Type System for a Safe Execution of Parallel Programs in BSML

Frédéric Gava
LACL, University Paris-East Créteil
Créteil, France
Frederic.Gava@univ-paris-est.fr

Louis Gesbert
MLstate
Paris, France
Louis.Gesbert@mlstate.com

Frédéric Loulergue
LIFO, University of Orléans
Orléans, France
Frederic.Loulergue@univ-orleans.fr

Abstract

BSML, or Bulk Synchronous Parallel ML, is a high-level language based on ML and dedicated to parallel computation. In this paper, an extended type system that guarantees the safety of parallel programs is presented. It prevents non-determinism and deadlocks by ensuring that the invariants needed to preserve the structured parallelism are verified. Imperative extensions (references, exceptions) are included, and the system is designed for compatibility with modules.

Categories and Subject Descriptors D3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages

General Terms Reliability, Theory

Keywords Functional Programming, Parallel Programming, Bulk Synchronous Parallelism, Formal Semantics

1. Introduction

The Bulk Synchronous Parallel ML (BSML) language [**?**] is a parallel extension of ML (a family of functional programming languages). BSML is an attempt at providing the right balance between the two opposite approaches of parallel programming, low-level and subject to concurrency issues, and high-level with loss of flexibility and efficiency. In the former, we find general-purpose libraries such as MPI [**?**], generally used with Fortran or C; these approaches are unsafe and leave the programmer responsible for deadlock or non-determinism issues. In the latter stand traditional algorithmic skeletons [**? ? ?**] where programs are safe but limited to a restricted set of algorithms.

BSML is implemented as an extension for Objective Caml [**?**], consisting of a pre-processor and a library. This makes the advanced general-purpose features of Objective Caml, such as multi-platform native-code compilation, available in the language; BSML also benefits from Ocaml type system, but in a limited way. Parallelism, in BSML, is strongly structured and follows the BSP (Bulk Synchronous Parallel [**? ? ?**]) paradigm. All communications in BSML are collective (require all processes) and deadlocks are avoided by a strict distinction between local and global computation. BSP also provides BSML with a simple and efficient cost model.

The formal semantics of BSML have made it possible to write proofs of parallel programs [**? ?**] using the Coq proof assistant [**? ?**]. However, given the extensions that have been added to BSML recently (references, exceptions), BSML does not have a type system that is able to prevent run-time problems due to an incorrect use of the parallelism in standard user programs. A previous static analysis [**?**] achieved this goal in a smaller language by building constraints along expressions, but it is not relevant with the current version of BSML anymore; the system presented here is less restrictive and more scalable, and is inherently compatible with references, exceptions and modules.

In order to present how to statically ensure the safety of parallel programs, we will start by a description of BSML itself. The different problems that may be caused by the parallelism are described in section **??**. The core of the type system is given in **??**, and how it applies to the more advanced features of the language is the subject of section **??**. A type inference algorithm is given in section **??**. We compare with related work (section **??**) and finally conclude in section **??**.

2. The BSML Approach to Parallelism

BSML is designed as an inclusive extension, by which we mean that a program in the host language will be compatible with BSML (with some exceptions that will be discussed later). Parallelism is achieved by the use of a data structure called *parallel vector* noted $\langle x_1, \ldots, x_p \rangle$ for "value x_i at processor i" on a machine with a fixed number p of processors. The parallel vector is the only way to obtain different values at different processors: all code outside is indeed *replicated* among the processors and assured to be consistent everywhere. A sequential host language program would thus be executed p times, once on each of the processors, returning – assuming the prerequisite that it is deterministic – a consistent replicated value at each of them.

Parallel vectors are manipulated through four primitives (figure **??**).

p, the number of processors of the parallel machine, is a constant defined at run-time. mkpar is used to build a parallel vector from a function; proj does the opposite, turning a parallel vector back to a replicated function. apply locally applies a vector of functions to a vector of arguments, enabling local application of functions. put is a versatile communication primitive, taking as arguments local functions that explicit what data to send to each processor, and returning local functions used to get, at each processor, the data received from the others.

An important difference exists between, on one part, mkpar and apply, and on the other part, put and proj. The first two, indeed,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HLPP'11, September 18, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0862-5/11/09...$10.00

$\mathsf{mkpar}:$	f	$\mapsto$	$\langle f\ 0, \ldots, f\ (p-1)\rangle$
$\mathsf{apply}:$	$\langle f_0, \ \ldots, \ f_{p-1}\rangle$ $\langle x_0, \ \ldots, \ x_{p-1}\rangle$	$\mapsto$	$\langle f_0\ x_0, \ldots, f_{p-1}\ x_{p-1}\rangle$
$\mathsf{put}:$	$\langle f_0, \ldots, f_{p-1}\rangle$	$\mapsto$	$\langle \mathsf{fun}\ i \to f_i\ 0, \ldots, \mathsf{fun}\ i \to f_i\ (p-1)\rangle$
$\mathsf{proj}:$	$\langle x_0, \ldots, x_{p-1}\rangle$	$\mapsto$	$\mathsf{fun}\ i \to x_i$

Figure 1. BSML Primitives

are performed asynchronously by all the processors, as they do not need any communication. put and proj, on the other hand, trigger a phase of collective communications, ended with a global barrier. This, in accordance with the BSP model, ends the current super-step and ensures the determinism of the execution.

In BSP, a program is a sequence of super-steps which are composed of three stages, in this order: asynchronous computation at each processor, global communication and global barrier. Super-steps in BSML are implicit, and always ended by put or proj, which wait for the barrier before making the communicated results available locally.

2.1 Examples

These minimal primitives are very suitable for semantics and proofs of programs, but they can, and are indeed intended to be embedded in higher-level functions.

- apply, for example, is most commonly used to apply the same function on the members of a vector:

  $$\begin{array}{l}\mathsf{let\ replicate}\ x = \mathsf{mkpar}(\mathsf{fun}\ pid \to x)\\ \mathsf{let\ parfun}\ f\ v = \mathsf{apply}(\mathsf{replicate}\ f)\ v\end{array}$$

  replicate turns a replicated value into a parallel vector, while parfun $f\ v$ applies f in parallel to all members of the vector v.
- Total-exchange is a well-known pattern in parallel computation: it makes local data from all processors available everywhere, and can be defined in BSML with:

  $$\mathsf{let\ totex}\ v = \mathsf{put}\ (\mathsf{parfun}\ (\mathsf{fun}\ v'\ i \to v')\ v)$$

  or alternatively let totex $v =$ replicate (proj v).
- A simple reduction on a vector of lists v using function f with a neutral element z could be performed with:

  $$\begin{array}{l}\mathsf{let\ proj\_list}\ v = \mathsf{List.map}\ (\mathsf{proj}\ v)\ [0, \ldots, p-1]\\ \mathsf{let\ fold\_parlist}\ v\ f\ z =\\ \quad \mathsf{let}\ v' = \mathsf{parfun}\ (\mathsf{fun}\ x \to \mathsf{List.fold\_left}\ x\ f\ z)\ v\ \mathsf{in}\\ \quad \mathsf{List.fold\_left}\ (\mathsf{proj\_list}\ v')\ f\ z\end{array}$$

  where List.fold_left and List.map are functions from the standard library of Objective Caml that perform list reduction and mapping of a function to a list, respectively. proj_list is used to convert the results of proj into a list, and fold_parlist performs local reductions, gathers the results globally and reduces again.

2.2 Definition of Mini-BSML

To fit the definition of the type system within this paper, we concentrate on a small core of the language that nevertheless comprehends BSML parallel characteristics.

$$\begin{array}{rcll} e & ::= & x & \textit{variables}\\ & \mid & c & \textit{constants}\\ & \mid & op & \textit{operators}\\ & \mid & \mathsf{fun}\ x \to e & \textit{functions}\\ & \mid & e\ e & \textit{function application}\\ & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e & \textit{definition}\\ & \mid & (e, e) & \textit{pair}\\ & \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e & \textit{conditional}\end{array}$$

Among the operators *op* are, at least:

$$op ::= \mathsf{mkpar} \mid \mathsf{apply} \mid \mathsf{put} \mid \mathsf{proj} \mid \mathsf{fst} \mid \mathsf{snd} \mid \cdots$$

The syntax of imperative features will be introduced in section **??**, where they will be discussed.

The full semantics of the language is not described within this paper. The curious reader is invited to refer to [**?**]; nevertheless, the core language used here is very standard apart from the parallel primitives.

3. Different Kinds of Unsafe BSML Programs

Programs in BSML are supposed to follow some invariants in order to keep the properties of BSP and the strict distinction between local and global execution. This section describes these invariants, and how they can be broken by programs that would be correctly typed in ML.

As it may already appear to the reader, the most important one is the consistency of replicated expressions. We say that we lose replicated consistency if an expression outside of a parallel vector holds different values at different processors. For example, in the following program (where v is a parallel vector).

$$\mathsf{if}\ a\ \mathsf{then\ put}\ v\ \mathsf{else}\ v$$

An inconsistent value for a would cause a different branch of execution at different processors, some of them waiting at a barrier (when $a = true$) and the others not, causing a deadlock. Loss of replication happens for example when using non-deterministic operations, *e.g.* random, or operations that are dependant on the local processor, *e.g.* system or i/o related. In the example above, defining $a = ((\mathsf{random\_int}\ 10) < 5)$ is likely to trigger the problem. A complete system for the management of global and local I/O for BSML is discussed in detail in [**?**].

3.1 Local execution of parallel primitives

BSML is built around the notion of global execution and parallel vectors. This makes the semantics of programs such as

$$\mathsf{mkpar}(\mathsf{fun}\ i \to \mathsf{mkpar}(\mathsf{fun}\ j \to i * j))$$

ambiguous: the parameter of the first mkpar is a function that is supposed to be executed locally. The second mkpar would thus be executed once on each processor; since this primitive is supposed to create a new parallel vector, that would require each of them to communicate with its neighbours to create locally a new global object, actually creating p parallel vectors containing p values each. The semantics of such a behaviour would be very complex, not to mention the loss in readability, performance and performance prediction possibilities.

The local execution of the other primitives would be even more difficult to make sense of: what would be the meaning of put used locally by a processor i and requesting communications between processors j and k ? To keep the language and the semantics simple and usable, the type system rules out local execution of the primitives, which will be called *invalid parallelism*.

3.2 Nested parallel vectors

Preventing invalid parallelism still allows the definition of nested parallel vectors, because global values can be accessed anywhere from the local code.

$$\mathsf{let}\, v = \mathsf{mkpar}(\mathsf{fun}\, i \to i)\, \mathsf{in}\, \mathsf{mkpar}(\mathsf{fun}\, j \to v)$$

is a valid program that would supposedly return the value $w = \langle\langle 0, \ldots, p-1\rangle, \ldots, \langle 0, \ldots, p-1\rangle\rangle$. This, however, does not faithfully describe the actual data since by definition each processor only holds one value of a vector. The data stored would rather look like $\langle\langle 0\rangle, \ldots, \langle p-1\rangle\rangle$. The expression proj w 2, instead of returning the expected vector v, thus returns the result $\langle 2, \ldots, 2\rangle$, which is a parallel vector which has actually lost its parallel information.

There is no good compromise between the clarity of the semantics and implementation concerns if nested parallelism is accepted. Therefore, our type system rules it out.

4. Definition of the Type System

We use a Hindley/Milner type system inspired from that of ML, with added effects [**?** **?**] and constraints [**?**]. In this section, we will first concentrate on the prevention of invalid parallelism, then introduce more constructs that are needed to avoid nested parallelism. In the core language described above, types are defined by:

τ	$::=$	α	*type variable*
	$\mid$	Base	*base type*
	$\mid$	$\tau \xrightarrow{\pi} \tau$	*function*
	$\mid$	$\tau \times \tau$	*pair type*
	$\mid$	τ par	*parallel vector*

Λ	$::=$	δ	*locality variable*
	$\mid$	ℓ	*local*
	$\mid$	g	*global*

The first and main concern of our type system is to prevent local execution of parallel primitives. For that purpose, we use effects. We write $\Gamma \vdash e : \tau/\pi$ if the expression e is typed τ with effect π in the context Γ. An effect π can remain latent in a function, in which case we write $\tau_1 \xrightarrow{\pi} \tau_2$.

We define *locality* Λ as given above: locality variable δ behaves in a way similar to type variables and takes values in Λ. Though they will be extended later, for now we consider simple effects indicating only the current locality, and take $\pi = \Lambda$.

4.1 Definitions

Type scheme. A type scheme σ is a type with universally quantified type and locality variables, written $\sigma = \forall\alpha_1 \cdots \alpha_k\delta_1 \cdots \delta_l.\tau$

Typing context. A context Γ is a map from language variables to type schemes

Substitution. Given two partial, finite domain applications φ^τ and φ^Λ, from type variables to types and from locality variables to localities respectively, we define a substitution φ as the extension of $\varphi^\tau \circ \varphi^\Lambda$ to types:

$$\begin{array}{rcl}
\varphi(\alpha) & = & \varphi^\tau(\alpha) \text{ if } \alpha \in \mathrm{Dom}(\varphi^\tau) \\
\varphi(\delta) & = & \varphi^\Lambda(\delta) \text{ if } \delta \in \mathrm{Dom}(\varphi^\Lambda) \\
\varphi(\tau_1 \xrightarrow{\pi} \tau_2) & = & \varphi(\tau_1) \xrightarrow{\varphi(\pi)} \varphi(\tau_2) \\
\varphi(\tau_1 \times \tau_2) & = & \varphi(\tau_1) \times \varphi(\tau_2) \\
\varphi(\tau\, \mathsf{par}) & = & \varphi(\tau)\, \mathsf{par} \\
\varphi(x) & = & x \text{ in all other cases}
\end{array}$$

Instance. Given a type scheme $\sigma = \forall\alpha_1 \cdots \alpha_k\delta_1 \cdots \delta_l.\tau$, we say that τ' is an instance of σ and write $\sigma \leq \tau'$ if, and only if there exists a substitution φ on $\{\alpha_1 \cdots \alpha_k\delta_1 \cdots \delta_l\}$ (by which we mean that any variables not in this set are left unchanged by φ), such that $\tau' = \varphi(\tau)$

Free variables The set $\mathcal{L}(\tau)$ of free variables in a type τ is defined by induction:

$$\begin{array}{rcl}
\mathcal{L}(\alpha) & = & \{\alpha\} \\
\mathcal{L}(\mathsf{Base}) & = & \emptyset \\
\mathcal{L}(\tau_1 \xrightarrow{\pi} \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \cup \mathcal{L}(\pi) \\
\mathcal{L}(\tau_1 \times \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\tau\, \mathsf{par}) & = & \mathcal{L}(\tau) \\
\mathcal{L}(\delta) & = & \{\delta\}
\end{array}$$

The definition of $\mathcal{L}$ is extended to type schemes by: $\mathcal{L}(\forall\alpha_1 \cdots \delta_l.\tau) = \mathcal{L}(\tau) \setminus \{\alpha_1 \cdots \delta_l\}$.

The locality of an expression denotes its presence on a single processor or on all of them. A function does not have a specific locality as a value (it is usually defined globally), but may be restricted to a specific one for its execution. Since this cannot be seen from its return type alone, we use latent effects on function arrows to denote it. mkpar, for example, should have a latent locality g since it can only be executed on the whole set of processors. Some functions (like the identity and most sequential functions) do not have a specific locality, in which case their locality is a free variable.

Other functions, on the other hand, may be inherently local if they cannot consistently be applied in a replicated way on all processors. The function $random\_int$, for example, would not return a valid replicated value if executed by all processors: we give it the type $unit \xrightarrow{\ell} int$.

Operators need to have statically defined types. These are stored in a map TC, from operators to type schemes.

$$\begin{array}{rcl}
TC(\mathsf{mkpar}) & = & \forall\alpha.(int \xrightarrow{\ell} \alpha) \xrightarrow{g} \alpha\, \mathsf{par} \\
TC(\mathsf{apply}) & = & \forall\alpha\beta\delta.(\alpha \xrightarrow{\ell} \beta)\, \mathsf{par} \xrightarrow{\delta} \alpha\, \mathsf{par} \xrightarrow{g} \beta\, \mathsf{par} \\
TC(\mathsf{put}) & = & \forall\alpha\delta.(int \xrightarrow{\ell} \alpha)\, \mathsf{par} \xrightarrow{g} (int \xrightarrow{\delta} \alpha)\, \mathsf{par} \\
TC(\mathsf{proj}) & = & \forall\alpha\delta.\alpha\, \mathsf{par} \xrightarrow{g} int \xrightarrow{\delta} \alpha \\
TC(\mathsf{fst}) & = & \forall\alpha\beta\delta.\alpha \times \beta \xrightarrow{\delta} \alpha \\
TC(\mathsf{fst}) & = & \forall\alpha\beta\delta.\alpha \times \beta \xrightarrow{\delta} \beta
\end{array}$$

These types make the locality constraints of the primitives clear: they are all restricted to global execution, while the functions they take as parameters must accept to be executed in a local scope, *i.e.* must not themselves execute primitives.

The rules of the type system are given in figure **??**. The first four are the most important:

DEFFUN stores the locality induced by a function in the type arrow, but does not constrain the current locality (the rule is true for any π_1): the function itself can be defined anywhere. APPFUN then

$$\text{ENVTYPE}\quad \frac{\Gamma(x) \leq \tau}{\Gamma \vdash x : \tau/\pi}$$

$$\text{LET-IN}\quad \frac{\Gamma \vdash e_1 : \tau_1/\pi \qquad \Gamma; (x : Gen(\tau_1, \Gamma)) \vdash e_2 : \tau_2/\pi}{\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau_2/\pi}$$

$$\text{DEFFUN}\quad \frac{\Gamma; (x : \tau_1) \vdash e : \tau_2/\pi}{\Gamma \vdash \mathsf{fun}\, x \to e : \tau_1 \xrightarrow{\pi} \tau_2/\pi_1}$$

$$\text{APPFUN}\quad \frac{\Gamma \vdash f : \tau_1 \xrightarrow{\pi} \tau_2/\pi \qquad \Gamma \vdash e : \tau_1/\pi}{\Gamma \vdash f\ e : \tau_2/\pi} \qquad \text{CONST}\quad \frac{}{\Gamma \vdash c : TC(c)/\pi}$$

$$\text{OP}\quad \frac{}{\Gamma \vdash op : TC(op)/\pi} \qquad \text{PAIR}\quad \frac{\Gamma \vdash e_1 : \tau_1/\pi \qquad \Gamma \vdash e_2 : \tau_2/\pi}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2/\pi}$$

$$\text{IFTHENELSE}\quad \frac{\Gamma \vdash e : bool/\pi \qquad \Gamma \vdash e_1 : \tau/\pi \qquad \Gamma \vdash e_2 : \tau/\pi}{\Gamma \vdash \mathsf{if}\, e \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2 : \tau/\pi}$$

Figure 2. Induction rules of the type system

forces the current locality and the locality of an applied function to be the same, forbidding the use of functions that are not compatible with the current locality.

ENVTYPE specifies that when taking a type from the environment, it can be assigned any locality π. Together with LET-IN, this allows to use the results of a parallel computation at the desired locality (after using proj). This is needed for valid programs like $\mathsf{let}\, x = \mathsf{proj}(\cdots)\ \mathsf{in}\ \langle \textit{expression using x locally} \rangle$.

$Gen(\tau, \Gamma)$ is defined in the usual way, except that it also generalises locality variables.

Definition: Generalisation is defined by

$$Gen(\tau, \Gamma) = \forall \alpha_1 \dots \alpha_k \delta_1 \dots \delta_l . \tau$$

with $\{\alpha_1 \dots \alpha_k \delta_1 \dots \delta_l\} = \mathcal{L}(\tau) \setminus \mathcal{L}(\Gamma)$.

The other rules are standard ML typing rules, and just propagate the current locality to sub-terms, which denotes the fact that the locality can only be changed by using the primitives, or applying functions that use them.

Take for example the previous expression:

$$\mathsf{mkpar}(\mathsf{fun}\, i \to \mathsf{mkpar}(\mathsf{fun}\, j \to i * j))$$

Using our type system, we can have:

$$\Gamma \vdash (\mathsf{fun}\, j \to i * j) : \mathbf{int} \xrightarrow{\ell} \mathbf{int}/g$$

and thus:

$$\Gamma \vdash \mathsf{mkpar}(\mathsf{fun}\, j \to i * j) : \mathbf{int}\ \mathsf{par}\ /g$$

the expression have thus the global effect, *i.e.*, the expression makes parallelism. We have thus:

$$\Gamma \vdash (\mathsf{fun}\, i\, \mathsf{mkpar}(\mathsf{fun}\, j \to i * j) : \mathbf{int}\ \mathsf{par}\ /g) : \mathbf{int} \xrightarrow{g} \mathbf{int}\ \mathsf{par}\ /\delta$$

and them following the application rule, there is no way to apply a mkpar to this expression since the effect over the arrow is global.

4.2 Preventing nested parallelism

The type system defined above prevents crashes due to local execution of parallel code, but still allows the definition and projection of nested parallel vectors. Since we do not want to prevent the access to global values from local code – which is a very useful feature –, and values that are members of parallel vectors can only be defined from the return values of local function, we choose to restrict the attribution of the ℓ locality to functions, depending on their return type. Note that this does not prevent values of type τ par from being manipulated locally; however, since these can only be "opened" by the use of primitives that are forbidden in this scope, no harm can be done.

This amounts to adding a constraint on its locality and return type to the definition of a function. We write $\tau \triangleleft \Lambda$ to specify that the type τ must be acceptable in the locality Λ, *i.e.* that if $\Lambda = \ell$, τ must not contain parallel vectors.

Definition: Locally acceptable types are a subset of all types, defined as

$$\dot{\tau} ::= \mathsf{Base} \mid \tau \xrightarrow{\pi} \tau \mid \dot{\tau} \times \dot{\tau} \mid \dot{\tau}\ \mathsf{array}$$

The constraint $\tau \triangleleft \ell$ means that τ should belong to $\dot{\tau}$.

We extend the effects π from the previous type system with sets of constraints as follows:

$$\pi ::= \Lambda[\mathrm{C}] \qquad \begin{array}{rcll} \mathrm{C} & ::= & \rho & \textit{row variable} \\ & \mid & \tau \triangleleft \Lambda; \mathrm{C} & \textit{constraint and rest of row} \end{array}$$

π now contains a set of constraints C along with the locality information. C is a row formed of constraints and terminated by a row variable, which is used for unification. With these extended effects, the function $\mathsf{fun}\, x \to x$ would be typed $\alpha \xrightarrow{\delta[\alpha \triangleleft \delta; \rho]} \alpha$, which shows that the local use of the function is restricted by the type α.

We can omit the terminating row variable when it is a free variable and write $\alpha \xrightarrow{\delta[\alpha \triangleleft \delta]} \alpha$. By allowing not to write the full row when it is empty (as in $\mathsf{fun}\, x \to 0 : \alpha \xrightarrow{\delta} int$), this stays in accordance with former type definitions.

We extend previous definitions to take row variables into account:

- Type schemes now quantify row variables:

$$\sigma = \forall \alpha_1 \cdots \alpha_k \delta_1 \cdots \delta_l \rho_1 \cdots \rho_m . \tau$$

- Substitutions φ are extended with a partial application φ^{C} from a finite domain of row variables to rows.

$$\begin{array}{rcl} \varphi(\rho) & = & \varphi^{\mathrm{C}}(\rho) \text{ if } \rho \in \mathrm{Dom}(\varphi^{\mathrm{C}}) \\ \varphi(\tau \triangleleft \Lambda; \mathrm{C}) & = & \varphi(\tau) \triangleleft \varphi(\Lambda); \varphi(\mathrm{C}) \end{array}$$

- Free variables $\mathcal{L}$ may include row variables too.

$$\begin{array}{rcl} \mathcal{L}(\tau_1 \xrightarrow{\pi[\mathrm{C}]} \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \cup \mathcal{L}(\pi) \cup \mathcal{L}(\mathrm{C}) \\ \mathcal{L}(\alpha \triangleleft \delta : \mathrm{C}) & = & \{\alpha, \delta\} \cup \mathcal{L}(\mathrm{C}) \\ \mathcal{L}(\rho) & = & \{\rho\} \end{array}$$

As a consequence of these new definitions, Gen also generalises row variables.

4.2.1 Equational theory on constraint rows

We need to define an equational theory on rows, to express the fact that the same constraints can be expressed in different ways.

First, order of constraints does not matter:

$$\gamma_1; \gamma_2; \mathrm{C} = \gamma_2; \gamma_1; \mathrm{C}$$

Several constraints can be ignored:

$$\begin{array}{rcl} \tau \triangleleft g; \mathrm{C} & = & \mathrm{C} \\ \tau_1 \xrightarrow{\pi} \tau_2 \triangleleft \Lambda; \mathrm{C} & = & \mathrm{C} \\ \mathsf{Base} \triangleleft \Lambda; \mathrm{C} & = & \mathrm{C} \\ \tau \triangleleft \ell; \tau \triangleleft \Lambda; \mathrm{C} & = & \tau \triangleleft \ell; \mathrm{C} \end{array}$$

Finally, the constraints can be distributed on subtypes until they either apply to type variables, or are of the form $\tau\ \mathsf{par} \lhd \Lambda$:

$$\tau_1 \times \tau_2 \lhd \Lambda; \mathrm{C} \quad = \quad \tau_1 \lhd \Lambda; \tau_2 \lhd \Lambda; \mathrm{C}$$

According to this theory, constraint rows can be reduced to rows containing only constraints of the form $\alpha \lhd \delta$, $\alpha \lhd \ell$, $\tau\ \mathsf{par} \lhd \delta$ or $\tau\ \mathsf{par} \lhd \ell$. The last case, $\tau\ \mathsf{par} \lhd \ell$, is unsatisfiable and should obviously be rejected ; the rules of the unification algorithm can, therefore, be limited to the three first cases.

4.2.2 Constraint construction by type inference

As before, the effects are built by unification during inference. The only rule that needs to be changed is DEFFUN, since we need to make the constraint on the return values of local functions explicit in the constraint row.

$$\text{DEFFUN}\ \frac{\Gamma; (x : \tau_1) \vdash e : \tau_2/\Lambda[\mathrm{C}] \qquad \mathrm{C} = \tau_2 \lhd \Lambda; \mathrm{C}'}{\Gamma \vdash \mathsf{fun}\, x \to e : \tau_1 \xrightarrow{\Lambda[\mathrm{C}]} \tau_2/\pi}$$

As example, we take the previous expression:

$$\mathsf{let}\, v = \mathsf{mkpar}(\mathsf{fun}\, i \to i)\, \mathsf{in}\, \mathsf{mkpar}(\mathsf{fun}\, j \to v)$$

We have:

$$\Gamma \vdash \mathsf{mkpar}(\mathsf{fun}\, i \to i) : \alpha\ \mathsf{par}\ / g$$

which is generalised in the environment Γ as $\forall\alpha.\alpha\ \mathsf{par}\ /[\alpha \lhd \ell]$. In this way, we have

$$\Gamma \vdash (\mathsf{fun}\, j \to v) : \beta \xrightarrow{[\alpha \lhd \ell]} \alpha\ \mathsf{par}$$

which is thus not a possible argument for mkpar.

4.3 Properties

As defined by Milner [**?**], the safety property of typing can be stated as "well-typed programs do not go wrong". This property is obtained if the two following properties hold [**?**]:

- subject reduction: the reduction of a well-typed expression preserves the typing,
- progress: any well-typed program that is not a value can be reduced.

Both properties hold for our type system. The operational semantics of BSML as well as the proofs of the properties can be found in [**?**].

5. Imperative and Extended Features

5.1 References

Objective Caml provides several very useful imperative features. The most commonly used are data structures that can be modified in-place (arrays, strings, mutable record fields and references). To make things simpler, we only consider references, as the others can be easily reduced to them.

References are manipulated through three operators: ref, to create a reference, store to change the value stored in one, and value to get the value currently stored:

e	$::=$	$\cdots$	
	$\mid$	$\mathsf{ref}\, e$	*reference declaration*
	$\mid$	$\mathsf{store}(x, e)$	*reference assignment*
	$\mid$	$\mathsf{value}\, x$	*dereference*

References can easily cause a loss of replicated consistency, as shows the following code:

$$\mathsf{let}\, a = \mathsf{ref}\, \mathit{false}\, \mathsf{in}\ \mathsf{let}\, x = \mathsf{mkpar}(\mathsf{fun}\, i \to \mathsf{store}(a, i > 0))\, \mathsf{in}\, \mathsf{value}\, a$$

a is a reference created globally, but which is written on locally. If it is then read back globally, we lose replicated consistency: This expression would have value false on processor 0 and $true$ on the others.

The problem occurs when references are written outside of the locality they were created in. For this reason, the type system annotates references with the locality they were created at:

$$\tau ::= \cdots \mid \tau\ \mathsf{ref}_\Lambda$$

The types of reference operators are then defined as:

$$\begin{aligned} TC(\mathsf{ref}) &= \forall\alpha\delta.\alpha \xrightarrow{\delta[\alpha \lhd \delta]} \alpha\ \mathsf{ref}_\delta \\ TC(\mathsf{store}) &= \forall\alpha\delta.\alpha\ \mathsf{ref}_\delta \times \alpha \xrightarrow{\delta} unit \\ TC(\mathsf{value}) &= \forall\alpha\delta_1\delta_2.\alpha\ \mathsf{ref}_{\delta_1} \xrightarrow{\delta_2[\alpha \lhd \delta_2]} \alpha \end{aligned}$$

Note that a reference can be read using value from any locality (with respect to the constraint on the type referenced), but can only be written from its native one.

A concern remains, however, for references put through the primitives can change their locality. For example, if put is used on a global reference, it is communicated *via* the network, and actually becomes a local copy of the original. This can be prevented at a small cost, by having the constraint $\lhd \ell$ forbid global references in local return types (this is the case if we do not extend the former definition of constraints to accept the new ref type).

Local references, also, can be sent to the global space using proj. We could prevent the use of proj on references (and arrays, strings and records with mutable fields in a more comprehensive language) in a similar way, but this would be too restrictive regarding the expressiveness of the language. A better solution is to have proj change the types of references from local to global (which is actually what happens in the semantics and in the implantation). We do this by adding an intermediary type construct glob.

$$\tau ::= \cdots \mid \mathsf{glob}(\tau)$$

$$TC(\mathsf{proj}) = \forall\alpha\delta.\alpha\ \mathsf{par} \xrightarrow{g} int \xrightarrow{\delta} \mathsf{glob}(\alpha)$$

The functions that were defined by induction previously (φ and $\mathcal{L}$) propagate naturally to the arguments of ref_Λ and glob.

$$\begin{aligned} \varphi(\tau\ \mathsf{ref}_\Lambda) &= \varphi(\tau)\ \mathsf{ref}_{\varphi(\pi)} \\ \varphi(\mathsf{glob}(\tau)) &= \mathsf{glob}(\varphi(\tau)) \\ \mathcal{L}(\tau\ \mathsf{ref}_\Lambda) &= \mathcal{L}(\tau) \cup \mathcal{L}(\pi) \\ \mathcal{L}(\mathsf{glob}(\alpha)) &= \{\alpha\} \end{aligned}$$

The type $\mathsf{glob}(\tau)$ actually means a transformation on τ: it can be reduced using the following rules, until it only applies to type variables.

$$\begin{aligned} \mathsf{glob}(\mathsf{Base}) &\Rightarrow \mathsf{Base} \\ \mathsf{glob}(\tau_1 \xrightarrow{\pi} \tau_2) &\Rightarrow \tau_1 \xrightarrow{\pi} \mathsf{glob}(\tau_2) \\ \mathsf{glob}(\tau_1 \times \tau_2) &\Rightarrow \mathsf{glob}(\tau_1) \times \mathsf{glob}(\tau_2) \\ \mathsf{glob}(\tau\ \mathsf{ref}_\Lambda) &\Rightarrow \mathsf{glob}(\tau)\ \mathsf{ref}_g \end{aligned}$$

Note that $\mathsf{glob}(\tau\ \mathsf{par})$ and $\mathsf{glob}(\tau\ \mathsf{ref}_g)$ are invalid, because the argument of glob is a local return value, that must therefore follow the constraint $\lhd \ell$.

The reader may have noticed that glob propagates to the right-hand side of arrows although constraints do not, and it is not actually part of the data stored by values of these types. Since a local function can return any local reference present in its context, and which will in fact be communicated and globalised along with it, this is needed. It is at the cost of a small restriction in some cases:

$\mathsf{fun}\, i \rightarrow \mathsf{ref}\, i$ for example, has its return type restricted to ref_g if projected, forbidding its local execution.

As last example, if we want to type the previous (bad) expression with references, the first reference a would have the effect g and passing it to the store operator, the sub-expression $(\mathsf{fun}\, i \rightarrow \mathsf{store}(a, i > 0))$ would have a global effect which is not a valid argument for mkpar.

5.2 Exceptions

Exceptions are another very useful imperative feature available in Objective Caml. Their main purpose is the recovery of errors in the course of execution of a program, but they are also sometimes used algorithmically for "exceptional" behaviours that make it convenient to jump back in the pile of calls.

A thorough analysis of exceptions in BSML is given in [**? ?**]. For our purpose here, we suppose that a base type exn is defined as well as a type exnset that holds associations between processor numbers and values of type exn. Values of type exn can be raised using the operator raise, which stops the current execution and gets up the stack until it meets an enclosing $\mathsf{try} \ldots \mathsf{with}\, x \rightarrow e$ construct. This construct catches the exception, and triggers the behaviour specified in e. If no exception is raised inside, it is simply ignored.

$$\begin{array}{rcll} e & ::= & \cdots & \\ & | & \mathsf{raise}\, e & \textit{exception raise} \\ & | & \mathsf{try}\, e\, \mathsf{with}\, x \rightarrow e & \textit{exception catch} \\ & | & \mathsf{trypar}\, e\, \mathsf{withpar}\, x \rightarrow e & \textit{exception set catch} \end{array}$$

We allow exceptions to be raised at any locality. An exception raised locally, however, could cause a global inconsistency, reason for which it is gathered and raised globally as a set of processor-exception pairs at the first request for communication. Such sets, of type exnset, are caught by trypar instead of try, but follow a similar pattern of exception propagation.

These new constructs are typed with:

$$\textsc{trywith}\quad \frac{\Gamma \vdash e_1 : \tau/\pi \qquad \Gamma; x : \mathsf{exn} \vdash e_2 : \tau/\pi}{\Gamma \vdash \mathsf{try}\, e_1\, \mathsf{with}\, x \rightarrow e_2 : \tau/\pi}$$

$$\textsc{trypar}\quad \frac{\Gamma \vdash e_1 : \tau/g \qquad \mathit{\Gamma}; x : \mathsf{exnset} \vdash e_2 : \tau/g}{\Gamma \vdash \mathsf{trypar}\, e_1\, \mathsf{withpar}\, x \rightarrow e_2 : \tau/g}$$

$$TC(\mathsf{raise}) = \forall \alpha\delta\rho.\, \mathsf{exn} \xrightarrow{\delta[\rho]} \alpha$$

The expression that triggers the exception and the expression executed when an exception is caught must have the same type. TRYWITH has no special effect on locality, while TRYPAR can only be used globally.

A concern with exceptions is that they may allow values to escape their scope. This has to be kept in mind when extending the type exn – which is, in Objective Caml, a variant type that can hold any other type –. In particular, the global propagation of local exceptions has a power equivalent to that of proj; this means that values that can be raised locally must be restricted in the same way that local return values are. A simple and yet not overly restrictive solution is to keep exceptions free of parallel vectors and references, by adding a constraint to the types included in exn.

5.3 Other features

The system described here does not, for the sake of simplicity, include record or variants types. Records pose no real problem: they are syntactic sugar over tuples, which can be simulated with the pairs of our system. A record type follows $\lhd \ell$ if, and only if all the types it contains do.

Variants are more complicated, but the constructs and functions we defined can be extended to them without particular problems: a variant type follows $\lhd \ell$ if all of its alternatives do, and glob propagates likewise on all of them.

Although they have not been described here, modules are very important in the construction of Objective Caml programs; the type system has been designed to be compatible with them. Modules can be compiled separately, and the inference system must be able to work on a partial program given summarised interfaces of the other modules used.

All values from other modules are considered to be stored in the environment Γ. Effects on functions and locality annotations on references thus appear even on foreign modules. The point can cause some difficulty is the validation of the $\lhd \ell$ constraint on non-disclosed types from foreign modules. A possible solution is to compute the result of this constraint in advance within the foreign module and write it as an annotation in the interface. The interaction with objects has not been yet studied.

6. Type Inference Algorithm

In HM(X) [**?**], inference is done by solving constraints. The problem of type inference is transformed into determining whether there exists a constraint C such that $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$, for given Γ, e, τ, Λ. The unknown here is the constraint, and not as one may thing, the type and the locality. As a matter of fact, by choosing fresh variables for τ and Λ in Γ, the existence of C will give us the existence of instances for these variables and the constraint will contain the necessary information.

As a first step, we need to build a mechanism for building a constraint C that should be necessary and sufficient to obtain $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$. This constraint is also the constraint the less specific to guarantee the result: any constraint C' also ensuring it should be such that $\mathrm{C}' \Vdash \mathrm{C}$.

6.1 Constraints generation

The constraint generation mechanism that follows is complete and coherent and it provides a property equivalent to the property of existence of principal type scheme in Damas/Milner. The coherence of the mechanism ensures that the generated constraint C is sufficient to ensure that $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$, and the completeness ensures that the constraint is the less specific.

Some constraints are not necessary to the definition of the type system but should be added to the system so we can ensure this property:

$$\begin{array}{rcll} \mathrm{C} & ::= & \cdots & \\ & & \mathsf{let}\, x : \sigma\, \mathsf{in}\, \mathrm{C} & \text{Introduction of type scheme} \\ & & x \preceq \tau & \text{Instantiation of type scheme} \end{array}$$

We use the notation $\sigma \preceq \tau$ to mean, if $\sigma = \forall \overline{\alpha\delta}[\mathrm{C}].\tau'$ with $\overline{\alpha\delta} \,\#\, \mathcal{L}(\tau)$, the constraint $\exists \overline{\alpha\delta}.(\mathrm{C} \wedge \tau = \tau')$. In other words, τ is an instance of σ.

Constraint $\mathsf{let}\, x : \sigma\, \mathsf{in}\, \mathrm{C}$ binds identifier x to type scheme σ inside C. It also assumes there exists an instance for σ. This leads to giving the meaning $\sigma \preceq \tau$ to all sub-constraints $x \preceq \tau$ where x is free inside C.

These two constraints allow to replace in usual Dalmas/Milner systems, the extension of the typing environment and the search for associations inside typing environments.

The constraint generated for Γ, e, τ, Λ is written $\mathbb{C}(\Gamma \vdash e : \tau/\Lambda)$. Generation is done by induction on the structure of e: nodes let lead to move a part of the current constraint inside a new type scheme in the environment, and this sub-constraint is afterwards included in the current constraint when the let-bound variable appears. It corresponds to typing rules LET-IN and VAR. The problem is thus simplified in $\mathbb{C}(e : \tau/\Lambda)$ (without environment Γ), the solution of the initial problem comes from $\mathsf{let}\,\Gamma\,\mathsf{in}\,\mathbb{C}(\Gamma \vdash e : \tau/\Lambda)$. Constraint generation is given in figure **??**.

We assume in these rules that variables on the right-hand term are always chosen to be fresh and distinct from left-hand term: the obtained constraints are equal modulo α-conversion. The proposed mechanism is very close to the mechanism of [**?**], the locality constrained being added naturally.

The first equation corresponds to the instantiation of a variables (typing rule: VAR): x has type τ in locality Λ if and only if τ is acceptable in this locality, and τ is an instance of the type scheme associated to x. This last constraint makes sense when a let previously bound x to a type scheme.

The second equation corresponds to the type rule DEFFUN; it gives an existential constraint, that is the constraint solver should determine the corresponding terms. It states that term $\mathsf{fun}\,x \rightarrow e$ will have type τ in locality Λ if and only if types α, β and locality δ can be found such that τ is $\alpha \xrightarrow{\delta} \beta$, that locality δ is more local than Λ, and that it can be ensured that e has type β in locality δ if x has type α. This last condition is ensured by constraint $\mathsf{let}\,x : \alpha\,\mathsf{in}\,\mathbb{C}(e : \beta/\delta)$: it is likely that e contains instances of x, that will introduce the sub-constraint $x \preceq \tau'$ in the generated constraint, enforcing the coherence of α and τ'.

The fourth rule corresponds to the typing rule LET-IN. It is important as it defines generalisation. It states that term $\mathsf{let}\,x = e_1\,\mathsf{in}\,e_2$ has type τ in locality Λ if and only if, assuming x has any type such that $\mathbb{C}(e_1 : \alpha/\Lambda)$, we are ensured that e_2 has type τ in locality Λ. By inductively assuming that $\mathbb{C}(e_1 : \alpha/\Lambda)$ is a sufficient and minimal constraint, the type scheme $\forall\alpha[\mathbb{C}(e_1 : \alpha/\Lambda)].\alpha$ is by construction a principal type scheme for e_1. Thus, α is the only type variable appearing free in $\mathbb{C}(e_1 : \alpha/\Lambda)$, and the only one needing to be generalised.

Other rules are based on our typing rules and ensure that their hypothesises hold by rewriting them as constraints as well as that the format of the final type is correct.

THEOREM 1 (Correctness of Constraint Generation).
$(\mathbb{C}(\Gamma \vdash e : \tau/\Lambda)), \Gamma \vdash e : \tau/\Lambda$.

THEOREM 2 (Completeness of Constraint Generation).
If $\mathrm{C}, \Gamma \vdash e : \tau/\Lambda$, *with* C *and* Γ *without free variables, and* Γ *is coherent in* C, *then* $\mathrm{C} \Vdash \mathbb{C}(\Gamma \vdash e : \tau/\Lambda)$.

The proof of these theorems are rather direct from the definition of the constraint generation mechanism, each step of the mechanism establishing a necessary and sufficient constraint. For a detailed proof, the reader can refer to [**?**]: the constraint we add to not interfere with the described systems.

Extensions. The exception extension presented in section **??** needs new rules for constraint generation (figure **??**, "Additional rules"). References do not need any new rule, but adding constraint $(\not\twoheadrightarrow)$ (in addition to glob and loc constructions).

6.2 Constraints Solving

We will not describe here a full constraint solver. We base our work on the version corresponding to HM(=) presented in [**?**].

The only difference between our constraints and the constraints of HM(=) for which we have a solver are: locality variables, $\tau \triangleleft \Lambda$ constraints, for the references extension, constraints $\tau(\not\twoheadrightarrow)$.

Locality variable are, at this level, like type variables. They can be considered as type variables taking only g or ℓ as value. They add no complexity to the constraint solver.

Constraints $\tau \triangleleft \Lambda$ and $\tau(\not\twoheadrightarrow)$ have a specific structure related to the structure of τ. We showed that during their definitions, these constraints propagate in the sub-term of a type. Therefore it is possible to define a normal form where they are only applied to type variables. Thus they only have a role during the unification of these type variables. These variables could be either invalidated (from example if $\alpha\,\mathsf{par}$ and β are tentatively unified under constraint $\beta \triangleleft \ell$) or raise new constraints.

We only need to modify the constraint unification algorithm that should solve type equations without introduction and type scheme instantiation; the main part of the solver remains unchanged. The solver in [**?**] offers a cyclic type detection: it is not so different from our approach. When the unification algorithm can bring together a variable and a type or a locality inside a same equivalence class, any sub-constraint $\triangleleft$ or $(\not\twoheadrightarrow)$ that may appear in the current constraint and involves the this variable would dealt with the corresponding substitution and would be immediately normalised:

$$\alpha = \tau \wedge \alpha \triangleleft \Lambda \Rightarrow \alpha = \tau \wedge \tau \triangleleft \Lambda$$
$$\delta = \Lambda \wedge \alpha \triangleleft \delta \Rightarrow \delta = \Lambda \wedge \alpha \triangleleft \Lambda$$
$$\alpha = \tau \wedge \alpha(\not\twoheadrightarrow) \Rightarrow \alpha = \tau \wedge \tau(\not\twoheadrightarrow)$$

These rules could be applied immediately after a new equality is formed during the solving process and are followed by the reduction of the right-hand side term, and possibly new applications of the rules.

7. Related Work

There are many approaches to high-level parallel programming and functional parallel programming. Few use the same model of parallel than BSML [**? ?**]. Most of the time the type system is quite close to the type system of the host language [**?**] as the model of parallelism is very different and nesting is allowed [**? ? ? ?**].

In our case, parallelism and communications are explicit. Our type system prohibits nested parallelism in order to optimise efficiency as well as to allow the *predictability of performances*. BSML is clearly a lower level programming language compared to algorithmic skeletons for example but it comes with a realistic cost model and is well adapt to the writing coarse-grain algorithms. Moreover, it can be used to implement higher-order parallel functions that could be used as algorithmic skeletons [**?**].

8. Conclusion and Future Work

BSML provides high-level structured parallel computation; since it is based on Objective Caml, it benefits from its type system, but as we have seen this is not enough to guarantee the preservation of its structured parallelism. This paper intends to be an important building stone for the language, providing a type system that ensures the safety of parallel programs while retaining advanced features.

We are currently working on: the formalisation of the type system presented here and the proof of its properties within the Coq proof assistant and the extension to other features of the operational semantics of pure functional BSML in Coq; the implementation of a full BSML compiler including the type system, built upon the Objective Caml compiler.

References

[] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

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

[] G. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[] D. Caromel, L. Henrio, and M. Leyton. Type safe algorithmic skeletons. In *16th Euromicro International Conference on Parallel, Dis-*

$$
\begin{aligned}
\mathbb{C}(x : \tau/\Lambda) &= x \preceq \tau \wedge \tau \lhd \Lambda \\
\mathbb{C}(\mathsf{fun}\, x \to e : \tau/\Lambda) &= \exists \alpha\beta\delta.((\mathsf{let}\, x : \alpha \,\mathsf{in}\, \mathbb{C}(e : \beta/\delta)) \wedge \delta \lhd \Lambda \wedge \tau = \alpha \xrightarrow{\delta} \beta) \\
\mathbb{C}(e_1\, e_2 : \tau/\Lambda) &= \exists \alpha.(\mathbb{C}(e_1 : \alpha \xrightarrow{\Lambda} \tau/\Lambda) \wedge \mathbb{C}(e_2 : \alpha/\Lambda)) \\
\mathbb{C}(\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau/\Lambda) &= \mathsf{let}\, x : \forall \alpha[\mathbb{C}(e_1 : \alpha/\Lambda)].\alpha \,\mathsf{in}\, \mathbb{C}(e_2 : \tau/\Lambda) \\
\mathbb{C}((e_1, e_2) : \tau/\Lambda) &= \exists \alpha\beta.(\mathbb{C}(e_1 : \alpha/\Lambda) \wedge \mathbb{C}(e_2 : \beta/\Lambda) \wedge \tau = \alpha \times \beta) \\
\mathbb{C}(\mathsf{if}\, e \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2 : \tau/\Lambda) &= \mathbb{C}(e : \mathsf{bool}/\Lambda) \wedge \mathbb{C}(e_1 : \tau/\Lambda) \wedge \mathbb{C}(e_2 : \tau/\Lambda) \\
\mathbb{C}([e_i]_i^n : \tau/\Lambda) &= \mathbb{C}(n : \mathsf{int}/\Lambda) \wedge \exists \alpha.(\bigwedge_i \mathbb{C}(e_i : \alpha/\Lambda) \wedge \tau = \alpha \,\mathsf{par}) \\
\mathbb{C}(\langle e_i \rangle_i : \tau/\Lambda) &= \exists \alpha.(\bigwedge_i \mathbb{C}(e_i : \alpha/\ell) \wedge \alpha \lhd \ell \wedge \tau = \alpha \,\mathsf{par})
\end{aligned}
$$

Additional rules:

$$
\begin{aligned}
\mathbb{C}(\mathsf{try}\, e \,\mathsf{with}\, x \to e' : \tau/\Lambda) &= \mathbb{C}(e : \tau/\Lambda) \wedge \mathsf{let}\, x : \tau_{\mathsf{exn}} \,\mathsf{in}\, \mathbb{C}(e' : \tau/\Lambda) \\
\mathbb{C}(\mathsf{trypar}\, e \,\mathsf{withpar}\, x \to e' : \tau/\Lambda) &= \mathbb{C}(e : \tau/g) \wedge \mathsf{let}\, x : \tau_{\mathsf{exn}} \,\mathsf{array} \,\mathsf{in}\, \mathbb{C}(e' : \tau/g) \wedge \mathit{\Lambda} = g \\
\mathbb{C}(\mathit{fail}\, e : \tau/\Lambda) &= \mathbb{C}(e : \tau_{\mathsf{exn}}/\Lambda) \\
\mathbb{C}(\mathit{failpar}\, e : \tau/\Lambda) &= \mathbb{C}(e : \tau_{\mathsf{exn}} \,\mathsf{array}\, /\Lambda)
\end{aligned}
$$

Figure 3. Constraints Generation

tributed and Network-Based Processing (PDP 2008), pages 45–53. IEEE Computer Society, 2008. doi: http://doi.ieeecomputersociety.org/10.1109/PDP.2008.29.

[] M. M. T. Chakravarty, R. Leshchinskiy, S. L. P. Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In N. Glew and G. E. Blelloch, editors, *Workshop on Declarative Aspects of Multicore Programming (DAMP 2007)*, pages 10–18. ACM, 2007. doi: 10.1145/1248648.1248652.

[] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):365–376, 2003.

[] F. Gava. External Memory in Bulk Synchronous Parallel ML. *Scalable Computing: Practice and Experience*, 6(4):43–70, December 2005.

[] F. Gava and I. Garnier. CPS implementation of a BSP composition primitive with application to the implementation of algorithmic skeletons. *International Journal of Parallel, Emergent and Distributed Systems*, 2011. doi: 10.1080/17445760.2010.481785.

[] F. Gava and F. Loulergue. A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting. *Future Generation Computer Systems*, 21(5):665–671, 2005.

[] L. Gesbert. *Développement systématique et sûreté d'exécution en programmation parallèle structurée*. PhD thesis, University Paris Est, LACL, 2009. URL `http://tel.archives-ouvertes.fr/tel-00481376`.

[] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski. Bulk Synchronous Parallel ML with Exceptions. *Future Generation Computer Systems*, 26:486–490, 2010. doi: 10.1016/j.future.2009.05.021.

[] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22 (2):340–377, 2000. doi: http://www.acm.org/pubs/citations/journals/toplas/2000-22-2/p340-leroy/.

[] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System release 3.10, 2007. web pages at caml.inria.fr.

[] R. Loogen, Y. Ortega-Mallen, and R. Pena-Mari. Parallel functional programming in eden. *Journal of Functional Programming*, 3(15): 431–475, 2005.

[] F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS)*, LNCS 3515, pages 1046–1054. Springer, 2005.

[] W. F. McColl. Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, 12:265–272, 1996.

[] Q. Miller. BSP in a Lazy Functional Context. In *Trends in Functional Programming*, volume 3. Intellect Books, may 2002.

[] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[] M. Odersky, M. Sulzmann, and M. Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

[] F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[] J. Reppy and Y. Xiao. Toward a parallel implementation of Concurrent ML. In *Workshop on Declarative Aspects of Multicore Programming (DAMP 2008)*, 2008. `http://clip.dia.fi.upm.es/Conferences/DAMP08/damp08proc.pdf`.

[] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.

[] J.-P. Talpin and P. Jouvelot. The Type and Effect Discipline. *Information and Computation*, 111(2):245–296, June 1994.

[] J. Tesson and F. Loulergue. A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation. In *11th International Conference on Computational Science (ICCS 2011)*, Procedia Computer Science, pages 36–45. Elsevier, 2011. doi: 10.1016/j.procs.2011.04.005.

[] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`.

[] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8):103, 1990.

[] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[] A. Zavanella. Skeletons, BSP and performance portability. *Parallel Processing Letters*, 11(4):393–405, 2001.