Available online at www.sciencedirect.com

ScienceDirect

Computer Languages, Systems & Structures 33 (2007) 179 – 192

COMPUTER LANGUAGES, SYSTEMS & STRUCTURES

www.elsevier.com/locate/cl

Skeleton-based parallel programming: Functional and parallel semantics in a single shot[☆]

Marco Aldinucci[a,*], Marco Danelutto[b]

[a] *Institute of Information Science and Technologies—CNR, Via Moruzzi 1, Pisa, Italy*
[b] *Computer Science Department, University of Pisa, Largo B. Pontecorvo 3, Pisa, Italy*

Received 1 October 2005; accepted 24 July 2006

Abstract

Semantics of skeleton-based parallel programming languages comes usually as two distinct items: a *functional semantics*, modeling the function computed by the skeleton program, and a *parallel semantics* describing the ways used to exploit parallelism during the execution of the skeleton program. The former is usually expressed using some kind of semantic formalism, while the latter is almost always given in an informal way. Such a 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 a system. We use Lithium, a skeleton-based full Java parallel programming environment, as the case study.
© 2006 Elsevier Ltd. All rights reserved.

Keywords: Algorithmical skeletons; Structured parallel programming; Labeled transition systems; Functional semantics; Parallel semantics

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 1990s, several research groups have 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.

[☆] 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. The preliminary version of this paper appeared in [1].

[*] Corresponding author. Tel.: +39 050 2212728; fax: +39 050 2212726.
E-mail addresses: aldinuc@di.unipi.it (M. Aldinucci), marcod@di.unipi.it (M. Danelutto).
URLs: http://www.di.unipi.it/∼aldinuc (M. Aldinucci), http://www.di.unipi.it/∼marcod (M. Danelutto).

1477-8424/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2006.07.004

Notable examples of this kind of environments are those based on the *algorithmical skeleton* concept. A skeleton, which is a known, recurrent pattern of parallelism exploitation, has been originally conceived by Cole [2]. Later on, it has been used by different research groups to design high-performance structured parallel programming environments [3–11]. 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 [12].

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 the parallelism exploitation pattern it models. As an example, a farm skeleton takes as a parameter the worker computation, 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, which in turn may be either sequential portions of code (sequential skeletons) or other parallel skeletons. 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 the previously cited frameworks have formal functional semantics. Typically, such semantics are given by providing a completely functional description of the skeletons used by means of a set of higher-order functions. In addition, some of these equip the functional behavior with a parallel behavior. These are mostly focused on the description of data parallel computations, as, for example, in BSP style computations [8,13–15]. Other works focused on task parallelism also (e.g. [16]), even if it might be hard to equip them with a simple cost model.

As an example, using Ocaml [17] 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 denoted 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 [18–20]. 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 [18,21–24]. For instance, one can easily derive that the following two programs actually compute the same result:

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

Quite often, the parallel semantics of the skeletons (especially for task parallel ones) is given in an informal way. For example, in the documentation of a skeleton environment, one may find a text looking like 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` above 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. Moreover, the parallel behavior of the two programs cannot be compared in any formal way.

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 [25]. 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 [25]. In particular, classical skeletons are included, such as pipeline and farm, divide and conquer, map (a sort of forall independent) and reduce. All the skeletons process a (finite) stream 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 stateless. Each output value computed by a skeleton only depends on the data passed as input to the skeleton itself. In other words, there is no concept of skeleton *state* nor static variables can be safely used in *Lithium* code to implement a skeleton state. Stateless skeletons model very common and useful parallelism exploitation patterns. As a side effect, the execution of *Lithium* skeletons can be implemented 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, except the ones explicitly and safely programmed by the user, e.g. using RMI servers to encapsulate shared data structures in such a way they can be used and accessed from different points in the skeleton code. 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. $x_1 \colon x_2 \colon x_3 \colon [\,] = [x_1, x_2, x_3]$ denotes a stream of the three values x_1, x_2, x_3). Tuples are denoted by angled braces.

$$\begin{aligned}
&\mathit{value} ::= \text{\it A Java value}\\
&\mathit{stream} ::= \mathit{value}\colon \mathit{stream} \,|\, [\,]\\
&\mathit{tuple}_k ::= \langle \mathit{value}_1, \ldots, \mathit{value}_k \rangle
\end{aligned}$$

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:

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

where the sequential Java functions (such as *f*) with no index have type $Object \rightarrow Object$, and auxiliary 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 that partition a value in a k-tuple and vice versa. *Lithium* provides the programmer with some simple partition multi-dimensional arrays in rows, columns and blocks. Moreover, programmers may directly code their partition (p) and gather (p^{-1}) functions writing them as Java methods. A formal definition, characterization and costing of partition and gather functions for arrays can be found in [15].

Lithium skeletons can be considered as 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;
- 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 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 are 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;
- the while skeleton is used to model loops.

Examples of how to design and code a *Lithium* program can be found in [25,26]. A *Lithium* program Δ transforms an (not empty) input *stream* into an output *stream*: $\Delta : stream \to stream$.

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 and required to be not empty. Values in the output stream are labeled according to 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. They describe where data items are mapped within the system, while labels on transitions (i.e. arrows) describe where they are computed. A transition is labeled with a set of labels, whose cardinality being the parallelism degree in a parallel execution.

We introduce the following syntactic categories representing the version of skeletons working on *value* ($\overline{\Delta} : value \to value$), not-evaluated (or partially evaluated) skeletal expressions, and labeled skeletal expressions:

$$\begin{aligned}\overline{\Delta} ::= {} & \overline{\mathsf{seq}}\ f \mid \overline{\mathsf{farm}}\ \overline{\Delta} \mid \overline{\mathsf{pipe}}\ \overline{\Delta}_1\ \overline{\Delta}_2 \mid \overline{\mathsf{comp}}\ \overline{\Delta}_1\ \overline{\Delta}_2 \mid \overline{\mathsf{map}}\ p^{-1}\ \overline{\Delta}\ p \\ & \mid \overline{\mathsf{d\&c}}\, t\ p^{-1} \overline{\Delta}\ p \mid \overline{\mathsf{while}}\ t\ \overline{\Delta} \mid \mathscr{R}^{\ell}\ \overline{\Delta}\end{aligned}$$

$$ske\_exp ::= \overline{\Delta}\ value\ : ske\_exp \mid \Delta\ stream \mid lab\_ske\_exp$$

$$lab\_ske\_exp ::= value : lab\_ske\_exp \mid \overline{\Delta}\ value : lab\_ske\_exp \mid \varepsilon$$

where ε is the empty expression. The *Lithium* operational semantics is defined in two steps: a program evaluated onto a stream is first reduced to a labeled skeletal expression (*lab_ske_exp*), by means of UNFOLD RULES (see Fig. 1). These rules just map a skeletal program Δ along stream items, transforming ske_exp into ske_exp. In the second step, the labeled skeletal expression is actually evaluated by means EXEC RULES (see Fig. 1).

Definition 1 (*Labeled skeletal expressions*). Given a skeletal expression $E \in ske\_exp, \Gamma \in lab\_ske\_exp$ is the labeled skeletal expression of E iff

$$E \mapsto^* \Gamma$$

where $\mapsto^*$ is the reflexive transitive closure of $\mapsto$ defined by UNFOLD RULES shown in Fig. 1. Note that $\Gamma = \overline{\Delta}_0 x_0 : \overline{\Delta}_1 x_1 : \cdots : \overline{\Delta}_k x_k$

All unfold rules spring from a common recursive schema: the stream is recursively unfolded in a sequence of values and the nested skeleton is applied to each item in the sequence. Overall, $\mapsto^*$ just provide a syntactic rewriting of

UNFOLD RULES ($\mapsto$)

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

$$\frac{\Delta\ (x:\tau)^{\ell} \mapsto \overline{\Delta}\ x^{\jmath} : \Delta\ \tau^{\jmath}}{C(\Delta\ (x:\tau)^{\ell}) \mapsto C(\overline{\Delta}\ x^{\jmath} : \Delta\ \tau^{\jmath})}\ \textit{context\_unfold, where} \quad C ::= - \mid \overline{\Delta}\ x : C$$

EXEC RULES ($\to$)

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

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

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

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

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

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

$\overline{7}$. $\overline{\mathsf{while}}\ t\ \overline{\Delta}\ x^{\ell} \xrightarrow{\ell} \begin{cases} \overline{\Delta}\ x^{\ell} & \textit{iff}\quad (t\ x) \\ \overline{\mathsf{while}}\ t\ \overline{\Delta}\ x^{\ell} & \textit{otherwise} \end{cases}$

$$\frac{\overline{\Delta}\ x^{\ell_2} \xrightarrow{\ell_2} y^{\ell_3}}{\mathcal{R}^{\ell_1}\ \overline{\Delta}\ x^{\ell_2} \xrightarrow{\ell_2} y^{\Phi(\ell_1\ell_2,0)}}\ \textit{relabel} \qquad \frac{\overline{\Delta}_1\ x^{\ell} \xrightarrow{\ell} y^{\jmath}}{\overline{\Delta}_2\ \overline{\Delta}_1\ x^{\ell} \xrightarrow{\ell} \overline{\Delta}_2\ y^{\jmath}}\ \textit{context}$$

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

$$\frac{\overline{\Delta}_i\ x_i^{\ell_i} \xrightarrow{\ell_i} y_i^{\jmath_i} \quad \forall i\ 1 \le i \le n \quad \wedge \quad \Gamma_1 \not\to \quad \wedge \quad \exists\ i,j\ 1 \le i,j \le n,\ \ell_i = \ell_j \Rightarrow i = j}{\Gamma_1 : \overline{\Delta}_1\ x_1^{\ell_1} : \cdots : \overline{\Delta}_n\ x_n^{\ell_n} : \Gamma \xrightarrow{\ell_1,\cdots,\ell_n} \Gamma_1 : y_1^{\jmath_1} : \cdots : y_n^{\jmath_n} : \Gamma}\ sp$$

Fig. 1. Each UNFOLD RULE has a twin rule (not shown in the figure) without the recursive term. $x, y, z \in$ *value*; $\tau \in$ *stream*; $\ell, \ell_i, \jmath, \jmath_i \ldots \in$ *label*; $\mathcal{O}, \Psi, \Phi$: *label** $\times$ *value* $\to$ *label**; $\Gamma_1, \Gamma_2 \in$ *lab_ske_exp*; $\alpha\overline{\Delta}\langle x_1, \ldots, x_n\rangle = \langle \overline{\Delta}x_1, \ldots, \overline{\Delta}x_n\rangle$ (a.k.a. apply-to-all [27]).

expressions by mapping skeletons along the stream (i.e. lazy functional map operator). Note that stream items are individually labeled in the map; this label is inherited from stream items in all cases except for rule 2, which relabels stream items according to an external function ($\mathcal{O}$) (described in Section 3.1).

Lemma 1. *Each skeletal expression E can be reduced to a unique labeled skeletal expression Γ deterministically obtained.*

Proof (*sketch*). By structural induction on terms: the set of labeled skeletal expressions derived from one skeletal expression is not empty since stream are finite and cannot be empty, and just one among exec rules can be applied to a single term. The derivation of a labeled skeletal expression is driven by syntax of terms. □

From labeled skeletal expressions we define both functional and parallel semantics as follows:

Definition 2 (*Computed stream*). Given a skeletal expression Γ, the computed stream of Γ, $[[\Gamma]]_f$ is the set

$$[[\Gamma]]_f = \{\Gamma_1 \in lab\_ske\_exp : \Gamma \rightarrow^* \Gamma_1 \ \wedge \ \Gamma_1 \not\rightarrow\},$$

where $\xrightarrow{\ell}$ defined by Exec Rules is shown in Fig. 1.

Definition 3 (*Computation labels*). Given a labeled skeletal expression Γ, the set of *computation labels* of Γ, $[[\Gamma]]_p$ is the set

$$[[\Gamma]]_p = \left\{ L_0; \cdots ; L_k : \exists\ \Gamma_1 \cdots \Gamma_{k+1} : \Gamma \xrightarrow{L_0} \Gamma_1 \wedge \forall i = 1 \cdots k\ \Gamma_i \xrightarrow{L_i} \Gamma_{i+1} \wedge \Gamma_{k+1} \not\rightarrow \right\}$$

where $L_i = \ell_0, \ldots, \ell_j$ is a sequence of labels, $\rightarrow$ is defined by Exec Rules shown in Fig. 1, and $\Gamma_{k+1} \not\mapsto$ is defined as $\nexists\ \Gamma' \in lab\_ske\_exp : \Gamma_{k+1} \mapsto \Gamma'$.

Definition 4 (*Functional and parallel semantics*). Given a skeletal expression $E = \Delta\ \tau$, $\tau \in$ stream, if Γ is the labeled skeletal expression of E, $[[\Gamma]]_f$ and $[[\Gamma]]_p$ are, respectively, the functional and parallel semantics of E.

3.1. Examples

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

$$\mathsf{farm}\ (\mathsf{seq}\ f)\ [x_1, x_2, x_3]^{\perp} \ \mapsto \overline{\mathsf{seq}}\ f\ x_1{}^{❶}\!:\!\mathsf{farm}\ (\mathsf{seq}\ f)\ [x_2, x_3]^{❶} \ \mapsto$$
$$\overline{\mathsf{seq}}\ f\ x_1{}^{❶}\!:\!\overline{\mathsf{seq}}\ f\ x_2{}^{❷}\!:\!\mathsf{farm}\ (\mathsf{seq}\ f)\ [x_3]^{❷} \mapsto \overline{\mathsf{seq}}\ f\ x_1{}^{❶}\!:\!\overline{\mathsf{seq}}\ f\ x_2{}^{❷}\!:\!\overline{\mathsf{seq}}\ f\ x_3{}^{❶}$$

The head of the stream is separated from the rest and re-labeled (from $\perp$ to ❶) 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 $\mathcal{O}$: *label** $\times$ *value* $\rightarrow$ *label** (namely the *oracle*) is an external function with respect to the LTS. It would represent the implementation-dependent features of each skeleton (e.g. scheduling policy, parallelism degree, etc.). The more general oracle manage resources by divination, i.e. choosing the optimal policy for each feature (e.g. unlimited parallelism degree, scheduling which deliver the optimal load-balancing, etc.). As an example, consider a farm with a round-robin scheduling over two workers.[1] An oracle function for this policy would cyclically return a label in a set of two labels.

The rest of the work is carried out by exec rules in the bottom half of Fig. 1. In addition to rules $\overline{1}$–$\overline{7}$, there are two main rules (*sp* and *dp*), and two auxiliary rules (*context*, *relabel*). Such rules define the order of reduction along labeled skeletal expressions. Let us describe each rule:

Rules $\overline{1}$–$\overline{7}$ define the semantics of $\overline{\Delta}\ value$. Note that it may be a sequential or parallel program, but the parallelism does not spring from the stream. At this point, all information leading to stream parallelism has been already transferred

[1] Farm equipped with round-robin scheduling policy is also named *deal* [11].

on labels and the relative parallelism is exploited by the topmost rule *sp*. Therefore, the possible paradigms are data parallelism and recursively applied data parallelism (i.e. $\overline{\text{d\&c}}$). These kinds of parallelism are managed by a single or iterated application of *dp* rule, respectively.

sp (*stream parallel*) rule describes evaluation order of labeled skeletal expressions along sequences separated by : operator. The meaning of the rule is the following: suppose that each term in the sequence may be rewritten in another term with a certain labeled transformation. Then, all such terms can be transformed in parallel, provided that they are adjacent, that all the stream up to the first term 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 $\overline{\texttt{map}}$ skeleton. The rule creates a tuple by means of the partition function p, and then requires the evaluation of all expressions composed by applying $\overline{\Delta}$ onto all elements of the tuple. All such expressions 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. These labels depend on the oracle Ψ that is external with respect to the semantics because it models the mapping of computations over processing elements.

relabel rule provides a relabeling facility by evaluating the function $\mathscr{R}^{\ell}$. 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 (along with the function to compute).

context rule establishes the evaluation order among nested expressions, in all cases except the ones treated by *relabel*. The rule imposes a strict evaluation of nested expressions (i.e. evaluating 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 $\Delta = \texttt{farm}\ (\texttt{pipe}\ f_1\ f_2)$ evaluated on the input stream $\tau = [x_1, x_2, x_3, x_4, x_5, x_6, x_7]^{\perp}$. Let us suppose that the oracle function returns a label chosen from a set of two labels. Pragmatically, since a farm skeleton represents the replication paradigm and a pipe skeleton the pipeline paradigm, the nested form $\texttt{farm}\ (\texttt{pipe}\ f_1\ f_2)$ 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 Fig. 2, the initial skeletal expression is unfolded to a labeled skeletal expression by means of unfold rules (Fig. 2, a $\mapsto^*$ b). We can then 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, b $\to$ c).

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, c $\to$ d $\to$ e). 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 4. Repeating the reasoning, we can completely reduce the formula to a sequence of singleton streams (Fig. 2, e $\to$ f $\to$ g). 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 reductions (Fig. 2, b $\to$ c) represent the start-up phase where the maximum parallelism available is 2, since the second stage of the pipeline has not fed with a task yet. Reductions (Fig. 2, c $\to$ d $\to$ e) represent steady state where all the four processing elements in the double-pipeline are working in parallel. Reductions (Fig. 2, e $\to$ f $\to$ g) represent end-up phase where tasks are flowing out of the pipeline thus progressively using less processing elements in the pipeline. Overall, the two semantics are the following:

$$[[\Delta\ \tau]]_f = z_1 : z_2 : z_3 : z_4 : z_5 : z_6 : z_7$$

$$[[\Delta\ \tau]]_p = \text{❶}, \text{❷}\ ;\ \text{❶❸}, \text{❷❸}, \text{❶}, \text{❷}\ ;\ \text{❶❸}, \text{❷❸}, \text{❶}, \text{❷}\ ;\ \text{❶❸}, \text{❷❸}, \text{❶}\ ;\ \text{❶❸}$$

From parallel semantics, we can easily understand that the stream has been evaluated in five steps exploiting a maximum parallelism of 4.

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

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

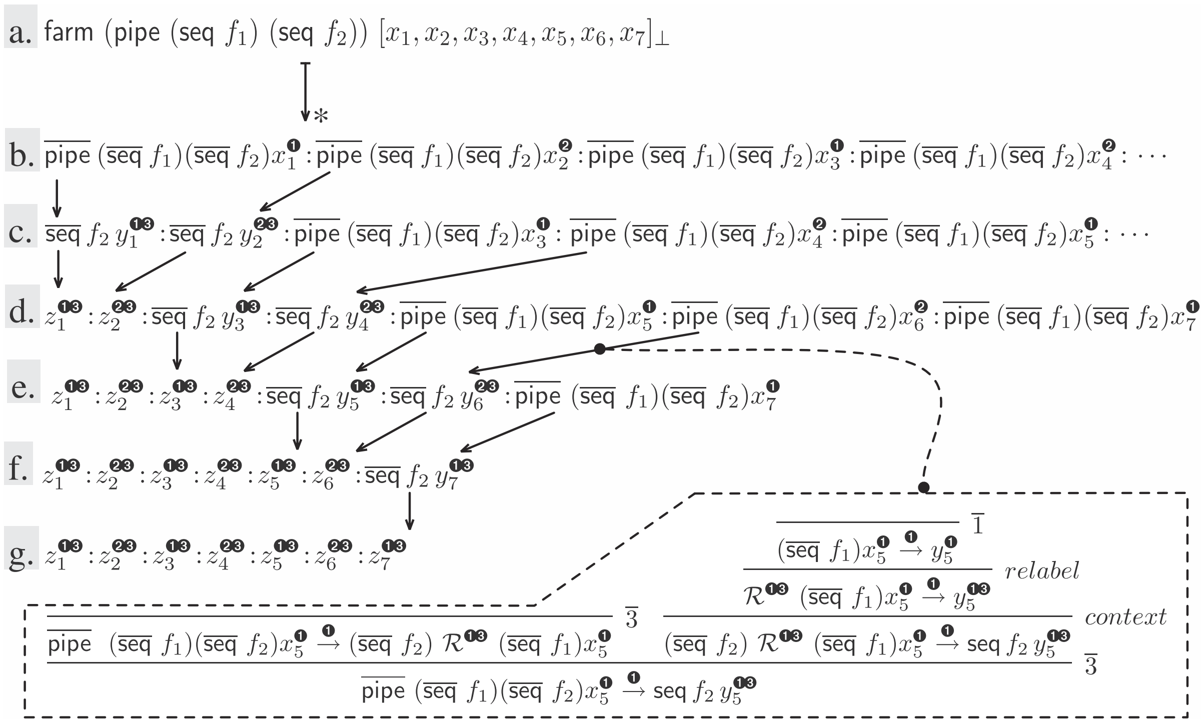


Fig. 2. The semantics of a *Lithium* program: a complete example.

Assume that the partition function p splits input data into two parts (e.g. an array divided into two slices) represented as a *tuple*$_2$, and p^{-1} joins them up. The *sp* rule can only be applied to the leftmost expression since the two expressions evaluate values with the same label $\perp$ (no stream parallelism):

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

where *dp* is applied as follows:

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

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 $y_a^{\text{❸,❹}} : y_b^{\text{❸,❹}}$. In this case, the oracle function Ψ for the map skeleton simply distributes each data parallel sub-execution to the same set of processing elements having the same cardinality of the data partitioning. In particular, $\Psi(\ell, x) = \alpha\ (\ell ++)\ \{\text{❸}, \text{❹}\}$, where $++$ is the string concatenation operator and $\perp$ its neutral element.[2] Note that if we nest the skeletal program in a skeleton supplying stream parallelism (e.g. farm map p^{-1} (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

[2] α = apply-to-all, see Fig. 1.

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:

$$\overline{\mathsf{map}}\, p^{-1}\,(\mathsf{seq}\, f)\, p\, x_a^{\text{❶}} : \overline{\mathsf{map}}\, p^{-1}\,(\mathsf{seq}\, f)\, p\, x_b^{\text{❷}} \xrightarrow{\text{❶❸},\text{❶❹},\text{❷❸},\text{❷❹}} y_a^{\text{❶❸},\text{❶❹}} : y_b^{\text{❷❸},\text{❷❹}}$$

Indeed, in this case $\Psi(\text{❶}, x_a) = \{\text{❶❸}, \text{❶❹}\}$ and $\Psi(\text{❷}, x_b) = \{\text{❷❸}, \text{❷❹}\}$. The two sets of labels are then joined up by the *sp* rule.

4. Parallelism and labels

Two relevant aspects of the proposed operational semantics schema are that

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

As described, both parallel and functional semantics are in two successive phases: a skeletal expression is rewritten into a labeled skeletal expression (Definition 1). Then, either parallel or functional semantics is evaluated according to Definition 4.

The functional semantics is confluent because relations defined in both phases are confluent. The first phase is defined by a fully deterministic relation (Lemma 1), thus it is confluent. In the following, we show that the second phase is also confluent and therefore we define the *canonical reductions*, namely those reductions obtained forcing $n = 1$ in any application of the *sp* rule (see Fig. 1). Observe that canonical reductions do not exploit the parallelism among different stream items; indeed, consecutive stream items are evaluated in sequence.

Lemma 2 (*Canonical reduction*). *Given a labeled skeletal expression Γ, there exists a unique canonical reduction of Γ.*

Proof (*sketch*). By definition, once fixed Γ, and external functions $\mathcal{O}$, Ψ, all exec rules except *sp* are deterministic and do not depend on labels. Fixing $n = 1$ in *sp* rule, also this rule is deterministic; each application reduces the leftmost-labeled skeletal expression, which has not yet reduced to a value. This reduction models the execution that exploits no stream parallelism. Note this is not true for data parallelism, which can be exploited also in the canonic execution. □

Proposition 1 (*Functional semantics confluence*). *Given a labeled skeletal expression Γ, all of its reductions have the same computed stream of its canonical reduction, thus any parallel execution of Γ yields the same computed stream.*

Proof (*sketch*). As described, all rules except *sp* are deterministic. Moreover, each application of *sp* rules reducing n terms can be simulated with n consecutive applications its canonic version, and the two reductions have the same computed stream. As an example for $n = 2$:

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

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

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

The proof can be trivially generalized by induction on the non-canonical reduction derivation. □

Note that the only difference between the two derivations in the proof is the set of transition labels. 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, that is, the output of our skeleton program.

Observe also confluence has been proved for finite stream, i.e. the kind of stream *Lithium* can manage. Other programming environments (as ASSIST [28,29]) are able to manage infinite streams. In this case, the confluence of functional semantics, at least as assumed in this paper, is not a well-defined propriety since the program never stops to consume input and to produce output. However, by relaxing the complete ordering between the two relations $\mapsto^*$ and $\rightarrow^*$, i.e. interleaving the application of unfold and exec rule may enable to obtain similar results. In particular, it is possible to focus on prefixes of output stream and to define on them partial confluence. We argue that different parallel behavior on prefixes of the input/output stream might be claimed to have equivalent functional semantics (where equivalent may be defined by some bisimulation, e.g. ground bisimulation). Actually, this topic is currently under investigation and should be consider as future work.

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* rules 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 a 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 enables 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 different parallel (although equivalent 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 its input 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 a 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 parallel implementations may be modeled.

Labels on transformations are derived from label on streams. Quite intuitively, a processing element must know 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-90s. In [30], 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 that 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 fenv = map (f env)
> where map fx = fx and map fx : rx = (fx) : (map frx).

Darlington's FARM skeleton happens to have the same name as the farm skeleton developed by our group [4,5] and by other groups [7,10] 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 parallelism in the computation of disjunct, completely independent task items. This can be easily expressed using our semantics by observing that the clause:

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

correctly modeling Darlington's FARM also 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:

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

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 Fig. 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 [18,25]). 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 simple one, but it is fundamental in the process of deriving the skeleton "normal form" (NF) [18]. The NF 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 fewer resources and delivers better performance (i.e. $[[\Delta\ \tau]]_f = [[\text{NF}(\Delta)\ \tau]]_f$, $[[\Delta\ \tau]]_p \neq [[NF(\Delta)\ \tau]]_p$). The *Lithium* system may automatically derive and execute the NF of any correct program simply by specifying a proper execution flag in the source code. 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 Fig. 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 true for any skeleton tree Δ. Thus, if Δ = pipe (seq f_1) (seq f_2):

$$[[\text{farm}\ \Delta\ \tau]]_f = [[\Delta\ \tau]]_f \quad [[\text{farm}\ \Delta\ \tau]]_p \neq [[\Delta\ \tau]]_p$$

since, for example, if $\tau = [x_1, x_2, x_3, x_4, x_5, x_6, x_7]$, and the farm exploits round-robin policy with two workers:

$$[[\text{farm}\ \Delta\ \tau]]_p = ❶, ❷ \ ;\ ❶❸, ❷❸, ❶, ❷ \ ;\ ❶❸, ❷❸, ❶, ❷ \ ;\ ❶❸, ❷❸, ❶ \ ;\ ❶❸$$

$$[[\Delta\ \tau]]_p = ❶ \ ;\ ❶, ❶❸ \ ;\ ❶, ❶❸ \ ;\ ❶, ❶❸ \ ;\ ❶, ❶❸ \ ;\ ❶, ❶❸ \ ;\ ❶, ❶❸ \ ;\ ❶❸$$

from which we can easily realize that the two programs exploits both different degrees of parallelism (max. 4 versus 2) and needed execution steps (5 versus 8). As a conclusion, we can state that:

(1) the equivalence “$\mathsf{farm}\ \varDelta \equiv \varDelta$” holds true 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 already discussed in Section 2), as well as most of the other existing skeletons systems [4,7,31,32], that is, it is proposed as a “pure” functional framework. In practice, writing complex applications either a global or local[4] 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 [28,29] 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. Anyway, 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 a 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.

To have a raw idea of how label information is used to implement stateful skeletons, consider the following example: assume a stream of images have to be processed by two filters removing a particular kind of noise. Filter *filter*$_1$ removes red eyes while filter *filter*$_2$ removed scratches. The overall computation can be modeled as a farm with pipeline worker, stage one being *filter*$_1$ and stage two being *filter*$_2$. Let us also suppose that *filter*$_1$ counts for removed red eyes and *filter*$_2$ counts for removed scratches. Therefore a state variable, say n_1, is to be updated by all the computations of *filter*$_1$ on the items of the input stream, and a state variable, say n_2, is to be updated by all the computations of *filter*$_2$.

Now, suppose we only have two computing elements and suppose also that the rule $\mathsf{farm}\ \varDelta \equiv \varDelta$ is used from left to right in such a way that the program actually evaluated is just a two-stage pipeline. In this case, we probably come up with labeling such as:

$$\ldots (\overline{\mathsf{seq}}\ \mathit{filter}_1)(x_{i+1})^{\text{❶}} : (\overline{\mathsf{seq}}\ \mathit{filter}_2)(\mathit{filter}_1(x_i))^{\text{❷}} \ldots$$

In this case, state variable n_1 (n_2) can be implemented locally on the processing element corresponding to label ❶ (n_2) will be performed by code running on that processing element.

In case the $\mathsf{farm}\ \varDelta \equiv \varDelta$ rule is not applied, instead, we could even come on with a labeling such that for all indexes i even:

$$\ldots (\overline{\mathsf{seq}}\ \mathit{filter}_2)((\overline{\mathsf{seq}}\ \mathit{filter}_1)(x_i))^{\text{❶}} : (\overline{\mathsf{seq}}\ \mathit{filter}_2)((\overline{\mathsf{seq}}\ \mathit{filter}_1)(x_{i+1}))^{\text{❷}} \ldots$$

(as an example, this may happen if the oracle function gives twice the same answer each time).

In this case, it is clear that a distributed implementation of both n_1 and n_2 is needed, as *filter*$_1$ and *filter*$_2$ are both computed on the node ❶ and on the node ❷, and therefore n_1 and n_2 have both to be accessed from both processing elements/nodes ❶ and ❷.

[3] This is just one case, of course: in order to complete the proof all the different possible $\varDelta$ must be considered. However, the simple case of $\varDelta = \mathsf{pipe}$ should make it clear how the full proof may work.

[4] W.r.t. the skeleton or to the processing element.

7. Conclusions

We proposed an operational semantics schema that can be used to model both functional and parallel behavior of skeletal programs in a uniform way. This schema is basically a labeled transition system, which is parametric with respect to an oracle function. The oracle function provides the labeled transition system with a configurable label generator establishing a mapping between data and computation and system resources. The features and the properties of our operational semantics schema have been described by modeling the *Lithium* skeleton set, that is, an existing set of commonly agreed skeletons that include both task parallel and data parallel skeletons. In particular, we showed how the operational semantics schema can be used to prove the correctness of skeleton expression rewriting rules, to analyze similarities and differences among skeletons coming from different skeleton frameworks and, eventually, we pointed out how the schema can be used to derive information useful to design parallel implementation of stateful skeletons.

Acknowledgments

Our acknowledgments go to Emilio Tuosto and Andrea Bracciali for many interesting discussions, and to the anonymous referees of the draft version of this paper for their helpful remarks.

References

[1] Aldinucci M, Danelutto M. An operational semantics for skeletons. In: Joubert GR, Nagel WE, Peters FJ, Walter WV, editors. Parallel computing: software technology, algorithms, architectures and applications, PARCO 2003, Advances in parallel computing, vol. 13. Dresden, Germany: Elsevier; 2004, p. 63–70.

[2] Cole M. Algorithmic skeletons: structured management of parallel computations, Research monographs in parallel and distributed computing. London: Pitman; 1989.

[3] Au P, Darlington J, Ghanem M, Guo Y, To H, Yang J. Co-ordinating heterogeneous parallel computation. In: Bouge L, Fraigniaud P, Mignotte A, Robert Y, editors. Proceedings of Euro-Par 1996. Berlin: Springer; 1996. p. 601–14.

[4] Bacci B, Danelutto M, Orlando S, Pelagatti S, Vanneschi M. P^{3}L: a structured high-level programming language and its structured support. Concurrency Practice and Experience 1995;7(3):225–55.

[5] Bacci B, Danelutto M, Pelagatti S, Vanneschi M. SkIE: a heterogeneous environment for HPC applications. Parallel Computing 1999;25(13–14):1827–52.

[6] Pelagatti S. Structured development of parallel programs. London: Taylor & Francis; 1998.

[7] Sérot J, Ginhac D. Skeletons for parallel image processing: an overview of the SKIPPER project. Parallel Computing 2002;28(12):1685–708.

[8] Loulergue F. Distributed evaluation of functional BSP programs. Parallel Processing Letters 2001;4:423–37.

[9] Klusik U, Loogen R, Priebe S, Rubio F. Implementation skeletons in Eden—low-effort parallel programming. In: IFL'00—International workshop on the implementation of functional languages. Lecture notes in computer science, vol. 2011. Aachen, Germany: Springer; 2000. p. 71–88.

[10] Kuchen H. A skeleton library. In: Monien B, Feldmann R, editors. Proceedings of Euro-Par 2002. Lecture notes in computer science, vol. 2400. Berlin: Springer; 2002. p. 620–9.

[11] Cole M. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Computing 2004;30(3): 389–406.

[12] MacDonald S, Anvik J, Bromling S, Schaeffer J, Szafron D, Taa K. From patterns to frameworks to parallel programs. Parallel Computing 2002;28(12):1663–84.

[13] Gava F, Loulergue F. A parallel virtual machine for bulk synchronous parallel ML. In: International conference on computational science (ICCS 2003). Lecture notes in computer science, vol. 2657. Berlin: Springer; 2003. p. 155–64.

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

[15] Cosmo RD, Pelagatti S, Li Z. A calculus for parallel computations over multidimensional dense arrays. Computer Languages, Systems and Structures, 2006, in press, doi: 10.1016/j.cl2006.07.05.

[16] Hidalgo-Herrero M, Ortega-Mallén Y. An operational semantics for the parallel language Eden. Parallel Processing Letters 2002;12(2): 211–28.

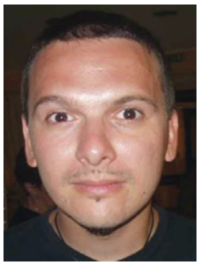
[17] The Ocaml home page, ⟨http://www.ocaml.org⟩; 2005.

[18] Aldinucci M, Danelutto M. Stream parallel skeleton optimization. In: Proceedings of the 11th IASTED international conference on parallel and distributed computing and systems (PDCS'99). Cambridge, MA, USA: IASTED/ACTA Press; 1999. p. 955–62.

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

[20] Gorlatch S, Lengauer C, Wedler C. Optimization rules for programming with collective operations. In: Proceedings of the 13th international parallel processing symposium & 10th symposium on parallel and distributed processing (IPPS/SPDP'99). Silver Spring, MD: IEEE Computer Society Press; 1999. p. 492–9.

[21] Aldinucci M. Automatic program transformation: the meta tool for skeleton-based languages. In: Gorlatch S, Lengauer C, editors. Constructive methods for parallel programming, advances in computation: theory and practice. NY, USA: Nova Science Publishers; 2002. p. 59–78 [chapter 5].
[22] Cole M, Hayashi Y. Static performance prediction of skeletal programs. Parallel Algorithms and Applications 2002;17(1):59–84.
[23] Skillicorn DB, Cai W. A cost calculus for parallel functional programming. Journal of Parallel and Distributed Computing 1995;28:65–83.
[24] Fradet P, Mallet J. Compilation of a specialized functional language for massively parallel computers. Journal of Functional Programming 2000;10(6):561–605.
[25] Aldinucci M, Danelutto M, Teti P. An advanced environment supporting structured parallel programming in Java. Future Generation Computer Systems 2003;19(5):611–26.
[26] Aldinucci M, Danelutto M, Dünnweber J, Gorlatch S. Optimization techniques for skeletons on grid. In: Grandinetti L, editor. Grid computing and new frontiers of high performance processing. Advances in parallel computing, vol. 14. Amsterdam: Elsevier; 2005.
[27] Backus J. Can programming be liberated from the von Neumann style? A functional programming style and its algebra of programs, Communications of the ACM 1978; 21(8):613–41.
[28] Vanneschi M. The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing 2002;28(12):1709–32.
[29] Aldinucci M, Coppola M, Danelutto M, Vanneschi M, Zoccolo C. ASSIST as a research framework for high-performance Grid programming environments. In: Cunha JC, Rana OF, editors. Grid computing: software environments and tools. Berlin: Springer; 2007. p. 230–56 [Chapter 10].
[30] Darlington J, To HW. Building parallel applications without programming. In: Leeds workshop on abstract parallel machine models; 1993.
[31] Darlington J, Field AJ, Harrison PG, Kelly PHJ, Sharp DWN, While RL, et al. Parallel programming using skeleton functions. In: Bode A, Reeve M, Wolf G, editors. Proceedings of the parallel architectures and languages Europe. Lecture notes in computer science, vol. 694. Berlin: Springer; 1993.
[32] MacDonald S, Szafron D, Schaeffer J, Bromling S. Generating parallel program frameworks from parallel design patterns. In: Bode A, Ludwing T, Karl W, Wismüller R, editors. Proceedings of Euro-Par 2000. Lecture notes in computer science, vol. 1900. Berlin: Springer; 2000, p. 95–105.



Marco Aldinucci got the Ph.D. in Computer Science in 2003, he has been researcher at the Institute of Information Science and Technologies of the Italian National Research Council (ISTI-CNR, 2003–2006), and he is currently research associate at Computer Science Department of the University of Pisa, Italy. He is author of about 40 papers appearing in journals and international refereed conference proceedings, together with more than 15 different co-authors. He has been and is currently participating in more than 10 national and international research projects concerning parallel computing and Grid topics, including the Grid.it Italian national project, CoreGRID EC Network of Excellence, GridComp EC-STREP, BEinGRID EC-IP, XtreemOS EC-IP, GridCoord EC-SSA.
His main research is focused on parallel/distributed computing in network of workstations and grids, and in particular on models and tools for high-level parallel programming, component-based frameworks, autonomic computing, and distributed shared memory systems. He contributed to the design and the development of a number of tools for parallel processing, including compilers, libraries and frameworks, both in industrial and academic teams.

Marco Danelutto received the Ph.D. in Computer Science in 1990 and since 1998 he is an associate professor at the Department of Computer Science of the University of Pisa.
His main research interests are in the field of parallel, distributed and grid architectures and in the design and implementation of structured parallel programming environments for such kind of architectures. In the 1990s, he has been one of the main designers of P3L, the Pisa Parallel Programming Language, a skeleton-based structured parallel programming environment targeting clusters and workstation networks. Recently, he actively participated in the development of the ASSIST programming environment and he currently maintains the muskel Java skeleton library. He is author of more than 80 scientific publications on international journals and conferences.
He is member of the program committees of several conferences, including Europar, that he organized and co-chaired in 2004. In the recent years, he has been the national coordinator of the activities of work package 8 in the GRID.it Italian national project, aimed at developing the ASSIST component-based parallel programming environment, and he is currently leading the Programming model Institute of the EU CoreGRID network of excellence and member of the CoreGRID executive committee.