New implementation of a parallel composition primitive for a functionnal BSP language

Example to the implementation of algorithmic skeletons

Ilias Garnier & **Frédéric Gava**

September 2008

TR–LACL–2008–4

Laboratoire d'Algorithmique, Complexité et Logique (LACL)
University of Paris–East

Université Paris 12 – Val de Marne, Faculté des Science et Technologie
61, Avenue du Général de Gaulle,
94010 Créteil cedex, France
Tel.: (33)(1) 45 17 16 47, Fax: (33)(1) 45 17 66 01

New implementation of a parallel composition primitive for a functionnal BSP language

Example to the implementation of algorithmic skeletons

Ilias Garnier **Frédéric Gava**

Laboratory of Algorithmics, Complexity and Logic (LACL)
University of Paris–East (Paris 12),
61 avenue du Génétal de Gaulle, P2 du CMC
94010 Créteil cedex, France
`gava@univ-paris12.fr`

Abstract

Bulk-Synchronous Parallel ML (BSML) is a ML based language 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 parallel primitives on a parallel data structure called parallel vector. One of these primitives is dedicated to express at most divide-and-conquer algorithms by allowing parallel composition of two BSP programs. Nevertheless, its implementation using system threads have a serious drawback which is the maximal number of possible threads in OS. This paper presents a new implementation of this primitive (called parallel superposition) based on a continuation-passing-style (CPS) transformation and a flow analysis. Exemple of application is done (with some benchmarks) to the application of the implemenatation of algorithmic skeletons (those of OCamlP3L) that will need an important number of calls of this primitive.

Contents

Chapter 1

Introduction

The increasing pervasiveness of multi-CPU systems makes the design of new and robust parallel programming languages more important. Creating such a language involves a tradeoff between the possibility to write predictable and efficient programs and the abstraction of such features to make programming safer and easier. An interesting compromise is Bulk-Synchronous Parallel ML[1] (a.k.a. BSML), an extension of ML to code bulk-synchronous algorithms which combines the high degree of abstraction of ML with the scalable and predictable performances of BSP. In BSML, deadlocks and non-determinism are avoided.

The Bulk-Synchronous Parallel[2] (BSP) paradigm's simplicity and elegance comes at a cost: the ability to synchronise a subset of the processors would break the BSP cost model. Subset synchronisation is used to recursively decompose computations into independent tasks (this is the divide-and-conquer paradigm). However, [36] proposes a natural way to fit divide-and-conquer algorithms into the BSP framework without using subset synchronisation by using sequentially interleaved threads of BSP computation, called *super-threads*. An adaptation of this method to BSML was proposed in [29]: the *parallel superposition*. The first implementation of this primitive was based on system threads [19], limiting the number of such threads and leading to efficiency problems.

We propose a new implementation of the parallel superposition. Our implementation is based on the efficient compilation of lightweight threads using a flow-directed CPS [30] transformation. We base our developments on a firm semantics ground, by proving the operational equivalence between the source and the result of the transformation. Finally, we show how algorithmic skeletons can be implemented (even naively) using this new primitive. This part briefly reviews the BSP model, the BSML language and informally presents the parallel superposition and its first implementation.

1.1 Generalities.

At the core of our implementation is the CPS transformation. CPS is a classic style of programming in which control is passed explicitly in the form of a continuation [1]. 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. For instance, the successor function, written (**fun** x→x+1) in direct style, becomes (**fun** x k→k(x+1)) in CPS style, where k is the extra continuation parameter. Programs can be systematically translated to semantically equivalent programs in CPS using a variety of CPS transformation algorithms [15]. The CPS transformation is widely used as an intermediate representation in compilers for functional languages [1], allowing aggressive optimisations that are significantly harder to perform on direct-style programs. First-class continuations are also an extremely powerful tool in the hands of the programmer, allowing rich and expressive control constructs to be built. In particular, cooperative lightweight threads [17, 32, 38] are easily encoded using continuations. We go further by guiding our transformation by a flow analysis, allowing to spare unrelated part of the program from the transformation.

[1]Currently, BSML is implemented as a parallel library for `OCaml`. See`http://bsmllib.free.fr`.

[2]We refer to [6, 33] for a gentle introduction to BSP.

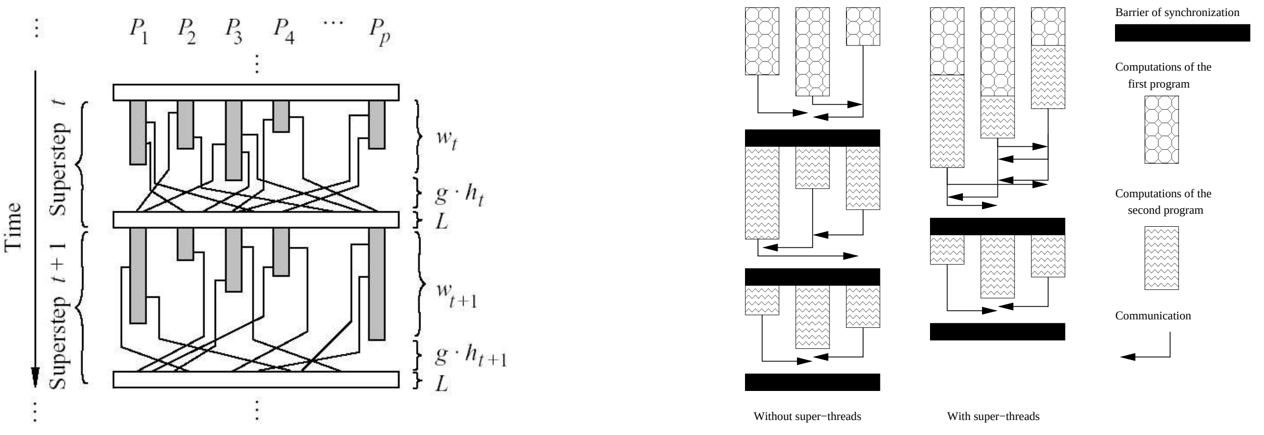


Figure 1.1: The BSP model of execution (left) and evaluation of the superposition (right)

1.2 The BSP model.

A BSP machine is a set of pairs CPU-memory distributed across a communication network. The execution of a BSP program is divided into super-steps (see left scheme in Fig. 1.1), each separated by a global synchronisation. A super-step consists of each processor doing some calculations on local data and communicating some data to other processors. The collective barrier of synchronisation event guarantee that all communications of data have completed before the start of the next super-step; this ensures the determinism of the parallel program.

A BSP computer is characterized by three parameters, which are given in function of the processor speed s:

- The number of processors p;
- The time taken by a global synchronisation l;
- The time taken for a collective exchange where each processor to send and/or receive at most one word g. This exchange is called a 1-relation. Thus, a communication phase where each processor sends and/or receives h words is bounded by $h \times g$.

Any BSP program's complexity is given in function of these parameters. If we were to allow subset synchronisation, we wouldn't be able to bound the execution time of a super-step so easily. This would in turn make the complexity analysis of BSP programs near impossible.

1.3 The BSML language.

1.3.1 Core BSP primitives.

BSML allows to program BSP algorithms in high-level fashion. BSML is based on 8 primitives, three of which are used to access the parameters of the machine :

bsp_p: int **bsp_l**: float **bsp_g**: float
mkpar: (int→α)→α **par**
apply: (α →β) **par**→α **par**→β **par**
put: (int→α) **par**→(int→α) **par** **proj**: α **par**→int→α
super: (unit→α)→(unit→β)→$\alpha * \beta$

A BSML program is built as a sequential program on a parallel data structure called parallel vector. Its ML type is α **par**, which expresses that it contains a value of type α at each of the $\mathbf{p}$ processors. Moreover, there is no nested data parallelism. To enforce this constraint, a type system was developed[3]. Implementation of these primitives rely either on MPI, PUB [7] or on the TCP/IP functions provided by the Unix module of `OCaml`.

The BSP asynchronous phase is programmed using the two primitives **mkpar** and **apply** so that (**mkpar** f) stores (f i) on process i (f is a sequential function): **mkpar** f $= \boxed{(\mathrm{f}\ 0)} \boxed{\cdots} \boxed{(\mathrm{f}\ \mathrm{i})} \boxed{\cdots} \boxed{(\mathrm{f}\ (\mathbf{p}-1))}$ and **apply**

[3]This is a part of the ongoing thesis of Louis Gesbert at the LACL: `http://research.antislash.info/english/`

applies a parallel vector of functions to a parallel vector of arguments: **apply** $\boxed{\cdots}\boxed{f_i}\boxed{\cdots}$ $\boxed{\cdots}\boxed{v_i}\boxed{\cdots}$ = $\boxed{\cdots}\boxed{(f_i\, v_i)}\boxed{\cdots}$

The first communication primitive is **put**. It takes as argument a parallel vector of functions which should return, when applied to i, the value to be sent to processor i. **put** returns a parallel vector with the vector of received values: at each processor these values are stored in a function which takes as argument a processor identifier and returns the value sent by this processor.

The second communication primitive **proj** is such that (**proj** vec) returns a function f where (f n) is the nth value of the parallel vector vec. Without this primitive, the global control cannot take into account data computed locally.

The primitive **super** (parallel superposition) allows the evaluation of two expressions as interleaved threads of BSP computations called super-threads. From the programmer's point of view, the semantics of **super** is the same as pairing but the evaluation of **super** $E_1\ E_2$ is different (see right scheme in Fig. 1.1): the phases of asynchronous computation of E_1 and E_2 are run; then the communication phase of E_1 is merged with that of E_2 and only one barrier occurs; if the evaluation of E_1 needs more super-steps than that of E_2 then the evaluation of E_1 continues (and *vice versa*).

The parallel superposition of E_1 and E_2 costs less than the evaluation of E_1 followed by the evaluation of E_2. The superposition is thus not only useful to express divide-and-conquer algorithms, but it can also be used to efficiently program parallel data structures [20], BSP scheduling *etc.*

1.3.2 Useful BSP functions.

The primitives described in this section constitute the core of the BSML language. The BSML library contains many others useful functions.

Often used asynchronous functions. The asynchronous function replicate creates a parellel vector which contains the same value everywhere. The **apply** primitive only handles the application of a parallel vector of functions taking one argument, and we define the applyn function to deal with n-ary functions.

$(* \, replicate : \alpha \rightarrow \alpha par \; and \; apply2 : \; (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha par \rightarrow \beta par \rightarrow \gamma par \, *)$

let replicate x = **mkpar**(fun pid → x) **and** apply2 vf v1 v2 = **apply** (**apply** vf v1) v2

It's also common to apply the same function to each element of a parallel vector. We provide such a primitive for functions of arity equal to 1 and 2.

$(* \, parfun : (\alpha \rightarrow \beta) \rightarrow \alpha par \; and \; parfun2 : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha par \rightarrow \beta par \, *)$

let parfun f v = **apply** (replicate f) v **and** let parfun2 f v1 v2 = **apply** (parfun f v1) v2

We often want to apply a different function at a specific process. applyat n f1 f2 applies function f1 at process n and f2 at others.

$(* \, applyat : int \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha par \rightarrow \beta \, *)$

let applyat n f1 f2 v = **apply** (**mkpar** (**fun** i → if i = n then f1 else f2)) v

Often used communication functions. As an example, we will describe replicated total exchange. Each processor contains a value (represented as a parallel vector of values) and the result of (rpl_total $\boxed{v_0}\boxed{\cdots}\boxed{v_{p-1}}$) is $[v_0, \ldots, v_{p-1}]$ - a replicated list of these values on each processor:

$(* \, rpl\_total : \alpha par \rightarrow \alpha list \, *)$

```
let rpl_total vec =
  let rpl_totex vec = compose noSome (proj (parfun (fun v →Some v) vec)) in
    List.map (rpl_totex vec) (procs ())
```

where compose f g x = f (g x), noSome (Some x) = x and procs () = $[0; 1; \ldots; bsp\_p() - 1]$.

Useful functions can then be defined, such as parfun_total which applies a sequential function to each element of a vector, totally exchanges these values and finally applies another sequential function:

parfun_total $f_1\ f_2$ $\boxed{v_0}\boxed{\cdots}\boxed{v_{p-1}}$ $\Rightarrow (f2\,[f_1 v_0; \ldots; f_1 v_{p-1}])$.

Our second example is the broadcasting of a value v from one processor i to other ones. It can be summarized as follows:

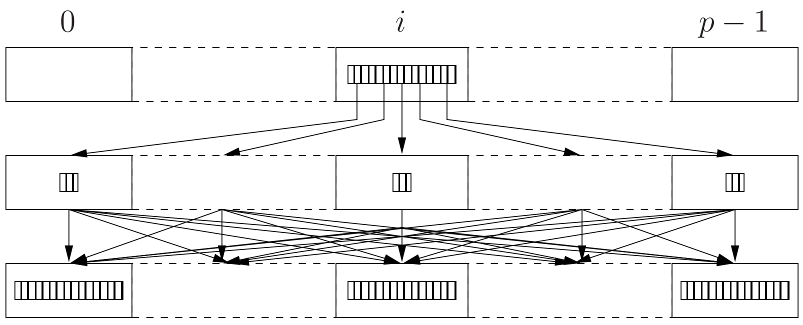
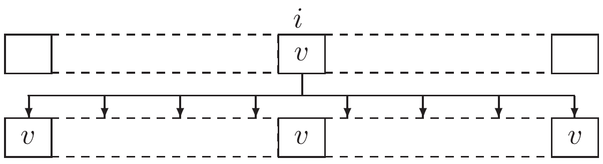


Figure 1.2: 2 phases broadcasting algorithm (also call with total exchange)



where processor i sends its own value v to other processors. This task can be done in one super-step using the following code:

```
(∗ bcast_direct: int→α par→α par ∗)
let bcast_direct root vv =
 if not (correct_number_of_processor root)
  then raise (Bcast_Error "Root␣is␣not␣a␣correct␣number␣of␣processor")
 else
  let mkmsg = applyat root (fun v dst →Some v) (fun _ dst →None) vv
   in parfun noSome (apply (put mkmsg) (replicate root))
```

When the size of value to broadcast is important, and following the parameters BSP of the machine, the two-phases algorithm described in [6, 21] can be more effective.

The Figure 1.2 shows the method used. The broadcasting in two super-steps proceeds as follows: the sending processor “cuts” the message into p sub-messages and sends one sub-message to each other processors (first super-step). Then, each processor sends its sub-message to other processors (total exchange) and to finish each processor “glue” the receveid sub-message to form the complete message.

In a first step, we define the first super-step, one that scatters the initial value of the processor. We code it with the function below, which takes as parameter a function that define how cutting the value to broadcast:

```
(∗ scatter :(α →int→β option)→int→α par→β par ∗)
let scatter_wide partition root v =
 if not (correct_number_of_processor root)
  then raise (Scatter_Error "Root␣is␣not␣a␣correct␣number␣of␣processor")
 else let mkmsg = applyat root partition (fun _ _ →None) in
        parfun noSome (apply (put (mkmsg v)) (replicate root))
```

Then, we can implant a generic version of this broadcasting algorithm, (generic because “cut” and “glue” functions are the first function parameters):

```
let bcast_totex_rpl_gen howTotex howApp partition paste root vv =
 if not (correct_number_of_processor root)
  then raise (Bcast_Error "Root␣is␣not␣a␣correct␣number␣of␣processor")
 else
  let phase1 = scatter_wide partition root vv in
  let phase2 = howTotex phase1 in
   howApp paste phase2

(∗ replicate_bcast_totex_gen: (α →int→β option)→((int→β )→γ )→int→α par→γ ∗)
let replicate_bcast_totex_gen partition paste root vv =
 bcast_totex_rpl_gen proj (fun f x →(f x)) partition paste root vv
```

It can easily be specialized as for example lists (but with time of "cutting" and "glueing" lists that are proportional to the length of the sending list) or with arrays (allowing a "cutting" and a "glueing" proportional to p).

Longer examples of BSML code can be found in the Chapter A of the appendix.

1.4 Older implementation.

1.4.1 Super-threads and evaluation strategy

Imperative features are widely used when it comes to program numerical code. `OCaml` is a call-by-value language, and in order to keep a deterministic semantics we must cleary define the order of evaluation in particular constructs. This is the case of our **super** primitive, as shown in the following expression :

```
let a=ref 0 in let _ = super (fun () →a:=1) (fun () →a:=2) in !a
```

Each process creates a replicate reference called `a` which contains the integer 0. Then, two super-threads are created, and each of them affects a different value to `a`. If no strategy is defined, the result would be indeterministic. The compositional nature of the BSP cost model would be lost if the number of super-steps of the program depended of `a`. This is due to the fact that the two super-threads are active and are allowed to modify this shared reference.

Having only one active super-thread and a deterministic strategy for the choice of this super-thread is the solution : the active super-thread is evaluated until it ends its computation or it needs communications, i.e., until it ends the first phase of a superstep. When communications are done, the first super-thread which has finished "its superstep" is re-started, i.e., it becomes the new current active super-thread.

Note that having only one active super-thread gives also better performances : there is no cost for changing every time of active super-thread.

1.4.2 Thread implementationa

Currently, the superposition is implemented (based on a semantics study) using system threads [19]. Each time a superposition is called, a new thread is created and share locks are used each time a communication primitive is called. This way, we simulate that only one super-thread is active and allow that all data (managed by the GC of `OCaml`) are available by each super-thread.

There are two drawbacks to this method. First, threads slow down the program: a global lock is used due to `OCaml`'s GC. Even if there is at most one active thread, we lost a fair amount of time because of the numerous system locks. The second and main drawback is the maximal number of possible threads in OS (e.g. 1024 for many Linux systems). For divide-and-conquer algorithms (initial goal of the superposition), it is not a problem because they mainly need a tree of recursive calls (each call need a new super-thread) and their sizes are logarithmic in the number of processors. But for a modern use of this primitive [19, 20], a greater number of super-threads than this maximal number can be run simultaneously in a single program.

These limitations would quickly depreciate the interest of this primitive and using the semantics property that only one super-thread is active, we now present another implementation which use a global continuation-passing-style transformation of BSML programs.

1.5 Outline

Part 2 presents our transformation as well as semantics results. A type system instrumented to perform flow analysis is described.

Our implementation is described in Part 3. We used a constraint-solving approach to type inference. Moreover, we describe the many transformations and optimisations used to bridge the gap of polymorphism that we implemented (such as monomorphisation), and their respective costs and benefits. We also dicuss the few technical limitations of our code transformer.

In Part 4, we present an interesting example of application for our new primitive: the implementation of algorithmic skeletons. The skeletons are a kind of "parallel design patterns" which can be easily composed to form safe parallel programs. These skeletons can be seen as a graph, which we recursively decompose using the parallel superposition.

Part 5 concludes this report by reviewing previous works on the implementation of the divide-and-conquer paradigm into parallel programming languages and the use of the CPS transformation to express concurrency.

Chapter 2

CPS transformation and flow analysis

Before describing our transformation, we present a core source language which is heavily inspired by ML, 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 **put** and **proj**, abstracting away communication handling. **yield** suspends the currently executing superthread (called thread in the next) and schedules the execution of the next thread, as defined by the **super** operational semantics. As can be seen from the syntax definition, **mkpar** and **apply** are ignored. These two primitives are orthogonal to the following work.

Syntax of the language. Expressions and values are as follow:

e ::=	x	*variables*	|	c	*constants*
|	$\lambda x.e$	*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** *and* **proj**
m ::=	$\kappa\ x\ \rightarrow\ e$	*matching branch*			

v ::=	c	*constants*	|	$\lambda v.e$	*functional values*
|	**fix** $f\ \lambda x.e$	*recursive functions*	|	(v_1, v_2)	*couples*
|	$\kappa\ v$	*constructor application*			

Annotations on expressions are possibly written in sub or superscript position. The big-step operational semantics for the core of our language is given in Fig. 2.1.

2.1 Continuation Passing Style.

The original CPS transform [30] was designed to study the various evaluation strategies for the lambda-calculus by making the control explicit, as a *continuation*: a function representing the evaluation context. It was then discovered that giving to the programmer or the compiler writer the ability to explicitly manipulate continuations was an expressive tool to perform various analysis or to encode various high-level constructs, such as exceptions or light-weight threads [32]. Below is the original CPS transformation, as defined in [30]:

$$\begin{array}{lcl}
[\![x]\!] & = & \lambda k.k\ x \\
[\![\lambda x.M]\!] & = & \lambda k.k\ (\lambda x.[\![M]\!]) \\
[\![M\ N]\!] & = & \lambda k.[\![M]\!]\ (\lambda m.[\![N]\!]\lambda n.m\ n\ k)
\end{array}$$

This transformation is strictly equivalent to the monadic one, which is presented in the next section.

2.2 Monadic CPS transformation.

Monads are a useful programming technique in functional languages [37]. They allow to extend a language while enforcing a correct operational behaviour. A monad is the data of three primitives: **run**, **ret** and **bind**, operating on a type $M\,\alpha$ - intuitively, the type of computations on values of type α. The **run** primitive has type $\forall\alpha.M\,\alpha \rightarrow \alpha$

CONST
$$c \Rightarrow c$$

LAMBDA
$$\lambda x.e \Rightarrow \lambda x.e$$

PAIR
$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{(e_1, e_2) \Rightarrow (v_1, v_2)}$$

CONSTR
$$\frac{e \Rightarrow v}{\kappa\, e \Rightarrow \kappa\, v}$$

LET
$$\frac{e_1 \Rightarrow v_1 \qquad [v_1/a]e_2 \Rightarrow v_2}{\textbf{let}\, a = e_1\, \textbf{in}\, e_2 \Rightarrow v_2}$$

APP
$$\frac{e_1 \Rightarrow \lambda x.e \qquad e_2 \Rightarrow v' \qquad [v'/x]e \Rightarrow v}{e_1\, e_2 \Rightarrow v}$$

FIX
$$\textbf{fix}\, h\, \lambda x.e \Rightarrow \textbf{fix}\, h\, \lambda x.e$$

MATCH
$$\frac{e \Rightarrow \kappa\, v \qquad [v/x]e' \Rightarrow v'}{\textbf{match}\, e\, \textbf{with} \mid \ldots \mid \kappa\, x \rightarrow e' \mid \ldots \Rightarrow v'}$$

OP
$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2 \qquad v = op\, v_1\, v_2}{op\, e_1\, e_2 \Rightarrow v}$$

SUPER
$$\frac{e_1() \Rightarrow v_1 \qquad e_2() \Rightarrow v_2}{\textbf{super}\, e_1\, e_2 \Rightarrow (v_1, v_2)}$$

YIELD
$$\textbf{yield} \Rightarrow ()$$

Figure 2.1: Big-step reduction rules

and executes a monadic program. **ret** , of type $\forall\, \alpha.\alpha \rightarrow M\, \alpha$ transforms a base value into a monadic one. Finally, **bind** allows chaining monadic computations as reflected by it's type $\forall\, \alpha, \beta.M\, \alpha \rightarrow\ (\alpha \rightarrow M\, \beta) \rightarrow M\, \beta.$

Monadic primitives. Threads are modelled as resumptions, meaning that they are either in a suspended state or terminated: **type** α thread=Terminated **of** α |Waiting **of** (unit$\rightarrow\alpha$ thread).

The usual CPS monad type is: $M_{cps}\, \alpha\ =\ \forall\beta.(\alpha \rightarrow \beta) \rightarrow \beta$ but in our case, the codomain of continuations is always threads: $M\, \alpha\ =\ \forall\, \beta.(\alpha \rightarrow thread\, \beta) \rightarrow thread\, \beta$. The primitives are defined as follow:

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

These primitives must also satisfy three laws:

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

Where $\approx$ is defined as: $a_1 \approx a_2\ =\ \forall k\, \exists\, a\, .(a_1\, k = a) \wedge (a_2\, k = a)$ extended by "concurrency-irrelevance": $\lambda k.Waiting\ k\ \approx\ $ **ret** $()$. We take $=$ to be β-convertibility. In our case, these laws were mechanically proved using the Coq proof assistant. See the appendix for the proof script.

Monadic transformation. We now straightforwardly proceed to the definition of the naive monadic transformation on expressions $T_0[\![e]\!]$:

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

The two concurrency primitives are then defined using first class continuations:

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

Where $a\,@\,b\ =\ \textbf{bind}\ \ a\ (\lambda v_a.\textbf{bind}\ \ b\ (\lambda v_b.v_a\ v_b)))$.

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**) and then schedules the execution of it's two sub-threads, until they are terminated.

Code size explosion. To illustrate the problem of the naive transformation, we give this trivial but clear example:

$$\begin{array}{lcl}
apply & = & \lambda f.\lambda x.fx \\
T_0[\![apply]\!] & = & \lambda k_0.k_0\lambda f.\lambda k_1.k_1\lambda x.(\lambda k_2.(\lambda k_3.k_3 f)(\lambda v.(\lambda v_f.\lambda k_4.(\lambda k_5.k_5 x)(\lambda v.(\lambda v_x.v_f v_x)v k_4))v k_2))
\end{array}$$

It is obvious that this transformation can't be used as is. Moreover, this code didn't need to be converted to CPS at all: it doesn't contain any concurrency primitive. Preserving these kind of expression from being converted is the aim of the transformation which is presented in the next section.

Soundness of the naive transformation. The soundness proof for the vanilla CPS transformation is equally applicable to our slightly modified setting. We won't expose the proof here, but we will nevertheless state the result. Let the CPS transformation on values be defined as:

$$[\![c]\!]_v = c \qquad [\![(v_1, v_2)]\!]_v = ([\![v_1]\!]_v, [\![v_2]\!]_v) \qquad [\![\kappa\ v]\!]_v = \kappa\ [\![v]\!]_v$$

$$[\![\lambda x.e]\!]_v = \lambda x.T_0[\![e]\!] \qquad [\![\textbf{fix}\ h\ \lambda x.e]\!]_v = \textbf{fix}\ h\ \lambda x.T_0[\![e]\!]$$

Theorem 2.2.1 (Soundness of the naive transformation) *If* $e \Rightarrow v$, $T_0[\![e]\!] \approx \textbf{\textit{ret}}\ \ [\![v]\!]_v$.

The small-step semantics of **super** make the "threads" appear in the reduction sequence, thus we use the big-step semantics rules, making the proofs much harder. However, the fact that **super** allows to reduce the number of global synchronisation only appears in the small-step semantics. The proof linking our big-step soundness proof to the small-step operational semantics of **super** can be found in [18].

2.3 Flow-directed cps transformation.

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

However, these reductions doesn't suffice. Aiming at intense parallel computing, we can't 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. Ours is inspired by [27]. 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.

2.3.1 Type-based flow analysis.

Definition of the type system. In the context of language compilation, flow analysis is usually used as a technique to efficiently implement closure conversion. Our flow analysis presents itself as an instrumented type system, yielding an straightforward flow inference algorithm.

Instead of tracking sets of lambda-abstractions, our flow analysis purpose is to decide if an expression is susceptible to reduce a **yield** expression. We define our flows as F ::= $\mathcal{P} \mid \mathcal{I}$. If an expression may reduce a **yield**, we tag it as *impure*: $\mathcal{I}$. If not, we tag it as *pure*, noted $\mathcal{P}$. We also define a total order $<_F$ on flows, defined by $\mathcal{I} <_F \mathcal{P}$.

Our type system is derived from the type system for CFA defined in [23]. We use ground, simple types annotated by flows. Let τ denote the syntactic family of types:

$$\begin{array}{rll}
\tau ::= & \langle F, const\_type \rangle & \\
\mid & \langle F, typename \rangle & \text{user-defined sum types} \\
\mid & \langle F, \tau_0 \rightarrow \tau_1 \rangle & \text{functions} \\
\mid & \langle F, \tau_0 * \tau_1 \rangle & \text{couples} \\
 & & \\
const\_type ::= & unit \mid int & \text{constants}
\end{array}$$

Each constructor κ has a domain type (the type of its argument) and a codomain type (the $typename$ to which κ belongs). These are denoted κ_{dom} and κ_{codom}. We also define two projection functions on types : $annot(\langle f, x \rangle)$ = x and flow($\langle f, x \rangle$) = f. These functions are readily extended to typed source terms. For any expressions a and b, we define $a \vee b = min(flow(a), flow(b))$. The inference rules for our type system are given in Fig. 2.2.

Soundness proof. Before stating and proving the soundness theorem, we will prove a lemma on type-preserving (and thus *flow preserving*) substitutions. This proof is mostly the same as in simply-typed lambda-calculus.

Lemma 2.3.1 (Typings are stable by substitution)
Let e be an expression such that $\Gamma, x \, : \, \tau \vdash e \, : \, \tau'$ holds, and let v be an expression such that $\Gamma \vdash v \, : \, \tau$. Then $\Gamma \vdash [v/x]e \, : \, \tau'$.

Theorem 2.3.2 (Soundness w.r.t. **yield** reductions)
*Let e be a well-typed expression. We have: a **yield** expression is reduced while normalizing $e \rightarrow \; flow \; e \; = \; \mathcal{I}$.*

Proof : see the appendix.

2.3.2 Partial CPS transformation.

In order to alleviate the overhead of the naive CPS, we use a flow-directed partial CPS transformation. The aim is to preserve "pure" expressions, while CPS-converting "impure" ones. The first option we considered was to directly use the transformation described in [27], relying on the types inferred during the flow analysis to generate appropriate padding code between CPS and non-CPS terms. Sadly, our analysis doesn't meet one of the criterion necessary to ensure the soundness of this transformation. It then appeared that our setting allows a much simpler transformation. The algorithm is quite simple : when transforming a pure expression, we simply wrap it into a **ret** .

Definition of the partial transformation.

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

The concurrency primitives **super** and **yield** are directly replaced by their definitions, and the primitive operators are always pure. 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.

Examples. We will only show the produced code and not the type derivations. The example from the previous section is simply wrapped into an abstraction:

$$\begin{array}{lcl}
apply & = & \lambda f.\lambda x.f x \\
T_1[\![apply]\!] & = & \textbf{ret}\ apply
\end{array}$$

We also give a more interesting variation of this function. We assume here that f and x are always applied to pure arguments, implying that flow(f) = flow(x) = flow($f\ x$) = $\mathcal{P}$.

$$\begin{array}{lcl}
apply_{\mathcal{I}} & = & \lambda f.\lambda x.\textbf{let}\,()\ =\ \textbf{yield}\ \textbf{in}\ f x \\
T_1[\![apply]\!] & = & \lambda k_0.k_0 \lambda f.\lambda k_1.k_1 \lambda x.\lambda k_2.(\lambda k_3.Waiting\ k_3)(\lambda ().(\lambda ().\lambda k_4.k_4(f\ x))\ ()\ k_2)
\end{array}$$

There is still many administrative redexes, but 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. Our final touch to the transformation is the use of a modified CPS transform which creates no administrative redex [14, 15].

2.3.3 Soundness for the partial transformation.

We redefine the partial CPS transformation on values as:

$$[\![v^{\mathcal{P}}]\!]_v = v \qquad [\![(\lambda x.e)^{\mathcal{I}}]\!]_v = \lambda x.T_1[\![e]\!] \qquad [\![(\textbf{fix}\, h\, \lambda x.e)^{\mathcal{I}}]\!]_v = \textbf{fix}\, h\, \lambda x.T_1[\![e]\!]$$
$$[\![(v_1, v_2)^{\mathcal{I}}]\!]_v = ([\![v_1]\!]_v, [\![v_2]\!]_v) \qquad [\![(\kappa\, v)^{\mathcal{I}}]\!]_v = \kappa\, [\![v]\!]_v$$

Once again, we will prove the soundness of T_1 on a subset of the source language. To this end, we need these two lemmas:

Lemma 2.3.3 *For all values v, $T_1[\![v]\!] =$ **ret** $[\![v]\!]_v$. Moreover, $[\![v]\!]_v$ is a value. (trivial proof)*

Lemma 2.3.4 (Extended monadic substitution) *For all values v,*

1. *If flow(x) = $\mathcal{I}$, $T_1[\![[v/x]a]\!]$ = $[[\![v]\!]_v/x]T_1[\![a]\!]$.*
2. *If flow(x) = $\mathcal{P}$, $T_1[\![[v/x]a]\!]$ = $[v/x]T_1[\![a]\!]$ (follows from the definition of $[\![]\!]_v$).*

We will now state the main soundness theorem.

Theorem 2.3.5 (Soundness of the partial CPS transformation) *If $t \Rightarrow v$, $T_1[\![t]\!] \approx$ **ret** $[\![v]\!]_v$.*

Proof : see the appendix for the proofs.

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

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

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

$$\frac{\begin{array}{c} \Gamma \vdash e : \kappa^j_{codom} \qquad \Gamma, x_i : \kappa^i_{dom} \vdash e_i : \tau \\ \Gamma, x_i : \kappa^i_{dom} \vdash \kappa^i x_i : \kappa^j_{codom} \qquad \text{if } flow(\tau) \leq_F flow(\kappa^j_{codom}) \qquad 1 \leq j \leq n,\ \forall i \in [1..n] \end{array}}{\Gamma \vdash \textbf{match}\, e \textbf{ with } \mid \kappa_i\ x_i \ \rightarrow \ e_i : \tau}$$

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

Figure 2.2: Inference rules

Chapter 3

New implementation

Our source language is BSML, a statically-typed derivative of Caml. Caml's type system is based on the Hindley-Milner type discipline, and provides a rich module system, including functors. We do not want to lose the flexibility provided by polymorphism. But even if we could extend our type system to handle let-polymorphism, we would lose some precision (application of polymorphic functions to different types would be "merged", and the worst case would be taken). A polyvariant flow analysis could be used, but it would be extremely heavy, both in algorithmic complexity and in implementation. Thus, we must take on defunctorisation and monomorphisation. Monomorphisation is the process of duplicating polymorphically typed functions for each needed domain type. It can potentially make the size of the program grow exponentially, but actual implementations (as MLton[1]) shows that practically, the size growth is manageable (about 30 %). In fine, to maximize the efficiency of the generated code, we use a process similar to monomorphisation called *monoflowisation*: after duplicating functions based on their types, we duplicate them based on their flows.

3.1 Imperative features.

We didn't treat imperative features in this report, suffice to say that every expression involved in an effectful operation is constrained to have a pure flow (our partial CPS transformation isn't defined on imperative programs yet). When encountering an impure loop, we must convert it into it's tail-recursive equivalent form. `OCaml` handles tail-recursion fine, so there is no risk of stack overflow (except ocamlopt on some architectures, when the number of arguments exceeds a certain threshold).

3.2 The module system.

3.2.1 Defunctorisation.

The source language provides parametric modularity (known as *functors*), but our transformation doesn't handle theses properly. In order to apply our transformation, we have to defunctorise the whole program. To this end, we use the already existing **Ocamldefun** program[2]. A nice side-effect is the increased possibilities in inlining by our back-end compiler (`OCaml`).

We give a small example of the relevance of defunctorisation in our setting. Imagining a similar example with for instance a generic matrix module parametrized by a set of operators isn't hard.

```
module type DummySig =
  sig
    val please_inline_me : float →float →float
  end

module Dummy : DummySig =
  struct
```

[1]`http://mlton.org/`
[2]`http://www.lri.fr/~signoles/ocamldefun/`

```
    let please_inline_me = (+.)
  end

module Functor (D : DummySig) =
  struct
    let fold array =
      Array.fold_left D.please_inline_me 0.0 array
  end

(* Instantiate Functor with Dummy. Sadly,
 * Dummy.please_inline_me isn't inlined. *)
module I1 = Functor(Dummy)

(* With defunctorisation, we would obtain: *)
module I2 =
  struct
    let fold array =
      Array.fold_left Dummy.please_inline_me 0.0 array
  end
(* This allows Dummy.please_inline_me to be inlined. *)
```

3.2.2 The module environment.

For clarity's sake, we won't present too deeply how we handle the module system. Our implementation slightly differs of `OCaml`'s in subtle ways, and we dropped several features (such as the **include** directive). Basically, we see the module environment as a stack of currently opened modules describing where we *currently* are in the module tree. We acknowledge that this part of our implementation needs a more severe review to conform to `OCaml`'s semantics, however most programs are handled just fine.

3.3 Polymorphic type inference.

Monomorphisation operates on a typed source tree. To this end, we extended our type system 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 [31], 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. Another gap to bridge is the *ground* nature of our original type system. Thus, the syntactic family of types τ is extended with type *variables*, noted v.

3.3.1 Type constraints.

As in [31], polymorphism is handled using constrained type schemes, whose meaning is roughly the set of all ground types admitted by the underlying expression. Constraints C and type schemes χ are defined as follow:

C ::=	$True$	*"empty" constraint*
$\mid$	$C_0 \wedge C_1$	*constraint conjunction*
$\mid$	$\tau_0 = \tau_1$	*equality constraint*
$\mid$	$\exists\, v.C$	*existential quantification of types*
$\mid$	$\exists_F\, v_f.C$	*existential quantification of flows*
$\mid$	$def\ x\ :\ \chi\ in\ C$	*binds a type scheme to x*
$\mid$	$inst\ x\ \chi$	*instantiates the scheme bound to x*
$\mid$	$\tau_0 \leq_F \tau_1$	*flow constraint*

χ ::=	τ	*ground types*
$\mid$	$\forall\, v[C].v$	*constrained type scheme*

Bear in mind that the types's shapes are irrelevant to the $\leq_F$ constraint.

$$
\begin{array}{lcl}
C_g[\![x]\!]_\tau & = & inst\; x\; \tau \\[1em]
C_g[\![i]\!]_\tau & = & \tau \;=\; \langle \mathcal{P}, int\rangle \\[1em]
C_g[\![()]\!]_\tau & = & \tau \;=\; \langle \mathcal{P}, unit\rangle \\[1em]
C_g[\![\lambda v.e]\!]_\tau & = & \exists X_0\; X_1.\exists_F f.def\; v\; :\; X_0\; in\; C_g[\![e]\!]_{X_1} \wedge\; \tau = \langle f, X_0 \to X_1\rangle \\
 & & \wedge\; f \leq_F\; flow(X_0)\; \wedge\; f \leq_F\; flow(X_1) \\[1em]
C_g[\![\textbf{fix}\; h\; \lambda x.e]\!]_\tau & = & \exists X_0\; X_1.\exists_F f.def\; h\; :\; \langle f, X_0 \to X_1\rangle\; in\; def\; x\; :\; X_0\; in\; C_g[\![e]\!]_{X_1} \\
 & & \wedge\; \tau = \langle f, X_0 \to X_1\rangle\; \wedge\; f \leq_F\; flow(X_0)\; \wedge\; f \leq_F\; flow(X_1) \\[1em]
C_g[\![e_1\; e_2]\!]_\tau & = & \exists X_0\; X_1.\exists_F f.C_g[\![e_1]\!]_{\langle f, X_0 \to X_1\rangle} \wedge\; C_g[\![e_2]\!]_{X_0}\; \wedge\; \tau = X_1 \\
 & & \wedge\; f \leq_F\; flow(X_0)\; \wedge\; f \leq_F\; flow(X_1) \\[1em]
C_g[\![\textbf{let}\; v\; =\; e_1\; \textbf{in}\; e_2]\!]_\tau & = & def\; v\; :\; \forall\, X[C_g[\![e_1]\!]_X\; \wedge\; flow(\tau) \leq_F\; flow(X)].X\; in\; C_g[\![e_2]\!]_\tau \\[1em]
C_g[\![(e_1, e_2)]\!]_\tau & = & \exists X_0\; X_1.\exists_F f.C_g[\![e_1]\!]_{X_0}\; \wedge\; C_g[\![e_2]\!]_{X_1} \wedge\; \tau = \langle f, X_0 * X_1\rangle \\
 & & \wedge\; f \leq_F\; flow(X_0)\; \wedge\; f \leq_F\; flow(X_1) \\[1em]
C_g[\![\kappa\; e]\!]_\tau & = & \exists X.\exists_F f.C_g[\![e]\!]_X\; \wedge\; (\tau = \langle f, \textbf{datacons}\; \kappa\; X\rangle)\; \wedge\; f \leq_F\; flow(X) \\[1em]
C_g[\![\textbf{match}\; e\; \textbf{with}\; \kappa_i\; x_i \to\; e_i]\!]_\tau & = & \exists X_0.C_g[\![e]\!]_{X_0}\; \wedge\; \exists Y_i.def\; x_i\; :\; Y_i\; in\; \wedge_i\, (C_g[\![\kappa_i\; x_i]\!]_{X_0}\; \wedge C_g[\![e_i]\!]_\tau) \\
 & & \wedge\; flow(\tau) \leq_F\; flow(X_0) \\[1em]
C_g[\![\textbf{op}\; e_1\; e_2]\!]_\tau & = & C_g[\![e_1]\!]_{\langle \mathcal{P}, int\rangle}\; \wedge\; C_g[\![e_2]\!]_{\langle \mathcal{P}, int\rangle}\; \wedge\; \tau = \langle \mathcal{P}, int \to \langle \mathcal{P}, int \to int\rangle\rangle \\[1em]
C_g[\![\textbf{super}\; e_1\; e_2]\!]_\tau & = & \exists X_0\; X_1\; X_l\; X_r.C_g[\![e_1]\!]_{X_l}\; \wedge\; C_g[\![e_2]\!]_{X_r} \wedge\; X_l = \langle \mathcal{I}, \langle \mathcal{P}, unit\rangle \to X_0\rangle \\[1em]
 & & \wedge\; X_r = \langle \mathcal{I}, \langle \mathcal{P}, unit\rangle \to X_1\rangle\; \wedge\; \tau = \langle \mathcal{I}, X_l \to \langle \mathcal{I}, X_r \to \langle \mathcal{P}, X_0 * X_1\rangle\rangle\rangle \\[1em]
 & & \wedge\; flow(X_0) = \mathcal{I} \wedge\; flow(X_1) = \mathcal{I} \\[1em]
C_g[\![\textbf{yield}]\!]_\tau & = & \tau = \langle \mathcal{I}, unit\rangle
\end{array}
$$

Figure 3.1: Constraint generation

3.3.2 Constraint generation.

The constraint generation algorithm C_g 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 parametrized by the expected type of the expression, as can be seen from the definition of C_g in Fig. 3.1. In order to simplify the presentation, we assume the existence of a **datacons** function which upon application to a data constructor κ and a domain type τ returns the type of the constructed value.

3.3.3 Constraint solving.

For any program P, the computed type constraint is $\mathcal{C} = \exists X.C_g[\![P]\!]_X$. $\mathcal{C}$ is then simplified into a list of type equations. This list is the unification problem we feed to our solver, which proceeds using a slightly adapted syntactic unification module, on top of an union-find algorithm. The flow equations are stored, as they can't be solved without the full types's shapes.

$$
\begin{array}{lcl}
C_s[\![True]\!]_\Gamma & = & \emptyset \\
C_s[\![C_0 \;\wedge\; C_1]\!]_\Gamma & = & C_s[\![C_0]\!]_\Gamma \cup C_s[\![C_1]\!]_\Gamma \\
C_s[\![\tau_0 \;=\; \tau_1]\!]_\Gamma & = & \{\tau_0 \;=\; \tau_1\} \\
C_s[\![\exists\, v.C]\!]_\Gamma & = & C_s[\![C]\!]_\Gamma \text{where } v \text{ is fresh} \\
C_s[\![\exists_F\, v_f.C]\!]_\Gamma & = & C_s[\![C]\!]_\Gamma \text{where } v_f \text{ is fresh} \\
C_s[\![def\ x\ :\ \chi\ in\ C]\!]_\Gamma & = & \text{if } \chi \text{ is a ground type, } C_s[\![C]\!]_{(x \,\mapsto\, \chi)::\Gamma} \\
 & & \text{if } \chi \text{ is a scheme } C_s[\![C]\!]_{(x \,\mapsto\, (\Gamma,\chi))::\Gamma} \\
C_s[\![inst\ x\ \tau]\!]_\Gamma & = & \text{if } \Gamma(x) \text{ is a ground type, } \{\Gamma(x) \;=\; \tau\} \\
 & & \text{if } \Gamma(x) \text{ is is a scheme } (\Gamma', \forall\, v[C].v), C_s[\![\exists\, v.C \wedge v \;=\; \tau]\!]_{\Gamma'}
\end{array}
$$

Figure 3.2: Constraint solving

The constraint solving algorithm is parametrized by an environment Γ from program variables to enriched constrained type schemes. When instantiating constraints, we must ensure that they are solved in their original environment, not the environment at instantiation point. Hence we enrich the type schemes with an environment, denoted by $(\Gamma, \forall\, v[C].v)$. The constraint solving algorithm C_s is defined in Fig. 3.2.

The unification module must solve the equations for the type *shapes* (as in any standard type inference system) but also for the *flows* contained in them. The method employed is simple: when unifying two terms, we compute the new flow as the minimum of the flows of the two terms being unified. The $\leq_F$ equations are accumulated during the solving phase and solved apart. There are many ways of achieving this, but we use an union-find tree parametrized with ad-hoc operations (merging two descriptors is defined as taking the minimum of two flows).

3.4 Monomorphisation.

The 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.

3.4.1 The instantiation graph.

In the following, we assume that each expression e is annotated with an unique integer id i, denoted e_i. We also assume that type schemes χ_i are annotated with the unique id i of the expression binding them. Finally, we ignore particular cases induced by the value restriction without loss of generality.

Any polymorphically typed function can in turn instantiate any previously defined polymorphic functions (except itself). Thus, the instantiation type of a function call may depend on the instantiation type of the enclosing function. This describes an instantiation graph. The main task of monomorphisation is to compute this graph, which is then easily used to specialize functions when they need to be. We define a *context* c to be either empty or equal to a couple (i, τ) formed by a polymorphic function and an instantiation type, where i is the unique id of the let-binding. The instantiation graph $\mathcal{G}$ is a mapping from non-empty contexts to instantiation types, indexed by program points : $\mathcal{G}\ (i_0, \tau)\ i_1$ is the type of the program point i_1 when the function i_0 is instantiated by τ. In order to compute $\mathcal{G}$ we parametrize C_s by a context, and the resolution rule for $inst$ is modified as follow :

$$
\begin{array}{lcl}
C_s[\![inst\ x_i\ \tau]\!]^c_\Gamma & = & \text{if } \Gamma(x)_j \text{ is a ground type,} \\
 & & \quad \text{If } c = \emptyset\text{, add } (j,\tau) \text{ to } \mathcal{G} \\
 & & \quad \text{If } c = (i_c, \tau_c)\text{, add a mapping from } c \text{ to } \tau \text{ indexed by } i \text{ in } \mathcal{G} \\
 & & \quad \{\Gamma(x) \;=\; \tau\} \\
 & & \text{if } \Gamma(x) \text{ is is a scheme } (\Gamma', (\forall\, v[C].v)_j), \\
 & & \quad \text{If } c = \emptyset\text{, add } (j,\tau) \text{ to } \mathcal{G} \\
 & & \quad \text{If } c = (i_c, \tau_c)\text{, add a mapping from } c \text{ to } \tau \text{ indexed by } i \text{ in } \mathcal{G} \\
 & & \quad C_s[\![\exists\, v.C \wedge v \;=\; \tau]\!]^{(j,\tau)}_{\Gamma'}
\end{array}
$$

Initially, the context parameter is empty.

3.4.2 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 x as we need. In order to avoid variable capture problem, we bind each specialized 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 monomorphisation. 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 $\mathcal{G}$. The e expression must also be recursively monomorphised, each point j in e beeing instantiated with the type $\mathcal{G}\ (i, \tau)\ j$.

Monomorphisation can make the code exponentially bigger, but this worst case is very unlikely in real-world, numerical code. On the other hand, we can then safely remove from the program any binding whose type still contains variables. Even more importantly, constraining each type to be ground allows `OCaml` to use efficient unboxed data representations wherever possible (e.g. arrays of floats). In our case, performing monomorphisation is a clear advantage from any standpoint.

3.5 Monoflowisation.

A step in the compilation process is what we call “monoflowisation”. This process is similar to monomorphisation, but instead of duplicating functions based on types, we duplicate them based on flows. This is of utmost importance for widely used functionals: if these functions 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. A solution would be to take the flows into account when doing the monomorphisation. Polymorphically-typed functions would then have a polyvariant behavior. From an implementer standpoint, it means that the instantiation graph should be able to distinguish instances based on their flows. Another approach would be to directly use a polyvariant flow, but it would have been far more difficult to prove and to implement.

3.6 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 may administrative redexes, which may greatly hamper the performance of the resulting program. To avoid them, we use the optimizing transformation defined in [15]. The transformation uses a “smart application” constructor $@_\beta$ which reduces on the fly administrative redexes: $\lambda x.b \mathbin{@_\beta} c$ = $[c/x]b$.

$$\begin{array}{lcl}
T_2[\![x]\!] \rhd k & = & k \mathbin{@_\beta} x \\
T_2[\![c]\!] \rhd k & = & k \mathbin{@_\beta} c \\
T_2[\![(\lambda x.e)^{\mathcal{I}}]\!] \rhd k & = & k \mathbin{@_\beta} (\lambda x.\lambda k.T_2[\![e]\!] \rhd k) \\
T_2[\![(\textbf{fix}\ h\ \lambda x.e)^{\mathcal{I}}]\!] \rhd k & = & k \mathbin{@_\beta} (\textbf{fix}\ h\ \lambda x.\lambda k.T_2[\![e]\!] \rhd k) \\
T_2[\![(e_1\ e_2^{\mathcal{I}})^{\mathcal{I}}]\!] \rhd k & = & T_2[\![e_1]\!] \rhd \lambda v_1.T_2[\![e_2]\!] \rhd \lambda v_2.v_1\ v_2\ k \\
T_2[\![(e_1\ e_2^{\mathcal{P}})^{\mathcal{I}}]\!] \rhd k & = & T_2[\![e_1]\!] \rhd \lambda v_1.v_1\ e_2\ k \\
T_2[\![(\textbf{let}\ v\ =\ e_1^{\mathcal{I}}\ \textbf{in}\ e_2)^{\mathcal{I}}]\!] \rhd k & = & T_2[\![e_1]\!] \rhd \lambda v.T_2[\![e_2]\!] \rhd k \\
T_2[\![(\textbf{let}\ v\ =\ e_1^{\mathcal{P}}\ \textbf{in}\ e_2)^{\mathcal{I}}]\!] \rhd k & = & \textbf{let}\ v\ =\ e_1\ \textbf{in}\ T_2[\![e_2]\!] \rhd k \\
T_2[\![(e_1,\ e_2)^{\mathcal{I}}]\!] \rhd k & = & T_2[\![e_1]\!] \rhd \lambda v_1.T_2[\![e_2]\!] \rhd \lambda v_2.k \mathbin{@_\beta} (v_1,\ v_2) \\
T_2[\![(\kappa\ e)^{\mathcal{I}}]\!] \rhd k & = & T_2[\![e]\!] \rhd \lambda v.k \mathbin{@_\beta} \kappa\ v \\
T_1[\![\textbf{match}\ e\ \textbf{with} & & \\
\quad \mid \kappa_i\ x_i \rightarrow\ e_i\ ]\!] \rhd k & = & T_2[\![e]\!] \rhd (\lambda v.\textbf{match}\ v\ \textbf{with} \\
 & & \qquad\qquad \mid \kappa_i\ x_i \rightarrow\ T_2[\![e_i]\!] \rhd k) \\
T_2[\![e^{\mathcal{P}}]\!] \rhd k & = & k\ e
\end{array}$$

This transformation generates no administrative redex, but has the flaw that we lose our semantic preservation theorem.

3.7 Implementation details and issues.

Our implementation language is the same as the target language : `OCaml`. The code itself is written in a purely functional style (the only exception being a unique id generator while creating the abstract syntax tree), allowing an “easier” job for future certified developments with e.g. Coq.

The most problematic part of the source language was constructs binding multiple variables at once, such as patterns. The constraint generation of these is quite involved, as is the monomorphisation of such bindings. We think that our whole code base’s size could be reduced by at least 30% if the source language were to have only one-variable bindings. Transforming the whole input code to an equivalent form with simple bindings could be worth considering, as it would also lower the gap between our formal developments and the language - but the added let-bindings could introduce the allocation of new values and a performance loss.

Also, mutually-recursive bindings are convertible into non-mutually recursive bindings by a simple transformation that we still haven’t implemented. The constraint generation for this kind of bindings would be a lot simpler if mutual recursion were to be hoisted. For the time beeing, we haven’t defined monomorphisation for these.

Chapter 4

Application to algorithmic skeletons

Algorithmic skeletons languages are generally defined by introducing a limited set of parallel patterns (also called operators or combinators) to be composed in order to build a full parallel application. In general, in skeleton languages, the only admitted parallelims is usually that of skeletons, in order to keep low the complexity of finding an efficient implementation. The expressiveness of the skeletal approach is achieved using as based skeletons a set of well-known and usefull parallel paradigm (farm, pipe, divide-and-conquer *etc.*).

We here present a naive implementation of the OCamlP3L skeletons language (P3L's set of skeletons for `OCaml`), based on our parallel superposition primitive.

4.1 Algorithmic skeletons approach

It observes that many parallel algorithms can be characterised and classified by their adherence to one or more of a number of generic patterns of computation and interaction. For example, many diverse applications share the underlying control and data flow of the pipeline paradigm [8].

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 these to enhance performance. The core principle of skeletal programming is conceptually straightforward. Its simplicity is a strength.

4.1.1 Definition, advantages and problems

The idea behind a strict skeletal approach is that such restriction of programs parallel structure does not affect the expressiveness of the methodology more than eliminating *goto* affected sequential imperative programming. On the contrary, skeletons programming helps in a making parallel software simple and easy to maintain since low-level details do not appear in the programs anymore.

Algorithmic skeletons are thus high-level primitives (also called "parallel patterns"[1]) designed to be safely composable[2]. Efficiency is usually easier to achieve since programs can be compiled by composing already optimised implementations. Skeletons languages are usually recognizes as a portable programming approach since they do not assume any particular features of the target machine.

Performance and easer programming have been the two main reasons to introduce skeletons, since the problem of exploiting the computational resources for parallel computing is harder than in the sequential case. A higher number of decisions should be taken when implementing parallel sofwares and so support is needed to make clever choices (e.g. mapping processes onto processors to optimise a given communication pattern is known to be in general a NP-hard problem and thus could be not not a full human decision procedure). The idea is that limiting the structure of the problem we also diminish the complexity of the implementation choices.

In this way, it promises to address many of the traditional issues within the parallel software engineering process:

- it will *simplify* programming by raising the level of abstraction;

[1]A generic definition of a pattern : it is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts.
[2]Definition of Murray Cole is *Each skeleton captures the essential structure of some particular problem solving style or technique.*

- it will enhance *portability* and *re-use* by absolving the programmer of responsibility for detailed realisation of the underlying patterns;
- it will improve *performance* by providing access to carefully optimised, architecture specific implementations of the patterns;
- it will offer scope for static and dynamic *optimisation*, by explicitly documenting information on algorithmic structure (e.g. sharing and dependencies) which would often be impossible to extract from equivalent unstructured programs.

Yet skeletal programming has still to make a substantial impact on mainstream practice in parallel applications programming. In contrast, MPI was designed to address similar issues (to varying degrees) and has proved very popular[3].

While initially targeting issues of synchronisation and nondeterminism more relevant to distributed computing, recent work has moved closer to the concerns of *High Performance Computing* (HPC) and the connection to skeletons has become increasingly apparent.

4.1.2 Disadvantages and mixing skeletons with ad-hoc parallelism

Despite all the advantages of the skeleton approach, they conserve some non-minor drawbacks. A well know disadvantage of skeleton language is the problem of efficiency of some combination of skeletons on certain archiectures. Finding the minimal set of skeletons (and have efficient and portable implementations) is still an open field of research.

Also, skeleton and pattern based parallel programming promise significant benefits but remain absent from mainstream practice because many parallel applications are not obviously expressible as instances of skeletons. Some have phases which require the use of less structured interaction primitives. Other applications have conceptually layered parallelism, in which skeletal behaviour at one layer controls the invocation of operations involving ad-hoc parallelism within. It is unrealistic to assume that skeletons can provide all the parallelism we need. Skeletons's languages must be constructed in the way to allow the integration of skeletal and ad-hoc parallelism in a welldefined way.

For example, MPI_Broadcast, MPI_Reduce and MPI's other collective operations are useful tools. However, the experienced parallel programmer is aware that there are other "patterns of computation and interaction" which occur in a range of applications but which are not catered for directly. For example, pipelines and task farms are well-established concepts, helpful during program design, but must be implemented directly in terms of MPI's simpler operations. The goal of the a skeleton library for MPI is to add such higher level collective operations to the MPI programmer's toolbox.

Despite parallel programming, using just skeleton to have parallel programs from sequential ones is quite reasonable when the goal is to build a stream processing network described by the skeletons. However, it has several drawbacks in the general case:

- **breaks uniformity** Though the skeletons look like ordinary functions, they are actually in different classes and can never been uniformly mixed together; hence, the programmers have to program in a style that strictly conforms with a two-level style, especially, in general, the skeletons cannot be invoked as ordinary functions from sequential code, even if they could have appropriate types.
- **may produce contrived programst** Many applications boil down to simple nested loops, some of which can be easily parallelized, and some cannot; in some cases, like numerical algorithms, what the user was really asking for, is the possibility of just parallelizing a particular very heavy computation deep inside the sequential code, while pure skeletons language enforced the user to rewrite all the program logic in a very unnatural way with some control parallel skeletons;
- **prevents sharing** In various numerical algorithms, some operation, like multiplying some vector v by the very same large matrix A, may be performed at different places of the sequential algorithm, and the user

[3]Noting the increasing stability and portability of direct parallel programming frameworks (and in particular MPI).

naturally wants to a way to assure that this computation be performed by the same processing resources (sharing the large matrix A). Most of pure skeleton languages does not allow the user to specify this sharing.

In this way, having skeleton in BSML would have the advantage of the BSP pattern of communications (collective ones) and the expressivity of the skeleton approach. Even, if the implementation is less efficient compare to a dedicated skeletons language (or MPI send/received implementation), programmer could compose skeleton when it is natural for him and used a BSP programming style when it is necessary. Note that, for the sequential parts (which cannot be parallelized) of a programs, BSML (and the BSP model) forces them to be replicated or in a specific processors but, in all the cases, as efficient as in a pure sequential implementation. This is the main advantage to have both BSP and skeletons paragdim in one shot.

For our purpose and to also have interesting benchmarks, we take for example the implemenation of the OCamlP3L skeletons language (P3L's set of skeletons for `OCaml`).

4.1.3 The P3L set of skeletons

Our work is about the P3L ("Pisa Parallel Programming Language") set of skeletons. P3L provides three kinds of skeletons: task parallel skeletons, data parallel skeletons and control skeletons. Each skeleton is a stream processor, i.e. a function which transforms an input stream of incoming data into an output stream of outgoing data. Skeletons can be composed to define the parallel behavior of programs.

Task parallel skeletons model the parallelism of independent processing activities related to different input data. They transform a stream of independent input data into a stream of results. Data parallel skeletons exploit parallelism in the computation of different parts of the same input data. Control skeletons are combinators which do not express parallelism *per se*, but orchestrate the interaction and control flow among the sequential and parallel parts of an application. The core P3L skeletons are:

- Task parallelism:
  - **pipe** A stream of tasks (the type of the stream is defined in the input list) flows along the stages, hence the input list of the stage i and the output list of the stage $i-1$ should coincide; The pipe exploits parallelism in the execution of a sequence of skeletons defining independent stages of a computation.
  - **farm** The computation on different input data items is executed in parallel over a set of worker; The stream of tasks with type defined in the inpuit list are distributed among the worker with some kind of load balancing strategies; The farm replicates a skeleton into a pool of identical copies (the farm workers) each one computing independent data items in the input stream.
- Data parallelism:
  - **map** modeling a data parallelism; a set of input structures is distributed according to a user-defined strategy and then the function is executed in parallel over a set of worker
  - **reduce** modeling binary tree computations of an associative operator on a n dimensional input parameter giving a $(n-1)$-th dimensional output parameter;
- Control parallelism:
  - **loop** modeling iterative computations; the execution of a function is repeated until a termination condition is true; at each iteration the argument of feedback becomes another input for the skeleton,
  - **seq** modelling the inclusion of a sequential code, written in the sequential host language; this code must have no side-effect

The possibility of nesting skeletons is the most relevant feature of such a language. The P3L set of skeletons has been added to many imperative language as C++ or Java. OCamlP3L is an extention of `OCaml` with an adapation of the P3L skeletons for functionnal programming. This set of skeletons would be our case study.

val seq : (unit $\rightarrow \alpha \rightarrow \beta$) $\rightarrow$unit $\rightarrow \alpha$ stream $\rightarrow \beta$ stream

val parfun : (unit $\rightarrow$unit $\rightarrow \alpha$ stream $\rightarrow \beta$ stream) $\rightarrow \alpha$ stream $\rightarrow \beta$ stream

val pardo : (unit $\rightarrow \alpha$) $\rightarrow \alpha$

val loop : (α $\rightarrow$bool) $*$ (unit $\rightarrow \alpha$ stream $\rightarrow \alpha$ stream) $\rightarrow$unit $\rightarrow \alpha$ stream $\rightarrow \alpha$ stream

val farm : (unit $\rightarrow \beta$ stream $\rightarrow \gamma$ stream) $*$ int $\rightarrow$unit $\rightarrow \beta$ stream $\rightarrow \gamma$ stream

val (|||) : (unit $\rightarrow \alpha$ stream $\rightarrow \beta$ stream) $\rightarrow$(unit $\rightarrow \beta$ stream $\rightarrow \gamma$ stream) $\rightarrow$unit $\rightarrow \alpha$ stream $\rightarrow \gamma$ stream

val mapvector : (unit $\rightarrow \beta$ stream $\rightarrow \gamma$ stream) $*$ int $\rightarrow$unit $\rightarrow \beta$ array stream $\rightarrow \gamma$ array stream

val reducevector : (unit $\rightarrow$(β $*$ β) stream $\rightarrow \beta$ stream) $*$ int $\rightarrow$unit $\rightarrow \beta$ array stream $\rightarrow \beta$ stream

Figure 4.1: The (complete) types of the OCamlP3L skeleton combinators

4.2 The OCamlP3L Skeletons

OcamlP3L imported the skeletal model proposed by P3L with some minor changes due to the functional nature of the OCaml. The OCamlP3L system is a programming environment that provides a skeletal model for OCaml and at the same time provides seamless integration of parallel programming and functional programming with advanced features like sequential logical debugging (i.e. functional debugging of a parallel program *via* execution of all parallel code onto a sequential machine) and strong typing, useful both as a testbed for innovative parallel programming style and a practical tool in building full-scale applications for scientific computation.

Figure 4.1 resumes the ML type of the OCamlP3L skeletons. Next of the text comes from the usual manual of OCamlP3L.

4.2.1 The seq skeleton

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. Any `OCaml` function with type (unit$\rightarrow \alpha \rightarrow \beta$) can be encapsulated in the **seq** skeletons as follows: seq f. The central point is that the function must be unary, i.e. functions working on more that one argument must collect them in a single tuple before being used in a **seq**.

4.2.2 The farm skeleton

The **farm** skeleton computes in parallel a function f over different data items appearing in its input stream. From a functional viewpoint, given a stream of data items $x_1, \ldots, x_n$ and a function f, the expression **farm**(f, k) computes $f(x_1), \ldots, f(x_n)$. Parallelism is gained by having k independent processes that compute f on different items of the input stream. If f has type (unit$\rightarrow \beta$ stream$\rightarrow \gamma$ stream), and k has type int, then **farm**(f, k) has type unit$\rightarrow \beta$ stream$\rightarrow \gamma$ stream. In terms of (parallel) processes, a sequence of data appearing onto the input stream of a farm is submitted to a set of worker processes. Each worker applies the same function (f, which can be in turn difined using parallel skeletons) to the data items received and delivers the result onto the output stream. The resulting process network looks like these of Figure 4.2 where the emitter process takes care of task-to-worker scheduling (possibly taking into account some load balancing strategy).

The `farm` function takes two parameters:

- the first denoting the skeleton expression representing the farm worker computation,
- the second denoting the parallelism degree the user decided for the farm, i.e. the number of worker processes that have to be set up in the farm implementation.

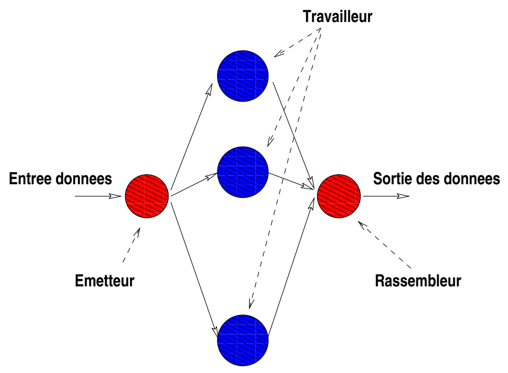


Figure 4.2: The process network of a **farm** skeleton

4.2.3 The pipeline skeleton

The pipeline skeleton is denoted by the infix operator |||; it performs in parallel the computations relative to different stages of a function composition over different data items of the input stream.

Functionally, f_1 ||| $f_2 \ldots$ ||| f_n computes $f_n(\ldots f_2(f_1(x_i))\ldots)$ over all the data items x_i in the input stream. Parallelism is now gained by having n independent parallel processes. Each process computes a function f_i over the data items produced by the process computing $f_i - 1$ and delivers its results to the process computing $f_i + 1$. If f_1 has type (unit→α stream→β stream), and f_2 has type (unit→β stream→γ stream), then f_1 ||| f_2 has type unit→α stream→γ stream.

In terms of (parallel) processes, a sequence of data appearing onto the input stream of a pipe is submitted to the first pipeline stage. This stage computes the function f_1 onto every data item appearing onto the input stream. Each output data item computed by the stage is submitted to the second stage, computing the function f_2 and so on and so on until the output of the $n-1$ stage is submitted to the last stage. Eventually, the last stage delivers its own output onto the pipeline output channel. The resulting process network looks like these of Figure 4.3.

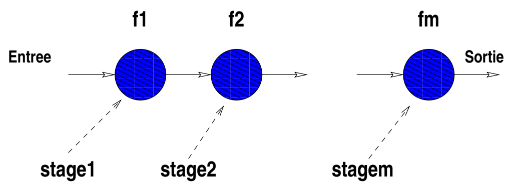


Figure 4.3: The process network of a **pipe** skeleton

4.2.4 The loop skeleton

The **loop** skeleton computes a function f over all the elements of its input stream until a boolean condition g is verified. A loop has type (α →bool) $*$ (unit →α stream →α stream) provided that f has type unit→α stream→α stream and g has type α →bool.

In terms of (parallel) processes, a sequence of data appearing onto the input stream of a loop is submitted to a *loop in* stage. This stage just merges data coming from the input channel and from the feedback channel and delivers them to the *loop body* stage. The loop body stage computes f and delivers results to the *loop end* stage. This latter stage computes g and either delivers (f x onto the output channel (in case (g (f x)) turns out to be `true`) or it delivers the value to the loop in process along the feedback channel (($g(fx)$) =`false`). The resulting process network looks like these of Figure 4.4.

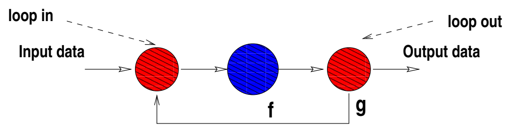


Figure 4.4: The process network of a **loop** skeleton

4.2.5 The map skeleton

The map skeleton is named **mapvector**; it computes in parallel a function over all the data items of a vector, generating the (new) vector of the results.

Therefore, for each vector X in the input data stream, **mapvector** (f, n) computes the function f over all the items of $X = [x_1, \ldots, x_n]$, using n distinct processes that compute f over distinct vector items $([f(x_1), \ldots, f(x_n)])$.

If f has type (unit→α stream→β stream), and n has type int, then **mapvector** (f, n) has type unit→α array stream→β

In terms of (parallel) processes, a vector appearing onto the input stream of a **mapvector** is split n elements and each element is computed by one of the n workers. Workers apply f to the elements they receive. A collector process is in charge of gluing together all the results in a single result vector (see Figure 4.5).

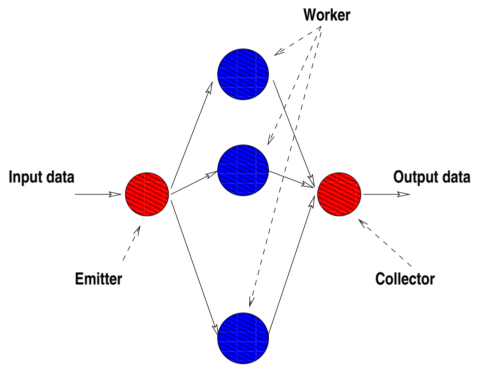


Figure 4.5: The process network of a **map** skeleton

Different strategies can be used to distribute a vector $[|x_1; \cdots; x_m|]$ appearing in the input data stream to the workers. As an example the emitter:

- may round robin each x_i to the workers $(w_1, \cdots, w_n)$. The workers in this case simply compute the function $f\ \alpha \rightarrow \beta$ over all the elements appearing onto their input stream (channel).
- may split the input data vector in exactly n sub-vectors to be delivered one to each one of the worker processes. The workers in this case compute an Array.map f over all the elements appearing onto their input stream (channel).

Summarizing, the emitter process takes care of (sub)task-to-worker scheduling (possibly implementing some kind of load balancing policy), while the collector process takes care of rebuilding the vector with the output data items and of delivering the new vector onto the output data stream. **mapvector** takes two arguments:

- the skeleton expression denoting the function to be applied to all the vector elements, and
- the parallelism degree of the skeleton, i.e. the number of processes to be used in the implementation.

4.2.6 The reduce skeleton

The reduce skeleton is named **reducevector**; it folds a function over all the data items of a vector.

Therefore, **reducevector** $(\oplus, n)$ computes $x_1 \oplus x_2 \oplus \ldots \oplus x_n$ out of the vector $x_1, \ldots, x_n$, for each vector in the input data stream. The computation is performed using n different parallel processes that compute f.

If $\oplus$ has type (unit→α $*\alpha$ stream→α stream), and n has type int, then **reducevector** $(\oplus, n)$ has type unit→α array strea

In terms of (parallel) processes, a vector appearing onto the input stream of a reducevector is processed by a logical tree of processes. Each process is able to compute the binary operator g. The resulting process network looks like the tree of Figure 4.6.

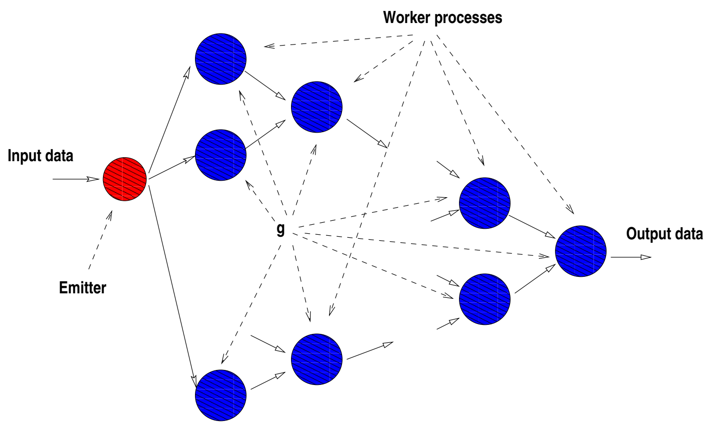


Figure 4.6: The process network of a **reduce** skeleton

In this case, the emitter process is the one delivering either couples of input vector data items or couples of sub-vectors of the input vector to the processes belonging to the tree base. In the former case, $\log(n)$ levels of processes are needed in the tree, in the latter one, any number of process levels can be used, and the number of sub-vectors to be produced by the emitter can be devised consequently.

The **reducevector** function takes two parameters as usual:

- the first parameter is the skeleton expression denoting the binary, associative and commutative operation (these properties must be ensured by the programmer to have a correct execution)
- the second is the parallelism degree, i.e. the number of parallel processes that have to be set up to execute the **reducevector** computation.

4.2.7 The parfun and pardo skeletons

The **parfun** (of type (unit→unit→α stream→β stream)→α stream→β stream) skeleton is the very dual of the **seq** skeleton. In simple words, one used to warp a regular function to be a skeleton unit with **seq**, now one can also wrap a full skeleton expression inside a **parfun** to obtain a regular stream processing function, usable with no limitations in any sequential piece of code. A **parfun** encapsulated skeleton function behaves exactly as a normal function that receives a stream as input value, and returns a stream as output value.

Finally, the **pardo** of type (unit→α)→α combinator defines the scope of the expressions that may use the **parfun** encapsulated skeleton expressions.

In order to have the **parfun** and **pardo** work correctly together the following scoping rule has to be sollowed:

- functions defined via the **parfun** combinator must be *defined before* the occurrence of the **pardo** combinator,
- those **parfun** defined functions can only be *executed within* the body of the functional parameter of the **pardo** combinator,
- no **parfun** can be used directly inside a **pardo** combinator.

Thus, due to the scoping rule in the **pardo**, the general structure of an OCamlP3L program looks like the following:

```
(* (1) Functions defined using parfun *)
let f = parfun(skeleton_expression)
let g = parfun(skeleton_expression)
```

```
(∗ (2) code referencing these functions under abstractions ∗)
let h x = ... (f ...) ... (g ...) ...
...
(∗ NO evaluation of code containing a parfun is allowed outside pardo ∗)
...
(∗ (3) The pardo occurrence where parfun encapsulated functions can be called. ∗)
pardo
 (fun () →
    (∗ NO parfun combinators allowed here ∗)
    (∗ code evaluating parfun defined functions ∗)
    ...
    let a = f ...
    let b = h ...
    ...)
(∗ finalization of sequential code here ∗)
```

4.2.8 Load balancing: the colors

In the OCamlP3L system, the combinators expressions govern the shape of the process network and the execution model assumes a “virtual” processor, for each process. The mapping of virtual to physical processors is delegated to the OCamlP3L system. The mapping is currently not optimized in the system. However, programs and machines can be annotated by the programmer using *colors*, which can pilote the virtual-to-physical mapping process.

The idea is to have the programmer to rank the relative “weight” of skeleton instances and the machine power in a range of integer values (the colors). Then, weights are used to generate a mapping in which load is evenly balanced on the partecipating machine according to their relative power.

We do not present this feature here (and refear to the OCamlP3L manual for more details) because we do not used it for our implementation.

4.3 BSML Implementation.

At this time, the approach taken when implementing these skeletons in BSML was relatively naive. But 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 stuck with TCP/IP. Thus, our skeletons can run on high-end parallel hardware.

For simplicity, we although generate the program in meta fashion as a simple string, we will use a MetaOCaml-like syntax: the meta-code will be quoted between “.<>.”.

4.3.1 Execution of process networks

The combination of P3L’s skeletons generate a process network (a graph). This network takes in input a stream of data. Then each datum is transformated by the network independtly of other data and finally the ouput is another stream of data of the same arity.

The most important information is that each execution of a process network is completly independant from another ones. In this way, they can be composed. We will used our parallel composition operator (the **superposition**) to do that. Thus, if we suppose that the stream contains n data, n times the execution of the network would be composed using the **superpositon**. The general execution of a network (skeleton pardo) is coded as follow:

```
let pardo eval_net datas =
 super_list (Array.to_list (Array.mapi
  (fun i d () →eval_net (ref (i mod (bsp_p()))) d) (Array.of_list datas)))
```

if we suppose that datas is the stream of data (here a list) and eval_net is the execution of a process network. Each evaluation is depending of a reference counter parameter which design the placement of the first computation. This placement of the tasks, using this counter, is just a naive round robin.

To n-compose execution of the networks, we used super_list which is a simple n-ary superposition:

```
let rec super_list = function
    [] →[]
 | hd::tl →let nhd,ntl = super hd (fun () →super_list tl)
                    in nhd::ntl
```

To execute just one network, we thus need as parameters a counter and a data. Then, we define a function that incremente this counter (modulo p), defined the triplet which represent the network (input,output and the parallel stream computation) from a skeleton expression and execute it. This is the goal of the $eval\_net$ function which have the following code (generated from a skeleton expression s):

```
 let eval_net place data =
    let noSome (Some x) = x in
    let incr_place () = place:=(!place+1) mod (bsp_p()) in
    let inf,outf,flow = .<bsml_trans s>. in
     noSome ((proj (flow (mkpar (fun pid →if pid=inf then Some data else None)))) outf)
```

flow is the execution of the network which have inf has processor input and outf as output. After the execution of the network, the result is globally exchanged using the primitive **proj**. The stream is created using **mkpar** such that only processor inf has a non empty value.

Now, let see how to produce this process network from a skeleton expression (function $bsml\_trans$).

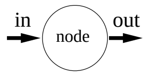
4.3.2 From skeletons to BSML codes

Our implementation takes a skeleton description and generates BSML code from it which symbolise the execution of the network. Our expressions of skeletons are represented by the following type:

```
type skel_tree =
    Seq of code_ocaml
 | Pipe of skel_tree ∗ skel_tree
 | Farm of int ∗ skel_tree
 | MapVector of int ∗ skel_tree
 | Loop of code_ocaml ∗ skel_tree
```

Noting that once can thinks that we can implement OCamlP3L's skeletons as a library of combinators for BSML. This is certainly possible but not in a natural way and this further work is not clear to be very usefull.

In our setting, a skeleton (sub-part of the network) is the data of an *input* CPU, an *output* CPU and a BSML function:



where "node" is a function that takes a data from processor "in" and return a data to processor "out". This will be implemented in BSML as a triplet int∗int∗(α option **par**→β option **par**) where the two first parameters are respectively input and output of the network and the last one, the function that compute the data on the stream. The P3L stream is implemented as a parallel vector of option value where one and only one processor keeps a non empty value (the data of the stream). The full stream could be thus a list of these vectors.

The BSML code is recursively generated on a skeleton expression using a function **bsml_trans** : $skel\_tree \rightarrow code\_BSML$, from which we will excerpt the relevant parts.

Each skeleton is parametrized by a placement reference (the place counter). This allows to distribute the tasks in a round robin fashion. Also, they are parametrized by a data variable, representing the data stream.

In order to transfer data from CPU to CPU, we define the tools function sendto:

```
let sendto outf inf data =
   apply (put (apply (mkpar (fun pid →
      if pid=outf then
         (fun d dest →if dest=inf then d else None)
      else
         (fun _ _ →None)))
      data)) (replicate outf)
```

which sends a data from processor outf to processor inf.

In Figure in the next of the text, the big arrow represent the function $bsml\_trans$ and its recursive calls (generate a new network from another networks).

4.3.3 Implementation of seq(f).

Given a sequential OCaml function f, the generated code for this skeleton is:

```
let pl=(!place) in
  incr_place ();
  (pl,pl,(fun data →
    apply (mkpar (fun pid →
      if pid=pl
      then (function Some d →(Some ((.<f>. ()) d)) | None →None)
      else (fun _ →None)))) data)
```

that is the resulting network is thus the triplet (pl,pl,(**fun** data →new_data)) where the function f only executes itself on the designated CPU (pl) designated by the counter place), returning None elsewhere.

4.3.4 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 (n in this skeleton expression) is thus all the time p. This is not a problem since be distributed the workers in a round robin manner (even if this is naive).

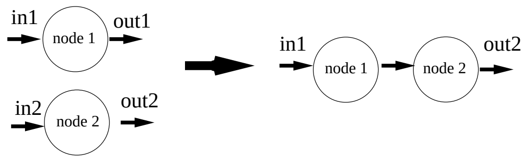
Thus, the code generated for **farm**(n, s) is simply the code generated for the skeleton s.

4.3.5 Implementation of pipeline(s_1,s_2).

The generated code for this skeleton is:

```
let in1,out1,flow1= .<bsml_trans s1>.
and in2,out2,flow2= .<bsml_trans s2>. in
  if (out1=in2) then
    (in1,out2,(fun data→ flow2 (flow1 data)))
  else
    (in1,out2,(fun data →flow2 (sendto out1 in2 (flow1 data))))
```

that is we recursively genereted the triplet representing the networks for s_1 and s_2 and compact then to generate a new triplet refleting the pipe network:



Then, if the output of s_1 and the input of s_2 are on the same CPU, we directly compose them. If they are on distinct CPUs, we perform a sendto to connect the output of s_1 to the input of s_2. This is exactly what is reflected in the code. As attended, the input of the resulting network is the input of the network of s_1 and the output that of s_2.

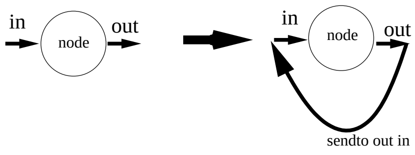
Noting that for a BSP machine with p processors and a skeleton expression using one pipe of two sequential processus, the tasks would be distributed on all processors (we suppose a typical stream of more than p elements). Then, a single barrier would occurs sending data from a processor to another one. This is clearly not the most efficient manner to execute the whole program but this is also clearly not an inefficient one.

4.3.6 Implementation of loop(f,s).

Our implementation of **loop** is a simple recursive function, which executes the s skeleton until the f condition holds true:

```
let i,o,task= .<bsml_trans s>. in
  (i,o, (let rec tmp data =
    if (proj (applyat i (fun d →.<f>. (noSome d)) (fun _ →false) data) i)
    then (sendto i o data)
    else tmp (sendto o i (task data))))
```

that is:



4.3.7 Implementation of map(n,s).

Our implementation of **mapvector** is probably the most interesting one. Once again, the n parameter (parallelism degree) is not unused due to a fix number of processor and our round robin strategy for the placement of the tasks. The method for this skeleton is as follow:

First, a new task is dynamically created for each element of the input vector of the stream and stored in the list of tasks call ntasks!. Each task is the execution of the network generated by the recursive call of $bsml\_trans$ on s.

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

Finnaly, once the processing terminated, the function tools **rebuild** gathers the results to the processor which has been designated to be the output of the full generated network. 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_trans 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)))
```

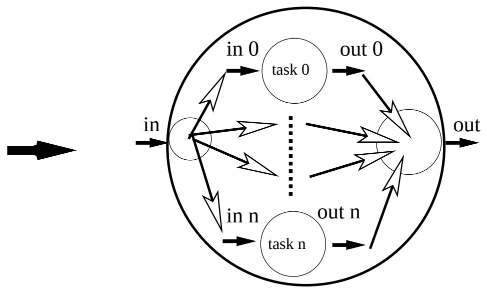
Where **rebuild** transforms a list of vector (data only on processor pl) to a vector of array (array only on processor pl):

```
let rebuild pl l_vector =
 let rec tmp = function
   [] →mkpar (fun pid →if pid=pl then Some [] else None)
 | hd::tl →parfun2 (function (Some d) →(function (Some l) →(Some (d::l)) | None →None) | None →(fun _ →None))
                       hd (tmp tl)
 in parfun (function Some l →Some (Array.of_list l) | None →None) (tmp l_vector)
```

```
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 4.7: Skeleton code fragment from a Poisson solver

This figure resume the idea of the implementation where n is the dynamic size of the input vector :



This skeleton is a good sample where system threads would not be sufficient: the size of typical data of skeletons programs would make the old implementation of the superposition unusable.

4.4 Examples

4.4.1 Code generation of a simple skeleton expression

Take the following skeleton expression:

```
Pipe((Seq (fun () x →float (x+1))),(Seq (fun () x →x∗.2.)))
```

Using our code generation, we will obtain the following BSML code:

```
let eval_net place data =
(∗ tools ∗)
 let noSome (Some x) = x in
(∗ increment counter ∗)
 let incr_place () = place:=(!place+1) mod (bsp_p()) in
(∗ first process of the pipe ∗)
 let inf,outf,flow = ((let in1,out1,flow1= (let pl=(!place) in incr_place ();(pl,pl,(fun data →
   apply (mkpar (fun pid →if pid=pl
     then (function Some d →(Some (((fun () x →float (x+1)) ()) d)) | None →None)
       else (fun _ →None)))) data))
(∗ second process of the pipe ∗)
 and in2,out2,flow2= (let pl=(!place) in incr_place ();(pl,pl,(fun data →
    apply (mkpar (fun pid →if pid=pl
     then (function Some d →(Some (((fun () x →x∗.2.) ()) d)) | None →None)
       else (fun _ →None)))) data)) in
(∗ combination of these twi processes ∗)
 if (out1=in2) then (in1,out2,(fun data→ flow2 (flow1 data)))
    else (in1,out2,(fun data →flow2 (sendto out1 in2 (flow1 data)))))) in
(∗ final execution of the network ∗)
  noSome ((proj (flow (mkpar (fun pid →if pid=inf then Some data else None)))) outf)
```

4.4.2 A PDE solver on multiple domains

Our second example is a parallel PDE solver which works on a set of subdomains, taken from [11]. On each subdomain it applies a fast Poisson solver written in C. The skeleton expression of the code is shown in Figure 4.7 and the coupling technique (and full equations) could be find in [11].

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 (Ubuntu as OS).

We present the benchmarks when the interface meshes match using random generated sub-domains (different cases on real life inputs are described in [11]). The principle of this extensibility test is as follow: increases the number of processors as well as size of data. The goal is to keep as much as possible a constant computation time, although the overall number of tasks is increased and communications are required to couple the global problem. 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 ... 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). There will be at least as many generated super-threads.

Performances (minutes and seconds) of OCamlP3l and skeletons in 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 that much. Barriers slow down the whole program but bulk-sending accelerates the communications: in the P3L running 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 division and distribution of the data.

Chapter 5

Conclusions

5.1 Related works.

5.1.1 Divide-and-conquer paradigms.

A general data-parallel formulation for a class of divide-and-conquer problems was evaluated in [2]. A combination of techniques are used to reorganise the algorithmic data-flow, providing great flexibility to efficiently exploit data locality and to reduce communications. But those techniques are only defined for a low-level parallel language, High Performance Fortran [28]. In [22], the authors present a new data-parallel C library for Intel's core-processors which have a divide-and-conquer primitive where some optimisations have been done using the BSP model.

Many algorithmic skeletons languages offers divide-and-conquer skeletons. Different optimisations have been designed for performance issues.

A methodology (and a language in [25]) based on a space-time mapping is presented in [24]. It uses a geometric computational model based on coordinate transformations with which time (the schedule) and space (the processor) can be made explicit. This technique may be applied to a class of divide-and-conquer recursions, resulting in a functional program skeleton and its parallel implementation with MPI. But cost prediction is too hard, making algorithmic optimisations harder than in the BSP model.

In [26], the proposed approach distinguished three levels of abstraction and their instantiations. (1), a ML like language defines the static parallel parts of the programs. The language comes with a partial evaluator which acts as a code transformer using MetaOCaml. (2), an implementation of a divide-and-conquer skeleton demonstrates how meta-programming can generate the appropriate set of communications for a particular process from an abstract specification. (3), the application programmer composes the program using skeletons, without the need to consider details of parallelism. However, no cost prediction nor efficient code generation are possible. For efficient code, [16] proposes the same approach using C++ templates. A task skeletons C++ language is compiled into a C++ MPI code but no divide-and-conquer skeleton is at this time provided. Also, in [4], the authors have implemented a divide-and-conquer skeleton using the functional and parallel language Eden.

5.1.2 CPS transformations.

CPS has been first introduced in [30] for semantics purpose and were massively used for various implementations of Scheme and ML the language [1]. The original CPS transformation was the most simple one; this transformation introduce too many unnecessary reductions (called administrative redexes) and more efficients CPS were defined in [14].

The idea of using CPS (or the call-cc operator[1]) for thread implementation comes from [38]. Then, many authors used them to implement multi-threaded extensions of sequential languages [34, 32] such as ML [12, 5], Java [3] or C [10]. Most of the time, a call-cc operator is used but some of them use a CPS transformation. In [17], the authors present some generic tricks to easily add efficient threads in a sequential language. But they suppose

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

a call-cc operator that does not exist in `OCaml` (at least not in an efficient form). Finally, the STALIN Scheme[2] compiler seems to implement a form of a flow-directed CPS conversion.

We can notice that the formal proof using a proof assistant that programs can be systematically translated to semantically equivalent CPS programs is a new field of research [15] and can be useful for our purpose.

5.2 Conclusion.

We have formally defined a new implementation of a multi-threading primitive (called parallel superposition) for a high-level BSP and data-parallel language. Efficiency concerns lead us to use lightweight threads, whose implementation relies on a global flow-directed CPS transformation. Our flow analysis is defined as an instrumented type system, allowing us to both guide and feed the partial CPS transformation. In order to compile polymorphically typed programs, we perform a monomorphisation pass which (in conjunction with defunctorisation) also fosters the efficiency of numerical code. Our implementation relies on a semantic investigation, allowing us to better trust our transformations - and it works on an important subset of the `OCaml` language.

The ease of use of this new implementation of the superposition will be experimented by implementing BSP algorithms described as divide-and-conquer algorithms in the literature and creating a less naive implementation of the ocamlp3l skeletons [13]. Our current implementation distributes the tasks in a simple round robin fashion; and in the interest of load balancing a smarter heuristic could be developed.

We will also investigate new kinds of optimisations such as a polyvariant flow analysis to generate less CPS code (which are less efficients in `OCaml` than direct-style ones).

[2] See ftp://ftp.ecn.purdue.edu/qobi/research-statement.pdf.

Bibliography

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

[2] M. Aumor, F. Arguello, J. Lopez, O. Plata, and L. Zapata. A data-parallel formulation for divide-and-conquer algorithms. *The Computer Journal*, 44(4):303–320, 2001.

[3] A. Begel, J. MacDonald, and M. Shilman. Picothreads: Light-weight threads in java. Technical report, Microsoft research, 2002.

[4] J. Berthold and R. Loogen. Analysing dynamic channels for topology skeletons in eden. Technical Report 0408, Institut für Informatik, Lübeck, September 2004. (IFL'04 workshop), C. Grelck and F. Huch eds.

[5] E. Biagioni, K. Cline, P. Lee, C. Okasaki, and C. Stone. Safe-for-space threads in standard ml. *Higher-Order and Symbolic Computation*, 11(2):209–225, 1998.

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

[7] O. Bonorden, B. Juurlink, I. Von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

[8] P. Brinch-Hansen. *Studies in Computational Science: Parallel Programming Paradigms*. Prentice-Hall, 1995.

[9] A. Chan, F. Dehne, and R. Taylor. Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *Journal of High Performance Computing Applications*, 2005. to appear.

[10] J. Chroboczek. Continuation Passing for C: A space-efficient implementation of concurrency. Technical report, PPS (University of Paris 7), 2005.

[11] F. Clément, V. Martin, A. Vodicka, R. Di Cosmo, and P. Weis. Domain Decomposition and Skeleton Programming with OCamlP3l. *Parallel Computing*, 32:539–550, 2006.

[12] E. C. Cooper and I. G. Morrisett. Adding threads to standard ml. Technical report, School of Computer Science, Carnegie Mellon University, 1990.

[13] R. Di Cosmo, Z. Li, S. Pelagatti, and P. Weis. Skeletal Parallel Programming with OcamlP3L 2.0. *Parallel Processing Letters*, 18(1), 2008.

[14] O. Danvy and L. R. Nielsen. Cps transformation of beta-redexes. *Information Processing Letterseses*, 94(5):217–225, 2005.

[15] Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In *Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR*, volume 4790 of *LNAI*, pages 211–225. Springer, 2007.

[16] J. Falcou, J. Serot, T. Chateau, and J. T. Lapreste. QUAFF : Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32(7-8):604–615, 2006.

[17] M. Gasbichler, E. Knauel, M. Sperber, and R. A. Kelsey. How to Add Threads to a Sequential Language Without Getting Tangled Up. In *Scheme Workshop 2003*, 2003.

[18] F. Gava. *Approches fonctionnelles de la programmation parallèle et des méta-ordinateurs. Sémantiques, implantations et certification*. PhD thesis, University Paris XII-Val de Marne, LACL, 2005.

[19] F. Gava. Implementation of the Parallel Superposition in Bulk-Synchronous Parallel ML. In Y. Shi, G.D.v. Albada, J. Dongarra, and P.M.A. Sloot, editors, *The International Conference on Computational Science (ICCS), Part I*, volume 4487 of *LNCS*, pages 611–619. Springer-Verlag, 2007.

[20] F. Gava. A Modular Implementation of Parallel Data Structures in BSML. *Parallel Processing Letters*, 18(1):39–53, 2008.

[21] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

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

[23] N. Heintze. Control-Flow Analysis and Type Systems. In A. Mycroft, editor, *Static Analysis Symposium (SAS)*, number 983 in LNCS. Springer, 1995.

[24] C. A. Hermann and C. Lengauer. On the space-time mapping of a class of divide-and-conquer recursions. *Parallel Processing Letter*, 6:525–537, 1996.

[25] C. A. Hermann and C. Lengauer. Hdc: A high-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2-3):239–250, 2000.

[26] C. A. Herrmann. Functional meta-programming in the construction of parallel programs. *Parallel Processing Letters*, 2005. to appear.

[27] J. Kim and K. Yi. Interconnecting between CPS terms and non-CPS terms. In A. Sabry, editor, *Third ACM SIGPLAN Workshop on Continuations (CW)*, number 545 in Technical Report. Computer Science Department, Indiana University, 2001.

[28] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[29] F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *International Conference on Computational Science (ICCS 2003), Part III*, number 2659 in LNCS, pages 223–232. Springer Verlag, june 2003.

[30] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[31] F. Pottier and D. Rémy. The Essence of ML Type Inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

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

[33] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

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

[35] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, Oxford University Computing Laboratory, 1998.

[36] A. Tiskin. A New Way to Divide and Conquer. *Parallel Processing Letters*, 11(4):409–422, 2001.

[37] P. Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–56, 1994.

[38] M. Wand. Continuation-based multiprocessing. In *Lisp Conference*, pages 19–28. ACM, 1980.

Appendix A

Longer BSML examples

To illustrate our programming language BSML, we present 2 classic problems: sieve of Eratosthenes and a parallel sorting.

A.1 Sieve of Eratosthenes

The sieve of Eratosthenes generates a list of primary numbers below a given integer n. We study 3 parallelization methods. We generate only the integers that are not multiple of the 4 first prime numbers and we classically iterate only to $\sqrt{n}$.

Fig. A.1 gives the BSML code of the 3 methods. We used the following functions: elim:int list→int→int list which deletes from a list all the integers multiple of the given parameter; final_elim:int list→int list→int list iterates elim; seq_generate:int →int→int list which returns the list of integers between 2 bounds; and select:int →int list→int list which gives the $\sqrt{n}$th first prime numbers of a list.

A.1.1 Logarithmic reduce method.

For our first method we use the classical parallel prefix computation (also call folding reduce) :

$$\textsf{scan}\ \oplus\ \boxed{v_0 \mid \cdots \mid v_{p-1}} = \boxed{v_0 \mid v_0 \oplus v_1 \mid \cdots \mid \oplus_{k=0}^{p-1} v_k}$$

We use a divide-and-conquer BSP algorithm (implemented using the **super** primitive) where the processors are divided into two parts and the scan is recursively applied to those parts; the value held by the last processor of the first part is broadcasted to all the processors of the second part, then this value and the values held locally are combined together by the associative operator $\oplus$ on the second part. In our computation, the sent values are first modified by a given function (select to just sent the $\sqrt{n}$th first prime numbers)

The parallel methods is thus very simple: each processor i holds the integers between $i \times \frac{n}{\mathbf{p}} + 1$ and $(i+1) \times \frac{n}{\mathbf{p}}$. Each processor computes a local sieve (the processor 0 contains thus the first prime numbers) and then our scan is applied. We then eliminate on processor i the integers that are multiple of integers of processors $i-1$, $i-2$, *etc.*

A.1.2 Direct method.

It is easy to see that our initial distribution (bloc of integers) gives a bad load balancing (processor $\mathbf{p}-1$ has the bigger integers which have little probability to be prime). We will distributes integers in a cyclic way: a is given to processor i where $a \bmod \mathbf{p} = i$). The second method works as follows: each processor computes a local sieve; then integers that are less to $\sqrt{n}$ are globally exchanged; a new sieve is applied to this list of integers (thus giving prime numbers) and each processor eliminates, in its own list, integers that are multiples of this $\sqrt{n}$th first primes.

A.1.3 Recursive method.

Our last method is based on the generation of the $\sqrt{n}$th first primes and elimination of the multiples of this list of integers. We generate this by a inductive function on n. We suppose that the inductive step gives the $\sqrt{n}$th first

```
let eratosthene_scan n =
  let p=bsp_p() in
  let listes = mkpar (fun pid→ if pid=0 then seq_generate (n/p) 10
                                else seq_generate ((pid+1)∗(n/p)) (pid∗(n/p)+1)) in
  let local_eras = parfun (local_eratosthene n) listes in
  let scan_era = scan_super final_elim (select n) local_eras in
   applyat 0 (fun l →2::3::5::7::l) (fun l→l) scan_era

let eratosthene_direct n =
 let listes = mkpar (fun pid→ local_generation n pid) in
 let etape1 = parfun (local_eratosthene n) listes in
 let selects = parfun (select n) etape1 in
 let echanges = replicate_total_exchange selects in
 let premiers = local_eratosthene n
                                 (List.fold_left (List.merge compare) [] echanges) in
 let etape2 = parfun (final_elim premiers) etape1 in
  applyat 0 (fun l→2::3::5::7::(premiers@l)) (fun l→l) etape2

let rec eratosthene n =
  if (fin_recursion n) then apply (mkpar distribution) (replicate (seq_eratosthene n))
  else
   let carre_n = int_of_float (sqrt (float_of_int n)) in
   let prems_distr = eratosthene carre_n in
   let listes = mkpar (fun pid →local_generation2 n carre_n pid) in
   let echanges = replicate_total_exchange prems_distr in
    let prems = (List.fold_left (List.merge compare) [] echanges) in
     parfun (final_elim prems) listes
let eratosthene_rec n =
 applyat 0 (fun l→2::3::5::7::l) (fun l→l) (eratosthene n)
```

Figure A.1: BSML code of the the parallel versions of the sieve of Eratosthenes.

primes and we perform a total exchange on them to eliminates the non-primes. End of this induction comes from the BSP cost: we end when n is small enough so that the sequential methods is faster than the parallel one.

After some benchmark (not presented here), we obtain a super-linear acceleration for the recursive method. This is due to the fact that, using a parallel method, each processor has a smaller list of integers and thus the garbage collector of `OCaml` is called less often.

A.2 Parallel sorting

Sorting is a classical problem of parallel algorithms which is complex and covers a huge range of program constructions. Many parallel algorithms, including the graphs [9], require that data be sorted on processors and also between them: the processor i contains data smaller than those of processor $i+1$. Moreover, we must that data are well distributed over the processors for a good load-balancing. It is not so easy to write a correct implementation of parallel sorting algorithms, even without any optimization, since a small mistake in such complex algorithms immediately has some catastrophic consequences.

Take a set of elements $\mathcal{X}$ of size n. We assume that the initial data structure x was been partitioned into p sub-structures $x^1, \ldots, x^p$ of size n/p with a sub-structure by processor. We note $\langle a, b\rangle$ an open interval *i.e.* the set of all elements $c \in \mathcal{X}$ such that $a < c < b$.

A naif algorithm would be to gather data on one processor, then to sort them and scatter on all processors this set of data. It is easy to see that this method is completly inefficient.

Many parallel sorting algorithm have been proposed with different complexities. Here, we are interested by the *sampling sort algorithm* (PSSR) in its BSP version [35].

The PSRS algorithm proceeds as follows. First, all sub-structures x^q (we assume that their lengths are $\geq p^3$) are sorted independently with a sequential sort algorithm on each processor q. The problem now consists of merging the p sorted sub-structures. Each process selects from its sub-structure $p+1$ elements (the first and last elements

```
(* generic functions :
   compare:(α →α →int) = compare two elements
   seq_sort:((α →α →int) →α →β ) = sequential sorting
   select:(int →β →γ ) = selection of a sample
   merge:((α →α →int) →γ list →β ) = merging the samples (sorted)
   to_be_send:((α →α →int) →int →γ →β → δ ) =construction of the
                                                     blocks to be send
   get:( δ →int → ϵ ) = select the ith block to be send
   merge_block:((α →α →int) → ϵ → ϵ → ϵ ) = merge 2 received blocks
                                             and return the final result
   vec:β Bsml.par = the parallel vector to sort *)

let tiskin_bsp_sample_sort_wide compare seq_sort select merge_samples to_be_send
                                get merge_block vec =
 (* number of processors *)
 let p=bsp_p() in
 (* merge the sending blocks at the end *)
 let final_merge f =
  let rec final n tmp =
     if n=p then tmp else final (n+1) (merge_block compare tmp (f n))
   in final 1 (f 0)
 in
  (* Super−step 1 *)
  let vec_sort = parfun (seq_sort compare) vec in
  let primary_sample = parfun (select p) vec_sort in
  let totex_prim_sample = replicate_total_exchange primary_sample in
  (* Super−step 2 *)
  let scd_sample = select p (merge_samples compare totex_prim_sample) in
  let elts_to_send = parfun (to_be_send compare p scd_sample) vec_sort in
  let to_send = put (parfun get elts_to_send) in
  (* Super−step 3 *)
   parfun final_merge to_send
```

Figure A.2: BSML code of a generic BSP sample sorting algorithm.

must be selected) for the primary sample and there is a total exchange of these samples. We note $\bar{x}_0^q, \ldots, \bar{x}_p^q$ the first sample of the sub-structure x^q (of processor q). It cut the sub-structure x^q into p primary blocks (of size n/p^2) and we note them $[\bar{x}_0^q, \bar{x}_1^q], \ldots, [\bar{x}_{p-1}^q, \bar{x}_p^q]$.

In the second super-step, each process reads the $p \times (p+1)$ primary samples, sorts them and selects $p+1$ secondary samples (in the same manner). We note these sorted sub-structures y^q. The second primary sample is noted $\bar{\bar{x}}_0, \ldots, \bar{\bar{x}}_p$. Note that this sample is the same on each processor. This sample cuts the elements of x (and not of x^q) into p secondary blocks which are open intervals $\langle \bar{\bar{x}}_0, \bar{\bar{x}}_1 \rangle, \ldots, \langle \bar{\bar{x}}_{p-1}, \bar{\bar{x}}_p \rangle$.

In the third super-step, each processor discards the values that do not belong to the assigned secondary block : each processor q takes the elements (from other processors) from the open interval $\langle \bar{\bar{x}}_q, \bar{\bar{x}}_{q+1} \rangle$ (with $0 \leq q < p$) and then merged the receveid values.

Figure A.2 shows a generic implementation of this BSP algorithm.

Appendix B

Proof of lemmas and theorems

Proof for lemma 2.3.1 (stability of typings by substitution). We proceed by induction on the derivation of $\Gamma, x : \tau \vdash e : \tau'$.

- If $e = x$, $[v/x]e = v$ and $\tau = \tau'$. By hypothesis, $\Gamma \vdash v : \tau'$.
- If $e = y$ where $x \neq y$, $[v/x]e = y$. The type of the expression is unchanged.
- If $e = a\ b$, $[v/x]e = [v/x]a\ [v/x]b$. By induction hypothesis, the types of a and b are preserved during the substitution. Using the typing rule for application, we deduce the type of $[v/x]e$:
$$\frac{\Gamma \vdash [v/x]a : \langle f_a, \tau_0 \to \tau_1 \rangle \quad \Gamma \vdash [v/x]b : \tau_0}{\Gamma, x : \tau \vdash [v/x]e : \langle f_a \vee flow(\tau_0), annot(\tau_1) \rangle}.$$
- If $e = \lambda y.b$, the type derivation is of the following shape:
$$\frac{\Gamma, x : \tau, y : \tau_0, \vdash b : \tau_1}{\Gamma, x : \tau \vdash e : \langle flow(\tau_0) \vee flow(\tau_1), \tau_0 \to \tau_1 \rangle}$$
  - If $x = y$, $[v/x]e = e$. The type of the expression is unchanged.
  - If $y \neq x$ and ($y \notin fv(v)$ or $x \notin fv(b)$), $[v/x]e = \lambda y.[v/x]b$. By induction hypothesis, the type of b is preserved during substitution. Thus, we can derive:
$$\frac{\Gamma y : \tau_0, \vdash [v/x]b : \tau_1}{\Gamma \vdash [v/x]e : \langle flow(\tau_0) \vee flow(\tau_1), \tau_0 \to \tau_1 \rangle}$$
  - If $y \neq x$ and ($y \in fv(v)$ and $x \in fv(b)$); with z fresh, $[v/x]e = \lambda z.[v/x]([z/y]b)$. We won't prove that substituting a fresh variable for another variable is type preserving, but we use this lemma to state that $\Gamma, x : \tau, z : \tau_0, \vdash ([z/y]b) : \tau_1$. We can then derive :
$$\frac{\Gamma, x : \tau, z : \tau_0, \vdash [z/y]b : \tau_1}{\Gamma, x : \tau \vdash \lambda z.[z/y]b : \langle flow(\tau_0) \vee flow(\tau_1), \tau_0 \to \tau_1 \rangle}$$
    Using the induction hypothesis, we can perform a type-preserving substitution such that:
$$\Gamma \vdash [v/x](\lambda z.[z/y]b) : \langle flow(\tau_0) \vee flow(\tau_1), \tau_0 \to \tau_1 \rangle$$
    As shown in a previous case, this reduces to:
$$\Gamma \vdash (\lambda z.[v/x]([z/y]b)) : \langle flow(\tau_0) \vee flow(\tau_1), \tau_0 \to \tau_1 \rangle$$

Proof for theorem 2.3.2 (soundness w.r.t. **yield** reductions). We will prove this theorem on a subset of the source language, namely the core lambda-calculus plus **yield** . Before proceeding to the proof, we state that **yield** is *not* a value, and we recall the small-step operational semantics of the lambda-calculus under the $\overset{\beta v}{\to}$reduction rule:

$$(\lambda x.e)\, v \overset{\beta v}{\to} [v/x]e \qquad \frac{e_2 \to e_2'}{e_1\, e_2 \to e_1\, e_2'}\ \textsc{AppRight} \qquad \frac{e_1 \to e_1'}{e_1\, v \to e_1'\, v}\ \textsc{AppLeft} \qquad \textbf{yield} \to ()$$

We proceed by induction on the reduction sequence of e.

- If e = **yield** , the theorem trivially holds.
- If $e = e_1\, v \rightarrow e'_1\, v$, and a **yield** is reduced in $e_1 \rightarrow e'_1$; by induction hypothesis, flow(e_1) = $\mathcal{I}$. Using the typing rule for application, flow(e) = $\mathcal{I}$.
- If $e = e_1\, e_2 \rightarrow e_1\, e'_2$, and a **yield** is reduced in $e_2 \rightarrow e'_2$; by induction hypothesis, flow(e_2) = $\mathcal{I}$. Using the typing rule for application, flow(e) = $\mathcal{I}$.
- If $e = (\lambda x.b)\, v \rightarrow [v/x]b$, a **yield** is reduced while normalizing $[v/x]b$ and flow($[v/x]b$) = $\mathcal{I}$. Typings are stable by substitution (Lemma 2.3.1), so flow(b) = $\mathcal{I}$. Using the typing rules for abstraction then application, we derive flow(e) = $\mathcal{I}$.

Proof for lemma 2.3.4 (extended monadic substitution). We proceed by induction on the shape of a:

- If $a = x$: the proof follows by reduction and application of lemma 2.3.3.
- If $a = c$ or a = **yield** : trivial case.
- If $a = c\, d$ where flow(a) = $\mathcal{I}$:

  We derive from Fig. 2.2 that flow(c) = $\mathcal{P}$is not possible.

  – If flow(d) = $\mathcal{I}$:
  $$\begin{aligned} T_1[\![[v/x]a]\!] &= T_1[\![[v/x]c\ [v/x]d]\!] \\ &= T_1[\![[v/x]c]\!]\ @\ T_1[\![[v/x]d]\!] && \text{(definition of } T_1\text{)} \\ &= [[\![v]\!]_v/x]T_1[\![c]\!]\ @\ [[\![v]\!]_v/x]T_1[\![d]\!] && \text{(ind. hyp.)} \\ &= [[\![v]\!]_v/x](T_1[\![c]\!]\ @\ T_1[\![d]\!]) \\ &= [[\![v]\!]_v/x](T_1[\![c\, d]\!]) \end{aligned}$$
  – If flow(d) = $\mathcal{P}$:
  $$\begin{aligned} T_1[\![[v/x]a]\!] &= T_1[\![[v/x]c\ [v/x]d]\!] \\ &= T_1[\![[v/x]c]\!]\ T_1[\![[v/x]d]\!] \\ &= [[\![v]\!]_v/x]T_1[\![c]\!]\ [[\![v]\!]_v/x]T_1[\![d]\!] \text{ (ind. hyp.)} \\ &= [[\![v]\!]_v/x](T_1[\![c]\!]\ T_1[\![d]\!]) \\ &= [[\![v]\!]_v/x](T_1[\![c\, d]\!]) \end{aligned}$$
- If $a = c\, d$ where flow(a) = $\mathcal{P}$:
  $$\begin{aligned} T_1[\![[v/x]a]\!] &= T_1[\![[v/x]c\ [v/x]d]\!] \\ &= \textbf{ret}\ \ [v/x](c\ d) \\ &= [v/x]\ \textbf{ret}\ \ (c\ d) = [v/x]\ T_1[\![(c\ d)]\!] \end{aligned}$$
- If $a = \lambda y.b$ where flow(a) = $\mathcal{I}$:

  – If $x = y$: easy.

  – If $x \neq y$ and ($y \notin fv(v)$ or $x \notin fv(b)$):
  $$\begin{aligned} T_1[\![[v/x]a]\!] &= T_1[\![\lambda y.T_1[\![[v/x]/b]\!]]\!] \\ &= \textbf{ret}\ \ \lambda y.T_1[\![[v/x]/b]\!] \end{aligned}$$
  At this point, two cases may arise. If flow(b) = $\mathcal{P}$, then flow(x) = $\mathcal{I}$and b contains no free occurences of x (proof by induction on the type derivation, easy and admitted). We can thus forget the substitution, and the proof follows easily. If flow(b) = $\mathcal{I}$, we can apply the induction hypothesis on b:
  $$\begin{aligned} T_1[\![[v/x]a]\!] &= T_1[\![\lambda y.[[\![v]\!]_v/x]T_1[\![b]\!]]\!] \\ &= [[\![v]\!]_v/\text{x}]T_1[\![\lambda y.T_1[\![b]\!]]\!] \end{aligned}$$
  – If $y \neq x$ and ($y \in fv(v)$ and $x \in fv(b)$) with z fresh: we must avoid variable capture, so we state that $a = \lambda z.[z/y]b$. Since $x \in fv(b)$, flow(b) = flow($[z/y]b$) = $\mathcal{I}$, allowing us to prune a case. Except that point, the proof is similar to the previous case.

- If a = $\lambda y.b$ where flow(a) = $\mathcal{P}$: the proof is similar to the pure application case.
- If a = **fix** $h\ \lambda y.b$: proof similar to the lambda-abstraction, with added cases to handle the fixpoint binder h.
- If a = **match** e **with** $|\ \ \kappa\, x_i\ \rightarrow\ e_i$. This proof is quite tedious if we don't work modulo α-conversion, we will thus adopt this hypothesis for the case at hand. The case where flow(a) = $\mathcal{P}$is as easy as the application case and won't be exposed.

  We first observe that the lemma holds for matching branches : $T_1[\![[v/x](\kappa\, x_i\ \rightarrow\ e_i)]\!]$ = $[[\![v]\!]_v$/x]$T_1[\![(\kappa\, x_i\ \rightarrow\ e_i)]\!]$. The proof is by case on wether $x = x_i$, and by induction hypothesis on e_i whenever $x \neq x_i$.

  Using this fact, the proof is easy, by using the induction hypothesis on e and the matching branches.
- If a = (c, d), a = $\kappa\ e$: easy.

This ends the proof of the extended monadic substitution lemma.

We also need these easily provable properties:

- Prop. 1: $(\lambda x.a)\ v \approx [v/x]a$
- Prop. 2: **bind** (**ret** v) $(\lambda x.b) \approx [v/x]b$ (first monadic law + $\overset{\beta v}{\rightarrow}$)
- Prop. 3: If $a\ \approx\ a'$, **bind** a $(\lambda x.b) \approx$ **bind** a' $(\lambda x.b)$.
- Prop. 4: If flow(a) = $\mathcal{P}$and $a\ \Rightarrow\ v$, $T_1[\![a]\!] \approx T_1[\![v]\!]$.
- Prop. 5: If $b\ \approx\ b'$, $ab\ \approx\ ab'$ (proof by induction on a).

Proof for theorem 2.3.5 (soundness of the partial CPS transformation). We proceed by induction on $e \Rightarrow v$. Please refer to Fig. 2.1 for the definition of the rules. The cases t = c, t = x, t = $\lambda x.b$, t = **fix** $h\ \lambda x.b$, t = (a, b), t = $\kappa\ a$ are trivial (an application of lemma 2.3.3 is enough). If flow(t) = $\mathcal{P}$, the result is immediate.

- LET rule:
  - If flow(e_1) = $\mathcal{I}$,
    $T_1[\![t]\!]$ = **bind** $T_1[\![e_1]\!]$ $(\lambda a.T_1[\![e_2]\!])$
    $\approx$ **bind ret** $[\![v_1]\!]_v$ $(\lambda a.T_1[\![e_2]\!])$ (ind. hyp. on e_1, Prop. 3)
    - If flow(e_2) = $\mathcal{P}$, a doesn't appear in e_2. The proof goes on as follow:
      $T_1[\![t]\!]$ $\approx T_1[\![e_2]\!]$ (forgetting the substitution)
      $\approx$ **ret** $[\![v_2]\!]_v$ (ind. hyp.)
    - If flow(e_2) = $\mathcal{I}$, we can apply the induction hypothesis such that:
      $T_1[\![t]\!]$ $\approx [[\![v_1]\!]_v/a]T_1[\![e_2]\!]$ = $T_1[\![[v_1/a]e_2]\!]$ (Prop. 2, Lemma 2.3.4)
      $\approx$ **ret** $[\![v_2]\!]_v$ (ind. hyp.)
  - If flow(e_1) = $\mathcal{P}$,
    $T_1[\![t]\!]$ = **let** $a\ =\ e_1$ **in** $T_1[\![e_2]\!]$
    $\approx [v_1/a]T_1[\![e_2]\!]$
    $\approx T_1[\![[v_1/a]e_2]\!]$ (Lemma 2.3.4)
    $\approx$ **ret** $[\![v_2]\!]_v$ (ind. hyp.)
- APP rule:

  Since flow(t) = $\mathcal{I}$, flow(e_1) = $\mathcal{I}$ (c.f. Fig. 2.2). Only the flow of e_2 may vary.
  - If flow(e_2) = $\mathcal{I}$,

$$
\begin{aligned}
T_1[\![t]\!] &= \textbf{bind } (T_1[\![e_1]\!])\ (\lambda v_1.\textbf{bind } (T_1[\![e_2]\!])\ (\lambda v_2.v_1\ v_2)) \\
&\approx \textbf{bind } (\textbf{ret } [\![\lambda x.e]\!]_v)\ (\lambda v_1.\textbf{bind } (T_1[\![e_2]\!])\ (\lambda v_2.v_1\ v_2)) && \text{(ind. hyp., Prop. 3)} \\
&\approx [[\![\lambda x.e]\!]_v/v_1](\textbf{bind } T_1[\![e_2]\!]\ (\lambda v_2.v_1\ v_2)) && \text{(Prop. 2)} \\
&\approx [\lambda x.T_1[\![e]\!]/v_1](\textbf{bind } T_1[\![e_2]\!]\ (\lambda v_2.v_1\ v_2)) && \text{(definition of } [\![]\!]_v) \\
&\approx \textbf{bind } (T_1[\![e_2]\!])\ (\lambda v_2.(\lambda x.T_1[\![e]\!])\ v_2) \\
&\approx \textbf{bind } (\textbf{ret } [\![v']\!]_v)\ (\lambda v_2.(\lambda x.T_1[\![e]\!])\ v_2) && \text{(ind. hyp., Prop. 3)} \\
&\approx [[\![v']\!]_v/v_2]((\lambda x.T_1[\![e]\!])v_2) && \text{(Prop. 2)} \\
&\approx ((\lambda x.T_1[\![e]\!])\ [\![v']\!]_v) \\
&\approx ([[\![v']\!]_v/x]T_1[\![e]\!])
\end{aligned}
$$

We have flow(e_2) = $\mathcal{I}$. Hence, flow(x) = $\mathcal{I}$ (c.f. Fig. 2.2).

$*$ If flow(e) = $\mathcal{P}$, e doesn't contain any free occurence of x, and we obtain:

$$
\begin{aligned}
T_1[\![t]\!] &\approx T_1[\![e]\!] && \text{(forgetting the substitution)} \\
&\approx \textbf{ret } [\![v]\!]_v = \textbf{ret } v && \text{(ind. hyp.)}
\end{aligned}
$$

$*$ If flow(e) = $\mathcal{I}$:

$$
\begin{aligned}
T_1[\![t]\!] &\approx T_1[\![[v'/x]e]\!] && \text{(Lemma 2.3.4)} \\
&\approx \textbf{ret } [\![v]\!]_v && \text{(ind. hyp.)}
\end{aligned}
$$

– If flow(e_2) = $\mathcal{P}$,

$$
\begin{aligned}
T_1[\![t]\!] &= T_1[\![e_1]\!]\ T_1[\![e_2]\!] \\
&\approx T_1[\![e_1]\!]\ T_1[\![v']\!] = T_1[\![e_1]\!]\ \textbf{ret } v' && \text{(Prop. 4, Prop. 5)} \\
&\approx T_1[\![\lambda x.e]\!]\ (\textbf{ret } v') = (\textbf{ret } \lambda x.T_1[\![e]\!])\ (\textbf{ret } v') && \text{(definition of } \approx\text{, ind. hyp.)} \\
&\approx [v'/x]T_1[\![e]\!] \\
&\approx T_1[\![[v'/x]e]\!] && \text{(Lemma 2.3.4)} \\
&\approx \textbf{ret } [\![v]\!]_v && \text{(ind. hyp.)}
\end{aligned}
$$

- MATCH rule:

  – If flow(e) = $\mathcal{I}$,

$$
\begin{aligned}
T_1[\![t]\!] &= \textbf{bind } T_1[\![e]\!]\ (\lambda v_e.\textbf{match} v_e \textbf{with } \kappa_i\ x_i\ \rightarrow\ T_1[\![e_i]\!]) \\
&\approx \textbf{bind } \textbf{ret } \kappa\ [\![v]\!]_v\ (\lambda v_e.\textbf{match} v_e \textbf{with } \kappa_i\ x_i\ \rightarrow\ T_1[\![e_i]\!]) \\
&\approx [\kappa\ [\![v]\!]_v/v_e](\textbf{match} v_e \textbf{with } \kappa_i\ x_i\ \rightarrow T_1[\![e_i]\!]) \\
&\approx \textbf{match } \kappa\ [\![v]\!]_v \textbf{with } \kappa_i\ x_i\ \rightarrow T_1[\![e_i]\!] \\
&\approx [[\![v]\!]_v/x]T_1[\![e']\!]
\end{aligned}
$$

Now, if flow(e') = $\mathcal{P}$then e' contains no free occurence of x and the lemma holds. Otherwise:

$$
\begin{aligned}
T_1[\![t]\!] &\approx T_1[\![[v/x]e']\!] && \text{(Lemma 2.3.4)} \\
&\approx \textbf{ret } v' && \text{(ind. hyp.)}
\end{aligned}
$$

  – If flow(e) = $\mathcal{P}$, then the branches have all a flow equal to $\mathcal{I}$.

$$
\begin{aligned}
T_1[\![t]\!] &= \textbf{bind } T_1[\![e]\!]\ (\lambda v_e.\textbf{match} v_e \textbf{with } \kappa_i\ x_i\ \rightarrow\ T_1[\![e_i]\!]) \\
&\approx \textbf{bind } \textbf{ret } \kappa\ v\ (\lambda v_e.\textbf{match} v_e \textbf{with } \kappa_i\ x_i\ \rightarrow\ T_1[\![e_i]\!]) \\
&\approx [\kappa\ v/v_e](\textbf{match} v_e \textbf{with } \kappa_i\ x_i\ \rightarrow T_1[\![e_i]\!]) \\
&\approx \textbf{match } \kappa\ v \textbf{with } \kappa_i\ x_i\ \rightarrow T_1[\![e_i]\!] \\
&\approx [v/x]T_1[\![e']\!]] \\
&\approx T_1[\![[v/x]e']\!]] && \text{(Lemma 2.3.4)} \\
&\approx \textbf{ret } v' && \text{(ind. hyp.)}
\end{aligned}
$$

- YIELD rule: follows from our definition of $\approx$.

- SUPER rule: assuming that **super** always terminates, an induction followed by a case analysis on the code of **super** shows that the final case is a **ret** (v_1, v_2).

This terminates the proof of soundness for our partial CPS transformation.

Appendix C

Coq script of Chapter 2

Using Coq version 8.1pl3.

```
Set Implicit Arguments.

Module Type MONAD.

  Parameter M : forall (A : Type), Type.
  Parameter ret : forall (A : Type), A →(M A).
  Parameter bind : forall (A B : Type), M A →(A →M B) →M B.
  Parameter run : forall (A : Type), M A →A.

  Parameter equivalence : forall (A : Type), M A →M A →Prop.

  Axiom is_reflexive : forall (A : Type) (x : M A), equivalence x x.
  Axiom is_symmetric : forall (A : Type) (x y : M A), equivalence x y →equivalence y x.
  Axiom is_transitive : forall (A : Type) (x y z: M A), equivalence x y →equivalence y z →equivalence x z.

  Axiom left_neutral : forall (A B : Type) (f : A →M B) (a : A),
        equivalence (bind (ret a) f) (f a).
  Axiom right_neutral : forall (A B : Type) (m : M A),
        equivalence (bind m (fun a : A => ret a)) m.
  Axiom assoc : forall (A B C : Type) (m : M A) (k : A →M B) (h : B →M C),
        equivalence (bind m (fun x : A => bind (k x) h)) (bind (bind m k) h).

End MONAD.

Inductive thread (a : Type) : Type :=
| Terminated : a →thread a
| Waiting : (unit →thread a) →thread a.

Inductive superthread_reduc (a b : Type) : thread a →thread b →Type :=
| Both_terminated : forall x0 x1,
      superthread_reduc (Terminated x0) (Terminated x1)
| Left_terminated : forall x susp,
      superthread_reduc (Terminated x) (susp tt) →
      superthread_reduc (Terminated x) (Waiting susp)
| Right_terminated : forall x susp,
      superthread_reduc (susp tt) (Terminated x) →
      superthread_reduc (Waiting susp) (Terminated x)
| Both_running : forall (s0 : unit →thread a) (s1 : unit →thread b),
      superthread_reduc (s0 tt) (s1 tt) →
      superthread_reduc (Waiting s0) (Waiting s1).

Lemma thread_terminates (A B : Type) :
  forall (t0 : thread A) (t1 : thread B), superthread_reduc t0 t1.
Proof.
induction t0.
induction t1.
exact (Both_terminated a a0).
exact (Left_terminated t (X tt)).
induction t1.
exact (Right_terminated t (X tt (Terminated a))).
exact (Both_running t t0 (X tt (t0 tt))).
Qed.

Module SuperMonad <: MONAD.

  Definition M (A : Type) := forall anstype, (A →thread anstype) →thread anstype.

  Definition ret (A : Type) : A →M A :=
      fun (x : A) anstype (k : A →thread anstype) => k x.

  Definition bind (A B : Type) : M A →(A →M B) →M B := fun (m : M A) (f : A →M B) =>
      fun ans k => m ans (fun (v : A) => f v ans k).

  Definition yield := fun ans (k : unit →thread ans) => Waiting k.

  Definition apply (A B : Type) : M (A →M B) →M A →M B :=
    fun f arg => bind f (fun vf => bind arg (fun varg => vf varg)).

  Definition run (A : Type) : M A →A :=
    fun (e : M A) =>
      let loop := (fix loop (m : thread A) :=
      match m with
        | Terminated x => x
        | Waiting s => loop (s tt)
      end)
     in loop (e A (fun x => Terminated x)).

Fixpoint super_aux (a b : Type)
     (t1 : thread a) (t2 : thread b)
     (h : (superthread_reduc t1 t2)) { struct h } : M (a ∗ b) :=
        bind yield (fun u =>
        match h return M (a ∗ b) with
        | Both_terminated x0 x1 =>
              ret (x0, x1)
        | Left_terminated x susp h0 =>
              super_aux h0
        | Right_terminated x susp h0 =>
              super_aux h0
        | Both_running s0 s1 h0 =>
              super_aux h0
        end).

Definition super := fun (A B : Type) => ret (fun f1 => ret (fun f2 =>
    let r1 : thread A := apply (ret f1) (ret tt) (fun x => Terminated x) in
    let r2 : thread B := apply (ret f2) (ret tt) (fun x => Terminated x) in
    match r1, r2 return M (A ∗ B) with
    | Terminated x1, Terminated x2 =>
      ret (x1, x2)
    | Terminated x, Waiting s =>
      super_aux (thread_terminates r1 r2)
    | Waiting s, Terminated x =>
      super_aux (thread_terminates r1 r2)
    | Waiting s0, Waiting s1 =>
      super_aux (thread_terminates r1 r2)
    end)).

(∗ Prove that super f g evaluates to (f tt, g tt). ∗)
Check super.

(∗ We need functional extensionality to prove the monadic laws ... ∗)
Axiom functional_extensionality :
    forall (A B : Type), forall (f g : A →B),
    (forall x,f x = g x) →f = g.

(* This lemma is much weaker than functional_extensionality. *)
Lemma eta_conversion :
    forall (A B: Type) (f : A →B), (fun x : A => f x) = f.
Proof.
intros.
assert (forall x, (fun x => f x) x = f x).
intros; reflexivity.
apply (functional_extensionality (fun x => f x) f).
intro;reflexivity.
Qed.

Definition equivalence : forall (A : Type), M A →M A →Prop :=
   fun (A : Type) a1 a2 => forall k,exists a, (a1 k = a) /\ (a2 k = a).

Lemma is_reflexive : forall (A : Type) (x : M A), equivalence x x.
Proof.
compute.
intros.
exists (x k).
auto.
Qed.

Lemma is_symmetric : forall (A : Type) (x y : M A), equivalence x y →equivalence y x.
Proof.
compute.
intros.
elim (H k).
intros.
elim H0.
intros.
rewrite H1.
rewrite H2.
exists x0; auto.
Qed.

Lemma is_transitive : forall (A : Type) (x y z: M A), equivalence x y →equivalence y z →equivalence x z.
Proof.
compute.
intros.
elim (H k).
intros.
elim (H0 k).
intros.
elim H1.
elim H2.
intros.
rewrite H4 in *.
rewrite H5 in *.
rewrite H3 in H6.
rewrite H6.
exists x0; auto.
Qed.

Lemma left_neutral : forall (A B : Type) (f : A →M B) (a : A),
          equivalence (bind (ret a) f) (f a).
Proof.
intros.
compute.
intro.
exists (f a k).
split.
```

```
apply eta_conversion.
reflexivity.
Qed.

Lemma right_neutral : forall (A B : Type) (m : M A),
        equivalence (bind m (fun a : A => ret a)) m.
Proof.
compute.
intros.
exists (fun k0 => m k (fun v => k0 v)).
split.
reflexivity.
symmetry.
assert (forall arg, (fun k0 => m k (fun v => k0 v)) arg = m k arg).
intro.
assert ((fun v => arg v) = arg).
rewrite <– eta_conversion.
reflexivity.
rewrite H.
reflexivity.
apply functional_extensionality.
exact H.
Qed.

Lemma assoc : forall (A B C : Type) (m : M A) (k : A →M B) (h : B →M C),
        equivalence (bind m (fun x : A => bind (k x) h)) (bind (bind m k) h).
Proof.
compute.
intros.
exists (fun k1 => m k0 (fun v => k v k0 (fun v0 => h v0 k0 k1))).
auto.
Qed.

End SuperMonad.
```