New Implementation of a BSP Composition Primitive with Application to the Implementation of Algorithmic Skeletons

Frédéric Gava
Laboratory of Algorithms, Complexity and Logic (LACL)
University of Paris-East
Créteil-Paris, France
Email: gava@univ-paris12.fr

Ilias Garnier
LIST laboratory
CEA Saclay, Essonne, France
Email: ilias.garnier@cea.fr

Abstract

BSML is a ML based language designed to code Bulk Synchronous Parallel (BSP) algorithms. It allows an estimation of execution time, avoids deadlocks and non-determinism. BSML proposes an extension of ML programming with a small set of primitives. One of these primitives, called parallel superposition, allows the parallel composition of two BSP programs. Nevertheless, its past implementation used system threads and have unjustified limitations. This paper presents a new implementation of this primitive based on a continuation-passing-style (CPS) transformation guided by a flow analysis. To test it and show its usefulness, we also have implemented the OCamlP3l algorithmic skeletons and compared their efficiencies with the original ones.

1. Introduction

Generalities. Since the paper "Go To Statement Considered Harmful", structured sequential programming is the norm. It is surprising to see that it is absolutely not the case for parallel programming [1]. Besides compiler-driven automatic parallelisation, programmers have kept the habit of using low-level parallel routines (as send/receive of MPI/PVM) or concurrent languages. In this way, they, less or more, managed the communications with the usual problems of (un)buffered or (un)blocking sending, which are source of deadlocks and non-determism[1]. Furthermore, programmers forbid optimisations that could be done if high-level structures as collective operators or skeletons [2] were to be used instead. High-level languages and tools are thus needed but sadly rarely used. The main reason of this fact is that they do generally not offer a sufficiently wide set of parallel structures for a practical and efficient programming.

The design of parallel programming languages is thus a tradeoff between the possibility for the programmer to control parallel aspects necessary for predictable efficiency (but which make programs more difficult to write, to prove correct and to port) and the abstraction of such features which are necessary to make parallel programming easier (but which hampers efficiency and performance prediction).

BSP framework. BSP[2] is a parallel model which offers a high degree of abstraction and allows an estimation of the execution time of its algorithms on a wide variety of architectures. BSML is an extension of ML to code this kind of algorithms using a a small set of primitives which are currently implemented as a parallel library[3] for the ML programming language Objective Caml (OCaml). Using a safe high-level language as ML to program BSP algorithms allows performance, scalability and expressivity.

One of the primitive, called superposition [4], is dedicated to the parallel composition of BSML expressions (notably for divide-and-conquer algorithms) without any need of subset synchronisation[4]. It is based on sequentially interleaved threads of BSP computations, called *super-threads* [5]. Informally, it is equivalent to pairing in BSML.

It was show in [6] that the parallel superposition is not only useful to divide-and-conquer BSP algorithms. This primitive can be used many times simultaneously in a single program and an efficient implementation is thus needed. To ensure a deterministic execution[5] of BSML programs, the semantics of the superposition forces us to have, at any time, only one active super-thread.

Currently, the super-threads are implemented over the system threads [7]. That limits the number of such threads and it leads to efficiency problems with OCaml. This restriction is unnecessary and to overcome this limitation, we present another implementation which uses a global continuation-passing-style (CPS) transformation of BSML programs.

CPS is a classic style of programming in which control

1. These properties are justified for concurrent computations but clearly not for parallel algorithms, i.e, high-performance applications.

2. We refer to [3] for a gentle introduction to the BSP model.

3. http://bsmllib.free.fr

4. Subset synchronisation of processors is usually justified by the necessity of the recursive decomposition of the computation into independent sub-problems; [5] argues that it is not really useful for BSP computing.

5. Determinism guarantees that program behaviour is identical on all nodes; this essentially eliminates an entire class of errors: data races.

is passed explicitly in the form of a continuation [8]. Instead of "returning" values, a function takes an extra argument, the continuation which represents what should be done with the result of the function and then passes it to another function. Programs can be systematically translated to semantically equivalent programs in CPS using a variety of algorithms [9]. As a programming device, CPS enables programmers to define advanced, application-specific control structures [10] such as co-routines [11], [12].

We followed a pragmatic approach in the design of our global CPS transformation and efficiency was one of our major concern[6]. Currently it works on a large subset of OCaml without objects, labels and functors. This transformation is also guided by a data flow analysis. Indeed, the superposition is transformed into a CPS construct, whereas most of the ML code does not have to be modified.

To benchmark our transformation, we have applied it to the implementation of algorithmic skeletons, those of the OCamlP3l [13] language[7]. Algorithmic skeletons languages are generally defined by introducing a limited set of parallel patterns to be composed in order to build easily a fully parallel application (see [2] for a survey). Even if the implementation is less efficient compared to dedicated skeletons languages (or a MPI send/receive implementation), the programmer can compose skeletons when it is natural for him and use a BSP programming style when it is necessary. Furthermore, as a performance test of our transformation, the implementation of skeletons have the advantage to generate an important number of super-threads.

Outline. First, we briefly review in Section 2 the BSP model, the BSML language and the past implementation with its restrictions. We give the semantics results of the CPS transformation in Section 3 and Section 4 is devoted to the implementation. Section 5 is dedicated to the benchmark of an implementation of OCamlP3l's skeletons using this transformation. Related work is discussed in Section 6. We end with conclusion and future works (Section 7).

2. Functional BSP programming

The Bulk-Synchronous Parallel Model. A BSP program is executed as a sequence of *super-steps* (see left scheme in Fig. 1), each one divided into three successive disjoint phases: each processor only uses its local data to perform sequential computations and to request data transfers to other nodes; the network delivers the requested data; a global synchronisation barrier occurs, making the transferred data available for the next super-step.

6. We know that OCaml code cannot be interrupted, so by adding an appropriate CPS in OCaml, we do not have to introduce an inefficient mechanism to save the execution context.

7. http://ocamlp3l.inria.fr/

Bulk-Synchronous Parallel ML (BSML). BSML is based on 8 primitives, three of which are used to access the BSP parameters of the machine and only four dedicated to BSP asynchronous phases and communications.

The last primitive called **super** (parallel superposition) allows the evaluation of two BSML expressions as interleaved threads of BSP computations called super-threads. From the programmer's point of view, the semantics of the superposition is the same as pairing but the evaluation of **super** $Prog_1$ $Prog_2$ is different (see right scheme in Fig. 1). In the left, pairing is just the sequential evaluation of the the two programs. Using the superposition, the phases of asynchronous computations of $Prog_1$ and $Prog_2$ occur; then the communication phase of the first program is merged with the second one and only one barrier occurs.

The parallel superposition is thus less costly than the evaluation of $Prog_1$ followed by the evaluation of $Prog_2$. These primitives constitute the core of BSML. Obviously, it is possible to define some useful functions.

Older implementation. In ML like languages, it is straightforward to add imperative features that can introduce non-deterministic results (deadlocks) in BSML. To avoid this, a strategy for the choice of the unique active super-thread has been added [7]: the active super-thread is evaluated until it ends its computations or it needs communications. When communications are done, the first super-thread which has finished "its past super-step" is re-evaluated, i.e., it becomes the new current active super-thread.

Currently, based on a semantics study, the superposition is implemented using system threads [7]. Each time a superposition is called, a new thread is created and share locks are used each time a communication primitive is called.

There is two drawbacks to this method. First, threads slows down the running of a OCaml program: a global lock is used due to the GC nature of OCaml. Second, many OS have a maximal number of possible threads (*e.g.* 1024 for OCaml in many Linux systems). That limits the use of this primitive if a greater number of super-threads than this maximal number are run simultaneously. These limitations would quickly depreciate the interest of this primitive. We now present another implementation which uses a global continuation-passing-style transformation.

3. CPS transformation and flow analysis

First, we present our ML-like core source language with the adjunction of two concurrency primitives: **yield** and **super**. We then proceed to the definition of the transformation to the target language, which is the same as the source minus the concurrency primitives. Here, **yield** replaces communication primitives, abstracting away communication handling. **yield** suspends the currently executing super-thread (called thread in the next) and schedules the execution of the next thread, as defined by the **super** operational semantics. Other

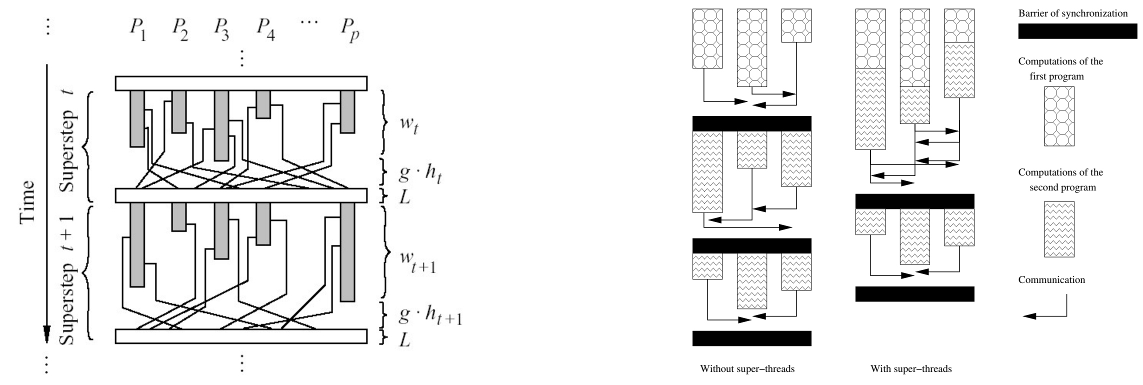


Figure 1. The model of execution of BSP (left) and of the superposition (right)

BSML primitives are ignored, they are orthogonal to the following work.

3.1. Semantics of the transformation

Expressions are as follow:

e ::=	x	*variables*
|	c	*constants*
|	$\lambda v.e$	*functional values*
|	**fix** $f\,\lambda x.e$	*recursive functions*
|	$e_1\, e_2$	*applications*
|	**let** $v = e_1$ **in** e_2	*local definitions*
|	(e_1, e_2)	*couples*
|	$\kappa\ e$	*constructor application*
|	**match** e **with** m_1 | ... | m_n	*pattern matching*
|	**op** $e_1\ e_2$	*arithmetic operators*
|	**super** $e_1\ e_2$	*superposition*
|	**yield**	*simulates* **put**
m ::=	$\kappa\, x \rightarrow e$	*matching branch*

Monads allow to extend a language while enforcing a correct operational behaviour [14]. A monad is the data of three primitives: run, ret and $bind$, operating on a type $M\,\alpha$. The run primitive has type $\forall\,\alpha.M\,\alpha \rightarrow \alpha$ and executes a monadic program. ret, of type $\forall\,\alpha.\alpha \rightarrow M\,\alpha$ transforming a base value into a monadic one. Finally, $bind$ allows chaining monadic computations as reflected by it's type $\forall\,\alpha, \beta.M\,\alpha \rightarrow (\alpha \rightarrow M\,\beta) \rightarrow M\,\beta$.

Our threads are modelled as resumptions, meaning that they are in a suspended state or terminated:

type α thread=Terminated **of** α | Waiting **of** (unit$\rightarrow\alpha$ thread)

The monadic type is always *thread*:

$$M\,\alpha = \forall\,\beta.(\alpha \rightarrow thread\,\beta) \rightarrow thread\,\beta.$$

The monadic primitives are thus defined as follow:

$$\begin{array}{lcl}
\textbf{ret}\ \mathrm{x} & = & \lambda k.kx \\
\textbf{bind}\ \mathrm{m\ f} & = & \lambda k.\mathrm{m}(\lambda v.fvk) \\
\textbf{run} & = & \lambda x.((\textbf{fix}\ loop\,\lambda t.\ \textbf{match}\ t\ \textbf{with} \\
\multicolumn{3}{l}{\quad \mid\ Terminated\ x \rightarrow x} \\
\multicolumn{3}{l}{\quad \mid\ Waiting\ s \rightarrow loop\ (s\ ()))\ (x\ (\lambda x.Terminated\ x)))}
\end{array}$$

They must at least satisfy these three monadic laws:

$$\begin{array}{lcl}
\mathbf{bind}\ (\mathbf{ret}\ a)\ f & \approx & f\ a \\
\mathbf{bind}\ a\ \lambda x.\mathbf{ret}\ x & \approx & a \\
\mathbf{bind}\ (\mathbf{bind}\ a\ (\lambda x.b))\ (\lambda y.c) & \approx & \mathbf{bind}\ a\ (\lambda x.\mathbf{bind}\ b\ (\lambda y.c))
\end{array}$$

where $\approx$ is defined as $\forall a_1, a_2, k\, \exists a\, (a_1\, k{\Rightarrow}a){\wedge}(a_2\, k{\Rightarrow}a)$ where $\Rightarrow$ is a simple big-step semantics [15]. In our case, these laws were mechanically proved using the Coq proof assistant (see [15] for the proof script).

We now straightforwardly proceed to the definition of the monadic transformation on expressions $T_0[\![e]\!]$, defined in Fig. 2 (left part). The two concurrency primitives are then defined using first class continuations in Fig. 2 (right part) where $a \,@\, b \equiv$ **bind** $a\ (\lambda v_a.$**bind** $b\ (\lambda v_b.v_a\, v_b)))$.

The operational behaviour of these primitives is clear: **yield** captures it's own continuation, and stores it into a suspension for further evaluation; **super** first suspends its own execution (using **yield**), then schedules the execution of it's two sub-threads until they are terminated.

Theorem 1: If $e \Rightarrow v$ then $T_0[\![e]\!] \approx$ **ret** v.
The proof can be found in [15].

3.2. Flow analysis for performance issues

The full transformation of a program to CPS considerably impedes performance. This overhead is usually alleviated using transformation-time reductions to eliminate the so-called "administrative redexes" on the programs.

However, that does not suffice. Aiming at numerical computing, we can not afford to transform unnecessary expressions. Observing how some very limited parts of the program need continuations, it seems natural to try to convert only the required expressions (in our case, only **yield** and **super** need them). We thus need a partial CPS transformation [16]. The expressions to be transformed are those susceptible to reduce a **yield** or **super** expression.

Since we must cope with higher-order functions, the partial CPS transformation is guided by a flow analysis which yields a straightforward flow inference algorithm whose purpose is to decide if an expression is susceptible to reduce a **yield**: we tag it as *impure* (*pure* otherwise). Our type system is derived from the type system for CFA defined in [17] and the rules can be found in Fig. 3 where τ are classical ML types, and flows F ::= $\mathcal{P}$ | $\mathcal{I}$ (pure or impure where $\mathcal{I} < \mathcal{P}$). More details can be found in [15].

$$\begin{array}{lcl}
T_0\llbracket x\rrbracket & = & \textbf{ret}\ \ x \\
T_0\llbracket c\rrbracket & = & \textbf{ret}\ \ c \\
T_0\llbracket \lambda v.e\rrbracket & = & \textbf{ret}\ \ \lambda v.T_0\llbracket e\rrbracket \\
T_0\llbracket \textbf{fix}\ f\,\lambda x.e\rrbracket & = & \textbf{ret}\ \ (\textbf{fix}\ f\,\lambda x.T_0\llbracket e\rrbracket) \\
T_0\llbracket e_1\,e_2\rrbracket & = & \textbf{bind}\ \ T_0\llbracket e_1\rrbracket\ (\lambda v_1.\textbf{bind}\ \ T_0\llbracket e_2\rrbracket\ (\lambda v_2.v_1\,v_2)) \\
T_0\llbracket \textbf{let}\,v\ =\ e_1\ \textbf{in}\ e_2\rrbracket & = & \textbf{bind}\ \ T_0\llbracket e_1\rrbracket\ (\lambda v.T_0\llbracket e_2\rrbracket) \\
T_0\llbracket (e_1,e_2)\rrbracket & = & \textbf{bind}\ \ T_0\llbracket e_1\rrbracket\ (\lambda v_1.\textbf{bind}\ \ T_0\llbracket e_2\rrbracket\ (\lambda v_2.(v_1,v_2))) \\
T_0\llbracket \kappa\ e\rrbracket & = & \textbf{bind}\ \ T_0\llbracket e\rrbracket\ (\lambda v_e.\kappa\ v_e) \\
T_0\llbracket \textbf{match}\ e\ \textbf{with} & & \\
\quad \mid \kappa_i\ x_i \rightarrow\ e_i\ \rrbracket & = & \textbf{bind}\ T_0\llbracket e\rrbracket\ (\lambda v_e.\textbf{match}\ v_e\ \textbf{with} \mid \kappa_i\ x_i \rightarrow\ T_0\llbracket e_i\rrbracket) \\
T_0\llbracket \textbf{op}\ e_1\ e_2\rrbracket & = & \textbf{bind}\ \ T_0\llbracket e_1\rrbracket\ (\lambda v_1.\textbf{bind}\ \ T_0\llbracket e_2\rrbracket\ (\lambda v_2.op\ v_1\ v_2))
\end{array}$$

$$\begin{array}{lcl}
\textbf{yield} & = & \lambda k.Waiting\ k \\
 & & \\
\textbf{super} & = & \textbf{let}\ loop\ =\ \textbf{fix}\ loop\,\lambda r_1.\lambda r_2. \\
 & & \textbf{bind yield}\ (\lambda\,()\ .\textbf{match}\ (r_1,\ r_2)\ \textbf{with} \\
\multicolumn{3}{l}{\mid\ (Terminated\,x_1,\ Terminated\,x_2)\ \rightarrow \textbf{ret}\ (x1,\ x2)} \\
\multicolumn{3}{l}{\mid\ (Terminated\,\_,\ Waiting\,s)\ \rightarrow loop\ r_1\ (s\ ())} \\
\multicolumn{3}{l}{\mid\ (Waiting\,s,\ Terminated\,\_)\ \rightarrow loop\ (s\ ())\ r_2} \\
\multicolumn{3}{l}{\mid\ (Waiting\,s_1,\ Waiting\,s_2) \rightarrow loop\ (s_1\ ())\ (s_2\ ()))\ \textbf{in}} \\
\multicolumn{3}{l}{\ \textbf{ret}\ \lambda f_1.\textbf{ret}\ \lambda f_2.} \\
\multicolumn{3}{l}{\quad \textbf{let}\,r_1\ =\ ((\textbf{ret}\ \ f_1)\,@\,(\textbf{ret}\ \ ()))\ (\lambda x.Terminated\,x)\ \textbf{in}} \\
\multicolumn{3}{l}{\quad \textbf{let}\,r_2\ =\ ((\textbf{ret}\ \ f_2)\,@\,(\textbf{ret}\ \ ()))\ (\lambda x.Terminated\,x)\ \textbf{in}} \\
\multicolumn{3}{l}{\qquad loop\ r_1\ r_2}
\end{array}$$

Figure 2. Monadic transformation

The flow-directed partial CPS transformation aims to preserve "pure" expressions, while CPS-converting "impure" ones. To preserve the operational equivalence, the code must be generated between CPS and non-CPS terms.

The partial transformation is quite simple, parallel primitives are directly replaced by their definitions, and the primitive operators are always pure. Therefore pure expressions are preserved from being transformed. On large real-world programs, most of the computation takes place in pure expressions, making the CPS part less of a burden:

$$\begin{array}{lcl}
T_1\llbracket x\rrbracket & = & \textbf{ret}\ \ x \\
T_1\llbracket c\rrbracket & = & \textbf{ret}\ \ c \\
T_1\llbracket \lambda v.e^{\mathcal{I}}\rrbracket & = & \textbf{ret}\ \ \lambda v.T_1\llbracket e\rrbracket \\
T_1\llbracket (\textbf{fix}\ h\,\lambda x.e)^{\mathcal{I}}\rrbracket & = & \textbf{ret}\ \ \textbf{fix}\ h\,\lambda x.T_1\llbracket e\rrbracket \\
T_1\llbracket (e_1\ e_2^{\mathcal{I}})^{\mathcal{I}}\rrbracket & = & \textbf{bind}\ \ T_1\llbracket e_1\rrbracket\ (\lambda v_1.\textbf{bind}\ \ T_1\llbracket e_2\rrbracket\ (\lambda v_2.v_1 v_2)) \\
T_1\llbracket (e_1\ e_2^{\mathcal{P}})^{\mathcal{I}}\rrbracket & = & T_1\llbracket e_1\rrbracket(\textbf{ret}\ \ e_2) \\
T_1\llbracket (\textbf{let}\,v\ =\ e_1^{\mathcal{I}}\ \textbf{in}\ e_2)^{\mathcal{I}}\rrbracket & = & \textbf{bind}\ \ T_1\llbracket e_1\rrbracket\ (\lambda v.T_1\llbracket e_2\rrbracket) \\
T_1\llbracket (\textbf{let}\,v\ =\ e_1^{\mathcal{P}}\ \textbf{in}\ e_2)^{\mathcal{I}}\rrbracket & = & \textbf{let}\,v\ =\ e_1\ \textbf{in}\ T_1\llbracket e_2\rrbracket \\
T_1\llbracket (e_1,\ e_2)^{\mathcal{I}}\rrbracket & = & \textbf{bind}\ \ T_1\llbracket e_1\rrbracket\ (\lambda v_1.\textbf{bind}\ \ T_1\llbracket e_2\rrbracket\ (\lambda v_2.\textbf{ret}\,(v_1,v_2))) \\
T_1\llbracket (\kappa\,e)^{\mathcal{I}}\rrbracket & = & \textbf{bind}\ \ T_1\llbracket e\rrbracket\ (\lambda v_e.\textbf{ret}\ \ \kappa\,v_e) \\
T_1\llbracket \textbf{match}\ e\ \textbf{with} & & \\
\quad \mid \kappa_i\ x_i \rightarrow\ e_i\ \rrbracket & = & \textbf{bind}\ \ T_1\llbracket e\rrbracket\ (\lambda v_e.\textbf{match}\ v_e\ \textbf{with} \\
 & & \qquad\qquad \mid \kappa_i\ x_i \rightarrow\ T_1\llbracket e_i\rrbracket) \\
T_1\llbracket e^{\mathcal{P}}\rrbracket & = & \textbf{ret}\ \ e
\end{array}$$

We observe that an impure expression is never embedded into a pure one. This property is induced by the type system: if any sub-expression e_i of an expression e is impure, so is e. We use this fact in the transformation: when encountering a pure expression, we simply wrap it into a **ret**.

The soundness proof for the partial CPS transformation is given in [16] and lies on the aforementioned consistency assumption. Our type system enforces that all variables bound to the same binder have the same flow, ensuring the soundness of our framework. This allows us to prove some useful lemmas on substitutions that make the following soundness theorem provable:

Theorem 2: If $t \Rightarrow v$ then $T_1\llbracket t\rrbracket \approx \textbf{ret}\ \ v$.
Proofs can be found in [15].

4. New implementation of the superposition

For lack of space, we do not give all the details. They can be found at [15] and full implementation is available at http://lacl.univ-paris12.fr/gava/cps-super-bsml-comp.tar.gz. Currently our implementation works on a large subset of OCaml without objects, labels and functors.

Imperative features. We did not treat imperative features in this report, suffice to say that every expression involved in an imperative operation is constrained to have the flow of its sub-expressions. When encountering an impure loop, we must convert it into its tail-recursive equivalent form. OCaml handles tail-recursion fine, so there is no added risk of stack overflow. The soundness of the transformation was not proved in the presence of imperative features, but we have not encountered any problems with them so far.

Defunctorisation. The OCaml language provides parametric modularity (known as *functors*), but our transformation does not handle them. In order to apply our transformation, we have to defunctorise the whole program. To this end, we use the already existing **Ocamldefun**[8] program. A nice side-effect is the increased possibilities in inlining and code specialisation by the back-end compiler (OCaml).

Monomorphisation. Our partial CPS transformation needs simple types. Thus, we need to monomorphise the whole program. After type inference, the syntax tree is annotated with either ground types or type schemes, which are introduced only at **let** bindings. Each of these bindings is possibly instantiated with different types. Monomorphisation is the act of duplicating these bindings for each instantiation type (duplicating polymorphically typed functions for each needed domain type). We must also specialise type declarations to take into account impure functions: we scan the whole program, registering each type used inside algebraic data constructors or records and instantiating declarations accordingly. We must then perform a topological sort to take into account the fact that a polymorphic type may be used with a type declared after. Monomorphisation can potentially make the size of the program grow exponentially, but actual implementations (as MLton[9]) shows that practically, the size growth is manageable (about 30 %).

Monoflowisation. In fine, to maximize the efficiency of the generated code, we use a similar process for flows: instead of duplicating functions based on types, we duplicate

8. http://www.lri.fr/~signoles/ocamldefun/index.fr.html
9. http://mlton.org/

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash c : \langle \mathcal{P}, \mathit{const\_type} \rangle} \qquad \frac{}{\Gamma \vdash \mathit{op} : \tau_{op}} \qquad \frac{\Gamma, x : \tau_1 \vdash z : \tau_2 \quad \text{if } f = x \vee z}{\Gamma \vdash \lambda x.z : \langle f, \tau_1 \rightarrow \tau_2 \rangle}$$

$$\frac{\Gamma \vdash z \qquad \Gamma \vdash z' : \tau_1 \quad \text{if } \mathit{annot}(z) = \tau_1 \rightarrow \tau_2 \text{ and } f = z \vee z'}{\Gamma \vdash (z\, z') : \langle f, \mathit{annot}(\tau_2) \rangle} \qquad \frac{\Gamma \vdash z : \tau_1 \qquad \Gamma \vdash z' : \tau_2 \quad \text{if } if = z \vee z'}{\Gamma \vdash (z, z') : \langle f, \tau_1 * \tau_2 \rangle}$$

$$\frac{\Gamma \vdash z : \tau_1 \qquad \Gamma, x : \tau_1 \vdash z' : \tau_2 \quad \text{if } \mathit{flow}(\tau_2) \leq_F \mathit{flow}(\tau_1)}{\Gamma \vdash \textbf{let}\, x \;=\; z \textbf{ in } z' : \tau_2} \qquad \frac{\Gamma, h : \langle f, \tau_0 \rightarrow \tau_1 \rangle, x : \tau_0 \vdash z : \tau_1 \quad \text{if } f = x \vee z}{\Gamma \vdash \textbf{fix } h\, \lambda x.z : \langle f, \tau_0 \rightarrow \tau_1 \rangle}$$

$$\frac{\Gamma \vdash e : \tau_\kappa \qquad \kappa : \tau_\kappa \rightarrow \langle \mathit{flow}(\tau_\kappa), \mu\, \alpha \,.\, \ldots + \tau_\kappa + \ldots \rangle}{\Gamma \vdash \kappa\, e : \langle \mathit{flow}(\tau_\kappa), \mu\, \alpha \,.\, \ldots + \tau_\kappa + \ldots \rangle} \qquad \frac{\Gamma \vdash e : \langle F, \mu\, \alpha \,.\, \tau_{\kappa_1} + \ldots + \tau_{\kappa_n} \rangle \qquad \Gamma, x_i : \tau_{\kappa_i} \vdash e_i : \tau \qquad i \in [1..n]}{\Gamma \vdash \textbf{match } e \textbf{ with } \mid \kappa_i\, x_i \;\rightarrow\; e_i : \tau}$$

$$\frac{\text{if } f = \mathit{min}(\mathit{flow}(\tau_0), \mathit{flow}(\tau_1))}{\Gamma \vdash \textbf{super} : \langle \mathcal{I}, \langle \mathcal{I}, \langle \mathcal{P}, \mathit{unit} \rangle \rightarrow \tau_0 \rangle \rightarrow \langle \mathcal{I}, \langle \mathcal{I}, \langle \mathcal{P}, \mathit{unit} \rangle \rightarrow \tau_1 \rangle \rightarrow \langle f, \tau_0 * \tau_1 \rangle \rangle \rangle} \qquad \frac{}{\Gamma \vdash \textbf{yield} : \langle \mathcal{I}, \mathit{unit} \rangle}$$

Figure 3. Inference rules

them based on flows. This is of utmost importance for widely used functions: if they are used with an impure argument throughout the code, they are flagged as impure for *every* call site (even with pure arguments). This is a consequence of our flow analysis being monovariant. Actually, the monoflowisation is performed directly during the monomorphisation. Another solution would be to use a polyvariant flow analysis, but it would be extremely heavy, both in algorithmic complexity and in implementation.

Polymorphic type inference. Monomorphisation operates on a typed source tree. To this end, we extended the type system defined in [15] to handle a caml-like language. Instead of modifying OCaml's type inference code, we chose to code from scratch a full-blown type inference system, handling let-polymorphism. Drawing upon [18], we decided to use a constraint-based inference algorithm. We use the (non-relaxed) value restriction to ensure the soundness of our analysis in presence of references.

As in [18], polymorphism is handled using constrained type schemes, whose meaning is roughly the set of all ground types admitted by the underlying expression. The constraint generation algorithm is defined inductively on expressions and is a quite natural encoding of the typing rules into the constraint language. This is no surprise since our type system is syntax-directed. It is also parametrised by the expected type of the expression. A formal definition of this algorithm can be found in [15].

Code duplication. Since we are typing the whole program, we know exactly each instantiation type for each binding, allowing us to create as many ground versions of them as we need. In order to avoid variable capture problem, we bind each specialised code to a fresh name, and update the instantiation points accordingly. The freshness is ensured by performing an alpha-conversion pass on the whole program after type inference and before monomorph(*flow*)isation. The instantiation graph stays valid, since we rely on unique ids. The duplication algorithm is simple: when encountering a let binding **let** $v_i = e$ **in** ... we instantiate the code for each node (i, τ) in the instantiation graph $\mathcal{G}$. The e expression must also be recursively monomorph(*flow*)ised, each point j in e being instantiated with the type $\mathcal{G}\ (i, \tau)\ j$.

Partial CPS transformation. Once the program is transformed into simply-typed form, we can apply the partial CPS transformation, as defined earlier. But the standard CPS transformation is known to generate too many administrative redexes, which may greatly hamper the performance of the resulting program. To avoid them, we use the optimizing transformation defined in [9].

5. Application to algorithmic skeletons

Anyone can observe that many parallel algorithms can be characterised and classified by their adherence to a small number of generic patterns of computation (farm, pipe, *etc.*). Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit with specifications which transcend architectural variations but implementations which recognise them to enhance performance [2]. The core principle of skeletal programming is conceptually straightforward. Its simplicity is a strength.

A well know disadvantage of skeleton languages is that the only admitted parallelism is usually that of skeletons while many parallel applications are not obviously expressible as instances of skeletons. Skeletons languages must be constructed as to allow the integration of skeletal and ad-hoc parallelism in a well defined way [2].

In this light, having skeletons in BSML would have the advantage of the BSP pattern of communications (collective ones) and the expressivity of the skeleton approach. For our purpose and to have interesting benchmarks, we take for model the implementation of the OCamlP3l skeletons language (P3L's set of skeletons for OCaml) and base them on our parallel superposition primitive.

```
val seq : (unit →α →β ) →unit →α stream →β stream
val loop:(α →bool)∗(unit→α stream→α stream)→unit→α strea→α stream
val farm : (unit →β stream →γ stream) ∗ int →unit →β stream →γ stream
val pipe (|||) : (unit →α stream →β stream) →
                 (unit →β stream →γ stream) →unit →α stream →γ stream
val mapvector : (unit →β stream →γ stream) ∗ int →unit →
                  β array stream →γ array stream
val reducevector : (unit →(β  ∗ β ) stream →β stream) ∗ int →unit →
                    β array stream →β stream
```

Figure 4. The types of the OCamlP3l skeletons

5.1. The OCamlP3l Skeletons

Fig. 4 [13] resumes the ML type of the OCamlP3l skeletons. We can describe the skeletons as follow.

The **seq** skeleton encapsulates an OCaml function f into a stream process which applies f to all the inputs received on the input stream and sends off the results on the output stream. **loop** computes a function f over all the elements of its input stream until a boolean condition g is verified.

The **farm** skeleton computes in parallel a function f over different data items appearing in its input stream. Parallelism is gained by having k independent processes.

The **pipeline** skeleton performs in parallel the computations relative to different stages of a function composition over different data items of the input stream.

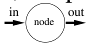
mapvector computes in parallel a function over all the data items of a vector, generating the new vector of the results. The **reducevector** works in the same manner but doing an array folding with a binary operator as argument.

5.2. BSML Implementation

There are already some advantages to using BSML-based skeletons: BSML can be used on a wide variety of communication libraries, such as PUB, MPI and TCP/IP whereas OCamlP3l is currently stuck with TCP/IP.

For simplicity, we generate the program in meta fashion as a simple string, we will use a MetaOCaml-like syntax: the meta-code will be quoted between ".<code>.". We do not present how all skeletons are implemented and utility functions. We refer to [15] for more details.

Execution of process networks. The combination of P3L's skeletons generates a process network. This network takes in input a stream of data. Then each datum is transformed by the network independently of other data and finally the output is another stream of data of the same arity. In this way, they can be composed. Thus, if we suppose that the stream contains n data, the execution of the network will be composed n times using the superposition.

For each execution of a network, we used a counter (place) that stores the placement of tasks and data in a round robin fashion. We then define a triplet which represent the network (input CPU, output CPU and the parallel stream node computation): 

where "node" is a function that takes a data from CPU "in" and return a data to CPU "out".

That will be implemented in BSML as a triplet where the P3L stream is implemented as a parallel vector of option values where only one processor keeps a non empty value (the data of the stream). The full stream is thus a list of these vectors. Now, to produce the process network we recursively generated a BSML code from the skeleton expression.

Implementation of seq(f). The generated code is the network (pl,pl,(**fun** data →new_data)) where the OCaml function f only executes itself on the designated CPU pl (designated by the counter), returning None elsewhere.

Implementation of farm(n,s). Because we have a fix number p of processors, we ignore the n parameter which represent the "number of workers". The parallelism degree is thus all the time p. In this way, the code generated for **farm**(n, s) is simply the code generated for the skeleton s.

Implementation of mapvector(n,s). This skeleton is probably the most interesting one. Once again, the parallelism degree n is unused. The method is as follow.

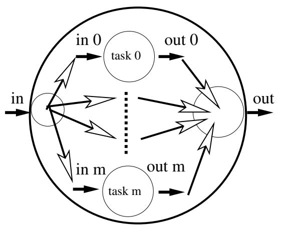
First, a new task is dynamically created for each element of the input vector of the stream and stored in a list of tasks, called ntasks. The BSML code for these tasks (produced by the sub-skeleton s) is generated from an inductive call.

Then, once all the tasks created, their execution are *superposed* using the superposition (**super_list** utility function). For each execution, the input processor of the network send a data of the vector to the processor that have been dynamically designated to execute the sub-network. The parallelism arises from data being distributed over all superposed tasks.

Finally, we gather the results to the network output processor (**rebuild** utility function). This is exactly what is reflected in the code:

```
let pl=(!place) in (pl,pl,(fun data →let ntasks = ref [] in
 let size = noSome ((proj (applyat pl (fun t →
Some (Array.length (noSome t))) (fun _ →Some 0) data)) pl) in
  for j=0 to (size−1) do
   incr_place ();
   let i,o,task= .<BSML code from s>. in
   let new_task= (fun () →sendto o pl (task (sendto pl i (parfun
                          (function Some t→Some t.(j) | None→None) data)))) in
    ntasks:=new_task::(!ntasks);
  done;
  rebuild pl (super_list !ntasks)))
```

This figure subsumes the idea of the implementation:



where m is the size of the input vector. This skeleton is a good sample: the size of typical data (arrays) would make the past implementation (using system threads) of the superposition unusable in practice.

5.3. Benchmark

Our example is a parallel PDE solver which works on a set of subdomains, taken from [13]. On each subdomain it applies a fast Poisson solver written in C. The skeleton

```
let PDE_solver =
  parfun (fun () →
    (loop ((fun (v,continue) →continue),
           seq(fun _ →fun (v,_) →v)
  ||| mapvector(seq(fun _ →compute_sub_domain),3)
  ||| seq(fun _ →projection) ||| seq(fun _ →bicgstab) ||| seq(fun _ →plot))))
```

Figure 5. Skeleton code fragment from a Poisson solver

expression of the code is shown in Figure 5 and the coupling technique (and full equations) can be found in [13].

All the tests were run on the new LACL cluster composed of 20 Intel Pentium dual core E2180 2Ghz with 2GBytes of RAM interconnected with a Gigabyte Ethernet network.

We present the benchmarks when the interface meshes match using random generated sub-domains (real life inputs are described in [13]). The principle of this extensibility test is as follow: increases the number of processors as well as size of data. In this context, for each input, one processor is associated with one sub-domain and the global domain is divided into 1, then into 2, 4, *etc.* sub-domains.

Various manners of decomposing the global domain in a structured way are explored. The number of sub-domains along the axis is denoted by N_x (resp. N_y, N_z) and each sub-domain possesses approximately 50000 cells (time to sequentially decompose a sub-domain is approximately linear). The number of generated super-threads would be too big for our past implementation.

Performances (minutes and seconds) of OCamlP3l and BSML (using its MPI implementation) are summarised in the following table:

(N_x, N_y, N_z)	Nb procs	OCamlP3l	BSML
$1 \times 1 \times 1$	1	20.56	21.29
$1 \times 1 \times 2$	2	24.06	27.63
$1 \times 1 \times 4$	4	24.78	28.23
$1 \times 1 \times 8$	8	25.05	28.97
$1 \times 1 \times 16$	16	26.53	30.67
$1 \times 2 \times 2$	4	20.78	25.14
$1 \times 2 \times 4$	8	24.45	28.36
$1 \times 2 \times 8$	16	25.56	29.84
$1 \times 4 \times 4$	16	26.89	29.89
$2 \times 2 \times 2$	8	25.88	27.21
$2 \times 2 \times 4$	16	27.89	32.75

As might be expected, OCamlP3l is faster than our naive implementation but not much. Barriers slow down the whole program but bulk-sending accelerates the communications: in the P3L run there exists a bottleneck due to the fact that sub-domains are centralised and therefore the amount of communication treated by one process may cause an important overhead. In BSML, the data are each time completely distributed, which reduces this overhead but causes a loss of time in the distribution of the data.

We did not benchmark the old implementation against the newer, because the older could not handle the number of concurrent threads. Our aim was not to beat OCamlP3l, whose implementation is far more complicated than ours but have both BSML and OCamlP3l.

6. Related works

Divide-and-conquer and skeletons paradigms. A general data-parallel formulation for a class of divide-and-conquer problems was evaluated in [19]. But those techniques are only defined for a low-level parallel language, High Performance Fortran. In [20], the authors present a new data-parallel C library for Intel's core-processors which have a divide-and-conquer primitive. Some optimisations in the implementation have been done using the BSP model.

In [21], the proposed approach distinguished two levels of abstraction: (1), a small skeleton language defines the static parallel parts of the programs (using MetaOCaml); (2), an implementation of a divide-and-conquer skeleton demonstrates how meta-programming can generate the appropriate set of communications. However, cost prediction nor efficient code generation are possible. For efficient code, [22] proposes the same approach using C++ templates but no divide-and-conquer skeleton is at this time provided.

[2] described how to add skeletons in MPI as well as some experiments (the eskel library). It also gives convincing and pragmatic arguments to mixed message passing and skeleton programming, using C. We think that using OCaml for parallel programming (high-performance applications) is not a bad choice since the generated code is often very competitive with the C counterparts.

CPS transformations. The original CPS transformation was the most simple one. This transformation introduce too many unnecessary administrative redexes and more efficients CPS were defined in [9], [23]. CPS were massively used for various implementation of ML languages [8].

Historically, the idea of using CPS (or a call-cc operator[10]) for thread implementation cames from [11]. These techniques were then massively used by many authors to implement concurrent extensions of sequential languages [24] such as ML [25] or C [26].

7. Conclusion and future works

Conclusion. In this paper we have defined a new implementation of a multi-threading primitive (called parallel superposition) for a high-level BSP and data-parallel language. This implementation uses a global CPS transformation which have been optimised using a flow analysis. Our presentation of the CPS transformation abstracts away from the details of BSP communications, as they are irrelevant to the semantics study. Different optimisations such as defunctorization and monomorphi(*flow*)sation have also been added for performance issues. Our implementation relies on semantics investigations, allowing us to better trust it. Furthermore, it works on an important subset of OCaml.

10. A call-cc (call-with-current-continuation) primitive is a control structure which is close to a CPS transformation.

The presented techniques are not novel, except our CPS transformation and the monoflowisation. We would like to emphasise that this transformation is not a subset of the one in [16]. More precisely, we do not constrain the flow of a binding variable to be the same as its binding expression, allowing impure functions to take pure arguments. Moreover, we find that the combination of all our transformations on a large subset of OCaml is quite new, if not on the pure theoretical front, at least as a tool (we think that it could be adapted to handle other constructs such as call/cc). Also, we are not aware of any implementation of P3L skeletons using the pure BSP paradigm: all implementations that we know of are implemented over pre-existing low level libraries.

Future works. The ease of use of this new implementation of the superposition will be experimented by developing less naive implementations of the OCamlP3l's skeletons as using a smarter heuristic for load balancing computations which will depend of the BSP's architecture parameters. We will also investigate a realistic polyvariant flow analysis to generate less CPS code.

References

[1] S. Gorlatch, "Send-receive considered harmful: Myths and realities of message passing," *ACM TOPLAS*, vol. 26, no. 1, pp. 47–56, 2004.

[2] M. Cole, "Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[3] R. H. Bisseling, *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.

[4] F. Loulergue, "Parallel Superposition for Bulk Synchronous Parallel ML," in *International Conference on Computational Science (ICCS 2003), Part III*, ser. LNCS, P. M. A. Sloot and al., Eds., no. 2659. Springer Verlag, june 2003, pp. 223–232.

[5] A. Tiskin, "A New Way to Divide and Conquer," *Parallel Processing Letters*, vol. 11, no. 4, pp. 409–422, 2001.

[6] F. Gava, "A Modular Implementation of Parallel Data Structures in BSML," *Parallel Processing Letters*, vol. 18, no. 1, pp. 39–53, 2008.

[7] ——, "Implementation of the Parallel Superposition in Bulk-Synchronous Parallel ML," in *The International Conference on Computational Science (ICCS), Part I*, ser. LNCS, Y. Shi, G. Albada, J. Dongarra, and P. Sloot, Eds., vol. 4487. Springer-Verlag, 2007, pp. 611–619.

[8] A. W. Appel, *Compiling with Continuations*. Cambridge University Press, 1992.

[9] Z. Dargaye and X. Leroy, "Mechanized verification of CPS transformations," in *Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR*, ser. LNAI, vol. 4790. Springer, 2007, pp. 211–225. [Online]. Available: http://gallium.inria.fr/~xleroy/publi/cps-dargaye-leroy.pdf

[10] H. Thielecke, "From control effects to typed continuation passing," *ACM SIGPLAN Notices*, vol. 38, no. 1, pp. 139–149, 2003.

[11] M. Wand, "Continuation-based multiprocessing," in *Lisp Conference*. ACM, 1980, pp. 19–28.

[12] O. Shivers, "Continuations and threads: Expressing machine concurrency directly in advanced languages," in *Second ACM SIGPLAN Workshop on Continuations*, 1997.

[13] F. Clément, V. Martin, A. Vodicka, R. D. Cosmo, and P. Weis, "Domain Decomposition and Skeleton Programming with OCamlP3l," *Parallel Computing*, vol. 32, pp. 539–550, 2006.

[14] P. Wadler, "Monads and composable continuations," *Lisp and Symbolic Computation*, vol. 7, no. 1, pp. 39–56, 1994.

[15] I. Garnier and F. Gava, "New Implementation of a Parallel Composition Primitive for a Functionnal BSP Language," LACL, University of Paris East, Tech. Rep. 5, 2008. [Online]. Available: http://lacl.univ-paris12.fr//Labo/TechReports/2008/TR-LACL-2008-5.pdf

[16] J. Kim and K. Yi, "Interconnecting between CPS terms and non-CPS terms," in *Third ACM SIGPLAN Workshop on Continuations, Technical Report*, A. Sabry, Ed., no. 545. Computer Science Department, Indiana University, 2001.

[17] N. Heintze, "Control-Flow Analysis and Type Systems," in *Static Analysis Symposium (SAS)*, ser. LNCS, A. Mycroft, Ed., no. 983. Springer, 1995.

[18] F. Pottier and D. Rémy, "The Essence of ML Type Inference," in *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, 2005, ch. 10, pp. 389–489. [Online]. Available: http://cristal.inria.fr/attapl/

[19] M. Aumor, F. Arguello, J. Lopez, O. Plata, and L. Zapata, "A data-parallel formulation for divide-and-conquer algorithms," *The Computer Journal*, vol. 44, no. 4, pp. 303–320, 2001.

[20] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou, "Ct: A Flexible Parallel Programming Model for Tera-scale Architectures," Intel Research, Tech. Rep., 2007.

[21] C. Herrmann, "Generating message-passing programs from abstract specifications by partial evaluation," *Parallel Processing Letters*, vol. 15, no. 3, pp. 305–320, 2005.

[22] J. Falcou, J. Serot, T. Chateau, and J. T. Lapreste, "QUAFF : Efficient C++ Design for Parallel Skeletons," *Parallel Computing*, vol. 32, no. 7-8, pp. 604–615, 2006.

[23] O. Danvy and L. R. Nielsen, "Cps transformation of beta-redexes," *Information Processing Letterseses*, vol. 94, no. 5, pp. 217–225, 2005.

[24] S. Srinivasan, "A thread of one's own," in *New Horizons in Compilers Workshop*, 2006.

[25] E. Biagioni, K. Cline, P. Lee, C. Okasaki, and C. Stone, "Safe-for-space threads in standard ml," *Higher-Order and Symbolic Computation*, vol. 11, no. 2, pp. 209–225, 1998.

[26] J. Chroboczek, "Continuation Passing for C: A space-efficient implementation of concurrency," PPS (University of Paris 7), Tech. Rep., 2005.