gorlatch, C. Wedler, and C. Lengauer. Optimization rules for programming with collective operations. In *proceedings of 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing (IPPS/SPDP'99)*, IEEE Computer Society Press, pages 492–499, 1999.

[12] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[13] C. M. Hoffmann and M. J. O'Donnell. Interpreter generation using tree pattern matching. In *Conference Record of the sixth Annual ACM Symposium on Principles of Programming Languages (POPL'79)*, pages 169–179, New York, USA, January 1979. ACM Press.

[14] S. R. Kasaraju. Efficient tree pattern matching. In *Proceedings of the 30th IEEE Annual Symposium on Foundations of Computer Science*, pages 178–183, Research Triangle Park, North Carolina, 1989. IEEE Computer Society Press.

[15] C. W. Kessler. Pattern-driven automatic program transformation and parallelization. In *Proc. 3rd EUROMICRO Workshop on Parallel and Distributed Processing*. IEEE Computer Society Press, January 1995.

[16] E. Mäkinen. On the subtree isomorphism problem for ordered trees. *Information Processing Letters*, 32:271–273, September 1989.

[17] T. Rauber and G. Rünger. A coordination language for mixed task and data parallel programs. In *proceedings of 3rd Annual ACM Symposium on Applied Computing (SAC'99)*, pages 146–155. ACM Press, 1999.

[18] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.

[19] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–11, August 1990.

[20] A. Zavanella. *Skeletons and BSP: Performance portability for parallel programming*. PhD thesis, Computer Science Department, University of Pisa, Italy, December 1999.