An operational semantics for skeletons[*]

M. Aldinucci[a] and M. Danelutto[b]

[a]Institute of Information Science and Technologies (ISTI) – National Research Council (CNR),
Via Moruzzi 1, I-56124 Pisa, Italy

[b]Department of Computer Science, University of Pisa,
Via Buonarroti 2, I-56127 Pisa, Italy

A major weakness of the current programming systems based on skeletons is that parallel semantics is usually provided in an informal way, thus preventing any formal comparison about program behavior. We describe a schema suitable for the description of both functional and parallel semantics of skeletal languages which is aimed at filling this gap. The proposed schema of semantics represents a handy framework to prove the correctness and validate different rewriting rules. These can be used to transform a skeleton program into a functionally equivalent but possibly faster version.

1. Introduction

Skeletons have been originally conceived by Cole [8] and then used by different research groups to design high-performance structured parallel programming environments [5,6,12]. A skeleton may be modeled as 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. A skeletal program is a composition of skeletons. Skeletons can be provided to the programmer either as language constructs [5–7] or as libraries [3,9,11].

The formal description of a parallel, skeletal language involves at least two key issues: *1)* the description of the input-output relationship of skeletons (*functional* semantics); *2)* the description of the parallel behavior of skeletons. The functional semantics enables the definition of semantics-preserving program transformations [2,4,1,10]. These transformations can also be driven by some kind of analytical performance models associated with skeletons [13], in such a way that only those rewritings leading to efficient implementations of the skeleton code are considered [2,1].

Almost all frameworks previously cited have a formal functional semantics. But none of them provide a complete and *uniform* description of the parallel semantics.

In this work we present a schema of operational semantics suitable for skeletal languages exploiting both data and stream parallel skeletons. The operational semantics is defined in term of a labeled transition system (LTS). It describes both functional and parallel behavior in a *uniform* and general way. We use a subset of the *Lithium* language as a test-bed to describe the methodology [3].

2. ***Lithium*** formal definition

Lithium extends the Java language by providing the programmer with both task parallel and data parallel skeletons [3] All the skeletons process a stream (finite or infinite) of input tasks to produce a stream of results. All the skeletons are assumed to be stateless, static variables are forbidden in *Lithium* code. No concept of "global state" is supported by the implementation, but the ones explicitly programmed by the user[2]. The *Lithium* skeletons are fully nestable. Each skeleton has one or more parameters that model the computations encapsulated in the related parallelism exploitation pattern. *Lithium* manages two new types in addition to Java types: streams and tuples that are

[*]This work has been partially supported by the Italian MIUR Strategic Project "legge 449/97" year 1999 No. 02.00470.ST97 and year 2000 No. 02.00640.ST97, and by the Italian MIUR FIRB Project *GRID.it* No. RBNE01KNFP.
[2]Such as RMI servers encapsulating shared data structures.

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. $\mathsf{while}\ f_{\mathsf{tc}}\ \Delta\ \langle x,\tau\rangle_\ell \xrightarrow{\ell} \begin{cases} \mathsf{while}\ f_{\mathsf{tc}}\ \Delta\ \ (\Delta\ \langle x\rangle_\ell :: \langle\tau\rangle_\ell) & \textit{iff}\ \ (f_{\mathsf{tc}}\ x) \\ \langle x\rangle_\ell ::\ \mathsf{while}\ f_{\mathsf{tc}}\ \Delta\ \langle\tau\rangle_\ell & \textit{otherwise} \end{cases}$

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

$$\frac{\Delta\ \langle x\rangle_{\ell_1} \xrightarrow{\ell_2} \langle y\rangle_{\ell_3}}{\mathcal{R}_\ell\ \Delta\langle x\rangle_{\ell_1} \xrightarrow{\ell_2} \langle y\rangle_\ell}\ \textit{relabel} \qquad \frac{E_1 \xrightarrow{\ell} E_2}{\Delta\ E_1 \xrightarrow{\ell} \Delta\ E_2}\ \textit{context}$$

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

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

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

denoted by angled braces and "$\lbag\ \rbag$" braces respectively:

$$\begin{array}{ll} \textit{value}\ ::= \textit{A Java value} & \textit{stream} ::= \langle\ \textit{values}\ \rangle \mid \langle\epsilon\rangle \\ \textit{values} ::= \textit{value} \mid \textit{value}\ ,\ \textit{values} & \textit{tuple}_k\ ::= \lbag\ \textit{stream}_1,\ \cdots,\ \textit{stream}_k\ \rbag \end{array}$$

A stream represents a sequence (finite or infinite) of values of the same type, whereas the tuple is a parametric type that represents a (finite, ordered) set of streams. Actually, streams appearing in tuples are always *singleton streams*, i.e. streams holding a single value. The set of skeletons (Δ) provided by *Lithium* are defined as follows:

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

where sequential Java functions (f, g) with no index have type *Object* $\rightarrow$ *Object*, and indexed functions $(f_{\mathsf{c}}, f_{\mathsf{d}}, f_{\mathsf{tc}})$ have the following types: f_{c} :*tuple*$_k$ $\rightarrow$ *stream*; f_{d} :*stream* $\rightarrow$ *tuple*$_k$; f_{tc} :*value* $\rightarrow$ *boolean.* In particular, $f_{\mathsf{c}}, f_{\mathsf{d}}$ represent families of functions that enable the splitting of a stream in k-tuples of singleton streams and vice-versa.

Lithium skeletons can be considered as a pre-defined higher-order functions. Intuitively, seq skeleton just integrates sequential Java code chunks within the structured parallel framework; farm and pipe skeletons model embarrassingly parallel and pipeline computations, respectively; comp models pipelines with stages serialized on the same processing element (PE); 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&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 via the f_c function. while skeleton model indefinite iteration. A skeleton applied to a stream is called a *skeletal expression*. Expressions are defined as follows: $exp ::= \Delta\ stream \mid \Delta\ exp \mid \mathcal{R}_\ell\ exp \mid f_\mathsf{c}\ (\alpha\,\Delta)\ f_\mathsf{d}\ stream$. The execution of a *Lithium* program consists in the evaluation of a $\Delta\ stream$ expression.

3. *Lithium* operational semantics

We describe *Lithium* semantics by means of a LTS. We define the *label* set as the string set augmented with the special label "$\perp$". We rely on labels to distinguish both streams and transitions. Input streams have no label, output stream are $\perp$ labeled. Labels on streams describe where data items are mapped within the system, while labels on transitions describe where they are computed. The *Lithium* operational semantics is described in Figure 1. The rules of the *Lithium* semantics may be grouped in two main categories, corresponding to the two halves of the figure. Rules 1–7 describe how skeletons behave with respect to a stream. These rules 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 $\mathsf{Skel} \in [\mathsf{seq}, \mathsf{farm}, \mathsf{pipe}, \mathsf{comp}, \mathsf{map}, \mathsf{d\&c}, \mathsf{while}]$, 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 appearing in the *params* list. For each of these rules a couple of *twin rules* exists (not shown in Figure 1):

$$\alpha)\ \mathsf{Skel}\ params\ \langle x, \tau\rangle \xrightarrow{\perp} \langle\epsilon\rangle_\perp ::\ \mathsf{Skel}\ params\ \langle x, \tau\rangle_\perp \qquad \omega)\ \mathsf{Skel}\ params\ \langle x\rangle_{\ell_1} \xrightarrow{\ell_1} \mathcal{F}\langle x\rangle_{\ell_2}$$

these rules manage the first and the last element of the stream respectively. Each triple of rules manages a stream as follows: the stream is first labeled by a rule of the kind α). Then the stream is unfolded in a sequence of singleton streams and the nested skeleton is applied to each item in the sequence. During the unfolding, singleton streams are labeled according to the particular rule policy, while the transition is labeled with the label of the stream before the transition (in this case ℓ_1). The last element of the stream is managed by a rule of the kind ω). Eventually resulting skeletal expressions are joined back by means of the :: operator.

Let us show how rules 1–7 work with an example. We evaluate farm (seq f) on the input stream $\langle x_1, x_2, x_3\rangle$. At the very beginning only the 2_α can be applied. It marks the begin of stream by introducing $\langle\epsilon\rangle_\perp$ (empty stream) and labels the input stream with $\perp$. Then rule 2 can be applied:

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

The head of the stream has been separated from the rest and has been re-labeled (from $\perp$ to 0) according to the $\mathcal{O}(\perp, x)$ function. Inner skeleton (seq) has been applied to this singleton stream, while the initial skeleton has been applied to the rest of the stream in a recursive fashion. The re-labeling function $\mathcal{O} : label \times value \to label$ (namely the *oracle*) is an external function with respect to the LTS. It would represent the (user-defined) data mapping policy. Let us adopt a two-PEs round-robin policy. An oracle function for this policy would cyclically return a label in a set of two labels. In this case the repeated application of rule 1 proceeds as follows:

$$\langle\epsilon\rangle_\perp :: \mathsf{seq}\ f\ \langle x_1\rangle_0 :: \mathsf{farm}\ (\mathsf{seq}\ f)\ \langle x_2, x_3\rangle_0 \xrightarrow{0}\xrightarrow{1} \langle\epsilon\rangle_\perp :: \mathsf{seq}\ f\ \langle x_1\rangle_0 :: \mathsf{seq}\ f\ \langle x_2\rangle_1 :: \mathsf{seq}\ f\ \langle x_3\rangle_0$$

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. As we shall see, in the former case the semantics models the maximally parallel computation of the skeletal expression.

Observe that using only rules 1–7 (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.

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

$$\mathsf{farm}\ (\mathsf{pipe}\ (\mathsf{seq}\ f_1)\ (\mathsf{seq}\ f_2))\ \langle x_1, x_2, x_3, x_4, x_5, x_6, x_7 \rangle \tag{1}$$

$$\langle \epsilon \rangle_\perp :: \mathsf{farm}\ (\mathsf{pipe}\ (\mathsf{seq}\ f_1)\ (\mathsf{seq}\ f_2))\ \langle x_1, x_2, x_3, x_4, x_5, x_6, x_7 \rangle_\perp \tag{2}$$

$$\begin{gathered}\langle \epsilon \rangle_\perp :: \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_1 \rangle_0 :: \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_2 \rangle_1 :: \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_3 \rangle_0 :: \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_4 \rangle_1 :: \\ \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_5 \rangle_0 :: \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_6 \rangle_1 :: \mathsf{pipe}\ (\mathsf{seq}\ f_1)(\mathsf{seq}\ f_2)\langle x_7 \rangle_0\end{gathered} \tag{3}$$

$$\begin{gathered}\langle \epsilon \rangle_\perp :: \mathsf{seq}\, f_2\, \mathcal{R}_{02}\, \mathsf{seq}\, f_1\, \langle x_1 \rangle_0 :: \mathsf{seq}\, f_2\, \mathcal{R}_{12}\, \mathsf{seq}\, f_1\, \langle x_2 \rangle_1 :: \mathsf{seq}\, f_2\, \mathcal{R}_{02}\, \mathsf{seq}\, f_1\, \langle x_3 \rangle_0 :: \mathsf{seq}\, f_2\, \mathcal{R}_{12}\, \mathsf{seq}\, f_1\, \langle x_4 \rangle_1 :: \\ \mathsf{seq}\, f_2\, \mathcal{R}_{02}\, \mathsf{seq}\, f_1\, \langle x_5 \rangle_0 :: \mathsf{seq}\, f_2\, \mathcal{R}_{12}\, \mathsf{seq}\, f_1\, \langle x_6 \rangle_1 :: \mathsf{seq}\, f_2\, \mathcal{R}_{02}\, \mathsf{seq}\, f_1\, \langle x_7 \rangle_0\end{gathered} \tag{4}$$

$$\begin{gathered}\langle \epsilon \rangle_\perp :: \mathsf{seq}\, f_2\, \langle f_1\, x_1 \rangle_{02} :: \mathsf{seq}\, f_2\, \langle f_1\, x_2 \rangle_{12} :: \mathsf{seq}\, f_2\, \mathcal{R}_{02}\, \mathsf{seq}\, f_1\, \langle x_3 \rangle_0 :: \mathsf{seq}\, f_2\, \mathcal{R}_{12}\, \mathsf{seq}\, f_1\, \langle x_4 \rangle_1 :: \\ \mathsf{seq}\, f_2\, \mathcal{R}_{02}\, \mathsf{seq}\, f_1\, \langle x_5 \rangle_0 :: \mathsf{seq}\, f_2\, \mathcal{R}_{12}\, \mathsf{seq}\, f_1\, \langle x_6 \rangle_1 :: \mathsf{seq}\, f_2\, \mathcal{R}_{02}\, \mathsf{seq}\, f_1\, \langle x_7 \rangle_0\end{gathered} \tag{5}$$

$$\langle \epsilon \rangle_\perp :: \langle f_2\,(f_1\,x_1)\,\rangle_{02} :: \langle f_2\,(f_1\,x_2)\,\rangle_{12} :: \langle f_2\,(f_1\,x_3)\,\rangle_{02} :: \langle f_2\,(f_1\,x_4)\,\rangle_{12} :: \langle f_2\,(f_1\,x_5)\,\rangle_{02} :: \langle f_2\,(f_1\,x_6)\,\rangle_{12} :: \langle f_2\,(f_1\,x_7)\,\rangle_{02} \tag{6}$$

$$\langle f_2\,(f_1\,x_1),\ f_2\,(f_1\,x_2)\,f_2\,(f_1\,x_3),\ f_2\,(f_1\,x_4)\,f_2\,(f_1\,x_5),\ f_2\,(f_1\,x_6),\ f_2\,(f_1\,x_7)\,\rangle_\perp \tag{7}$$

Figure 2. The semantic of a *Lithium* program: a complete example.

sp (stream parallel) rule describes evaluation order of skeletal expressions along sequences separated by :: operator. 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 $f_\mathsf{c}\ (\alpha\,\Delta)\ f_\mathsf{d}$ *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 PE to send the result of a computation to another PE (along 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. $join_\epsilon$ does the same work on the first element of the stream.

Let us consider a more complex example. Consider the semantics of $\mathsf{farm}\ (\mathsf{pipe}\ f_1\ f_2)$ evaluated on the input stream $\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7 \rangle$. Let us suppose that the oracle function returns a label chosen from a set of two labels. Pragmatically, since farm skeleton represent the replication paradigm and pipe skeleton the pipeline paradigm, the nested form $\mathsf{farm}\ (\mathsf{pipe}\ f_1\ f_2)$ 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. As shown in Figure 2, the initial expression is unfolded by rule 2_α ((1) $\rightarrow$ (2)) then reduced by many applications of rule 2 ((2) $\rightarrow^*$ (3)). Afterward the term can be rewritten by rule 3 ((3) $\rightarrow^*$ (4)). At this point, we can reduce the formula using the *sp* rule. *sp* 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 *sp* to the leftmost pair of the previuos expressions :

$$\langle\epsilon\rangle_\perp :: \mathsf{seq}\, f_2\, \mathcal{R}_{02}\, \mathsf{seq}\, f_1\, \langle x_1\rangle_0 :: \mathsf{seq}\, f_2\, \mathcal{R}_{12}\, \mathsf{seq}\, f_1\, \langle x_2\rangle_1 :: \mathsf{seq}\, f_2\, \mathcal{R}_{02}\, \mathsf{seq}\, f_1\, \langle x_3\rangle_0 :: \cdots \xrightarrow{\perp}$$
$$\langle\epsilon\rangle_\perp :: \mathsf{seq}\, f_2 \langle f_1\, x_1\rangle_{02} :: \mathsf{seq}\, f_2\, \langle f_1\, x_2\rangle_{12} :: \mathsf{seq}\, f_2\, \mathcal{R}_{02}\, \mathsf{seq}\, f_1\, \langle x_3\rangle_0 :: \cdots$$

Observe that due to the previous reduction ((4) $\rightarrow$ (5) in Figure 2) two new stream labels appear (02 and 12). Now, we can repeat the application of *sp* rule as in the previous step. This time, it is possible to find four adjacent expression that can be rewritten with (pairwise) different labels $(0, 1, 02, 12)$. Notice that even on a longer stream this oracle function never produces more than four different labels, thus the maximum number of skeletal expressions reduced in one step by *sp* is four. Repeating the reasoning the we can completely reduce the formula to a sequence of singleton streams ((5) $\rightarrow^*$ (6)), that, in turn can be transformed by many application of the *join* rule ((6) $\rightarrow^*$ (7)).

Let us analyze the whole reduction process: from the initial expression to the final stream of values we applied three times the *sp* rule. In the first application of *sp* 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[3].

The succession of *sp* applications 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 *sp* applications, in fact expanding the steady state phase of modeled system.

4. Parallelism and labels

The first relevant aspect of the proposed schema is that 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. It can be completely understood looking at the application of two rules: *dp* and *sp* that respectively control data and stream parallelism. We call the evaluation of either *dp* or *sp* rule a *par-step*. *dp* rule acts as an apply-to-all on a tuple of data items. Such data items are generated partitioning a single task of the stream by means of an user-defined function. The parallelism comes from the reduction of all elements in a tuple (actually singleton streams) in a single *par-step*. A single instance of the *sp* rule enable the parallel evolution of adjacent terms with different labels (i.e. computations running on distinct PEs). The converse implication also holds: many transitions of adjacent terms with different labels *might* be reduced with a single *sp* application. However, notice that *sp* rule may be applied in many different ways even to the same term. In particular the number of expressions reduced in a single *par-step* may vary from 1 to n (i.e. the maximum number of adjacent terms exploiting different labels). These different applications lead to both different proofs and parallel (but functional) semantics for the same term. This degree of freedom enables the language designer to define a (functionally confluent) family of semantics for the same language covering different aspects of the language. For example, it is possible to define the semantics exploiting the maximum available parallelism or the one that never uses more than k PEs.

At any time, the effective parallelism degree in the evaluation of a given term in a *par-step* can be counted by inducing in a structural way on the proof of the term. Parallelism degree in the conclusion of a rule is the sum of parallelism degrees of the transitions appearing in the premise (assuming one the parallelism degree in rules 1–7). The parallelism degree counting may be easily formalized in the LTS by using an additional label on transitions.

The LTS proof 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 PEs following a given discipline. Stream labels trace tasks in their journey, and *sp* establishes that tasks with the same label, i.e. on the same PE, cannot be computed in parallel. Labels are assigned by the oracle function by rewriting a label into another one using its own internal policy. The oracle abstracts the mapping of data onto PEs, and it can be viewed as a parameter of the transition system used to model several policies in data

[3]Second and third *sp* application are not shown in the example.

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). 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. Labels on transformations are derived from label on streams. Quite intuitively, a processing element must known a data item to elaborate it. The re-labeling mechanism enables to describe a data item re-mapping. In a *par-step*, transformation labels point out which are the PEs currently computing the task.

5. Conclusion

We propose an operational semantics schema that can be used to describe both functional and parallel behavior of skeletal programs in a uniform way. This schema is basically a LTS, which is parametric with respect to an oracle function. The oracle function provides the LTS with a configurable label generator that establishes a mapping between data and computation and system resources.

We use the *Lithium* – a skeletal language exploiting both task and data parallelism – as a test-bed for the semantics schema. We show how the semantics (built according to the schema) enables the analysis of several facets of *Lithium* programs such as: the description of functional semantics, the comparison in performance and resource usage between functionally equivalent programs, the analysis of maximum parallelism achievable with infinite or finite resources. To our knowledge, there is no other work discussing a semantics with the same features in parallel skeleton language framework (and, in general, in the structured parallel programming world as well).

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.

[2] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimization. In *Proc. of the 11th IASTED Intl. Conference on Parallel and Distributed Computing and Systems (PDCS'99)*, pages 955–962, Cambridge, Massachusetts, USA, November 1999. IASTED/ACTA press.

[3] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003.

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

[5] 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 Euro-Par 1996*, pages 601–614. Springer-Verlag, 1996.

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

[7] G. H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196(1–2):71–107, April 1998.

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

[9] M. Danelutto and M. Stigliani. SKElib: parallel programming with skeletons in C. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *Proc. of Euro-Par 2000*, number 1900 in LNCS, pages 1175–1184. Springer-Verlag, September 2000.

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

[11] H. Kuchen. A skeleton library. In B. Monien and R. Feldmann, editors, *Proc. of Euro-Par 2002*, number 2400 in LNCS, pages 620–629. Springer-Verlag, 2002.

[12] J. Sérot and D. Ginhac. Skeletons for parallel image processing: an overview of the SKIPPER project. *Parallel Computing*, 28(12):1685–1708, December 2002.

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