Skeleton based parallel programming: functional and parallel semantics in a single shot *

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

[a] *Institute of Information Science and Technologies (ISTI-CNR), Via Moruzzi 1, I-56124 Pisa, Italy*
[b] *Department of Computer Science, University of Pisa, Via Buonarroti 2, I-56127 Pisa, Italy*
{Marco.Aldinucci,Marco.Danelutto}@di.unipi.it

Abstract

Different skeleton based parallel programming systems have been developed in past years. The main goal of these programming environments is to provide programmers with handy, effective ways of writing parallel applications. In particular, skeleton based parallel programming environments automatically deal with most of the difficult, cumbersome programming problems that must be usually handled by programmers of parallel applications using traditional programming environments (e.g. environments such as those based on MPI). The semantics of the skeleton based programming environments is usually provided in two distinct items: a formal *functional semantics*, precisely modeling the function computed by the skeleton program, *and* an informal *parallel semantics* describing the ways used to exploit parallelism during the execution of a given skeleton program. The separation of functional and parallel semantics seriously impairs the possibility of programmers to use the semantic tools to prove properties of programs. In this work we show how a formal semantic framework can be set up that handles both functional and parallel aspects of skeleton based parallel programs. The framework is based on a labeled transition system. We show how different properties related to skeleton programs can be proved using such system. We use the Lithium skeleton based, full Java parallel programming environment as the case study.
Key words: algorithmical skeletons, structured parallel programming, labeled transition systems, functional semantics, parallel semantics.

*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.

1 Introduction

Parallel programming environments provide programmers ways of expressing concurrency/parallelism and communications and/or data sharing. In traditional environments the programmers either explicitly use such mechanisms to set up parallel application code (e.g. using MPI/PVM) or completely (or almost completely) rely upon the language compiler and run time to handle and exploit parallelism (e.g. using HPF).

Since the '90, several research groups proposed alternative programming environments, namely *structured* parallel programming environments. Those environments ask programmers to explicitly deal with the *qualitative* aspects of parallelism exploitation but rely on compiler tools and run time support to manage all the parallelism exploitation related aspects, that is concurrent activities set up, mapping and scheduling, communication management, etc.

Notable examples of this kind of environments are those based on the *algorithmical skeleton* concept. The concept of skeleton, that is a known, recurrent, efficient pattern of parallelism exploitation has been originally conceived by Cole [11]. Later on, it has been used by different research groups to design high-performance structured parallel programming environments [8, 9, 10, 26, 22, 20]. Other research groups, moving from the *parallel design pattern* concept, produced parallel programming environments with features very similar to those of skeleton based parallel programming environments [24].

Basically, structured parallel programming systems allow a parallel application to be coded by properly composing a set of basic parallel skeletons. These basic skeletons usually include skeletons modeling embarrassingly parallel computations (farms), computations structured in stages (pipelines) as well as common data parallel computation patterns (map/forall, reduce, scan). Each skeleton is parametric; in particular, it accepts as a parameter the kind of computation to be performed according to parallelism exploitation pattern it models. As an example, a farm skeleton takes as a parameter the worker, i.e. the computation to be performed on the single input task (data item). As a further example, a pipeline takes as parameters the pipeline stages. Such parameters may be either parameters modeling sequential portions of code (sequential skeletons) or even other skeletons, in turn. Therefore, a farm skeleton may take as a worker a two stage pipeline. The composition of the two expresses embarrassingly parallel computations where each input task (data item) is processed by two stages. Parallelism is exploited both by using different resources to compute independent input tasks and by using different resources to compute the first and the second stage onto a single input task.

The formal description of a parallel, skeletal language involves at least two key issues:

- the description of the input-output relationship of skeletons (*functional* semantics);
- the description of the parallel behavior of skeletons.

Almost all frameworks previously cited have a formal functional semantics. Typically, such semantics is given by providing a completely functional description of the skeletons used by means of a set of higher order functions. Also, some of them equips functional behaviour with parallel behaviour. These are mostly focused on description of data parallel computation, as for example in BSP style computations [22, 17, 23, 13]. Other works focused on task parallelism also [19], even if they hardly migth be equipped with a simple cost model.

As an example, using OCaml [2] as the functional formalism, farm and pipeline skeletons are usually described as follows:

```
let farm f x = (f x);;
let pipeline f g x = (g (f x));;
let stream_parallel f x::y =
    (f x)::(stream_parallel f y);;
```

This kind of formalization of skeleton functional semantics allows programmers to "compute" the function computed by a skeleton program. It also allows to reason about program equivalence, in terms of the results computed, or to define semantics-preserving program transformations [5, 7, 18]. These transformations can also be driven by some kind of analytical performance models associated with skeletons, in such a way that only those rewritings leading to efficient implementations of the skeleton code are considered [3, 12, 27, 16]. For instance, one can easily derive that two programs such as:

```
let progA f g = stream_parallel (pipeline f g);;
let progB f g = stream_parallel (farm (pipeline f g));;
```

actually compute the same results.

Quite often the parallel semantics of the skeletons (especially for task parallel ones) is given in an informal way. Somewhere, in the skeleton environment documentation, you can find a text telling you that:

> *The farm skeleton uses a set of independent processing elements to compute the input tasks. Each time a new input task is available one of these resources is selected for the execution of the task, possibly using some kind of auto scheduling policy. The pipeline skeleton uses independent processing elements to compute the different stages in such a way that computation of stage i relative to task j can proceed concurrently (in parallel) with both the computation of stage $i-1$ for task $j+1$ and the computation of stage $i+1$ for task $j-1$.*

From this, we can evince that `progA` exploits less parallelism that `progB`. In particular, the first program only computes in parallel the first and the second stage of the pipeline relative to the two different input tasks i and $i+1$. The second program, instead, computes in parallel stages relative to different tasks i_1 and i_1+1 to i_k and i_k+1 where k is the number of parallel workers set up for farm execution. However, a certain effort has to be made to reach this conclusion. In particular there is no formal tool that we can use to understand how the computation of such programs is performed in parallel.

In this work we present a schema of operational semantics suitable for skeletal languages exploiting both data and stream parallel skeletons, i.e. covering the more common and used parallelism exploitation patterns. The operational semantics is defined in terms of a labeled transition system (LTS). It describes both functional and parallel behavior in a *uniform* and general way. After presenting and discussing the operational semantics schema, we will show how it can be used to *compute* different properties of parallel programs that cannot be computed just using the functional semantics. In order to show the possibilities offered by the proposed operational semantics schema, we use a subset of the *Lithium* language as a test-bed [6]. Despite the fact we describe *Lithium* subset semantics here, the operational semantics schema proposed can be used to describe any other skeleton based parallel programming environment.

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. Section 5 outlines how the operational semantics schema can be used. Section 6 discusses how the semantics may be extended to manage both a shared and private state; eventually Section 7 concludes.

2 Our case study: *Lithium*

Lithium is a pure Java library providing the programmer with both task parallel and data parallel skeletons [6, 1]. In particular, classical skeletons are included, such as pipeline and farms, divide and conquer, map (a sort of forall independent) and reduce. All the skeletons process a stream (finite or infinite) of input tasks to produce a stream of results. Pipeline and farm skeletons only exploit parallelism in the execution of different tasks, divide and conquer, map and reduce skeletons, both express parallelism in the execution of different tasks *and* in the execution of subtasks of a single task. All the skeletons are assumed to be stateless. There is no concept of skeleton *state* nor static variables can be safely used in *Lithium* code to implement a skeleton state. This because the execution of *Lithium* skeletons happens in

different JVMs running onto a set of processing elements distributed across a network. Also, no concept of "global state" is supported by the implementation, but the ones explicitly programmed by the user[1]. Last but not least, the *Lithium* skeletons are fully nestable. *Lithium* programmers can write programs where a farm has pipeline workers and some of the pipeline stages are data parallel, for instance. Therefore in Lithium 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. Streams are denoted by using Haskell-like lists cons notation (e.g. $x_1 : x_2 : x_3 : [\,] = [x_1, x_2, x_3]$ denotes a stream of the three values x_1, x_2, x_3) Tuples are denoted by angled braces.

$$\begin{aligned} \mathit{value} &::= \textit{A Java value} \\ \mathit{stream} &::= \mathit{value} : \mathit{stream} \mid \mathit{value} : [\,] \\ \mathit{tuple}_k &::= \langle\ \mathit{value}_1, \cdots, \mathit{value}_k\ \rangle \end{aligned}$$

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 values. The set of skeletons (Δ) provided by *Lithium* are defined as follows:

$$\begin{aligned} \Delta ::= {} & \mathsf{seq}\ f \mid \mathsf{farm}\ \Delta \mid \mathsf{pipe}\ \Delta_1\ \Delta_2 \\ & \mid \mathsf{comp}\ \Delta_1\ \Delta_2 \mid \mathsf{map}\ p^{-1}\ \Delta\ p \mid \mathsf{d\&c}\ t\ p^{-1}\ \Delta\ p \\ & \mid \mathsf{while}\ t\ p^{-1}\ \Delta\ p \end{aligned}$$

where the sequential Java functions (such as f) with no index have type $\mathit{Object} \rightarrow \mathit{Object}$, and indexed functions (p, p^{-1}, t) have the following types: p^{-1} :$\mathit{tuple}_k \rightarrow \mathit{value}$; p :$\mathit{value} \rightarrow \mathit{tuple}_k$; t :$\mathit{value} \rightarrow \mathit{boolean}$. p and p^{-1} respectively representing families of functions which partion a value in a k-tuple and vice-versa.

Lithium skeletons can be considered as a pre-defined higher-order functions. Informally, they can be described as follows:

- the seq skeleton is used to encapsulate sequential portions of code defining functions from `Objects` to `Objects`
- the farm skeleton models embarrassingly parallel computations over a stream of input data. Each input data (task) is computed independently of the others
- the pipe skeleton models pipelines with an arbitrary number of stages, processing a stream of input data
- the comp skeleton actually represents a pipe whose stages are computed sequentially

[1] e.g. RMI servers encapsulating shared data structures can be used and accessed from different points in the skeleton code.

- the map skeleton exploits data parallelism by dividing input data into (possibly overlapping) subsets and computing a partial result out of each subset. The partial results are used to compute (build) the final result. In particular p decomposes the input data into a set of possibly overlapping data subsets, the inner skeleton computes a result out of each subset and the p^{-1} function rebuilds a unique result out of these results
- the d&c skeleton models plain divide and conquer. Here, the input data is divided into subsets by p and each subset is computed recursively and concurrently until the t condition does not hold true. At this point results of sub-computations are conquered via the p^{-1} function
- last but not least, the while skeleton is used to model cycles.

The execution of a *Lithium* program consists in the evaluation of a Δ *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. The input stream is $\perp$ labeled. Values in the output stream are labeled accordingly with processing elements that figured them out (in the case of a distributed memory parallel system this may be interpreted with their storage position). Labels on values and streams are denoted by superscripts and describe where data items are mapped within the system, while labels on transitions (i.e. arrows) describe where they are computed. A transition may be labeled either with single labels or set of them, the latter case modelling a data parallel execution on a set of processing elements. The *Lithium* operational semantics is described in Figure 1. The rules of the *Lithium* semantics may be grouped in two main categories: Unfolding Rules and Exec Rules.

Unfolding rules 1–7 describe how skeletons behave with respect to a stream. These rules spring from a common recursive schema: the stream is unfolded in a sequence of values and the overlined nested skeleton is applied to each item in the sequence, where an overlined skeleton $\overline{\mathsf{Ske}}$ represents a version of Ske working on values instead of streams. During the unfolding, head value and the tail of the stream are labeled according to the particular rule policy, while the transition is labeled with the $\perp$ label meaning the transition is not a real computation but just a data distribution (with the only exception of rule 1). Unfolding rules can be applied in any context[2], and they produce an empty stream when evaluated on an empty stream.

[2] A simple context inference for unfolding rules should be added to be fully formal, we omit it for the sake of clearness.

UNFOLDING RULES

1. $\mathsf{seq}\ f\ x{:}\tau^{\ell} \xrightarrow{\ell} f\ x^{\ell} : \mathsf{seq}\ f\ \tau^{\ell}$
2. $\mathsf{farm}\ \Delta\ x{:}\tau^{\ell} \xrightarrow{\perp} \overline{\mathsf{farm}}\ \overline{\Delta}\ x^{\mathcal{O}(\ell,x)} : \mathsf{farm}\ \Delta\ \tau^{\mathcal{O}(\ell,x)}$
3. $\mathsf{pipe}\ \Delta_1\Delta_2\ x{:}\tau^{\ell} \xrightarrow{\perp} \overline{\mathsf{pipe}}\ \overline{\Delta}_1\ \overline{\Delta}_2\ x^{\ell} : \mathsf{pipe}\ \Delta_1\ \Delta_2\ \tau^{\ell}$
4. $\mathsf{comp}\ \Delta_1\Delta_2\ x{:}\tau^{\ell} \xrightarrow{\perp} \overline{\mathsf{comp}}\ \overline{\Delta}_1\ \overline{\Delta}_2\ x^{\ell} : \mathsf{comp}\ \Delta_1\ \Delta_2\ \tau^{\ell}$
5. $\mathsf{map}\ p^{-1}\Delta\ p\ x{:}\tau^{\ell} \xrightarrow{\perp} \overline{\mathsf{map}}\ p^{-1}\ \overline{\Delta}\ p\ x^{\ell} : \mathsf{map}\ p^{-1}\ \Delta\ p\ \tau^{\ell}$
6. $\mathsf{d\&c}\ t\ p^{-1}\ \Delta\ p\ x{:}\tau^{\ell} \xrightarrow{\perp} \overline{\mathsf{d\&c}}\ t\ p^{-1}\ \overline{\Delta}\ p\ x^{\ell} : \mathsf{d\&c}\ t\ p^{-1}\ \Delta\ p\ \tau^{\ell}$
7. $\mathsf{while}\ t\ p^{-1}\Delta\ p\ x{:}\tau^{\ell} \xrightarrow{\perp} \overline{\mathsf{while}}\ t\ p^{-1}\ \overline{\Delta}\ p\ x^{\ell} : \mathsf{while}\ t\ p^{-1}\ \Delta\ p\ \tau^{\ell}$

EXEC RULES

$\overline{1}$. $\overline{\mathsf{seq}}\ f\ x^{\ell} \xrightarrow{\ell} f\ x^{\ell}$

$\overline{2}$. $\overline{\mathsf{farm}}\ \overline{\Delta}\ x^{\ell} \xrightarrow{\ell} \overline{\Delta}\ x^{\mathcal{O}(\ell,x)}$

$\overline{3}$. $\overline{\mathsf{pipe}}\ \overline{\Delta}_1\overline{\Delta}_2\ x^{\ell} \xrightarrow{\ell} \overline{\Delta}_2\ \mathcal{R}^{\mathcal{O}(\ell,x)}\ \overline{\Delta}_1\ x^{\ell}$

$\overline{4}$. $\overline{\mathsf{comp}}\ \overline{\Delta}_1\overline{\Delta}_2\ x^{\ell} \xrightarrow{\ell} \overline{\Delta}_2\ \overline{\Delta}_1\ x^{\ell}$

$\overline{5}$. $\overline{\mathsf{map}}\ p^{-1}\Delta\ p\ x^{\ell} \xrightarrow{\ell} p^{-1}\ (\alpha\ \overline{\Delta})\ p\ x^{\ell}$

$\overline{6}$. $\overline{\mathsf{d\&c}}\ t\ p^{-1}\ \overline{\Delta}\ p\ x^{\ell} = \begin{cases} \overline{\Delta}\ x^{\ell} & \textit{iff}\ \ (t\ x) \\ p^{-1}\ (\ \alpha\ (\ \overline{\mathsf{d\&c}}\ t\ p^{-1}\ \overline{\Delta}\ p\ )\ )\ p\ x^{\ell} & \textit{otherwise} \end{cases}$

$\overline{7}$. $\overline{\mathsf{while}}\ t\ p^{-1}\ \overline{\Delta}\ p\ x^{\ell} = \begin{cases} \overline{\Delta}\ x^{\ell} & \textit{iff}\ \ (t\ x) \\ \overline{\mathsf{while}}\ t\ p^{-1}\ \overline{\Delta}\ p\ x^{\ell} & \textit{otherwise} \end{cases}$

$$\frac{\overline{\Delta}\ x^{\imath} \xrightarrow{\imath} y^{\jmath}}{\mathcal{R}^{\ell}\ \overline{\Delta}\ x^{\imath} \xrightarrow{\imath} y^{\ell}}\ \textit{relabel} \qquad \frac{\overline{\Delta}_1\ x^{\ell} \xrightarrow{\ell} y^{\imath}}{\overline{\Delta}_2\ \overline{\Delta}_1\ x^{\ell} \xrightarrow{\ell} \overline{\Delta}_2\ y^{\imath}}\ \textit{context}$$

$$\frac{p\ x^{\ell} \xrightarrow{\ell} \langle\ y_1^{\ell_1}, \cdots, y_n^{\ell_n}\ \rangle \quad \overline{\Delta}\ y_i^{\ell} \xrightarrow{\ell_i} z_i^{\ell} \quad p^{-1}\ \langle\ z_1^{\ell}, \cdots, z_n^{\ell}\ \rangle \xrightarrow{\ell} z^{\ell}\ ,\ i = 1..n,\ \Psi(l,y) = \ell_1, \cdots, \ell_n}{p^{-1}\ (\alpha\ \overline{\Delta})\ p\ x^{\ell} \xrightarrow{\ell_1,\cdots,\ell_n} z^{\ell}}\ dp$$

$$\frac{\overline{\Delta}_i\ x_i^{\ell_i} \xrightarrow{\ell_i} y_i^{h_i} \quad \forall i\ 1 \le i \le n \quad \wedge \quad \exists\ i,j\ 1 \le i,j \le n,\ \ell_i = \ell_j \Rightarrow i = j}{\gamma : \overline{\Delta}_1\ x_1^{\ell_1} : \cdots : \overline{\Delta}_n\ x_n^{\ell_n} : \Gamma \xrightarrow{\ell_1,\cdots,\ell_n} \gamma : y_1^{h_1} : \cdots : y_n^{h_n} : \Gamma}\ sp$$

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

Let us show how rules 1–7 work with an example. We evaluate farm (seq f) on the input stream $[x_1, x_2, x_3]^{\perp}$

$\mathsf{farm}\ (\mathsf{seq}\ f)\ [x_1, x_2, x_3]^{\perp} \xrightarrow{\perp} \overline{\mathsf{seq}}\ f\ x_1{}^{\bullet_1} : \mathsf{farm}\ (\mathsf{seq}\ f)\ [x_2, x_3]^{\bullet_1} \xrightarrow{\perp}$
$\overline{\mathsf{seq}}\ f\ x_1{}^{\bullet_1} : \overline{\mathsf{seq}}\ f\ x_2{}^{\bullet_2} : \mathsf{farm}\ (\mathsf{seq}\ f)\ [x_3]^{\bullet_2} \xrightarrow{\perp} \overline{\mathsf{seq}}\ f\ x_1{}^{\bullet_1} : \overline{\mathsf{seq}}\ f\ x_2{}^{\bullet_2} : \overline{\mathsf{seq}}\ f\ x_3{}^{\bullet_1}$

The head of the stream is separated from the rest and re-labeled (from $\perp$ to $\bullet_1$) according to the $\mathcal{O}(\perp, x_1)$ function. Inner skeleton $\overline{\mathsf{seq}}$ has been applied to this value, while the initial skeleton has been applied to the rest of the stream in a recursive fashion. The re-labeling function $\mathcal{O} : \mathit{label} \times \mathit{value} \to \mathit{label}$ (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 processing elements round-robin policy. An oracle function for this policy would cyclically return a label in a set of two labels.

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.

The rest of the work is carried out by Exec Rules in the bottom half of figure 1. There are two main rules (*sp* and *dp*), two auxiliary rules (*context*, *relabel*, and a rule for each overlined skeleton. Such rules define the order of reduction along aggregates of skeletal expressions. Let us describe each rule:

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 all the stream up to the first expression of the sequence has been completely evaluated (the sequence has a prefix of values or no prefixes), and that all the transformation labels involved are pairwise different.

dp (data parallel) rule describes the evaluation order for the $\overline{\mathsf{map}}$ skeleton. Basically the rule creates a tuple by means of the partition function p, 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 p^{-1} function gathers all the elements of the evaluated tuple in a single value. The transition is labeled with a set of labels indicating the processing elements working on this evaluation.

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

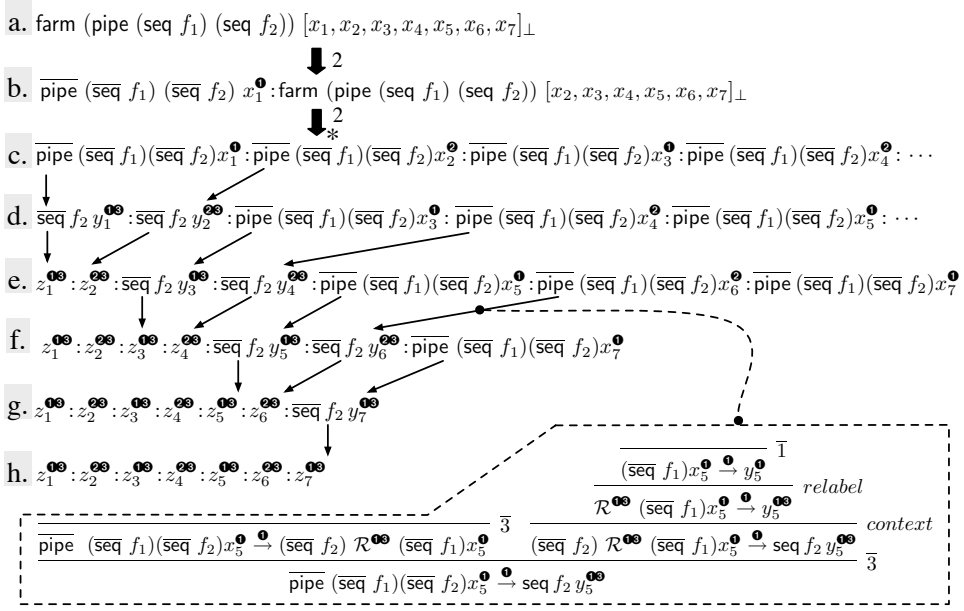

Figure 2: The semantics of a *Lithium* program: a complete example.

computation to another processing element (along with the function to compute).

context rules establishes the evaluation order among nested expressions, in all other cases but the ones treated by *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.

As an example consider the semantics of farm (pipe f_1 f_2) evaluated on the input stream $[x_1, x_2, x_3, x_4, x_5, x_6, x_7]^\perp$. 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 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. As shown in Figure 2, the initial expression is unfolded by rule 2 (Fig. 2 a → b). The same rule is then iterated up to the complete unfolding of the stream (Fig. 2 b →* c). 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 (❶ and ❷), thus we apply *sp* to the leftmost pair of expressions (Fig. 2 c → d).

Due to this reduction two new stream labels appear (❶❸ and ❷❸). Now, we can repeat the application of *sp* rule as in the previous step (Fig. 2 d $\rightarrow$ e $\rightarrow$ f). This time, it is possible to find four adjacent expressions that can be rewritten with (pairwise) different labels (❶, ❷, ❶❸, ❷❸). 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 (Fig. 2 f $\rightarrow$ g $\rightarrow$ h). The succession of *sp* applications in the transition system matches the expected behavior for the double pipeline paradigm. In particular the semantics correctly models start-up, steady-state and end-up phases of the computation. The reduction (Fig. 2 a $\rightarrow$ b) represents the start-up phase where the maximum parallelism available is 2 due to a bouble in second stage of the pipeline. Reductions (Fig. 2 d $\rightarrow$ e $\rightarrow$ f) represent steady-state where all the 4 processing elements in the double-pipeline are working in parallel. Reductions (Fig. 2 f $\rightarrow$ g $\rightarrow$ h) represent end-up phase where tasks are flowing out of the pipeline thus progressively using less processing elements in the pipeline.

Let us describe how *sp* and *dp* works together along an example. We evaluate $\mathsf{map}\ p^{-1}\ (\mathsf{seq}\ f)\ p$ on the input stream $[x_a, x_b]^{\perp}$. The expresssion can be unfolded by two applications of rule 5 (see Fig. 1) to

$$\overline{\mathsf{map}}\, p^{-1}\, (\mathsf{seq}\, f)\, p\, x_a^{\perp} : \overline{\mathsf{map}}\, p^{-1}\, (\mathsf{seq}\, f)\, p\, x_b^{\perp}$$

Assume partition function p split input data in two parts (e.g. an array divided in two slices) represented as a $tuple_2$, and p^{-1} joins them up. The *sp* rule can be applied to the leftmost expression only since the two expressions evaulate values with the same label $\perp$ (no stream parallelism):

$$\dfrac{\dfrac{}{\overline{\mathsf{map}}\, p^{-1}\, (\mathsf{seq}\, f)\, p\, x_a^{\perp} \xrightarrow{\perp} p^{-1}\ (\alpha\ (\overline{\mathsf{seq}}\ f))\ p\ x_a^{\perp}}\ \overline{5} \quad \dfrac{\cdots}{p^{-1}\ (\alpha\ (\overline{\mathsf{seq}}\ f))\ p\ x_a^{\perp} \xrightarrow{\perp} y_a^{❸,❹}}\ dp}{\overline{\mathsf{map}}\, p^{-1}\, (\mathsf{seq}\, f)\, p\, x_a^{\perp} : \overline{\mathsf{map}}\, p^{-1}\, (\mathsf{seq}\, f)\, p\, x_b^{\perp} \xrightarrow{❸,❹} y_a^{❸,❹} : \overline{\mathsf{map}}\, p^{-1}\, (\mathsf{seq}\, f)\, p\, x_b^{\perp}}\ sp$$

where *dp* is applied as follows:

$$\dfrac{p\ x_a^{\perp} \xrightarrow{\perp} \langle x_{a1}^{❸}, x_{a2}^{❹} \rangle \quad \dfrac{}{\overline{\mathsf{seq}}\ f x_{a1}^{❸} \xrightarrow{❸} y_{a1}^{❸}}\ \overline{1} \quad \dfrac{}{\overline{\mathsf{seq}}\ f x_{a2}^{❹} \xrightarrow{❹} y_{a2}^{❹}}\ \overline{1} \quad p^{-1}\ \langle y_{a1}^{❸}, y_{a2}^{❹} \rangle \xrightarrow{❸,❹} y_a}{p^{-1}\ (\alpha\ (\overline{\mathsf{seq}}\ f))\ p\ x_a^{\perp} \xrightarrow{❸,❹} y_a^{❸,❹}}\ dp$$

exploiting a data parallelism on two processing elements (two reduction at the same time). The *sp* rule, and in turn *dp* rule, must be applied to sort out the output stream $y_a^{❸,❹} : y_b^{❸,❹}$.

Note that if we nest the skeletal program in a skeleton supplying stream parallelism (e.g. $\mathsf{farm}\ \mathsf{map}\ p^{-1}\ (\mathsf{seq}\ f)\ p$), we can exploit both stream and data parallelism. As an example if we assume a round-robin policy for the farm as before we reduce the initial expression to a pair of expressions that can be evaluated at the same time by the *sp*, in turn, each of them exploits data parallelism. Overall we have 4 as parallelism degree as also suggested by the labels in the output stream:

$$\overline{\mathsf{map}}\, p^{-1}\, (\mathsf{seq}\, f)\, p\, x_a^{\textcircled{1}} : \overline{\mathsf{map}}\, p^{-1}\, (\mathsf{seq}\, f)\, p\, x_b^{\textcircled{2}} \overset{\textcircled{13},\textcircled{14},\textcircled{23},\textcircled{24}}{\longrightarrow} y_a^{\textcircled{13},\textcircled{14}} : y_b^{\textcircled{23},\textcircled{24}}$$

Observe that even interleaving the application of rule 5 and *dp* does not change the output stream. This behaviour is also true in the general case since – as we shall discuss – the semantic is functionally confluent.

4 Parallelism and labels

Two relevant aspects of the proposed schema are that

- the functional semantics is confluent;
- the functional semantics is independent from labeling functions.

In order to enforce confluence we can arrange the application of rules in two phases: an expression must be first completely reduced by using unfolding rules only, then by all other rules. Unfolding rules are deterministic because they are syntax directed and their application produce a sequence of expressions with at most one expression that can be reduced by themselves, i.e. the rightmost one. No other rules produce terms that can be reduced by unfolding rules. Unfolding rules produces a sequence of expressions joined by cons. The only rule than can reduce this term is the *sp* rule. Moreover *sp* rule reduces each expression independently: the only possible choice consists of the number of expressions that can be reduced in a single application of the rule. It is trivial to prove (by modus ponens) that the applications of *sp* with n reduction may be simulated with n consecutive application with 1 reduction, which can always be applied. As an example, a *sp* application with $n = 2$:

$$\frac{\dfrac{\cdots}{\overline{\Delta}_1\ x_1^{\ell_1} \overset{\ell_1}{\to} y_1^{h_1}} \qquad \dfrac{\cdots}{\overline{\Delta}_2\ x_2^{\ell_2} \overset{\ell_2}{\to} y_2^{h_2}}}{\gamma : \overline{\Delta}_1\ x_1^{\ell_1} : \overline{\Delta}_2\ x_2^{\ell_2} : \Gamma \overset{\ell_1,\ell_2}{\to} \gamma : y_1^{h_1} : y_2^{h_2} : \Gamma}\ sp$$

may be simulated with two reductions with $n = 1$:

$$\frac{\dfrac{\dfrac{\cdots}{\overline{\Delta}_1\ x_1^{\ell_1} \overset{\ell_1}{\to} y_1^{h_1}}}{\gamma : \overline{\Delta}_1\ x_1^{\ell_1} : \overline{\Delta}_2\ x_2^{\ell_2} : \Gamma \overset{\ell_1}{\to} \gamma : y_1^{h_1} : \overline{\Delta}_2\ x_2^{\ell_2} : \Gamma}\ sp \qquad \dfrac{\dfrac{\cdots}{\overline{\Delta}_2\ x_2^{\ell_2} \overset{\ell_2}{\to} y_2^{h_2}}}{\gamma : y_1^{h_1} : \overline{\Delta}_2\ x_2^{\ell_2} : \Gamma \overset{\ell_2}{\to} \gamma : y_1^{h_1} : y_2^{h_2} : \Gamma}\ sp}{\gamma : \overline{\Delta}_1\ x_1^{\ell_1} : \overline{\Delta}_2\ x_2^{\ell_2} : \Gamma \overset{\ell_2}{\to} \gamma : y_1^{h_1} : y_2^{h_2} : \Gamma}$$

Note that the only difference is the transition label. As a matter of a fact the two reductions have the same functional semantics but they describe two different parallel behaviors. Pragmatically the iterated application of *sp* with $n = 1$ describes the linearization of a parallel execution and can be assumed as canonical derivation.

Since the canonical derivation does not depend on labels, the functional semantics does not depend on labels. 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.

Another important issue 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 processing elements). 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 processing elements.

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 processing elements 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 processing elements, and 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). 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 processing elements currently computing the task.

5 How to use the labeled transition system

The *Lithium* semantics discussed in section 3 can be used to prove *Lithium* program correctness, as expected. Here we want to point out how the transition system can be used to evaluate analogies and differences holding between different skeleton systems and to evaluate the effect of rewriting rules on a skeleton program.

Concerning the first point, let us take into account, as an example, the skeleton framework introduced by Darlington's group in mid '90. In [15] they define a farm skeleton as follows: *"Simple data parallelism is expressed by the* FARM *skeleton. A function is applied to each element of a list, each element of which is thought of as a task. The function also takes an environment, which represents data which is common to all tasks. Parallelism is achieved by distributing the tasks on different processors.*

FARM : $(\alpha \to \beta \to \gamma) \to \alpha \to (\ \langle\ \beta\ \rangle \to \langle\ \gamma\ \rangle\ )$
FARM f env = map (f env)
where map f x = f x *and* map f x:rx = (f x):(map f rx)".

Darlington's FARM skeleton happens to have the same name as the farm skeleton developed by our group [9, 10] and by other groups ([26, 21]) but it has a slightly different semantics. It actually exploits data parallelism, in that parallelism comes from parallel computation of items belonging to the same task (input data item), i.e. the list x parameter, whereas other farm skeletons express embarrassingly parallel computations exploiting par-

allelism in the computation of disjunct, completely independent task items. This can be easily expressed using our semantics by observing that the clause:

$$\mathsf{FARM}\ p^{-1}\ \Delta\ p\ x^{\ell} \xrightarrow{\ell} p^{-1}\ (\alpha\ \Delta)\ p\ x^{\ell}$$

exactly models both the functional and the parallel behavior of *Lithium* map provided that p is the function taking a list and returning a tuple of singletons, p^{-1} is the function taking a tuple of singletons and returning a list and that Δ is actually a sequential function f. However, in order to make the FARM being "stream aware", we must rather consider a clause such as:

$$\mathsf{FARM}\ p^{-1}\Delta\ p\ x{:}\tau^{\ell} \xrightarrow{\ell} p^{-1}\ (\alpha\Delta)\ p\ x^{\ell}{:}\ \mathsf{FARM}\ p^{-1}\ \Delta\ p\ \tau^{\ell}$$

recursive on the stream parameter. At this point we should immediately notice that this is the same clause modeling our map skeleton (cf. rule 5 in Figure 1).

Concerning the possibility of using the LTS to prove correctness of skeleton rewriting rules, let us take as an example the equivalence "farm $\Delta \equiv \Delta$" (see [5, 6]). The equivalence states that farms (stream parallel ones, not the Darlington's ones) can be inserted and removed anywhere in a skeleton program without affecting the functional semantics of the program. The equivalence looks like to be a really simple one, but it is fundamental in the process of deriving the skeleton "normal form" (see [5]). The normal form is a skeletal expression that can be automatically derived from any other skeletal expression. It computes the same results of the original one but uses less resources and delivers better performance. The *Lithium* system may automatically derive and execute the normal form of any correct program.

The validity of these equivalences can be evaluated using the labeled transition system and the effects on parallelism degree in the execution of left and right hand side programs can be evaluated too. The validity of the equivalence can be simply stated by taking into account rule 2 in Figure 1. It simply unfolds the stream and applies Δ to the stream items, as we can expect for Δ applied to the same stream, apart for the labels. As the labels do not affect the functional semantics, we can derive that the equivalence "farm $\Delta \equiv \Delta$" actually holds for any skeleton tree Δ. We can say more, however. Take into account expression a. in Figure 2. We derived its semantics and we concluded that four different (non-bottom) labels can be obtained: ❶❸, ❷❸, ❶, ❷. Rewriting (**??**) by using the equivalence from left to right we get: "pipe (seq f_1) (seq f_2) $[x_1, x_2, x_3, x_4, x_5, x_6, x_7]$". From this expression we can derive exactly the same output but holding only two distinct labels: $0, 1$. These two labels happen to be the two labels modeling the two stages of the original pipeline. As a conclusion we can state that:

1. the equivalence "farm $\Delta \equiv \Delta$" holds and functional semantics is not affected by it;

2. parallel semantics is affected by the equivalence in that the parallelism degree changes.[3]

6 Managing a state

Lithium only provides *stateless* skeletons, as well as most of the other existing skeletons systems [9, 14, 26, 25], that is, it is proposed as a "pure" functional framework. 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 [28, 4] explicitly 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.

7 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.

[3] This just one case, of course, in order to complete the proof all the different possibles Δ must be considered. However the simple case of $\Delta =$ pipe should made clear how the full proof may work.

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. Last, we gave an idea of how the LTS features can be exploited to model skeletons with state. 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] The lithium home page. `http://www.di.unipi.it/~marcod/Lithium/`, 2002.

[2] The Ocaml home page. `http://www.ocaml.org`, 2003.

[3] 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.

[4] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The implementation of ASSIST, an environment for parallel and distributed programming. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Proc. of the Euro-Par 2003*, number 2790 in LNCS, pages 712–721. Springer, August 2003.

[5] 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.

[6] 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.

[7] 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.

[8] 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, *Proc. of Euro-Par 1996*, pages 601–614. Springer-Verlag, 1996.

[9] 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.

[10] 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.

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

[12] M. Cole and Y. Hayashi. Static performance prediction of skeletal programs. *Parallel Algorithms and Applications*, 17(1):59–84, 2002.

[13] R. Di Cosmo, S. Pelagatti, and Z. Li. A calculus for parallel computations over multidimensional dense arrays. *Computer Languages, Systems and Structures*, 2004. (to appear).

[14] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, R. L. While, and Q. Wu. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Proc. of the Parallel Architectures and Langauges Europe (PARLE'93)*, number 694 in LNCS. Springer-Verlag, June 1993.

[15] J. Darlington and H. W. To. Building parallel applications without programming. In *Leeds Workshop on Abstract Parallel Machine Models 93*, 1993.

[16] P. Fradet and J. Mallet. Compilation of a specialized functional language for massively parallel computers. *Journal of Functional Programming*, 10(6):561–605, November 2000.

[17] F. Gava and F. Loulergue. A parallel virtual machine for bulk synchronous parallel ML. In *Intl. Conference on Computational Science (ICCS 2003)*, number 2657 in LNCS, pages 155–164. Springer Verlag, June 2003.

[18] 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.

[19] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters (World Scientific Publishing Company)*, 12(2):211–228, 2002.

[20] U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation Skeletons in Eden — Low-Effort Parallel Programming. In *IFL'00 — Intl. Workshop on the Implementation of Functional Languages*, number 2011 in LNCS, pages 71–88, Aachen, Germany, September 2000. Springer.

[21] 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.

[22] F. Loulergue. Distributed evaluation of functional BSP programs. *Parallel Processing Letters*, 4:423–437, 2001.

[23] F. Loulergue, Z. Hu, and K. Kakehi. An implementation of the diffusion algorithmic skeleton with the BSMLlib library. Technical Report METR-2004-06, Department of Mathematical Informatics, University of Tokyo, 2004.

[24] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Taa. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1684, december 2002.

[25] S. MacDonald, D. Szafron, J. Schaeffer, and S. Bromling. Generating parallel program frameworks from parallel design patterns. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *proc of Euro-Par 2000*, number 1900 in LNCS, pages 95–105. Springer-Verlag, September 2000.

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

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

[28] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, december 2002.