Università di Pisa
Dipartimento di Informatica

Technical Report: TR-02-13

An operational semantics for skeletons

M. Aldinucci [a] M. Danelutto [a]

[a] Dept. of Computer Science, University of Pisa,
corso Italia, 40, I-56125 Pisa, Italy

July 19, 2002

ADDRESS: Corso Italia 40, 56125 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

An operational semantics for skeletons

M. Aldinucci [a] M. Danelutto [a]

[a] Dept. of Computer Science, University of Pisa,
corso Italia, 40, I-56125 Pisa, Italy

July 19, 2002

Abstract

In this work we present a schema of an operational semantics suitable for skeleton-based parallel languages that support both task and data parallelism. The semantics describes both functional and parallel behavior of the skeletal language in a uniform way by means of a labeled transition system. We use Lithium language (namely a Java skeleton framework) as the test-bed language to describe the methodology. The proposed schema can also be used to deal with skeletons with state.

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 modeling a parallel computation out of them. Cole introduces the skeleton concept in the late 80's [11]. Cole's skeletons represent parallelism exploitation patterns that can be used (instantiated) to model common parallel applications. Later, different authors acknowledged 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 [15, 7]. Recently, the skeleton concept evolved, and became the coordination layer of structured parallel programming environments [6, 8, 21]. Skeletons can be provided to the programmer either as language constructs [6, 8] or as libraries [12, 13, 18].

The compiling tools of the skeleton language or the skeleton libraries take care of automatically deriving/executing actual, efficient parallel code out of the skeleton application without any direct programmer intervention [20, 13, 18]. Different rewriting techniques have been developed, exploiting skeleton compositional semantics, that allow skeleton programs to be transformed/rewritten into equivalent ones achieving different, possibly better performances when implemented on the target architecture [10, 17]. These transformations can also be driven by some kind of analytical performance models, in such a way that only those rewritings leading to efficient implementations of the skeleton code are considered.

The research community proposed several development frameworks based on the refinement of skeletons [15, 16, 22]. In such frameworks, the user starts by writing an initial skeletal program/specification. Afterward, the initial specification is passed to a cost-driven transformation process with the aim of improving the performance of the parallel program. The transformations are carried out using semantic-preserving rewriting rules. A rich set of rewriting rules [2, 3, 5, 17] and cost models [22, 24] for various skeletons have recently been developed. On this ground, the automatic optimization by semantic-preserving rewriting of skeleton-based languages has been experimented [5, 1].

The formal description of a parallel, skeleton-based language involves at least two key issues:

1. the description of the input-output relationship of skeletons (namely the *functional* semantics);

2. the description of the parallel behavior of skeletons.

The significance of the first point is evident: there is no way to describe and to prove the correctness of (semantic-preserving) rewriting rules without some formal description of objects we are dealing with. The second point copes with a more quantitative aspect: the exact definition of the effective parallelism exploited by each skeleton. By using this parameter performance comparison among different but functionally equivalent skeletons can be performed. Almost all the frameworks previously cited adopt at least a functional formal description of the proposed language; such descriptions rely on the solid ground of sequential languages semantics.

With respect to parallel behavior, some skeleton classes have reached mature formal descriptions, whereas others haven't yet found a satisfactory formalization. There are two main skeleton classes, actually matching the two main paradigms in parallelism exploitation:

- *data parallelism*, which is exploited over several sub-parts of a single data item;

- *task parallelism*, modeling computations whose parallel activities come from the computation of different and independent data items (*stream parallelism* in the case data items come from a stream).

Data parallel skeletons can be elegantly described by the apply-to-all and other classic functional iterators (such as those proposed by Backus in [9] and used in the Bird-Meertens formalism [10]). On the contrary, task parallel skeletons (in particular stream parallel ones) have historically found no formal description both general and precise in performance modeling. In many cases, new skeletons have been introduced with only an informal definition of the exploited parallelism, giving to the application programmer a very intuitive description of the real skeleton behavior, but failing to give him a quantitative performance characterization. In other cases, performance models of stream parallel skeletons have been precisely developed for a given implementation. As an example, performance models for stream parallel skeleton class have been extensively studied assuming that each skeleton is implemented as a fixed topology process network [19, 20].

In this work we present an operational semantics for a skeleton-based language exploiting both data and stream parallel skeletons. The operational semantics describes both functional and parallel behavior of the skeleton language in a uniform and general way. The operational semantics is defined in term of a labeled transition system. We use Lithium language (namely a Java skeleton framework [14, 4]) as test-bed to describe the methodology.

The paper is organized as follows: Section 2 introduces Lithium skeleton framework, Section 3 discusses the operational semantics schema for Lithium, Section 4 outlines how labels can be used to properly handle Lithium parallel features. Eventually, Section 5 outlines how state can be handled using the proposed operational semantics.

2 Lithium skeletons

Lithium provides both task parallel and data parallel skeletons. All the skeletons work on streams, i.e. process a stream[1] of input tasks to produce a stream of results. All the skeletons are assumed to be stateless. No concept of "global state" is supported by the implementation, but the ones explicitly programmed by the user, possibly exploiting plain Java mechanisms (e.g. RMI servers encapsulating shared data structures, whose services/methods are invoked by code used to express sequential computations within skeletons). In particular, static (class) variables cannot be used in the definition of Lithium code to share information across different concurrent/parallel entities. The Lithium skeletons are fully nestable. Each skeleton has one or more skeleton parameters that model the computations encapsulated in the related parallelism exploitation pattern.

Lithium: a formal definition. Lithium extends the Java language. Lithium manages two new types in addition to Java types: streams and tuples. We denote streams by angled braces and tuples by curly braces:

$$
\begin{array}{lcl}
\mathit{value} & ::= & \mathit{A\ Java\ value\ (of\ a\ given\ type)} \\
\mathit{values} & ::= & \mathit{value} \mid \mathit{value}\ ,\ \mathit{values} \\
\mathit{stream} & ::= & \langle\ \mathit{values}\ \rangle \mid \langle\rangle \\
\mathit{tuple}_k & ::= & \{\ \mathit{stream}_1, \cdots, \mathit{stream}_k\ \}
\end{array}
$$

Streams represent sequences (finite or infinite) of values of the same type, whereas tuple is a parametric type that represents a (finite, ordered) set of streams. Actually, streams appearing in tuples are always *singleton streams*, i.e.

[1] Both finite and infinite streams, i.e. lists and non-strict lists, respectively

streams holding a single value[2].

Skeletons (Δ) provided by Lithium are defined as follows:

$$\Delta ::= \mathsf{seq}\ f \mid \mathsf{farm}\ \Delta \mid \mathsf{pipe}\ \Delta_1\ \Delta_2 \mid \mathsf{comp}\ \Delta_1\ \Delta_2 \mid \mathsf{map}\ f_{\mathsf{d}}\ \Delta\ f_{\mathsf{c}} \mid$$
$$\mathsf{d\&c}\ f_{\mathsf{tc}}\ f_{\mathsf{d}}\ \Delta\ f_{\mathsf{c}} \mid \mathsf{for}\ i\ \Delta \mid \mathsf{while}\ f_{\mathsf{tc}}\ \Delta \mid \mathsf{if}\ f_{\mathsf{tc}}\ \Delta_1\ \Delta_2$$

$$f, f_{\mathsf{c}}, f_{\mathsf{d}}, f_{\mathsf{tc}} ::= \textit{Sequential user-defined Java functions}^3$$

where sequential Java functions (f, g) with no index have type $Object \to Object$ (i.e. they are `Object f(Object x)` functions), and indexed functions $(f_{\mathsf{c}}, f_{\mathsf{d}}, f_{\mathsf{tc}})$ have the following types:

$f_{\mathsf{c}} : tuple_k \to stream$
$f_{\mathsf{d}} : stream \to tuple_k$
$f_{\mathsf{tc}} : stream \to boolean\ value$

In particular, $f_{\mathsf{c}}, f_{\mathsf{d}}$ represent families of functions that enable the splitting of a singleton stream in tuples of singleton streams and vice-versa, respectively. A pair of $f_{\mathsf{c}}, f_{\mathsf{d}}$ functions appearing in the same skeleton have to have the same value for the k parameter. As an example $f_{\mathsf{c}}, f_{\mathsf{d}}$ may be used to split an array in its rows/columns and join them back to the original array shape.

We consider a skeleton as a pre-defined higher-order function. The exact machinery used to declare a skeleton heavily depends on the programming framework (Java, in the Lithium case). Intuitively, seq skeleton just encapsulates sequential portions of code in such a way they can be used as parameters of other skeletons; farm and pipe skeletons model usual task farm (*alias* embarrassingly parallel) computations and computations organized in *stages*, respectively; comp models pipelines with stages executed serially (on the same processing element); map models data parallel computations: f_d decomposes the input data into a set of possibly overlapping data subsets, the inner skeleton computes a result out of each subset and the f_c function rebuilds a unique result out of these results; d&c models divide and conquer computations: input data is divided into subsets by f_d and each subset is computed recursively and concurrently until the f_{tc} condition does not hold true. At this point results of sub-computations are conquered (combined) via the f_c function. Finally for, while and if skeletons model finite iteration, indefinite iteration and conditional.

A skeleton applied to a stream is called a *skeletal expression* (expression for short). Expressions are defined as follows:

$$exp ::= \Delta\ stream \mid \Delta\ exp \mid \mathcal{R}_\ell\ exp \mid f_{\mathsf{c}}\ (\alpha\,\Delta)\ f_{\mathsf{d}}\ stream$$

Actually, the Lithium programmer may only write programs as expressions in the form Δ *stream*. In fact, a lithium program is a skeletal expression:

$$Lithium\_prog ::= \Delta : stream \to stream$$

and processes a stream of input data producing a stream of output results. Expressions such as the $\mathcal{R}_\ell\ exp$ and $f_{\mathsf{c}}\ (\alpha\,\Delta)\ f_{\mathsf{d}}\ stream$ will only appear in the operational semantics, as described in the Section 3, and will be used to deal

[2] We avoid to define a new sort for this type for the sake of simplicity.

with parallelism exploitation in the evaluation of Lithium programs.

3 Lithium operational semantics

We describe Lithium semantics by means of a labeled transition system. We define the label set as the string set augmented with the special label $\perp$ acting as the empty label:

$label ::= strings \cup \{\perp\}$

We rely on labels to distinguish both streams and transitions. Labeled streams are streams (as defined in the previous section) indexed by a label. Input and output stream are always $\perp$ labeled. Labels on transitions are used to enforce the serialization among some of all the possible transitions in the system. We shall extensively discuss on the labels rôle along the presentation of transition system rules in this Section and in Section 4.

The Lithium operational semantics is described in Figure 1. The rules of the Lithium semantics may be grouped in two main categories, corresponding at the two halves of the figure.

Rules 1–9 (on top of the figure) describe how skeletons behave with respect to a stream. All rules in this set spring from the following common schema:

$$\mathsf{Skel}\ params\ \langle x, \tau \rangle_{\ell_1} \xrightarrow{\ell_1} \mathcal{F}\langle x \rangle_{\ell_2} :: \ \mathsf{Skel}\ params\ \langle \tau \rangle_{\ell_3}$$

where $\mathsf{Skel} \in [\mathsf{seq}, \mathsf{farm}, \mathsf{pipe}, \mathsf{comp}, \mathsf{map}, \mathsf{d\&c}, \mathsf{for}, \mathsf{while}, \mathsf{if}]$, and the infix stream constructor $\langle \sigma \rangle_{\ell_i} :: \langle \tau \rangle_{\ell_j}$ is a non strict operator that sequences skeletal expressions. In general, $\mathcal{F}$ is a function of the skeletons appearing in the *params* list. Furthermore, for each rule of this kind, a *twin rule* matching singleton streams is defined (not shown in the figure for the sake of clearness):

$$\mathsf{Skel}\ params\ \langle x \rangle_{\ell_1} \xrightarrow{\ell_1} \mathcal{F}\langle x \rangle_{\ell_2}$$

Each pair of rules manages a stream as follows: the stream is unfolded in a sequence of singleton streams, and the nested skeleton is then applied to each item in the sequence. Eventually resulting skeletal expressions are joined back by means of the :: operator. During the unfolding, singleton streams are labeled accordingly to the particular rule policy, while the transition is labeled with the label of the stream before the transition (in this case ℓ_1).

As already stated, there are two basic aspects in the semantics: functional behavior and parallel behavior. To describe the parallel behavior we introduce (in abstract way) the concept of processing element. In each step of the computation, the only activities that may proceed in parallel are those mapped on different processing elements. Processing elements are identified by labels. With regards to streams, data mapping is abstracted by streams labels, while computations mapping is abstracted by transitions labels. As an example, a stream

1. $\mathsf{seq}\ f\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \langle f\,x\rangle_\ell :: \mathsf{seq}\ f\ \langle\tau\rangle_\ell$
2. $\mathsf{farm}\ \Delta\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \Delta\ \langle x\rangle_{\mathcal{O}(\ell,\mathrm{x})} :: \mathsf{farm}\ \Delta\ \langle\tau\rangle_{\mathcal{O}(\ell,\mathrm{x})}$
3. $\mathsf{pipe}\ \Delta_1\ \Delta_2\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \Delta_2\ \mathcal{R}_{\mathcal{O}(\ell,\mathrm{x})}\ \Delta_1\ \langle x\rangle_\ell :: \mathsf{pipe}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell$
4. $\mathsf{comp}\ \Delta_1\ \Delta_2\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \Delta_2\ \Delta_1\ \langle x\rangle_\ell :: \mathsf{comp}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell$
5. $\mathsf{map}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} f_\mathsf{c}\ (\alpha\ \Delta)\ f_\mathsf{d}\ \langle x\rangle_\ell :: \mathsf{map}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle\tau\rangle_\ell$
6. $\mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle x\rangle_\ell :: \mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle\tau\rangle_\ell$
   $$\mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}\ \langle y\rangle_\ell = \begin{cases} \Delta\ \langle y\rangle_\ell & \textit{iff}\ \ (f_\mathsf{tc}\ y) \\ f_\mathsf{c}\,(\alpha\,(\mathsf{d\&c}\ f_\mathsf{tc}\ f_\mathsf{c}\ \Delta\ f_\mathsf{d}))\ f_\mathsf{d}\ \langle y\rangle_\ell & \textit{otherwise} \end{cases}$$
7. $\mathsf{for}\ i\ \Delta\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \underbrace{\Delta(\Delta(\cdots(\Delta}_{i\ \text{times}}\langle x\rangle_\ell)\cdots)) :: \mathsf{for}\ i\ \Delta\ \langle\tau\rangle_\ell$
8. $\mathsf{while}\ f_\mathsf{tc}\ \Delta\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \begin{cases} \mathsf{while}\ f_\mathsf{tc}\ \Delta\ \ (\Delta\ \langle x\rangle_\ell :: \langle\tau\rangle_\ell) & \textit{iff}\ \ (f_\mathsf{tc}\ x) \\ \langle x\rangle_\ell ::\ \mathsf{while}\ f_\mathsf{tc}\ \Delta\ \langle\tau\rangle_\ell & \textit{otherwise} \end{cases}$
9. $\mathsf{if}\ f_\mathsf{tc}\ \Delta_1\ \Delta_2\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \begin{cases} \Delta_1\langle x\rangle_\ell :: \mathsf{if}\ f_\mathsf{tc}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell & \textit{iff}\ \ (f_\mathsf{tc}\ x) \\ \Delta_2\langle x\rangle_\ell :: \mathsf{if}\ f_\mathsf{tc}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell & \textit{otherwise} \end{cases}$

$$\langle\sigma\rangle_{\ell_1} :: \langle\tau\rangle_{\ell_2} \xrightarrow{\perp} \langle\sigma,\tau\rangle_\perp \quad \textit{join}$$

$$\frac{\Delta\ \langle x\rangle_{\ell_1} \xrightarrow{\ell_2} \langle y\rangle_{\ell_3}}{\mathcal{R}_\ell\ \Delta\langle x\rangle_{\ell_1} \xrightarrow{\ell_2} \langle y\rangle_\ell}\ \textit{relabel}$$

$$\frac{\Theta_1\langle x\rangle_{\ell_1} \xrightarrow{\ell} \Theta_2\langle y\rangle_{\ell_2}}{\Delta\ \Theta_1\langle x\rangle_{\ell_1} \xrightarrow{\ell} \Delta\ \Theta_2\langle y\rangle_{\ell_2}}\ \textit{context}$$

$$\frac{f_\mathsf{d}\,\langle x\rangle_\ell \xrightarrow{\ell} \{\langle y_1\rangle_\ell,\ \cdots\langle y_n\rangle_\ell\} \quad f_\mathsf{c}\,\{\langle z_1\rangle_\ell,\ \cdots\langle z_n\rangle_\ell\} \xrightarrow{\ell} \langle z\rangle_\ell \quad \Delta\langle y_i\rangle_\ell \xrightarrow{\ell} \langle z_i\rangle_\ell\ ,\ \forall i:\ 1\le\ i\le n}{f_\mathsf{c}\ (\alpha\Delta)\ f_\mathsf{d}\,\langle x\rangle_\ell \xrightarrow{\ell} \langle z\rangle_\ell}\ \textit{dp}$$

$$\frac{E_i \xrightarrow{\ell_i} E'_i \quad \forall i\ 1\le i\le n \quad \wedge \quad \exists i,j\ 1\le i,j\le n,\ \ell_i=\ell_j \Rightarrow i=j}{\langle\sigma\rangle_\perp :: E_1 :: \cdots E_n :: \Gamma \xrightarrow{\perp} \langle\sigma\rangle_\perp :: E'_1 :: \cdots E'_n :: \Gamma}\ \textit{eval}$$

Figure 1: Lithium operational semantics. $x,y\ \in\ \textit{value}$; $\sigma,\tau\in\ \textit{values}$; $E\in\ \textit{exp}$; $\Gamma\in\ \textit{exp}^*$; $\ell,\ell_i,\ldots\in\ \textit{label}$; $\mathcal{O}:\ \textit{label}\ \times\ \textit{value}\ \rightarrow\ \textit{label}$.

carryng the label 1 is local to the processing element 1, while a transition labeled with 2 represents a computation running on 2. The two concepts are not equivalent: at a given step, a data item may reside on a processing element waiting to be computed, or maybe a processing element may decide to bounce a data item to another one, thus relabeling the data item. We shall return back on pragmatic aspects of labeling schema in Section 4.

Let us show how rules 1–9 work with an example. We evaluate farm (seq f) on the input stream $\langle x_1, x_2, x_3, x_4, x_5 \rangle_\perp$. The first step is the following:

$$\mathsf{farm}\ (\mathsf{seq}\ f)\ \langle x_1, x_2, x_3, x_4, x_5 \rangle_\perp \xrightarrow{\perp} \mathsf{seq}\ f\ \langle x_1 \rangle_0 :: \mathsf{farm}\ (\mathsf{seq}\ f)\ \langle x_2, x_3, x_4, x_5 \rangle_0$$

The head of the stream has been separated from the rest and has been re-labeled (from $\perp$ to 0) accordingly with the $\mathcal{O}(\perp, x)$ function. Inner skeleton (seq) has been applied to such singleton stream, while the initial skeleton has been rewritten in a tail-recursive fashion. The $\mathcal{O}$ function (namely the *Oracle* function) has type $\mathcal{O} : \mathit{label} \times \mathit{value} \rightarrow \mathit{label}$, and it is basically a re-labeling function. The *Oracle* is an external function with respect to the labeled transition system. It would represent the (user-defined) policy used by a skeleton to label singleton streams, i.e. the data mapping policy. Let us suppose that the defined policy consists in distributing data items of the input stream round-robin onto three processing elements. An oracle function for this policy would cyclically return a label in a set of three fresh labels. In this case the repeated application of rule 1 (and its twin rule) proceeds as follows:

$$\begin{array}{l}\mathsf{seq}\ f\ \langle x_1 \rangle_0 :: \mathsf{farm}\ (\mathsf{seq}\ f)\ \langle x_2, x_3, x_4, x_5 \rangle_0 \xrightarrow{0}\xrightarrow{1}\xrightarrow{2}\xrightarrow{0} \\ \mathsf{seq}\ f\ \langle x_1 \rangle_0 :: \mathsf{seq}\ f\ \langle x_2 \rangle_1 :: \mathsf{seq}\ f\ \langle x_3 \rangle_2 :: \mathsf{seq}\ f\ \langle x_4 \rangle_0 :: \mathsf{seq}\ f\ \langle x_5 \rangle_1\end{array}$$

The oracle may have an internal state, and it may implement several policies of label transformation. As an example the oracle might always return a fresh label, or it might make decisions about the label to return on the basis of the x value. In case a new, fresh label is always returned by the oracle function, the semantics models the maximally parallel computation of the skeletal expression, as each different label is meant to represent a different processing element.

Observe that using only rules 1–9 (and their twins) the initial skeleton expression cannot be completely reduced (up to the output stream). Applied in all the possible ways, they lead to an aggregate of expressions *(exp)* glued by the :: operator. It is worth reminding that an expression may be a skeleton applied to a stream or the two *meta-skeletons*: $\mathcal{R}_\ell\ \mathit{exp}$ and $f_\mathsf{c}\ (\alpha\,\Delta)\ f_\mathsf{d}\ \mathit{stream}$, where the former abstracts a re-labeling function and the latter a data parallel application pattern.

The rest of the work is carried out by the five rules in the bottom half of figure 1. There are two main rules (*dp* and *eval*) and three auxiliary rules (*context*, *relabel* and *join*). Such rules define the order of reduction along aggregates of skeletal expressions. Let us describe each rule:

eval rule describes evaluation order of skeletal expressions[4] along sequences separated by ::. The meaning of the rule is the following: suppose that each skeletal expression in the sequence may be rewritten in another skeletal expression with a certain labeled transformation. Then all such skeletal expressions can be transformed in parallel, provided that they are adjacent, that the first expression of the sequence is a stream of values, and that all the transformation labels involved are pairwise different.

dp (data-parallel) rule describes the evaluation order for the $f_{\mathsf{c}}\ (\alpha\,\Delta)\ f_{\mathsf{d}}\ \mathit{stream}$ expression. Basically the rule creates a tuple by means of the f_{d} function, then requires the evaluation of all expressions composed by applying Δ onto all elements of the tuple. All such expression are evaluated in one step by the rule (apply-to-all). Finally, the f_{c} gathers all the elements of the evaluated tuple in a singleton stream. Labels are not an issue for this rule.

relabel rule provides a relabeling facility by evaluating the meta-skeleton $\mathcal{R}_\ell$. The rule does nothing but changing the stream label. Pragmatically, the rule imposes to a processing element to send the result of a computation to another processing element (together with the function to compute).

context rules establishes the evaluation order among nested expressions, in all other cases but the ones treated by *dp* and *relabel*. The rule imposes a strict evaluation of nested expressions (i.e. evaluated the arguments first). The rule leaves unchanged both transition and stream labels with respect to the nested expression.

join rule does the housekeeping work, joining back all the singleton streams of values to a single output stream of values.

Again, let us show how the rules work by means of a concrete example. Consider the semantics of farm (pipe f_1 f_2) evaluated on the input stream $\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7\rangle_\perp$. Let us suppose that the oracle function returns a label chosen in a set of two fresh labels. Pragmatically, since farm skeleton represent the replication paradigm and pipe skeleton the pipeline paradigm, the nested form farm (pipe f_1 f_2) basically matches the idea of a multiple pipeline (or a pipeline with multiple independent channels). The oracle function defines the parallelism degree of each paradigm: in our case two pipes, each having two stages. The initial expression would be initially unfolded by rule 2 and 3 (and twins):

farm (pipe (seq f_1) (seq f_2)) $\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7\rangle_\perp \ \longrightarrow^*$

pipe (seq f_1)(seq f_2)$\langle x_1\rangle_0$ :: pipe (seq f_1)(seq f_2)$\langle x_2\rangle_1$ :: pipe (seq f_1)(seq f_2)$\langle x_3\rangle_0$ ::
pipe (seq f_1)(seq f_2)$\langle x_4\rangle_1$ :: pipe (seq f_1)(seq f_2)$\langle x_5\rangle_0$ :: pipe (seq f_1)(seq f_2)$\langle x_6\rangle_1$ ::
pipe (seq f_1)(seq f_2)$\langle x_7\rangle_0 \longrightarrow^*$

seq $f_2\ \mathcal{R}_{02}$ seq $f1\ \langle x_1\rangle_0$ :: seq $f_2\ \mathcal{R}_{12}$ seq $f1\ \langle x_2\rangle_1$ :: seq $f_2\ \mathcal{R}_{02}$ seq $f1\ \langle x_3\rangle_0$ ::

[4] referred as "sequence" from now on

$\mathsf{seq}\ f_2\, \mathcal{R}_{12}\, \mathsf{seq}\ f1\, \langle x_4 \rangle_1 :: \mathsf{seq}\ f_2\, \mathcal{R}_{02}\, \mathsf{seq}\ f1\, \langle x_5 \rangle_0 :: \mathsf{seq}\ f_2\, \mathcal{R}_{12}\, \mathsf{seq}\ f1\, \langle x_6 \rangle_1 ::$
$\mathsf{seq}\ f_2\, \mathcal{R}_{02}\, \mathsf{seq}\ f1\, \langle x_7 \rangle_0$

At this point, we can reduce the formula using the *eval* rule. *eval* requires a sequence of adjacent expressions that can be reduced with differently labeled transitions. In this case we can find just two different labels $(0, 1)$, thus we apply *eval* to the leftmost pair of expressions:

$$\dfrac{\dfrac{\dfrac{\mathsf{seq}\ f_1\ \langle x_1 \rangle_0 \xrightarrow{0} \langle f_1\ x_1 \rangle_0}{\mathcal{R}_{02}\, \mathsf{seq}\ f_1\ \langle x_1 \rangle_0 \xrightarrow{0} \langle f_1\ x_1 \rangle_{02}}\ relabel}{\mathsf{seq}\ f_2\, \mathcal{R}_{02}\, \mathsf{seq}\ f_1\ \langle x_1 \rangle_0 \xrightarrow{0} \mathsf{seq}\ f_2\ \langle f_1\ x \rangle_{02}}\ context \quad \dfrac{\dfrac{\mathsf{seq}\ f_1\ \langle x_2 \rangle_1 \xrightarrow{1} \langle f_1\ x_2 \rangle_1}{\mathcal{R}_{12}\, \mathsf{seq}\ f_1\ \langle x_2 \rangle_1 \xrightarrow{1} \langle f_1\ x_2 \rangle_{12}}\ relabel}{\mathsf{seq}\ f_2\, \mathcal{R}_{12}\, \mathsf{seq}\ f_1\ \langle x_2 \rangle_1 \xrightarrow{1} \mathsf{seq}\ f_2\ \langle f_1\ x \rangle_{12}}\ context}{\begin{array}{l}\langle\rangle :: \mathsf{seq}\ f_2\, \mathcal{R}_{02}\, \mathsf{seq}\ f_1\ \langle x_1 \rangle_0 :: \mathsf{seq}\ f_2\, \mathcal{R}_{12}\, \mathsf{seq}\ f_1\ \langle x_2 \rangle_1 :: \mathsf{seq}\ f_2\, \mathcal{R}_{02}\, \mathsf{seq}\ f_1\ \langle x_3 \rangle_0 :: \cdots \xrightarrow{\perp} \\ \langle\rangle :: \mathsf{seq}\ f_2 \langle f_1\ x_1 \rangle_{02} :: \mathsf{seq}\ f_2\ \langle f_1\ x_2 \rangle_{12} :: \mathsf{seq}\ f_2\, \mathcal{R}_{02}\, \mathsf{seq}\ f_1\ \langle x_3 \rangle_0 :: \cdots\end{array}}\ eval$$

eval application on the first two expressions gives:

$\mathsf{seq}\ f_2\ \langle f_1\ x_1 \rangle_{02} :: \mathsf{seq}\ f_2\ \langle f_1\ x_2 \rangle_{12} ::$
$\mathsf{seq}\ f_2\, \mathcal{R}_{02}\, \mathsf{seq}\ f_1\ \langle x_3 \rangle_0 :: \mathsf{seq}\ f_2\, \mathcal{R}_{12}\, \mathsf{seq}\ f_1\ \langle x_4 \rangle_1 :: \mathsf{seq}\ f_2\, \mathcal{R}_{02}\, \mathsf{seq}\ f_1\ \langle x_5 \rangle_0 ::$
$\mathsf{seq}\ f_2\, \mathcal{R}_{12}\, \mathsf{seq}\ f_1\ \langle x_6 \rangle_1 :: \mathsf{seq}\ f_2\, \mathcal{R}_{02}\, \mathsf{seq}\ f_1\ \langle x_7 \rangle_0$

Observe that due to the previous transformation two new stream labels are appeared (02 and 12). Now, we can can repeat the application of *eval* rule as in the previous step. This time, it is possible to find four adjacent expression that can be rewrited with (pairwise) different labels $(0, 1, 02, 12)$. It is worth noticing that, even on a longer stream this labeling schema for the oracle function never produces more than four different labels, thus the maximum number of skeletal expressions reduced in one step by *eval* is four. Repeating the reasoning the we can completely reduce the formula to a sequence of singleton streams, that, in turn can be transformed by repeated application of *join* rule::

$$\begin{array}{l}\langle f_2\,(f_1\,x_1)\,\rangle_{02} :: \langle f_2\,(f_1\,x_2)\,\rangle_{12} :: \langle f_2\,(f_1\,x_3)\,\rangle_{02} :: \langle f_2\,(f_1\,x_4)\,\rangle_{12} :: \\ \langle f_2\,(f_1\,x_5)\,\rangle_{02} :: \langle f_2\,(f_1\,x_6)\,\rangle_{12} :: \langle f_2\,(f_1\,x_7)\,\rangle_{02} \xrightarrow{\perp} \\ \langle f_2\,(f_1\,x_1),\ f_2\,(f_1\,x_2)\,\rangle_{\perp} :: \langle f_2\,(f_1\,x_3)\,\rangle_{02} :: \langle f_2\,(f_1\,x_4)\,\rangle_{12} :: \\ \langle f_2\,(f_1\,x_5)\,\rangle_{0p} :: \langle f_2\,(f_1\,x_6)\,\rangle_{1p} :: \langle f_2\,(f_1\,x_7)\,\rangle_{0p} \to^* \\ \langle f_2\,(f_1\,x_1),\ f_2\,(f_1\,x_2)\,f_2\,(f_1\,x_3),\ f_2\,(f_1\,x_4)\,f_2\,(f_1\,x_5),\ f_2\,(f_1\,x_6),\ f_2\,(f_1\,x_7)\,\rangle_{\perp}\end{array}$$

and this is a stream of values labeled with $\perp$, i.e. the output stream.

Let us analyze the whole reduction process: from the initial expression to the final stream of values we applied three times the *eval* rule. In the first application of *eval* we transformed two skeletal expression in one step; in the second application four skeletal expressions have been involved; while in the last one just one expression has been reduced[5].

The succession of *eval* application in the transition system matches the expected behavior for the double pipeline paradigm. The first and last applications match pipeline start-up and end-up phases. As expected the parallelism exploited is reduced with respect to the steady state phase, that is matched by the second application. A longer input stream would rise the number of *eval* applications, in fact expanding the steady state phase of modeled system.

4 Parallelism and labels

The first relevant aspect of the semantics concerns the functional semantics. The functional semantics is independent of the labeling function. Changing the oracle function (i.e. how data and computations are distributed across the system) may change the number of transitions needed to reduce the input stream to the output stream, but it cannot change the output stream itself.

The second aspect concerns parallel behavior. Parallel behavior can be completely understood looking at the application of two rules: *dp* that disciplines data parallelism and *eval* that disciplines task parallelism on streams. Since all skeletons work on streams (included data parallel ones), the parallelism analysis starts by *eval*. Let us consider the synchronous case: In the *eval* rule only transitions with different labels (i.e. running on distinct processing elements) may evolve in the same step. The converse implication also holds: if two computations run on distinct processing elements they *might* be reduced with a single *eval* application. It is worth noticing that the result may be made stronger. The *eval* rule has a degree of freedom, in fact it does not require that the number of expressions to be reduced is the *maximum* (n, say), but any value in $[1, n]$. By enforcing the rule to pick the maximum possible number of expressions to be reduced, it is possible to rephrase the result as follows: if two computations run on distinct processing elements they *are* reduced with a single *eval* application.

dp acts as an apply-to-all on a tuple of data items. Such data items springs from a single task in the input stream by partitioning it using a user-defined function. The parallelism comes from the reduction of all elements in a tuple (actually singleton streams) in a single step.

At any time, the effective parallelism degree in the computation of a Lithium program can be (recursively) computed by considering the number of expressions that can be possibly reduced in parallel by *eval*. In case these expressions do not contain sub-terms that can be reduced by *dp*, this is the actual parallelism degree; otherwise the parallelism degree is given by the sum of the number of items appearing in the tuples generated by the reduction of these expressions via the *dp* rule.

Moreover, in synchronous implementations a direct mapping between time macro-step and implementation timing mechanism may be drawn. As an example time macro-step maps directly on BSP super-step concept.

[5]second and third *eval* application are not shown in the example

The labels-*eval* machinery subsumes a generic skeleton implementation: the input of skeleton program comes from a single entity (e.g. channel, cable, etc.) and at discrete time steps. To exploit parallelism on different tasks of the stream, tasks are spread out in processing elements following a given discipline. Labels trace tasks in their journey, and *eval* establishes that tasks with the same label, i.e. on the same processing element, cannot be computed in parallel. There are several important details about label management:

- the label set effectively used in a computation depends on the oracle function. It can be a statically fixed set or it can change cardinality during the evaluation. On this ground, a large class of implementations may be modeled.
- there are two kind of labels: on transformations (arrows) and on streams. Given one time macro-step, a transformation label points out which are the processing elements currently computing the task. Stream label gives a weaker information: it indicates what is currently the processing element which is assumed to elaborate the task. Actually the task may be relabeled before the elaboration, i.e. the task can be bounced from the candidate processing element towards another processing element.
- labels are assigned by the oracle function: the function transforms a label into another one using its own internal policy. Oracle function abstracts the mapping of data onto processing elements. It can be viewed as a parameter of the transition system used to model several policies in data mapping. As an example a farm may take items from the stream and spread them in a round-robin fashion to a pool of workers. Alternatively, the farm may manage the pool of workers by divination, always mapping a data task to a free worker. Such kind of policy may be used to establish an upper bound in the parallelism exploited.

5 Managing a state

Lithium language, as well as many others skeleton based languages, is proposed as a "pure" functional framework: all skeletons are assumed to be stateless. In practice, writing complex applications either a global or local (to the skeleton or to the processing element) state would remarkably reduce the programming complexity. A simple task as for example "counting the number of data items on a stream having a given property" turns out to be much more complex than expected (a place acting as accumulator, i.e. a state, would be useful in this case).

New generation skeleton frameworks such as Assist [23] explicitely includes a "global state" concept. In other skeleton frameworks the programmer is used to adopt more tricky solutions, as for example exploiting a persistent state locally to the processing element by means of static variable/class declarations although in this case the programmer must be aware of the underlying skeleton implementation in order to be sure that this trick actually works.

A discussion about the opportunity of introducing a state in a functional framework, as well as the effectiveness of different approaches to implement it, goes beyond our aims. However, we want to point out that the proposed operational semantics may effectively support both global and local state concept with just few adjustments. As labels can be used to represent the places where computations actually happen, they can also be used to model/drive global and local state implementation. In particular, they can be used to have a precise idea of where the subjects and objects of state operations (the process/thread issuing the state operation and the process(es)/thread(s) actually taking care of actually performing the operation) are running. This allows a precise modeling of the operations needed to implement skeleton state.

6 Conclusion

We proposed an operational semantics schema that models both functional semantics of the Lithium skeletons and parallel behavior in Lithium program execution. This schema is basically a labeled transition system, which is parametric with respect to a label oracle function. This oracle function basically provides fresh labels when required.

The system can be used to compute functional semantics of a Lithium skeleton program. In this case, labels are not significant. By taking into account labels, a precise modeling of the parallel beahvior of a Lithium program execution is achieved. In particular, analysis on the parallelism degree achieved in the computation can be performed, when proper oracle functions are used. We discussed how oracle functions can be designed that model the execution of Lithium programs with infinite or finite resources (processing elements). Last, we gave an idea of how the labeled transition system features can be exploited to model skeletons with state.

References

[1] M. Aldinucci. Automatic program transformation: The meta tool for skeleton-based languages. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, chapter 5, pages 59–78. Nova Science Publishers, NY, USA, 2002. (To appear. Draft copy: `ftp://ftp.di.unipi.it/pub/Papers/aldinuc/meta_book_dft.ps.gz`).

[2] 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.

[3] 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.

[4] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. to appear in Future Generation Computer Systems, North Holland, 2002.

[5] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Towards parallel programming by transformation: The FAN skeleton framework. *Parallel Algorithms and Applications, Gordon & Breach (Taylor & Francys group)*, 16(2–3):87–122, 2001.

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

[7] 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.

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

[9] J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Programming Style and its Algebra of Programs. *Communications of the ACM*, 21(8), August 1978.

[10] R. S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computing Science*. NATO ASI Series, 1988. International Summer School directed by F. L. Bauer, M. Broy, E. W. Dijkstra and C. A. R. Hoare.

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

[12] M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Parallel Functional Programming with Skeletons: the OCAMLP3L experiment. In *ACM Sigplan Workshop on ML*, pages 31–39, 1998.

[13] M. Danelutto and M. Stigliani. SKElib: parallel programming with skeletons in C. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 Parallel Processing*, LNCS, No. 1900, pages 1175–1184. Springer Verlag, August/September 2000.

[14] M. Danelutto and P. Teti. Lithium: a structured parallel programming environment in Java. In P. M. A. Sloot, C. J. Kenneth Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 844–853. Springer Verlag, April 2002.

[15] 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 A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93 Parallel Architectures and Langauges Europe*, volume 694 of *LNCS*. Springer-Verlag, June 1993.

[16] S. Gorlatch and S. Pelagatti. A transformational framework for skeletal programs: Overview and case study. In J. Rohlim, editor, *proceedings of Parallel and Distributed Processing. Workshop held in conjunction with IPPS/SPDP'99*, volume 1586 of *LNCS*, pages 123–137, Berlin, 1999. Springer-Verlag.

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

[18] H. Kuchen. A skeleton library. Technical Report 6/02-I, Angewandte Mathematik und Informatik, University of Munster, 2002.

[19] S. Pelagatti. A methodology for the development and the support of massively parallel programs. Technical Report TD-11/93, Dept. of Computer Science – Pisa, 1993. PhD Thesis.

[20] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor&Francis, 1998.

[21] 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.

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

[23] M. Vanneschi. ASSIST: an environment for parallel and distributed portable applications. Technical Report TR-02-07, Dept. of Computer Science, University of Pisa, May 2002.

[24] A. Zavanella. *Skeletons and BSP: Performance portability for parallel programming*. PhD thesis, Computer Science Department, University of Pisa, Italy, March 2000.