Pattern Matching and Exceptions Handling for Bulk Synchronous Parallel ML

Dabrowski, F.

Technical Report TR-2003-06

Laboratory of Algorithms, Complexity and Logic
University of Paris XII, Val-de-Marne
61, avenue du Général de Gaulle
F-94010 CRÉTEIL Cedex – France
Tel: +33 (0)1 45 17 16 47
Fax: +33 (0)1 45 17 66 01

Pattern Matching and Exceptions Handling for Bulk Synchronous Parallel ML

Dabrowski, F.

Technical Report TR-2003-06

Laboratory of Algorithms, Complexity and Logic
University of Paris XII, Val-de-Marne
61, avenue du Général de Gaulle
F-94010 CRÉTEIL Cedex – France
Tel: +33 (0)1 45 17 16 47
Fax: +33 (0)1 45 17 66 01

Filtrage, Exceptions et Bulk Synchronous Parallel ML

Frédéric Dabrowski

Stage du DEA “Programmation”

sous la direction de
Frédéric Loulergue
Laboratoire d’Algorithmique, Complexité et Logique
61, avenue du général de Gaulle
94010 Créteil cedex – France
loulergue@univ-paris12.fr

Avril-Septembre 2003

Remerciements

Je tiens à remercier tout particulièrement Frédéric Loulergue pour la qualité de son encadrement, toujours au juste équilibre entre liberté et aiguillage vers des voies plus raisonnables. Je remercie également Frédéric pour ce qui, bien plus qu'un simple encadrement, a été un enseignement scientifique, méthodique et technique.

Merci, également, à Frédéric Gava pour d'intéressantes discussions sur les thèmes du projet.

Je remercie, évidemment, les enseignants du DEA Programmation pour l'excellente qualité de leur enseignement, qualité qui ne manque pas d'ouvrir de nombreuses portes aux étudiants de ce DEA. Je recommande ce DEA à tout étudiant soucieux d'un enseignement rigoureux et de qualité dans un domaine aussi passionant que l'informatique théorique.

Je dois, bien sûr, remercier également l'équipe enseignante de l'université Paris XII et particulièrement ceux dont les enseignements portant sur un aspect théorique de l'informatique sont parvenus à me montrer un aspect de cette discipline autre que celui des nuits blanches passées à la recherche de bug dans des programmes dont le code aurait fait peur à tout utilisateur de langages fonctionnels.

Je remercie enfin ma "Mamie", sans qui rien n'aurait été possible.

Présentation

Ce stage a été effectué au LACL (laboratoire d'algorithmique, complexité et logique) de l'université Paris XII dans le cadre du projet CARAML (CoordinAtion et Répartition des Applications Multiprocesseurs en objective camL, http://www.caraml.org), qui est l'un des projets de l'ACI Globalisation des ressources informatiques et des données. Ce projet a pour but le développement de bibliothèques pour le calcul haute-performance et globalisé autour du langage CAML de l'INRIA : bibliothèques de primitives parallèles et globalisées, bibliothèques applicatives orientés SGBD et calcul numérique et exemples d'application à la simulation moléculaire. Il réunit des chercheurs des universités Paris 6,7 (PPS, Preuve programmation et système), Paris 12 (LACL), de l'université d'Orléans (LIFO, Laboratoire d'informatique fondamentale d'Orléans) et de l'INRIA. Le stage, qui s'incrit plus particulièrement dans le cadre du développement de bibliothèques de primitives parallèles et globalisées, repose sur la bibliothèque `BSMLlib`, développée par Frédéric Loulergue au LACL et destinée à la programmation parallèle suivant le modèle BSP (Bulk Synchronous Parallel). Le langage BSML (Bulk Synchronous Parallel ML), basé sur un λ-calcul étendu par un ensemble de primitives parallèles, a déjà fait l'objet de plusieurs extensions, en particulier en ce qui concerne l'incorporation de traits impératifs au langage et la réalisation d'un sytème de type polymorphes (travaux réalisés par Frédéric Gava, doctorant au LACL). Le sujet du stage proposait d'étudier les possibilités d'extension du langage BSML (Bulk Synchronous Parallel ML) par des fonctionnalités de gestion des exceptions. En particulier, le langage BSML reposant sur deux niveaux, local et global, la question se posait de savoir dans quelle mesure il serait possible de rattraper des exceptions aux deux niveaux et surtout de rattraper au niveau global, des exceptions levées au niveau local. Initialement, la possibilité de pouvoir rattraper au niveau global des exceptions levées localement conduisait à se poser la question du filtrage de "vecteurs" parallèles ce qui a conduit à l'étude de cette possibilité en première partie du stage. Finalement, la solution retenue pour la gestion des exceptions n'utilise pas cette possibilité mais elle sera néanmoins ajoutée au langage, en particulier pour la transformation d'un des opérateurs du langage, la conditionnelle globale. L'étude de ces extensions du langages BSML devait en premier lieu permettre de déterminer l'expressivité souhaitée et de formaliser ces idées dans un BSλ-calcul étendu, ce calcul servant de fondement théorique au langage BSML. Cette étude devait être suivit de l'implémentation de ces extensions pour la bibliothèque `BSMLlib`.

Résumé

Le langage BSML (Bulk Synchronous ML) est un langage data-parallel fonctionnel pour la programmation d'algorithmes BSP (Bulk Synchronous Parallel) en mode dit direct. Dans un algorithme BSP en mode direct, la structure physique des processus est rendue explicite. Le temps d'exécution peut alors être estimé d'après le code source et les blocages et l'indéterminisme sont évités. La bibliothèque `BSMLlib`, l'implémentation actuelle du langage BSML , autorise en tant qu'extension d'Objective Caml, l'utilisation du mécanisme de gestion des exceptions accompagnant ce langage. Cependant, l'interaction des exceptions Objective Caml avec le BSλ-calcul (le modèle théorique servant de fondement au langage BSML) n'a pas encore été étudiée et soulève des problèmes de sureté. En particulier, l'utilisation d'opérations de synchronisation collectives nécessite la participation de tous les processus lors de l'appel de l'une de ces opérations, dans le cas contraire, les processus impliqués dans cet appel sont bloqués. Le langage BSML, sans exceptions, assure que tous les processus participent à un tel appel et ainsi que les risques de blocage sont écartés (en dehors du risque de panne). Lorque l'on introduit les exceptions d'Objective Caml, cette propriété de sureté ne tient plus. Ainsi, il est nécessaire d'étudier une nouvelle sémantique, adaptée à la gestion des exceptions, pour retrouver cette propriété. Ce travail présente une telle sémantique dans laquelle la participation de tous les processus lors d'un appel à une opération de synchronisation est garantie et les risques de blocage sont écartés. On présentera également une sémantique permettant le filtrage des vecteurs parallèles introduits par le langage BSML. Cette sémantique a été étudiée dans le cadre d'une approche du traitement des exceptions qui n'a pas été retenue ici, mais ses fonctionnalités seront néanmoins ajoutées au langage BSML.

Abstract

The BSML (Bulk Synchronous ML) language is a data-parallel functional language for programming BSP (Bulk Synchronous algorithms) algorithms in so-called direct mode. In a direct mode BSP algorithm, the physical structure of processes is made explicit. The execution time can then be estimated and dead-locks and indeterminism are avoided. The `BSMLlib` library, the current implementation of the BSML language, permits, as an extension of Objective Caml, the use of the exceptions handling mechanism that comes with this language. However, the interaction of Objective Caml exceptions with the BSλ-calculus (the theoretical model underlying the BSML language) has not yet been studied and yields some safety issues. In particular, the use of collective synchronization operations needs the participation of all processes during the call to one of these operation, should the opposite occur, processes involved in this call are locked. The BSML language, without exceptions, ensures that all processes participate to such a call and thus that dead-locks are avoided (except for process failure). When one introduces Objective Caml exceptions, this safety property does not hold any more. Thus it is needed to study a new semantics, suitable to exceptions handling, to recover this property. The present work introduces such a semantics in which the participation of all processes is ensured and dead-lock issues are avoided. We will also introduce a semantics allowing the pattern-matching of BSML parallel vectors. This semantics has been studied in the framework of a previous work on exceptions handling which has not been retained here but its functionalities will be nethertheless add to the BSML language.

Contents

Chapter 1

Introduction

Although many parallel machines are available (Cray T3 series, SGI Origin series, Fujitsu AP series, clusters of PC,...), the development of softwares on this kind of architectures is still a delicate task. These systems are very complex, mainly because of indeterminism and deadlock. These concurrency related issues increase the development cost and the risk of run-time failure. Thus it is an important task to build some new theoretical models and programming languages for those systems. While developing such languages, one has first to decide the level of abstraction to offer. A first solution, totally devoid of concurrency, is to let the user to write his programs in a sequential language and to use a compiler with an automatic parallelization method to detect the parallelism in source. This approach has the great advantage to permit existing code reuse (what is significant if we consider, for example, the huge quantity of existing libraries for numerical computation in Fortran). Several methods for automatic parallelization exist, from the basic loop transformation to the more recent geometrical method. This approach, although very attractive is hard to generalize and its interest is challenged by constraints it imposes on programming methodology to be efficient.

For example, in [Col89], the author gives the factorial definition as an example:

$$\begin{cases} factorial\ 0 \ = \ 1 \\ factorial\ (S\ n) \ = \ (S\ n) * factorial\ n \end{cases}$$

With methods based on dependence graph analysis, for example, a compiler could have difficulties to extract parallelism from this algorithm whereas there exists an efficient but less intuitive solution which could easily yields to a parallelized treatment:

$$\begin{cases} factorielle\ 0 \ = \ 1 \\ factorielle\ n \ = \ produit\ 1\ n \\ produit\ a\ a \ = \ a \\ produit\ a\ b \ = \ (produit\ a\ (\frac{a+b}{2})) * (produit(a\ (\frac{a+b}{2}) + 1)b) \end{cases}$$

This example leads us to consider an other solution for the development of languages for parallel software development which is diametrically opposed to the previous one. This solution relies in the extension of an existing language by a set of communication primitives as in, for example, MPI [SG98] or Concurrent ML [**?**]. Although very efficient from the performance point of view, this approach leaves to the user the whole responsibility for concurrency handling. He's in charge firstly of problem decomposition, secondly of communication between process what throws him into previously observed issues and requires a high skill level. The first approach, of a too high level of abstraction, precludes an intelligent use of parallelism while the second approach doesn't restricts its complexity.

Starting from that observation, an intermediary approach, restricting the user task to problems decomposition has been investigated. This approach, provides to the user a set of higher level functions named skeletons. These functions depends of a sequential code which is used for parallel computation. For example, the well-know map function which applies a given function to all elements of a given list becomes a skeleton of the distribution of a list, the application of a function to elements of sub-lists and the gathering of resulting sub-lists. In this approach, the operational parallel semantic is implicit and the user writes his programs by composition of such higher order function. It is so possible, while having the possibility of to choose the problem decomposition to avoid concurrency related issues.

However, one can still object to this approach that it may be strongly difficult to find a subset of parallel computational skeletons which would be sufficient to express, in an efficient way, an acceptable quantity of algorithms. A symptom of this is that one has to build a specific library for a given field.

The `BSMLlib` [BLH00, HL02, Lou02] concept is quite close to this last approach and yet more general, the bricks of parallel computation are more elementary. In the `BSMLlib`, the Objective Caml language [Ler02, **?**] is extended by a parallel data-type named data field and a set of primitive on this data-type:

`mkpar`	data fields creation
`apply`	point to point application
`get`	data-field values exchange
`ifat`	global conditional

Rather than to restrict programming to composition of high level algorithmic structures, the `BSMLlib` proposes to restrict it to communication primitives in a synchronous data-parallel framework. This methodology yields to a more expressive language, particularly it permits to express skeletons and to extend it in a safe way. Moreover, `BSMLlib` programming follows the Bulk Synchronous Parallel [Val90, McC96, SHM97] (BSP) cost model which leads to portable performance evaluation (the BSP cost model is function of three parameters depending on the target architecture).

Pattern-matching and exceptions handling are very attractive features of functional languages. In sequential languages, exceptions handling is known to be very useful in terms of safety, efficiency and expressivity. Parallel programming may also take strong benefits from this feature, in [DL94], the authors show that the use of exceptions leads to higher performance in this paradigm. This paper compares the use of algorithms that run quickly and often give the right answer as well as other, slower, algorithms that are always right. They present the following paradigm which takes advantage of the faster algorithm when possible and use the slower if needed

1. Use the fast algorithm to compute an answer
2. Quickly and reliably access the accuracy of the computed answer
3. In the unlikely event the answer is not accurate enough, recompute it slowly but accurately.

Since the `BSMLlib` is an extension of Objective CAML, it is possible to use the exceptions handling mechanism of this language in BSML programs. However, the use of that Objective Caml feature yields some safety issues and particularly dead locks issues. The BSML language presents two level, local and global. A BSML program consist in the distributed evaluation of as many copies as process of the the same sequential program. Parallel vectors are used to avoid explicit access to process names, and are the only place in a BSML program where a process is able to work out its own behavior. Collective synchronization operations which need to be called by every processes, (should the opposite occur, the program would be locked) are used on parallel vectors and can not be called by a strict subset of processes since the nesting of parallel vectors is not allowed. However, this safety property does not holds any more when one introduce Objective Caml exceptions in BSML programs. In particular, since parallel vectors in the `BSMLlib` implementation are common terms nested in the constructor `Par` of the concrete type `par`, an exception raised in a parallel vector, not caught within this vector, would propagates outside global structure of parallel vectors. When exceptions escapes from vectors structure, the safety of BSML programs is not ensured any more. One problem which may occurs is that if an exception propagates locally through an operation involving a synchronization between processes, then when a process would try to synchronize with process on which an exception has been raised, these would be locked. One can also notice that it becomes possible, when exceptions are introduced, that a process works out its own behavior even outside a parallel vector. For example, one may write a program where a process raises, within a parallel vector, an exception which propagates the process name outside the parallel vector. It would then be possible to handle that exception outside the parallel vector and to work out the behavior of the process.

While thinking about exceptions handling, the question of the matching of parallel vectors of exceptions was brought. We then, first, studied the pattern matching

of parallel values. As for exceptions, it is possible to use the Objective Caml pattern-matching facilities but we need specific constructions permitting a global pattern-matching of parallel vectors in order to have a global control of processes behavior. We could, for example, write programs where the Objective Caml pattern-matching would be used within parallel vectors but the matching would then be local and the choice made by processes would not be a concerted and would have no impact on the global behavior. The exceptions handling mechanism presented here, doesn't use the matching of parallel vectors but since pattern-matching is a very expressive programming construction, we have decided to keep this work which will be add to the `BSMLlib` implementation.

In chapter 2, we present a state of the art of pattern-matching and exceptions handling within sequential and concurrent frameworks. In chapter 3, BSP cost model and BSλ-calculus [LHF00] (the theoretical basis of the `BSMLlib`) and the `BSMLlib` itself are introduced. Chapter 4 presents a BSλ-calculus with pattern matching . Chapter 5 presents a BSλ-calculus with exceptions handling facilities. Chapter 6 presents conclusion and future works.

Chapter 2

State of the Art

2.1 Pattern Matching

2.1.1 Calculi with Pattern Matching

Pattern Matching as Cut Elimination [CK99]

Pattern Matching as Cut Elimination
Serenella Cerrito and Delia Kesner
CNRS and LRI

In this paper, the authors present a Typed Pattern Calculus with Explicit Substitution, called TPC_{ES} were both the typing rules and the reduction rules relies on the Gentzen's sequent calculus for intuitionnistic minimal logic. This calculus follows the principle of the Curry-Howard isomorphism wich interprets types as proposition and proof as programs, usually between simply typed λ-calculus and natural deduction. The Gentzen sequent calculus (we consider here its intuitionnistic version) is an illustration of symmetries of logic and permits to extend the Curry-Howard isomorphism to more complicated abstraction than in simply type λ-calculi. Contrary to natural deduction, there is no elimination rules but rather left introduction rules on top of right ones. This calculus can so be used to build and type nested patterns and terms. For exemple, with the $\times_{left} rule$

$$\frac{\frac{\Gamma, x : A, y : B \vdash x : A}{\Gamma, (x, y) : A \times B \vdash x : A}}{\Gamma \vdash \lambda(x, y) : A \times B.x : A \times B \to A}$$

The main reproach the authors make to usual programming languages is that there is no notion of globally typing a set of clauses (rewrite rules). They underline the fact that ad-hoc treatment of exhaustivity and non-redundance is a symptome of this abscence of "pattern of a given type" that would cover all the possible constructors. The main contribution of this paper is to bring a calculus with pattern-matching

facilities where exhaustivity and non-redundance are assured by the type system. In a previous paper the authors presented a first version of such a calculus but the notion of reduction didn't correspond to normalization sequent proof, but to normalization of proof in natural deduction. There was also an external match operator wich mapped a substitution to a nested pattern and a term. This new calculus is based on a computationnal interpretation of the Gentzen sequent proof and allows to model pattern-matching via cut-elimination. The cut operation over a non atomic formula is replaced by some cuts operation over its sub-formulae. The external operator match is so not needed any more and the substitution is also internalized by the use of explicit substitution wich can also be modeled via cut-elimination in Gentzen sequent calculus.

The TPC_{ES} enjoys usual basic properties such as confluence, subject reduction and strong normalization. It is also shown that the typing is still decidable despite that (because of the gentzen proof system) there is several derivations of typing judgment. The authors show that the typing can be made by decomposing the left members of proof what permits to use usual typing rules.

A Lambda-calculus with Patterns [VH]

A LAMBDA-CALCULUS WITH PATTERNS
Pimpen Vekkakiva, Chulalongkorn University, Thailand
Mark E. Hall, Hastings College, USA

In this paper, the authors introduce a λ-calulus with patterns,allowing matching and by case function definition. The calculus uses two main kinds of reduction which concerns simple (resp. compound) abstractions. A compound abstraction corresponding to the by case definition of function and the simple one to usual abstraction. The calculus doesn't introduce mechanical matching transformation. The reduction of the application of a simple abstraction $(\lambda P.e)\ e'$ if defined by $e[\sigma_{(p,e')}]$ where σ is a substitution such that $\sigma_{(p,e')}(p) = e'$. The reduction of the application of a compound abstraction $(\lambda P.e|a)\ e'$ is either the reduction of $(\lambda P.e)\ e'$ if a substitution such as the previous one exists and $a\ e'$ otherwhise. There is no reduction rule for the case of matching-failure so the evaluation in that case simply stop. Furthermore, the calculus is not typed. The notion of simple and compound abstraction are formalized as well as the notion of redex in the two case (an abstraction is a redex if a substitution exist). This calculus enjoys the Church-Rosser property.

Pure Patterns Type Systems [BCKL03]

Pure Patterns Type Systems Gilles Barthe and Horatiu Cirsteas, Claude Kirshner and Luigi Liquori
LORIA, INRIA and University Nancy II

In this papers, the authors generalize Pure Type Systems (PTS) by introducing patterns to obtain a "Pure Patterns Type Systems. The main properties of this new system is to make possible to have not linear patterns and to allow free variables in the patterns of an abstraction. The confluence of the calculus is achieved through a modification of the Van Oostrom's Rigid Pattern Condition which restrict the set of terms which can be used as patterns. In an abstraction, a context is add to describe the subset of variables which have to be bounded by the abstraction while others are free. In this calculus, the matching is delayed and a rule is dedicated to solving the corresponding "equations". If an equation cannot be solved the term is simply not reduced. The delay introduce in the pattern matching permits not to have to match the argument with the pattern but to use a evolving constraint system which is blocked as soon as the matching is not possible any more.

2.1.2 Calculi with Exceptions Handling Facilities

A Simple Calculus of Exception Handling [de 95]

A Simple Calculus of Exception Handling
Philippe de Groote, INRIA,CRIN,CNRS

In this paper the author present a formal description of exception handling à la ML. From an analysis of this calculus he presents what can be see from the logical point of view as a inconsistency, the fact that some raised exceptions may not be caught and then that a program evaluation may lead to an uncaught exception. The type system uses, as usual, the type $\perp$ for raised exceptions and it seems then legitime, in the Curry-Howard spirit, to say that a program should not evaluate like that. This issue comes from the fact that some bound variables become free during evaluation. The author adress this problem by proposing a modified operationnal semantics called $\lambda^{\rightarrow}_{exn}$. The calculus is shown to enjoy Church-Rosser and subject reduction properties. The semantics is also shown to be conservative as far as non-exceptional values are concerned and may be seen as an interesting ML-like language extension. An other interesting result is presented. The evaluation is conservative when dealing with non-exceptionnal values but one may wonder about other cases. Since a program which evaluates to an uncaught exception in ML does not any more, it would be possible that such a program loops in the semantics but it doesn't, it is shown that any infinite sequence of reduction in $\lambda^{\rightarrow}_{exn}$ induces an infinite sequence of β-reduction in the simply typed λ-calculus.

2.1.3 Parallel Languages with Exceptions Handling Facilities

Concurrent exception handling [Iss01]

Concurrent Exceptions Handling

Valérie Issarny, INRIA, UR Rocquencourt

In this paper, the author proposes a general concurrent exception handling mechanism. Processes communicate by message passing and the nesting of processes is allowed. This model presents two cases of exceptions handling. The first one concerns the communication between processes, the author remarks that the occurence of an exception within a process may interfere with the behavior of other processes. For example, if two processes have to synchronise at a point of their execution . If one of the processes raises an exception, then this occurence may prevent achievement of a synchronous communication expected by the other and then leads to a dead-lock. To avoid such dead-lock, the proposed model introduces the notion of *global exception*. In that model, if a process raises an exception which may prevent the realisation of a synchronisation operation, then its excecution terminates signaling a global exception. When the other process will try to communicate with the terminated one then it will catch the global exception. Finally when a process catches a global exception then the handler for this exception, if exists, acts the same way as in the sequential case.

The second proposition of the model concerns the case of multiprocedure where multiple processes participate to the computation of the result (A multi-procedure launch multiple processes to get a result). In that case, during the computation many exceptions may be raised and the whole cooperating calculus must be terminated. The model introduces the notion of *concerted exception* which results from a set of concurrently raised exceptions. When one declares a multiprocedure, one must stipulates an external *resolution function* which maps concurrent exceptions to a concerted exception. If some concurrent exceptions are raised during the evalutation of the multi-procedure, then it returns the result of the application of the resoltution function to the array `exn` of concurrent exceptions where `exn[i]` holds the exception raised by processe i if one has been raised, `null` otherwise.

Asynchronous Exceptions in Haskell [MJMR01]

Asynchronous Exceptions in Haskell
Simon Marlow and Simon Peyton Jones, Microsoft Research, Cambridge
Andrew Moran, Oregon Graduate Institute
John Reppy, Bell Labs, Lucent Technologies

This paper introduces an asynchronous exceptions handling mechanism for concurrent Haskell. By asynchronous exception, the authors mean exceptions which may be raised as the result of an external event and may occur at any point during execution, for example when the user types Control-C. Asynchronous exceptions cannot be considered as part of the semantics of a expression at the opposite of synchronous exceptions which are more tractable as for example divide by zero, pattern-matching failure and explicitly raised exceptions. Thus the handling of asyn-

chronous exceptions is a quite difficult task and raises some difficulties the authors try to address in this paper.

Concurrent Haskell is extended by the possibility of to raise explicitly asynchronous exceptions from one process to an other. This is done by the introduction of the operator `throwTo :: Exception -> ThreadId -> IO ()` where ThreadId is a data type corresponding to thread identifiers returned by the modified forkIO introduced in this language. In Concurrent Haskell forkIO is used to create new threads but does not returns identifiers. The meaning of `throwTo e t` is that `e` is raised as soon as possible on the thread identified by `t`. A call to `throwTo` is synchronous, in the sense that it waits for the exception to be delivered. This paper also introduce a specific exception to kill processes.

Asynchronous exceptions are caught as usual but some instruction permit to put restrictions on the time at which such an exception can be signal to the receiving thread. `block` and `unblock`

If an exception is raised in a computation within a `block` then the thread which raised it will be blocked until asynchronous exceptions be allowed by a `unblock`. `block` and `unblock` may be arbitrarily nested. In this paper, the authors also give some high level constructions based on the previous one and some indication about the implementation of ground constructions.

Exception Handling During Asynchronous Method Invocation [KO02]

Exception Handling During Asynchronous Method Invocation
Aaron W. Keen and Ronald A. Olsson
University of California

This paper introduce an exception handling mechanism in the framework of asynchronous method invocation. The main issue addressed in this paper is about the call to methods which may raise an exception but whom the caller may be terminated when the exception propagates out. This issue is addressed by the introduction of handler object which must be associated to asynchronous method. When such a method is called then an handler object is concurrently running and is in charge of handling exceptions raised within this method, even if the caller is terminated. This work is done in the `JR` programming language which is a `JAVA` extension.

Chapter 3

Functional Bulk Synchronous Parallelism

3.1 Bulk Synchronous Parallelism

The Bulk Synchronous Parallel (BSP) model [Val90, McC96, SHM97] describes: an abstract parallel computer, a model of execution and a cost model. A BSP computer has three components: a homogeneous set of processor-memory pairs, a communication network allowing inter processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. A wide range of actual architectures can be seen as BSP computers.

The performance of the BSP computer is characterized by three parameters (expressed as multiples the local processing speed):

- the number of processor-memory pairs **p**
- the time **l** required for a global synchronization
- the time **g** for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation (communication phase where every processor receives/sends at most h words) in time $g \times h$.

Those parameters can easily be obtained using benchmarks [HMa98].

A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjointed phases (Fig. 3.1):

1. Each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes;
2. the network delivers the requested data transfers;
3. a global synchronization barrier occurs, making the transferred data available for the next super-step.

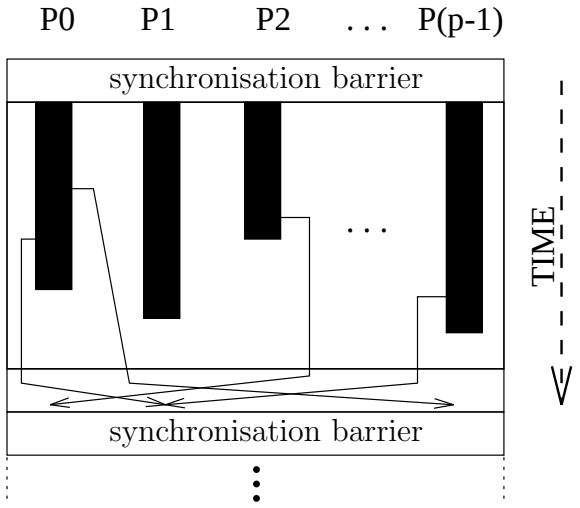

Figure 3.1: A BSP superstep

The execution time of a super-step s is, thus, the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

$$\text{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} \times g + l$$

where $w_i^{(s)}$ = local processing time on processor i during super-step s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during super-step s.

The execution time $\sum_s \text{Time}(s)$ of a BSP program composed of S super-steps is, therefore, a sum of 3 terms:

$$W + H \times g + S \times l \text{ where } \left\{ \begin{array}{rcl} W & = & \sum_s \max_i w_i^{(s)} \\ H & = & \sum_s \max_i h_i^{(s)}. \end{array} \right.$$

In general, W, H and S are functions of p and of the size of data n, or of more complex parameters like data skew. To minimize execution time, the BSP algorithm design must jointly minimize the number S of super-steps, the total volume h with imbalance of communication and the total volume W with imbalance of local computation.

Bulk Synchronous Parallelism (and the Coarse-Grained Multicomputer, CGM, which can be seen as a special case of the BSP model) is used for a large variety of applications: scientific computing [**?**, **?**], genetic algorithms [BV99] and genetic programming [DK96], neural networks [RS98], parallel databases [BBH99], constraint solvers [GHMR98], *etc.* It is to notice that "A comparison of the proceedings of the eminent conference in the field, the ACM Symposium on Parallel Algorithms and Architectures, between the late eighties and the time from the mid nineties to

today reveals a startling change in research focus. Today, the majority of research in parallel algorithms is within the coarse-grained, BSP style, domain" [Edi99].

3.2 BSλ-calculi

In this section we introduce an extension of the classical λ-calculus called the BSλ-calculus. This calculus introduces operations for data-parallel programming but with explicit processes in the spirit of BSP. We now describe the BSλ syntax, its reductions with operational motivations and an important extension.

3.2.1 The BSλ-calculus

Syntax

We consider a set $\dot{V}$ of *local variables and a set* $\bar{V}$ *of* global variables. Let $\dot{x}, \dot{y}, \ldots$ denote local variables and $\bar{x}, \bar{y}, \ldots$ denote global variables from now on. x will denote a variable which can be either local or global. The syntax of BSλ begins with *local terms t from the λ-calculus representing programs or values stored in a processor's local memory. The set* $\dot{\mathcal{T}}$ *of local terms is given by the following grammar:*

$$\begin{array}{rcll} t & ::= & \dot{x} & \\ & | & t\ t & \textit{application} \\ & | & \lambda\dot{x}.t & \textit{lambda abstraction} \\ & | & c & \textit{constants} \end{array}$$

where $\dot{x}$ *denotes an arbitrary local variable. We will abbreviate to* $(t_1 \rightarrow t_2, t_3)$ *the conditional terms* $t_1\ t_2\ t_3$*. The processors names are closed λ-terms in such a way that the data fields could be intentionally expressed per a special constant named* π*. We assume for the sake of simplicity a finite set* $\mathcal{N} = \{0, \ldots, p-1\}$ *which represents the set of processors names, note* n_i *for a processor name belonging to* $\mathcal{N}$ *and p the finite number of processor.*

The principal BSλ terms E are called global and represent *data fields, i.e. some functions from a fixed set of processors to values. The set* $\bar{\mathcal{T}}$ *of global terms is given by the following grammar:*

$$\begin{array}{rcll} E & ::= & \bar{x} & \\ & | & E\ E & \textit{application} \\ & | & E\ e & \textit{application} \\ & | & \lambda\bar{x}.E & \textit{lambda abstraction} \\ & | & \lambda\dot{x}.E & \textit{lambda abstraction} \\ & | & \pi\ e & \\ & | & E\#E & \\ & | & E?E & \\ & | & (E \xrightarrow{e} E, E) & \end{array}$$

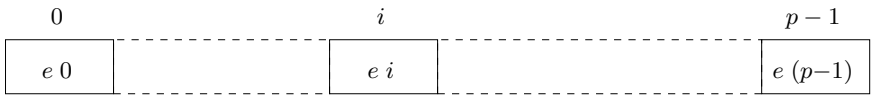

Figure 3.2: The term $\pi\ e$

where $\bar{x}$ denotes an arbitrary local variable. Terms of the form $(\lambda\bar{x}.E)\ e$ (respectively $(\lambda\dot{x}.E)\ E$) constitute implicit errors because they represent a local argument to a global → global function (respectively a global argument to a local → global function). Now we give the denotational meaning.

The term $\pi\ e$ represents a data field whose values are given by the function e (figure 3.2).

*The global terms denote some functions from $\mathcal{N}$ to local values, some functions between them, some functions from local terms to such functions. In particular, the denotation of $\pi\ e$ at processor n_i, the value of $e\ n_i$ (figure 3.2). The forms $E_1\#E_2$ and $E_1?E_2$ are respectively called parallel application (*apply-par) and *get. Apply-par represents point-wise application of a field function to a fields values (the pure computation phase of a BSP super-step). Get represents the communication phase of a BSP super-step, i.e., a collective data exchange with a barrier synchronisation. In $E_1?E_2$, the resulting data field contains values from E_1 taken at processor names defined in E_2. The last form of global terms defines synchronous conditional expressions. The meaning of $(E_1 \to E_2, E_3)$ is E_2 (respectevely E_3) if the data field denoted by E_1 has* **true** *(respectively* **false***) value at processor name denoted by e. We will formalise this in the next section.*

Rules

We now define the reduction of BSλ terms.

The reduction of local terms is simply β-reduction, obtained from the local β-contraction rule apply to any sub-term.

$$(\beta) \quad (\lambda\dot{x}.e)e' \to e[\dot{x} \leftarrow e']$$

The reduction of global terms is defined by syntax-directed rules and context rules (any sub-terms) which determine the applicability of the former. First, there are rules for global β-reduction:

$$(B) \quad (\lambda\bar{x}.E)\ E' \to E[\bar{x} \leftarrow E']$$
$$(B') \quad (\lambda\dot{x}.E)\ e' \to E[\dot{x} \leftarrow e']$$

Because the terms $(\lambda\dot{x}.E)\ E'$ is syntaxively correct, but the substitution $E[\dot{x} \leftarrow E']$ is not, we need these two rules.

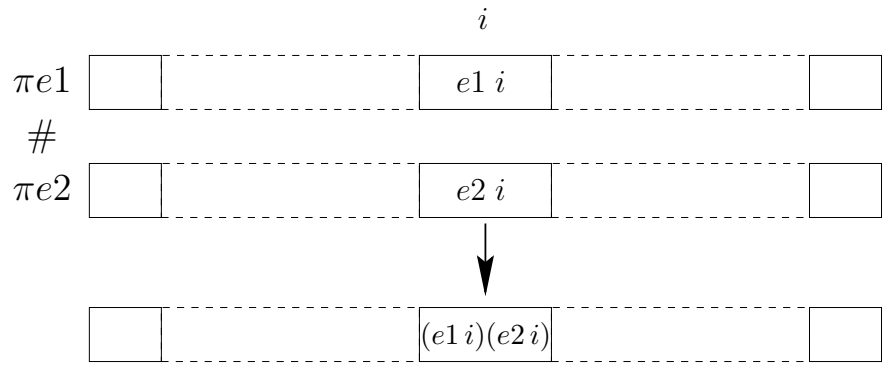

Figure 3.3: The rule ($\#\pi$)

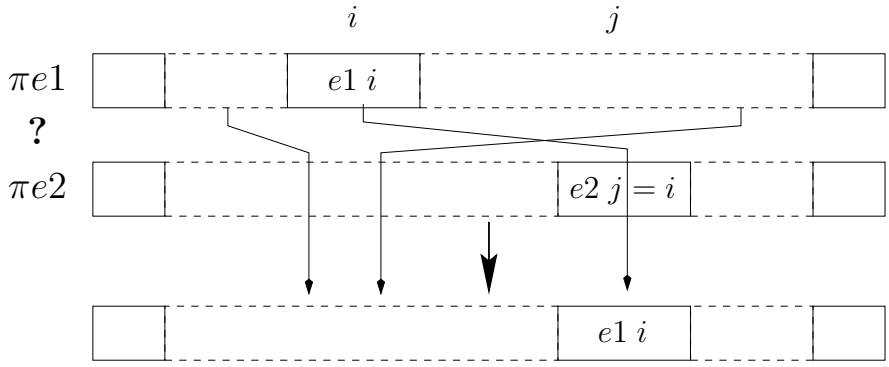

Figure 3.4: The rule ($?\pi$)

There are also axioms for the interaction of the data fields with other BSP operations:

($?\pi$) $(\pi\ e_1)?(\pi\ e_2) \rightarrow \pi(\lambda\dot{x}.e_1(e_2\ \dot{x}))$ *where* $\forall i \in \mathcal{N},\ (e_2\ i) \rightarrow n \in \mathcal{N}$ *(figure 3.3)*

($\#\pi$) $(\pi\ e_1)\#(\pi\ e_2) \rightarrow \pi(\lambda\dot{x}.(e_1\ \dot{x})(e_2\ \dot{x}))$ *(figure 3.4)*

These two equations encode the denational signification of the BSP's operators on the fields. In particular, get is the functional composition in the π. The value of $\pi\ e_1?\pi\ e_2$ at processor n_i is the value of $e_1(e_2\ n_i)$, i.e. the value of $\pi\ e_1$ at processor name given by the value of $e_2\ n_i$. Notice that, in pratical, this represents an operation whereby every processor receives one and only one value from one and only one other processor.

Next, the global conditional is defined by two rules:

$$(\xrightarrow{e}) \quad ((\pi\ e) \xrightarrow{e'} E_1, E_2) \rightarrow E_1 \quad \text{where } e\ e' \rightarrow \mathbf{true}$$
$$(\xrightarrow{e}) \quad ((\pi\ e) \xrightarrow{e'} E_1, E_2) \rightarrow E_2 \quad \text{where } e\ e' \rightarrow \mathbf{false}$$

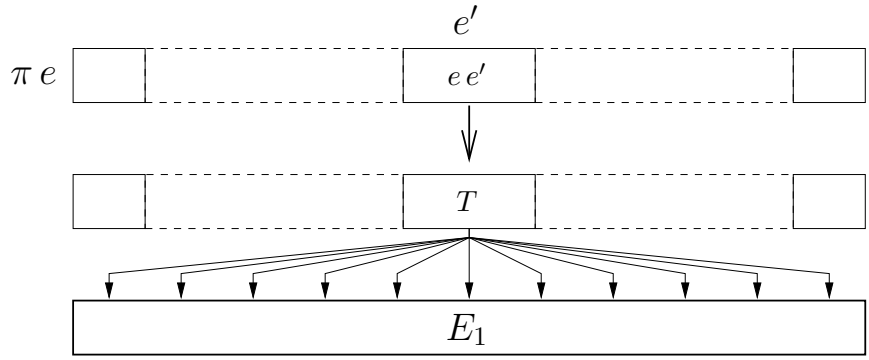

Figure 3.5: The rule ($\xrightarrow{e}$)

where e' belongs to $\mathcal{N}$ ((Figure 3.5).

The two cases generate the following bulk-synchronous computation: first a pure computation phase where all processors evaluate the local term e' yielding to n; then processor n evaluates e n giving v'. If $v' =$ **true** (respectively **false**, then the processor n broadcasts the order for global evaluation of E_1 (respectively E_2); otherwise the computation fails.

Theorem 1 *The BSλ-calculus is confluent. [LHF00]*

3.2.2 The BSλ_p-calculus

The BSλ_p-calculus is a confluent extension of the BSλ-calculus where the basic parallel objects are enumerated and correspond to the processor. Thus, its parallel data structures are flat and map directly to physical processors. Now, the data fields are enumerated on the processor's names $0, \ldots, p-1$ and the abstract term $\pi\ e$ of the BSλ is remplaced by $\langle (e\ 0), \ldots, (e\ (p-1)) \rangle$ where the length of the sub-term list is equal to p. In the rules, we will identify terms modulo renaming of bound variables and we will use Barendregt's variable convention: if terms $t_1, \ldots, t_n$ occur in a certain context then in these terms all bound variables are chosen to be different from free variables. The rules of this calculus is applicable in context:

1. $(\lambda \dot{x}.e)e' \rightarrow e[\dot{x} \leftarrow e']$
2. $(\lambda \bar{x}.E)\ E' \rightarrow E[\bar{x} \leftarrow E']$
3. $(\lambda \dot{x}.E)\ e' \rightarrow E[\dot{x} \leftarrow e']$
4. $\langle t_0, \ldots, t_{p-1} \rangle \# \langle u_0, \ldots, u_{p-1} \rangle \rightarrow \langle t_0\ u_0, \ldots t_{p-1}\ u_{p-1} \rangle$
5. $\langle t_0, \ldots, t_{p-1} \rangle ? \langle n_0, \ldots, n_{p-1} \rangle \rightarrow \langle t_{n_0}, \ldots t_{n_{p-1}} \rangle$
6. $(\langle t_0, \ldots, \overbrace{S}^{n}, \ldots, t_{p-1} \rangle \xrightarrow{n} E_1, E_2) \rightarrow T$

The global condition is defined by two rules where n belongs to $\mathcal{N}$ and T is E_1 (respectively E_2) when S is **true** (respectively **false**). Those two rules are necessary to express algorithms of the form:

Repeat
Parallel Iteration
Until Max of local errors $< \epsilon$

Because without them, the global control cannot take into account data computed locally, i.e. global control cannot depend on data. For the exchange of data, the BSλ_p has another instruction that could replace the *get instruction:* put noted ! that has the following rule:

$$! \langle f_0, \ldots, f_{p-1} \rangle \rightarrow \langle \ldots, \underbrace{\lambda j.((j = 0 \rightarrow (f_0\ i), (j = 1 \rightarrow (f_1\ i), (\ldots, \mathbf{nc}))\ldots))}_{i}, \ldots \rangle$$

where **nc** is the non-communication constant. With all these formal definitions, we could describe the library that has been implemented.

3.3 The BSMLlib library

BSML is a data parallel funtional language for programming BSP algorithms. BSMLlib is based on the following elements ([Lou02]). It is without the pid variable of SPMD programs, but uses an externally-bound variable `bsp_p:unit->int` so that the value of `bsp_p()` is p, the static number of processes. The value of this variable does not change during execution. There is also a polymorphic type constructor `par` such that `'a par` represents the type of p-wide vectors of objects of type `'a`, one per process. The nesting of `par` types is prohibited. Our type system enforces this restriction. This improves on the earlier design DPML/Caml Flight [HF93] in which the global parallel control structure `sync` had to be prevented *dynamically* from nesting.

Parallel objects are created by

```
mkpar:  (int -> 'a) -> 'a par
```

so that `(mkpar f)` stores `(f i)` on process i for $i = 0, 1, \ldots, (p-1)$.

A BSP algorithm is expressed as a combination of asynchronous local computations and phases of global communication with global synchronization. Readers familiar with BSPlib will observe that we ignore the distinction between a communication request and its realization at the barrier. Asynchronous phases are programmed with

```
apply:  ('a -> 'b) par -> 'a par -> 'b par
```

whose semantics is that of a map over the parallel structures. In other words `apply (mkpar f) (mkpar e)` stores `(f i) (e i)` on process i. Neither the implementation of `BSMLlib`, nor its semantics [Lou01] prescribes a synchronization barrier between two successive uses of `apply`.

The communication and synchronization phases are expressed by

```
put:  (int -> 'a option) par -> (int -> 'a option) par
```

where `'a option` is defined by: `type 'a option = None|Some of 'a`.

Consider the expression:

$$\texttt{put(mkpar(fun i->fs}_i\texttt{))} \tag{3.1}$$

To send a value `v` from process `j` to process `i`, the function $\texttt{fs}_j$ at process `j` must be so that ($\texttt{fs}_j$ `i`) evaluates to `Some v`. To send no value from process `j` to process `i`, ($\texttt{fs}_j$ `i`) must evaluate to `None`.

Expression (3.1) evaluates to a parallel vector containing a function $\texttt{fd}_i$ of delivered messages on every process. At process `i`, ($\texttt{fd}_i$ `j`) evaluates to `None` if process `j` sent no message to process `i` or evaluates to `Some v` if process `j` sent the value `v` to the process `i`.

The full language would also contain a synchronous conditional operation

```
ifat:  (bool par) * int * 'a * 'a -> 'a
```

such that `ifat (v,i,v1,v2)` will evaluate to `v1` or `v2` depending on the value of `v` at process `i` . But Objective Caml is an eager language and this synchronous conditional operation cannot be defined as a function. That is why the core `BSMLlib` contains the function: `at:bool par -> int -> bool` to be used only in the contruction: `if (at vec pid) then... else...` where `(vec:bool par)` and `(pid:int)`.

The meaning of `if (at vec pid) then expr1 else expr2` is that of `ifat(vec,pid,expr1,expr2)`.

We give some examples of BSML programs we'll use in following sections.

```
let replicate x = mkpar (fun pid->x)
```

Create a vector whom all components hold the same value.

```
let parfun f v = apply (replicate f) v
```

Apply a sequential function to all components of a vector.

```
let totex vec =
  parfun (fun f src -> match (f src) with Some x->x)
    (put (parfun (fun data dst->Some data) vec))
```

Total exchange.

```
let gather root vv =
  let mkmsg pid v dest = if dest=root then Some v else None in
    put(apply (mkpar mkmsg) vv)
```

gathers the values of parallel vector vv at process n as a function.

```
let gather_list root vv =
  let procs_at_root = mkpar (function root->procs() | _ ->[]) in
  let to_list = parfun (compose noSome) (gather root vv) in
    parfun2 List.map to_list procs_at_root
```

gathers the values of parallel vector vv at process n as a list.

Chapter 4

BSλ-calculi with Pattern Matching

4.1 Overview

4.1.1 Current State of the `BSMLlib` library

In the Current BSMLlib library, pattern matching is possible because it is possible in Objective Caml. Thus one can write

```
let rec length l = match l with
   | [] -> 0
   | h::t -> 1 + length t
```

In this case, the matching of a parallel value (l is a parallel value since it is a list of parallel vectors) is possible because the patterns match only the structure of the list (wich is the same at each process) and could be applied to any kind of list.

We can also use pattern matching on a parallel value in applying a usual Objective Caml pattern at each process :

$$\texttt{parfun length (mkpar(fun i->l}_\texttt{i}\texttt{))}$$

In this case, the matching cannot change the global behavior of the program because the matching is done on usual sequential values, even if those values are inside a parallel vector.

When one writes program using the BSMLlib library, one would like to be allowed to write pattern matching of parallel values in the two following cases.

First the matching of parallel vectors at the global level, with the possibility to have patterns which depend on the internal structure of parallel values. Take as an exemple, a usual Objective Caml pattern P. $\langle P \rangle$ would be a pattern which matches a parallel vector whose components are sequential values which may be matched by P. Of course the right side of a $\langle P \rangle \rightarrow E$ must be in this case a parallel expression, otherwhise the nesting of parallel vectors would be allowed.

Another interesting pattern matching of parallel values would be the global matching of of parallel vectors where the patterns may be different on each processor.

We first try to design an extension of the BSMLlib with the $\langle P \rangle$ pattern construct as syntactic sugar. As seen in the previous exemple, the local matching of the internal structure is not enough because it doesn't change the global behavior of a program, so if one considers the following exemple :

```
type number = F of float | I of int
let v1 = mkpar(fun pid->if (pid/2) = 0
            then I pid
            else F (float_of_int pid))
let f1 e = match e with
  <I i> -> <i>
| <F f> -> replicate 0
```

The local matching of the patterns $\langle P \rangle$ would lead to a different evaluation of the program at each process, some of them would match their value with the pattern `I i` where as some other would match their value with the pattern `F f` what is certainly not the expected behavior. With a correct implementation, none of the pattern would match `v1`, thus communication are needed to determine whether a $\langle P \rangle$ pattern matches an expression or not. Each process must check locally the pattern matching and then say to a chosen process (for example process 0) if the pattern matches the expression or not. Then a `if at` construction is used to change the global behavior. For exemple

```
let f2 e = match e with
   | <Some v> -> (<v>, <v>)
```

is syntactic sugar for the BSMLlib program:

```
let f2' e =
 let local_match e =
  match e with |Some v -> true |_ -> false
 and extract_v e =
  match e with |Some v -> v in
 let vl = gl 0 (parfun local_match e) in
 let vb = parfun (reduce (&&)) vl in
 if at vb 0
 then let vv = parfun extract_v e in (vv,vv)
 else raise Parallel_Match_failure
```

where `gather_list` has type `int -> 'a par -> 'a list par` and `gather_list l i vv` gathers the values of parallel vector `vv` at process `n` and whose

semantics is given by :

$$\texttt{gather\_list n}\ \langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle\ =\ \langle\ []\ ,\ldots,\ \underbrace{[\texttt{v}_0;\ldots;\texttt{v}_{p-1}]}_{n}\ ,\ldots,\ []\ \rangle$$

and $\texttt{reduce}\ \oplus\ [\texttt{v}_1;\ldots;\texttt{v}_n]\ =\ [\texttt{v}_1;\texttt{v}_1\oplus\texttt{v}_2;\ldots;\oplus_{0<k\leq n}\texttt{v}_k]$.

In the sequential case, each pattern is tried until one pattern matches the given expression. If we take the same strategy in the parallel case, each try costs two BSP super-steps which is too expensive. It is possible to reduce this cost to one super-step but in this case the program is not well typed: it is possible implement the new `match with` construct as a *primitive* but it is no more syntactic sugar.

Another possibility is to first check locally all the patterns then exchange the booleans which indicate whether a pattern matches the expression or not and determine the first j such as the j^{th} pattern matches the expression on all processes. For example:

```
let f3 e = match e with
  | <Some v> -> <v>
  | <None>   -> (replicate 0)
```

is syntactic sugar for the program given in figure 4.1.

```
let f3' e =
 let local_checkP1 e = match e with | Some v -> true | _ -> false
 and local_checkP2 e = match e with | None -> true | _ -> false in
 let apply_check e = map (fun f->f e) [local_checkP1;local_checkP2] in
 let l1 =parfun apply_check e in  let l2 = total_exchange l1 in
 let vj = let rec aux j l =
  if j=bsp_p() then raise Parallel_Match_failure
  else let h,l' = split(map (fun (h::l)->h,l) l) in
       let b = reduce (&&) h in
       if b then j else aux (j+1) l' in
  parfun (aux 0) l2 in
 if at (parfun (fun x ->x=0) vj) 0 then
   let extract_v e = match e with | Some v -> v in
       let vv=parfun extract_v e in vv
 else if at (parfun (fun x ->x=1) vj) 0 then replicate 0
 else raise Parallel_Match_failure
```

where

$$\begin{cases} \texttt{map f}\ [\texttt{e}_1;\ldots;\texttt{e}_n] & = & [\texttt{f e}_1;\ldots;\texttt{f e}_n] \\ \texttt{split}\ [(\texttt{e}_1,\texttt{e}'_1);\ldots;(\texttt{e}_n,\texttt{e}'_n)] & = & ([\texttt{e}_1;\ldots;\texttt{e}_n],[\texttt{e}'_1;\ldots;\texttt{e}'_n]) \\ \texttt{total\_exchange}\ \langle\ e_0\ ,\ldots,\ e_{p-1}\ \rangle & = & \langle\ [e_0;\ldots;e_{p-1}]\ ,\ldots,\ [e_0;\ldots;e_{p-1}]\ \rangle \end{cases}$$

Figure 4.1: `f3'` program

This program is again not very efficient since we must use as many `if at` as patterns. Thus for n patterns, the cost would be in the worst case, the cost of the total exchange $n \times (p-1) \times g + L$ plus the cost of each `if at`: $n \times ((p-1) \times g + L)$.

4.1.2 Extension with the `match at with` Primitive

To improve the efficiency of a program such as the last one of the previous section, we need to make the number of super-steps independent on the number of patterns. For that, we need to introduce a new primitive in the language: a `match e at n with` construction whose parallel cost would be the same as a `if at`. The end of the `f3'` function would then became:

```
match vj at 0 with
 | 0 -> let extract_v e =
        match e with | Some v -> v in
     let vv = parfun extract_v e in vv
 | 1 -> replicate 0
 | _ -> raise Parallel_Match_failure
```

and cost would be: $n \times (p-1) \times g + L + (p-1) \times g + L$. This new construction is a straightforward generalization of the `if at` construct and it will be included in the next release of the `BSMLlib` library [**?**].

It can be make even more efficient but no more as syntactic sugar. One can notice that the j value is a parallel vector which contains the same value everywhere. Thus as in the implementation of the `BSMLlib` [Lou02] the type `' a par` is defined as `'a`, in the implementation of the core library, the following program, which uses the usual pattern matching of Objective Caml, would be correct (but it is an incorrect user program, because it would not be well typed):

```
match vj with
 | 0 -> let extract_v e =
        match e with | Some v -> v in
     let vv = parfun extract_v e in (vv)
 | 1 -> replicate 0
 | _ -> raise Parallel_Match_failure
```

The communication and synchronization cost of this last version would be $n \times (p-1) \times g + L$.

If we add the `match at with` construction, the `if at` is not needed any more since a program such as :

```
if E at n then E1 else E2
```

can now be written :

```
match E at n with
  | true -> E1
  | false -> E2
```

The expressivity of the language is slightly augmented since a program such as

```
match vv at n with
  x -> f x
```

where f is a function from local to global terms, cannot be written with the `if at` construct since the right part of the match case depends on a component of the parallel vector vv and would request a `if at` for each possible value.

In conclusion of this section, the matching of parallel values using the new parallel pattern $\langle P \rangle$ where P is a usual Objective Caml pattern must be implemented as a *primitive* of the `BSMLlib` library in order to attain reasonable performance. Designing it as syntactic sugar on top on the current `BSMLlib` library would lead to poor performances.

4.2 BSλ_p-calculus with global matching

In this section, we introduce a BSλ_p-calculus (where p is the processes number) extended by pattern-matching facilities, particularly the pattern-matching of parallel values. The BSλ_p-calculus syntax is extended by a set of global patterns allowing the matching of the internal structure of parallel values. In 4.2.1, we introduce the syntax and an informal description of its dynamic semantics. In 4.2.2, we present the static semantics of this calculus and,in 4.2.3, its formal dynamic semantics. As usual, $\mathcal{N}$ denotes the set of processes names.

4.2.1 Syntax

Let $x, y, z, \ldots$ range over a set V of variables. We define, respectively p and e, the sets of patterns and terms of our calculus, v the set of values. Let c ranges over a set of constants, which may be integers or booleans. Finally E ranges over a set of constructors which are supposed, without loss of generality, of arity 1. A constructor of arity 0 may be seen as a constructor of arity 1 whom argument is (), the only element of a type `unit`.

The by case function definition is introduced by the mean of two kinds of abstraction. The simple abstraction, in which the usual variable that bounds occurrences under the λ is replaced by a pattern p that bounds variables occurring in p in the expression under the λ. In a simple abstraction the term between brackets denotes a guard which must evaluates to $\mathbf{T}$ to permit reduction.

The β-reduction uses the external `match` operator to unify the argument and the λ-abstraction's pattern. If the `match` operator returns a substitution σ for pattern p

$$
\begin{array}{rcl}
p & ::= & x \mid \_ \mid c \mid (p,\ p) \mid E\ p \mid p@e \mid \langle p \rangle \\
a & ::= & \lambda p[e].e \mid (\lambda p[e].e)|a) \\
e & ::= & x \mid c \mid (e,\ e) \mid E\ e \mid a \mid e\ e \\
 & \mid & \pi\ e \mid \overbrace{\langle e,\ e,\ \ldots,\ e \rangle}^{p} \mid e \,\#\, e \mid e \,?\, e \\
v & ::= & c \mid (v, v) \mid E\ v \mid a \mid \overbrace{\langle v,\ v,\ \ldots,\ v \rangle}^{p}
\end{array}
$$

Figure 4.2: Syntax

and term e, then the application of a simple abstraction $\lambda p[e_1].e_0$ to a term e which evaluates to v, evaluates to $e_0\sigma$. If the `match` operator return $\bot_{\mathrm{m}}$ which denotes match failure then the application does not reduce.

This first kind of abstraction permits the matching of arguments, we also introduce compound abstractions to make effectively possible by case definition. A compound abstraction is defined recursively as a couple $(\lambda p[e_1].e_0|\ a_2)$ where $\lambda p[e_1].e_0$ is a simple abstraction and a_2 may be a simple abstraction as well as a compound abstraction. The meaning of this construction is that when it is applied to a term e reducing to v it uses the `match` operator to unify p and v, as for a simple abstraction. It denotes the same value if the matching returns a substitution and denotes $a_2\ v$ otherwise.

We do not address, in this calculus, the problem of matching failure because we will, as in ML-like language, use exceptions to recover this kind of run-time error. When it is not possible to match a pattern with a value, then the term simply does not reduces any more.

In the BSλ-calculus, the `if at` construction is used to take into account data computed locally in the global control. This construction is restricted to boolean values. The new construction $p@e$ which is the main addition to the BSλ-calculus introduced here, formalizes the `match par` construction presented in the previous section and permits to take into account data computed locally more efficiently. As we have seen in the previous section, this construction could be had as syntactic sugar but performances achieved in that case would not be satisfying. Since this construction is used for global control, the result of such a matching is available on each process as a (distributed) local term, but the typing rules must ensured that it is used only within a global term to avoid global terms nesting.

We give the following simple example

$$(\lambda x@0.\pi(\lambda y.x))(\pi(\lambda x.x)) \quad \text{must evaluates to} \quad \langle 0,\ 0,\ \ldots,\ 0 \rangle$$

The pattern _ is the pattern which matches any value as in Objective Caml. Patterns

must be linear, i.e each variable occurring in a pattern p must occur at most once in p and the nesting of delocalized patterns is prohibited.

We also add pattern of the form $\langle p \rangle$ that matches a parallel vector whom all components match the pattern P. For example :

$$(\lambda\langle(x,y)\rangle.\langle x\rangle)\langle(v_0,w_0),\ldots,(v_{p-1},w_{p-1})\rangle \quad \text{must evaluates to} \quad \langle v_0,\ldots,v_{p-1}\rangle$$

In BSλ-calculus, as in λ-calculus, only one variable is bind by an abstraction ($\lambda x.e$, binds x in e). With the introduction of patterns, the abstraction is generalized, an expression such as $\lambda p[e_1].e_0$ defines bindings whose scope is e_1 and e_0 for all the variables occurring in p . Thus for a pattern p, we have to define the set of bounded variables (fig 4.3), the usual definition of free variables occurring in a term (fig 4.4) must use this generalization of bindings.

$$\begin{array}{rcl} Var(x) & = & \{x\} \\ Var(\_) & = & \emptyset \\ Var(c) & = & \emptyset \\ Var((p_1,\ p_2)) & = & Var(p_1) \cup Var(p_2) \end{array} \qquad \begin{array}{rcl} Var(E\ p) & = & Var(p) \\ Var(p@e) & = & Var(p) \\ Var(\langle p\rangle) & = & \{\langle x\rangle / x \in Var(p)\} \end{array}$$

Figure 4.3: Bounded variables

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(\langle x\rangle) & = & \{\langle x\rangle\} \\ FV(c) & = & \emptyset \\ FV((e_1,e_2)) & = & FV(e_1) \cup FV(e_2) \\ FV(E\ e) & = & FV(e) \\ FV(\lambda p[e_1].e_0) & = & (FV(e_1) \cup FV(e_0)) \backslash Var(p) \\ FV((\lambda p[e_1].e_0|a)) & = & FV(\lambda p[e_1].e_0) \cup FV(a) \\ FV(\pi e) & = & FV(e) \\ FV(e_1\ \#\ e_2) & = & FV(e_1) \cup FV(e_2) \\ FV(e_1\ ?\ e_2) & = & FV(e_1) \cup FV(e_2) \end{array}$$

Figure 4.4: Free variables

The substitution of a variable x by a term e in a term is defined by induction on the term structure in fig 4.5:

4.2.2 Static Semantics

As we mentioned in previous sections, it is important to avoid global terms nesting as well as delocalized patterns nesting. These properties are ensured by the type system introduced in this section. A relation on types is used to avoid such nesting

$$
\begin{array}{rcl}
x\{e/x\} & = & e \\
y\{e/x\} & = & y \text{ if } y \neq x \\
\langle x\rangle\{e/\langle x\rangle\} & = & e \\
\langle y\rangle\{e/\langle x\rangle\} & = & y \text{ if } y \neq x \\
x\{e/\langle y\rangle\} & = & x \\
\langle x\rangle\{e/y\} & = & \langle x\rangle \\
c\{e/x\} & = & c \\
(e_1, e_2)\{e/x\} & = & (e_1\{e/x\}, e_2\{e/x\}) \\
(E\ e_1)\{e/x\} & = & E\ e_1\{e/x\}
\end{array}
$$

$$
(\lambda p[e_1].e_0)\{e/x\} = \begin{cases} \lambda p[e_1\{e/x\}].e_0\{e/x\} \text{ if } x \notin Var(p) \\ (\lambda p[e_1].e_0) \text{ otherwise} \end{cases}
$$

$$
\begin{array}{rcl}
(\lambda p[e_1].e_0|a)\{e/x\} & = & ((\lambda p[e_1].e_0)\{e/x\}|a\{e/x\}) \\
(e_1\ e_2)\{e/x\} & = & e_1\{e/x\}\ e_2\{e/x\} \\
(\pi e_1)\{e/x\} & = & \pi e_1\{e/x\} \\
(e_1 \# e_2)\{e/x\} & = & e_1\{e/x\} \# e_2\{e/x\} \\
(e_1 \,?\, e_2)\{e/x\} & = & e_1\{e/x\} \,?\, e_2\{e/x\}
\end{array}
$$

Figure 4.5: Substitution

by restricting the use of typing rules. We define types τ of our calculus and the relation $\mathcal{R}$ on these types by

$$\tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \tau \to \tau \mid \tau * \tau \mid \tau\ \texttt{par}$$

$$
\begin{array}{c}
\texttt{int}\ \mathcal{R}\ \tau \\
\texttt{bool}\ \mathcal{R}\ \tau \\
\texttt{unit}\ \mathcal{R}\ \tau \\
\tau_1 * \tau_2\ \mathcal{R}\ \tau \text{ iff } \tau_1\ \mathcal{R}\ \tau \text{ and } \tau_2\ \mathcal{R}\ \tau \\
\tau\ \mathcal{R}\ \tau'\ \texttt{par}
\end{array}
$$

where τ, τ', τ_1 and τ_2 are non functional types and

$$
\begin{array}{c}
\tau\ \mathcal{R}\ \tau_1 \to \tau_2 \text{ iff } \tau\ \mathcal{R}\ \tau_2 \\
\tau_1 \to \tau_2\ \mathcal{R}\ \tau \text{ iff } \tau_2\ \mathcal{R}\ \tau
\end{array}
$$

A type τ is said to be local, noted $\tau \in \mathcal{L}$ if τ is `int`,`bool`, `unit` or if $\tau\ \mathcal{R}\ \tau'$ for $\tau' \in \mathcal{L}$. For each constant c, $\mathcal{T}(c)$ is the type of c.

$$
\begin{array}{rclrcl}
\mathcal{T}(n) & = & \texttt{int} \text{ if } n \text{ is an integer} & \mathcal{T}(\mathbf{F}) & = & \texttt{bool} \\
\mathcal{T}(\mathbf{T}) & = & \texttt{bool} & \mathcal{T}(()) & = & \texttt{unit}
\end{array}
$$

$$(1)\frac{}{x : \tau \vdash x : \tau} \quad (2)\frac{\tau \in \mathcal{L}}{\langle x \rangle : \tau \ \mathtt{par} \vdash \langle x \rangle : \tau \ \mathtt{par}} \quad (3)\frac{}{\vdash c : \mathcal{T}(c)}$$

$$(4)\frac{\Gamma_{\neg x} \vdash e : \tau_2}{\Gamma_{\neg x},\ x : \tau_1 \vdash e : \tau_2} \qquad (5)\frac{\Gamma \vdash e : \tau_2}{\Gamma,\ \_ : \tau_1 \vdash e : \tau_2} \qquad (6)\frac{\Gamma \vdash e : \tau}{\Gamma,\ c : \mathcal{T}(c) \vdash e : \tau}$$

$$(7)\frac{\Gamma_{\neg\langle x \rangle} \vdash e : \tau_2 \quad \tau_1 \in \mathcal{L}}{\Gamma_{\neg\langle x \rangle},\ \langle x \rangle : \tau_1 \ \mathtt{par} \vdash e : \tau_2} \quad (8)\frac{\Gamma \vdash e : \tau_2 \quad \tau_1 \in \mathcal{L}}{\Gamma,\ \langle \_ \rangle : \tau_1 \ \mathtt{par} \vdash e : \tau_2} \quad (9)\frac{\Gamma \vdash e : \tau_2}{\Gamma,\ \langle c \rangle : \mathcal{T}(C) \ \mathtt{par} \vdash e : \tau_2}$$

$$(10)\frac{\Gamma,\ p : \tau_1 \vdash e1 : \tau_2 \quad \Gamma,\ \tau_1 \vdash e_1 : \mathtt{bool} \quad \tau_1 \ \mathcal{R} \ \tau_2}{\Gamma \vdash \lambda p : A[e_1].e_0 : (\tau_1 \to \tau_2)}$$

$$(11)\frac{\Gamma \vdash \lambda p[e_1].e_0 : \tau \quad \Gamma \vdash a : \tau}{\Gamma \vdash (\lambda p[e_1].e_0 | a) : \tau}$$

$$(12)\frac{\Gamma \vdash e1 : (\tau_1 \to \tau_2) \quad \Gamma \vdash e2 : \tau_1}{\Gamma \vdash e1\ e2 : \tau_2}$$

$$(13)\frac{\Gamma \vdash e1 : \tau_1 \quad \Gamma \vdash e2 : \tau_2}{\Gamma \vdash (e1,\ e2) : \tau_1 * \tau_2} \qquad (14)\frac{\Gamma,\ p : \tau_1,\ q : \tau_2 \vdash e : \tau_3}{\Gamma,\ (p,\ q) : (\tau_1 * \tau_2) \vdash e : \tau_3}$$

$$(15)\frac{\Gamma \vdash e : \tau \quad \tau = T(E)}{\Gamma \vdash E\ e : \mathcal{T}(E)} \qquad (16)\frac{\Gamma,\ p : \tau_1 \vdash e : \tau_2 \quad \tau_1 = \mathcal{T}(E)}{\Gamma,\ E\ p : \mathcal{T}(E) \vdash e : \tau_1}$$

$$(17)\frac{\Gamma,\ \langle p \rangle : \tau_1 \ \mathtt{par},\ \langle q \rangle : \tau_2 \ \mathtt{par} \vdash e : \tau}{\Gamma,\ \langle (p,\ q) \rangle : (\tau_1 * \tau_2) \ \mathtt{par} \vdash e : \tau} \quad (18)\frac{\Gamma,\ \langle p \rangle : \tau_1 \ \mathtt{par} \vdash e : \tau_2 \quad \tau = \mathcal{T}(E)}{\Gamma,\ \langle E\ p \rangle : \mathcal{T}(E) \ \mathtt{par} \vdash e : \tau_2}$$

$$(19)\frac{\Gamma,\ p : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash n : \mathtt{int}}{\Gamma,\ p@n : \tau_1 \ \mathtt{par} \vdash e : \tau_2}$$

$$(20)\frac{\Gamma \vdash e : (int \to \tau) \quad \tau \in \mathcal{L}}{\Gamma \vdash \pi\ e : \tau \ \mathtt{par}} \qquad (21)\frac{\forall i \in \mathcal{N}.\Gamma \vdash e_i : \tau}{\Gamma \vdash \langle v_0,\ v_1,\ \ldots,\ v_{p-1} \rangle : \tau \ \mathtt{par}}$$

$$(22)\frac{\Gamma \vdash e1 : \tau_1 \ \mathtt{par} \quad \Gamma \vdash e2 : \mathtt{int\ par}}{\Gamma \vdash e1\,?\,e2 : \tau_1 \ \mathtt{par}}$$

$$(23)\frac{\Gamma \vdash e1 : (\tau_1 \to \tau_2) \ \mathtt{par} \quad \Gamma \vdash e2 : \tau_1 \ \mathtt{par}}{\Gamma \vdash e_1 \,\#\, e_2 : \tau_2 \ \mathtt{par}}$$

Figure 4.6: Static Semantics

For each constructor E, $\mathcal{T}(E)$ is the type t if E is a constructor of type $\tau' \to t$ and the type τ' is used in the relation $\mathcal{R}$ in place of t.

A typing judgment has the form $\Gamma \vdash e : \tau$ where Γ, called a pattern assignment, is a set of elements of the form $p : B$. A pattern assignment Γ is said to be linear if for all $p : B \in \Gamma$, p is linear. We write $Var(\Gamma)$ to denote the set $\bigcup_{p:\tau \in \Gamma} Var(p)$.

As we have mentioned, we use some conditions in typing rules to avoid the nesting of global terms. In rule (2) and (7), the construction $\langle x \rangle$ has type τ par where τ must be a local type. The rule (10) permits no to have function which maps data of non local types to data of local types. This restriction is needed because the application of such a function to a non local argument would lead to a local term which would hide some global constructions. Finally, the most obvious restriction is in rule (20) and is use to avoid direct global term nesting. It can be seen in [KPT96] that the introduction of patterns does not remove decidability properties of type systems since a pattern may be decomposed in a set of variable for which a derivation exists iff it exists for the pattern. of patterns

4.2.3 Dynamic Semantics

We first present the semantic of the external operator match which is used for the unification of a pattern and a value in applications.

The rules (m_1) to (m_8) have the same behavior as in the sequential case and don't need further explanations. The rule (m_9) is used for the matching of a given component of a parallel value with a local pattern and thus simply returns the usual matching for this case as a (distributed) local term. The rule (m_{10}) brings the local structure of a vector to the global level in the case of uniform matching of a parallel value with a local pattern. As mentioned in the condition, the matching must be possible on each component, for example

$$\frac{match((1,2),(x,y)) \to_m \{1/x, 1/y\} \quad match((3,4),(x,y)) \to_m \{3/x, 4/y\}}{match(\langle(1,2),(3,4)\rangle, \langle(x,y)\rangle) \to_m \{\langle 1,3\rangle/x, \langle 2,4\rangle/y\}}$$

If one process fails to match its value, then the rule (m_{11}) is used.

$$\frac{match((3,4),(1,y)) \to_m \bot_m}{match(\langle(1,2),(3,4)\rangle, \langle(1,y)\rangle) \to_m \bot_m}$$

We define the relation (p, v, α) such as (p, v, α) iff $\texttt{match}(p, v) \to_m \alpha$.

Proposition 1 *If $\Gamma, p : A \vdash e : B$ and $\Gamma \vdash v : A$ and if (p, v, α) and (p, v, α') then $\alpha = \alpha'$, where α is a substitution or $\bot_m$.*

Proposition 2 *If $e \to \alpha$ and $e \to \alpha'$ then $\alpha = \alpha'$ where α is a value or $\bot$.*

As we have seen previously, when pattern matching fails, the "calling" term simply does not reduce. There is also an other case for which a reduction could not

$$\gamma_{\alpha\beta} = \begin{cases} \sigma_1 \cup \sigma_2 \text{ if } \alpha = \sigma_1,\ \beta = \sigma_2 \\ \bot_{\mathrm{m}} \text{ if } \alpha \text{ or } \beta = \bot_{\mathrm{m}} \end{cases} \qquad \alpha' \in \{\sigma, fail\}$$

$$(m_1)\frac{}{\mathbf{match}(x,\ v) \to_m \{(v,\ x)\}} \quad (m_2)\frac{}{\mathbf{match}(\langle x\rangle,\ v) \to_m \{(v,\ \langle x\rangle)\}} \quad (m_3)\frac{}{\mathbf{match}(\_,\ v) \to_m \emptyset}$$

$$(m_4)\frac{}{\mathbf{match}(c,\ c) \to_m \emptyset} \quad (m_5)\frac{c \neq v}{\mathbf{match}(c,\ v) \to_m \bot_m}$$

$$(m_6)\frac{\mathbf{match}(p_1,\ v_1) \to_m \alpha \quad \mathbf{match}(p_2,\ v_2) \to_m \beta}{\mathbf{match}((p_1,\ p_2),(v_1,\ v_2)) \to_m \gamma_{\alpha\beta}}$$

$$(m_7)\frac{\mathbf{match}(p,\ v) \to_m \alpha'}{\mathbf{match}(E\ p,\ E\ v) \to_m \alpha'} \quad (m_8)\frac{v \neq Cv'}{\mathbf{match}(E\ p,\ v) \to_m \bot_m}$$

$$(m_9)\frac{\mathbf{match}(p,\ v_j) \to_m \alpha'}{\mathbf{match}(\langle p@j\rangle,\ \langle v_0, v_1, \ldots v_{p-1}\rangle) \to_m \alpha'}$$

$$(m_{10})\frac{\forall i \in \mathcal{N}.\mathbf{match}(p,\ v_i) \to_m \sigma_p^i}{\mathbf{match}(\langle p\rangle,\ \langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle) \to_m \sigma)} \quad \sigma(\langle x\rangle) = \langle \sigma_p^0(x),\ \sigma_p^1(x),\ \ldots,\ \sigma_p^{p-1}(x)\rangle$$

$$(m_{11})\frac{\exists i \in \mathcal{N}.\mathbf{match}(p,\ v_i) \to_m \bot_m}{\mathbf{match}(\langle p\rangle,\ \langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle) \to_m \bot_m}$$

Figure 4.7: Semantics (1)

$$(24)\frac{e_2 \to v \quad \mathbf{match}(p,\ v) \to \sigma \quad e_1\sigma \to \mathbf{T} \quad e_0\sigma \to v_0}{(\lambda p[e_1].e_0)e_2 \to v_0}$$

$$(25)\frac{e_2 \to v \quad \mathbf{match}(p,\ v) \to \sigma \quad e_1\sigma \to \mathbf{T} \quad e_0\sigma \to v_0}{(\lambda p[e_1].e_0|a)e_2 \to v_0}$$

$$(26)\frac{e_2 \to v \quad \mathbf{match}(p,\ v) \to \sigma \quad e_1\sigma \to \mathbf{F} \quad ae_2 \to v_0}{(\lambda p[e_1].e_0|a)e_2 \to v_0}$$

$$(27)\frac{e_2 \to v \quad \mathbf{match}(p,\ v) \to \bot_{\mathrm{m}}}{(\lambda p[e_1].e_0|a)e_2 \to ae_2}$$

$$(28)\frac{}{v \to v}$$

$$(29)\frac{e \to v}{E\ e \to E\ v} \quad (30)\frac{e_1 \to v_1 \quad e_2 \to v_2}{(e_1, e_2) \to (v_1, v_2)} \quad (31)\frac{\forall i \in \mathcal{N}.(e\ i) \to w_i}{\pi\ e \to \langle w_0, w_1, \ldots, w_{p-1}\rangle}$$

$$(32)\frac{e_1 \to \langle f_0,\ f_1,\ \ldots,\ f_{p-1}\rangle \quad e_1 \to \langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle \quad \forall i \in \mathcal{N}.f_i v_i \to w_i}{e_1\ \#\ e_2 \to \langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle}$$

$$(33)\frac{e_1 \to \langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle \quad e_2 \to \langle m_0,\ m_1,\ \ldots,\ m_{p-1}\rangle \quad \forall i \in \mathcal{N}.m_i \in \mathcal{N}}{e_1\ ?\ e_2 \to e_1 \to \langle v_{m_0},\ v_{m_1},\ \ldots,\ v_{m_{p-1}}\rangle}$$

Figure 4.8: Semantics (2)

apply, if one of values denoting source processes in a communication (?) are not all a process name. These two issues will be addressed when we will introduce exceptions handling in the BSλ_p-calculus. For now we just define cases for which we know a reduction may be blocked and we check that for all other cases a well-typed term reduces to a value.

Definition 1 *An expression e is said to be blocking if*

- $e = (\lambda p[e_1].e_0)\ e'$, $e' \to v$ *and*
  $\texttt{match}(p,\ v) = \bot_m$
  or $\texttt{match}(p,\ v) = \sigma$ *and* $e_1\sigma$ *admits a blocking sub-term*
  or $\texttt{match}(p,\ v) = \sigma$ *and* $e_1\sigma \to \mathbf{F}$
  or $\texttt{match}(p,\ v) = \sigma,\ e_1\sigma \to \mathbf{T}$ *and* $e_0\sigma$ *admits a blocking sub-term*
- $e = (\lambda p[e_1].e_0)|a)\ e$, $e' \to v$ *and* $(\lambda p[e_1].e_0)\ e$ *and* $a\ e$ *are blocking.*
- $e = \pi\ e'$, $e' \to v$ *and exists* i *such that* $v\ i$ *is blocking.*
- $e = e_1\ ?\ e_2$, $e_1 \to v$, $e_2 \to \langle m_0,\ m_1,\ \ldots, m_{p-1}\rangle$ *where* $\exists i.m_i \notin \mathcal{N}$.
- $e = e_1 \# e_2$, $e_1 \to \langle f_0,\ f_1,\ \ldots, f_{p-1}\rangle$, $e_2 \to \langle v_0,\ v_1,\ \ldots, v_{p-1}\rangle$ *and exists* i *such that* $f_i\ v_i$ *is blocking.*

Lemma 1 *Rules (4) to (9), (14) and (16) to (19) commute with rules (10) to (13), (15) and (20) to (22).*

Proposition 3 *If* $\Gamma \vdash e : \tau$, *where* e *is a closed term, then* $\exists v.e \to v$ *or* e *admits a blocking sub-term.*

proof, proposition 1 : By case on the structure of p.

- If $(x,\ v,\ \alpha)$ and $(x,\ v,\ \alpha')$ then $\alpha = \{(v,\ x)\} = \alpha'$
- If $(\_,\ v,\ \alpha)$ and $(x,\ v,\ \alpha')$ then $\alpha = \emptyset = \alpha'$
- If $(c,\ v,\ \alpha)$ and $(c,\ v,\ \alpha')$ then $\begin{cases} \text{If } v = c,\ \alpha = \emptyset = \alpha' \\ \text{If } v \neq c,\ \alpha = \bot_m = \alpha' \end{cases}$
- If $((p_1,\ p_2),\ v,\ \alpha)$ and $((p_1,\ p_2),\ v,\ \alpha')$ then $v = (v_1,\ v_2)$ and $\alpha = \gamma_{\beta_1,\,\beta_2}$, $\alpha' = \gamma_{\delta_1,\,\delta_2}$ with $(p_1,\ v_1,\ \beta_1)$,$(p_1,\ v_1,\ \delta_1)$, $(p_2,\ v_2,\ \beta_2)$, $(p_2,\ v_2,\ \delta_2)$, by I.H. $\beta_1 = \delta_1$ and $\beta_2 = \delta_2$.
  where $\gamma_{\alpha\beta} = \begin{cases} \sigma_1 \cup \sigma_2 \text{ if } \alpha = \sigma_1,\ \beta = \sigma_2 \\ \bot\text{m} \text{ if } \alpha \text{ or } \beta = \bot\text{m} \end{cases}$

- If $(E\ p,\ v,\ \alpha)$ and $(E\ p,\ v,\ \alpha')$ then $\begin{cases} \text{If } v = E\ v' \text{ then } (p,\ v,\ \alpha) \text{ and } (p,\ v,\ \alpha') \\ \text{and by induction hypothesis, } \alpha = \alpha' \\ \\ \text{If } (v \neq E\ v') \text{ then } \alpha = \bot_m = \alpha' \end{cases}$

- If $(p@j,\ v,\ \alpha)$ and $(p@j,\ v,\ \alpha')$, we have $v = \langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle$ and $(p,\ v_j,\ \alpha_j)$ and $(p,\ v_j,\ \alpha'_j)$ and by induction hypothesis, $\alpha_j = \alpha'_j$ and then $\alpha = \alpha'$.

- If $(\langle p\rangle,\ v,\ \alpha)$ and $(\langle p\rangle,\ v,\ \alpha')$ then $v = \langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle$. We distinguish two cases

  – If $\forall i \in \mathcal{N}.(p,\ v_i,\ \sigma_i) \wedge (p,\ v_i,\ \sigma'_i)$, then by I.H.,

$$\begin{array}{rrcl} \forall i \in \mathcal{N}.\sigma_i & = & \sigma'_i \\ & \text{and} & \\ \forall x \in Var(p). \quad \sigma(x) & = & \langle \sigma_0(x),\ \sigma_1(x),\ \ldots,\ \sigma_{p-1}(x)\rangle \\ & = & \langle \sigma_0(x),\ \sigma_1(x),\ \ldots,\ \sigma_{p-1}(x)\rangle \\ & = & \sigma'(x). \end{array}$$

  – If $\exists i \in \mathcal{N}.(p,\ v_i,\ \alpha_i) \wedge (p,\ v_i,\ \alpha'_i)$ and $\alpha_i = \bot_m$ (resp. $\alpha'_i = \bot_m$) then by I.H. $\alpha'_i = \bot_m$ (resp $\alpha_i = \bot_m$) and $\alpha = \bot_m = \alpha'$.

proof, proposition 2 : Trivial with proposition 1.

proof, lemma 1: We give only some cases, other are similar.

rules (5) and (10).

$$\dfrac{\dfrac{\begin{array}{ccc} D & & D' \\ \vdots & & \vdots \\ \Gamma,\ p : \tau_1 \vdash e_0 : \tau_2 & & \Gamma,\ p : \tau_1 \vdash e_1 : \texttt{bool} \end{array} \quad \tau_1\ \mathcal{R}\ \tau_2}{\Gamma \vdash \lambda x[e_1].e_0 : \tau_1 \to \tau_2}}{\Gamma,\ x : \tau \vdash \lambda x[e_1].e_0 : \tau_1 \to \tau_2}$$

$$\dfrac{\begin{array}{ccc} D & & D' \\ \vdots & & \vdots \\ \dfrac{\Gamma,\ p : \tau_1 \vdash e_0 : \tau_2}{\Gamma,\ x : \tau,\ p : \tau_1 \vdash e_0 : \tau_2} & & \dfrac{\Gamma,\ p : \tau_1 \vdash e_1 : \texttt{bool}}{\Gamma,\ x : \tau,\ p : \tau_1 \vdash e_1 : \texttt{bool}} \end{array} \quad \tau_1\ \mathcal{R}\ \tau_2}{\Gamma,\ x : \tau \vdash \lambda x[e_1].e_0 : \tau_1 \to \tau_2}$$

rules (19) and (11)

$$\dfrac{\dfrac{\begin{array}{ccc} D & & D' \\ \vdots & & \vdots \\ \Gamma,\ p : \tau \vdash \lambda p : \tau_1[e_1].e_0 : \tau' & & \Gamma,\ p : \tau \vdash a : \tau' \end{array}}{\Gamma,\ p : \tau \vdash (\lambda p : \tau_1[e_1].e_0 | a) : \tau'} \quad \begin{array}{c} D'' \\ \vdots \\ \Gamma \vdash n : \texttt{int} \end{array}}{\Gamma,\ p@n : \tau\ \texttt{par} \vdash (\lambda q : \tau_1[e_1].e_0 | a) : \tau'}$$

$$\dfrac{\dfrac{\begin{array}{c}D\\ \vdots\\ \Gamma,\ p:\tau \vdash \lambda p:\tau_1[e_1].e_0:\tau'\end{array} \quad \begin{array}{c}D''\\ \vdots\\ \Gamma \vdash n:\texttt{int}\end{array}}{\Gamma,\ p@n:\tau\ \texttt{par} \vdash \lambda p:\tau_1[e_1].e_0:\tau'} \quad \dfrac{\begin{array}{c}D'\\ \vdots\\ \Gamma,\ p:\tau \vdash a:\tau'\end{array} \quad \begin{array}{c}D''\\ \vdots\\ \Gamma \vdash n:\texttt{int}\end{array}}{\Gamma,\ p@n:\tau\ \texttt{par} \vdash a:\tau'}}{\Gamma,\ p@n:\tau\ \texttt{par} \vdash (\lambda q:\tau_1[e_1].e_0|a):\tau'}$$

proof, proposition 3: By lemma 1, we can consider derivations where the last rule is a right introduction rule.

- $\Gamma \vdash \lambda p[e_1].e_0 : \tau_1 \to \tau_2$, $\lambda p[e_1].e_0$ is a value.
- $\Gamma \vdash (\lambda p[e_1].e_0|a) : \tau$, $(\lambda p[e_1].e_0|a)$ is a value.
- $\Gamma \vdash e_1\ e_2 : \tau_2$, then we have $\Gamma \vdash e_1 : \tau_1 \to \tau_2$ $\Gamma \vdash e_2 : \tau_2$. By I.H, e_1 and e_2 reduce to values or admit a blocking sub-term.

  If e_1 or e_2 admit a blocking sub-term then e_1 or e_2 admits a blocking sub-term of $e_1\ e_2$.

  If e_1 and e_2 reduces to values, then e_1 reduce to a simple or a compound abstraction since $\Gamma \vdash e_1 : \tau_1 \to \tau_2$. e_2 reduces to a value v.

  - e_1 reduce to a simple abstraction $\lambda p[e_1].e_0$

    If $\begin{cases}\texttt{match}(\texttt{p},\texttt{v}) = \bot_m,\\ \texttt{match}(\texttt{p},\texttt{v}) = \sigma \text{ and } e_1\sigma \text{ admits a blocking sub-term}\\ \texttt{match}(\texttt{p},\texttt{v}) = \sigma \text{ and } e_1\sigma \to \mathbf{F}\\ \texttt{match}(\texttt{p},\texttt{v}) = \sigma,\ e_1\sigma \to \mathbf{T} \text{ and } e_0\sigma \text{ admits a blocking sub-term}\end{cases}$
    then $(e_1\ e_2)$ admits a blocking sub-term.
    If $\texttt{match}(\texttt{p},\texttt{v}) = \sigma$ and $e_1\sigma \to \mathbf{T}$ then $e_1\ e_2$ reduces to $e_0\sigma$.
- $\Gamma \vdash (e_1,\ e_2) : \tau_1 * \tau_2$, then $\Gamma \vdash e_1 : \tau_1$ and $\Gamma \vdash e_2 : \tau_2$. By I.H. e_1 and e_2 reduces to values or admit a blocking sub-term.

  If e_1 and e_2 reduces to values, then $(e_1,\ e_2)$ reduce to a value, otherwise $(e_1,\ e_2)$ admits a blocking sub-term.
- $\Gamma \vdash E\ e : \mathcal{T}(E)$, then $\Gamma \vdash e : \mathcal{T}(E)$. By I.H. e reduces to value or admits a blocking sub-term and so $E\ e$ does.
- $\Gamma \vdash \pi\ e : \tau\ \texttt{par}$, then $\Gamma \vdash e : \texttt{int} \to \tau$. By I.H. e reduces to a value or admits a blocking sub-term.

  If e reduce to a value, then if $\exists i \in \mathcal{N}.e\ i$ is blocking $\pi\ e$ is blocking. otherwise $e\ i$ reduce to values and $\pi\ e$ reduces to a value.

- $\Gamma \vdash e_1 \ ?\ e_2 : \tau$ `par`, then $\Gamma \vdash e_1 : \tau$ `par` and $\Gamma \vdash e_2$: `int par`. By I.H. e_1 and e_2 reduces to values or admit a blocking sub-term.

  If e_1 or e_2 admit a blocking sub-term then $e_1 \ ?\ e_2$ admits a blocking sub-term.

  If e_1 and e_2 reduce to values, then $e_1 = \langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle$ and $e_1 = \langle m_0,\ m_1,\ \ldots,\ m_{p-1}\rangle$.

  If $\exists i \in \mathcal{N}.m_i \notin \mathcal{N}$, then $e_1?e_2$ is blocking otherwise $e_1?e_2$ reduces to $\langle v_{m_0},\ v_{m_1},\ \ldots,\ v_{m_{p-1}}\rangle$.

- $\Gamma \vdash e_1 \# e_2 : \tau_2$, then $\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2$ and $\Gamma \vdash e_2 : \tau_2$.

  By I.H., e_1 and e_2 reduce to values or admit a blocking sub-term.

  If e_1 or e_2 admit a blocking sub-term, then $e_1 \# e_2$ admits a blocking sub-term.

  If e_1 and e_2 reduce to values, then $e_1 = \langle f_0,\ f_1,\ \ldots,\ f_{p-1}\rangle$ and $e_1 = \langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle$.

  if exists i such that $f_i or e_i$ is blocking, then $e_1 \# e_2$ is blocking, otherwise $f_i\ v_i$ reduces to a value w_i for all i in $\mathcal{N}$ and $e_1 \# e_2$ reduce to $\langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle$.

- $\Gamma \vdash \langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle$, $\langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle$ is a value.

Chapter 5

BSλ-calculi with Exceptions Handling

5.1 Introduction

Exceptions handling in the framework of the BSML language raises some difficulties. The actual implementation, the BSMLlib, is based on the Objective Caml and so it is possible to use its exception mechanism. However this is not well suited to the data-parallel framework of BSML and may lead to several run-time errors. The evaluation of a BSML program, according to the data-parallel model, consists in the evaluation of the same sequential (a global term) on data depending of process number (local terms within parallel vectors). In this kind of execution model, there is some collective operation (each process must call them) the processes use to communicate. In languages, such as MPI, the notion of collective operation poses problems. Indeed, if one of the processes involved in the program evaluation doesn't realize the call to such an operation then all other processes are locked. Yet, in such languages, the user has explicitly access to process name and then is able to work out the behavior of a copy of the program at a specific process. It is then possible that one of processes doesn't participate to a call of a collective operation. In the BSML language, the notion of parallel vector, which are vectors of p value (one by processes) is introduced. There is no more explicit access to processes names, each component of the vector is a term depending on the process number (a function which maps processes names to values) and there is no possibility to use the process name outside of a vector. At the program level (local terms) there is so no possibility to work out a specific behavior at a process. Since parallel vectors cannot be nested, there is no possibility for process not to realize a call to one of the collective operations which are all operations on parallel vectors. However, it is possible to work out the global behavior of a program according to the value of a vector at a specific process but in that case all processes have the same behavior.

The use of Objective Caml exceptions in BSML is not safe because the previous property is not assured any more when one introduces them. If an exception is

raised outside a parallel value, then we are sure that all processes have raised it.At the opposite, if an exception is raised within a parallel vector then it is possible that only a subset of processes have raised it (the evaluation in a parallel value may depend on the process number) and then one may ask what happens in that case. If we look at the implementation of the current BSMLlib, a Parallel vector at a process is a value nested in the concret type `par` defined in Objective Caml by

$$type\ 'a\ par = Par\ of\ 'a$$

If we follow the Objective Caml semantics, then an exception raised within a parallel vectors propagates out from its "structure" until it meets a handler. In that case it is now possible that a call for a collective operation to be dismissed during the propagation and a dead-lock may then occur. Look at the following exemple

```
p ::= get (mkpar(fun x-> if x = 0 then
                         raise (Exception) else x))
          (mkpar(fun x->1))
```

and its evaluation at process 0 and 1

$$\begin{array}{rcl} p & \Rightarrow_0 & \texttt{get}\ (\texttt{raise}\ (Exception))\ (\texttt{mkpar}(\texttt{fun}\ x \rightarrow 1)) \\ & \Rightarrow_0 & \texttt{raise}\ (Exception) \end{array}$$

$$\begin{array}{rcl} p & \Rightarrow_1 & \texttt{get}\ (Par\ 1)\ (\texttt{mkpar}(\texttt{fun}\ x \rightarrow 1)) \\ & \Rightarrow_1 & \texttt{get}\ (Par\ 1)\ (Par\ 1) \end{array}$$

The evaluation is blocked at process one while at process 0 it has terminated.

Thus, it is necessary to assure new safety properties in BSML programs evaluation while introducing exceptions. We have to avoid that exceptions raised locally may propagate outside vectors structure without global control, particularly we have to avoid that collective operation may be dismissed. It is also necessary to distinguished two kind of exceptions according to the type of data they care, otherwhise the nesting of parallel vector would be allowed what is strictly forbidden in BSML semantics. In next sections, we propose a formal semantics which is well-suited for BSML programs evaluation in the framework of exception mechanism. Intuitively, a process that raises locally an exception must let it propagates locally until it meets a synchronisation (collective) operation and then it waits the end of the current BSP-superstep to transmit the exception at the global level so that processes may recover all together. But is is not enought to ensure safety, some problems may also occur if some processes raise an exception out of a parallel vector while some process are not "actives", for exemple the synchronisation operation waited for the wake-up of inactive processes could be dismissed and then such an exception must be allowed to propagate throught collective operation only if all processes are actives. Since BSML is a flat language, we don't want to introduce a priority on processes names

and then it is not possible to distinguish a particular exception between all exceptions raised during a superstep. Thus we introduce the notion of set of exceptions, when some processes raise some local exceptions during a superstep, then the set of all locally raised exception is transmitted at the global level. We introduce a new `try`-like construction which permits to catch such sets of exceptions and to recover according to the whole set of locally raised exception or only to a subset of it.

5.2 Syntax

Let $x,\ y,\ z, \ldots$ range over a set V of variables. We define a finite set $E = \{E_1,\ E_2,\ \ldots,\ E_n,\ Not\_Raised\}$ of constructors which we can be supposed of arity 1 without loss of generality. The **BSλ_p^{exn}-expressions** e, where p is the processes number, are defined by

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda x.e \mid e\ e \mid () \\
& \mid & \pi\ e \mid \overbrace{\langle e,\ e,\ \ldots,\ e\rangle}^{p} \mid e \# e \mid e\ ?\ e \\
& \mid & E\ e \mid [E_{k_1}(e,e),\ E_{k_2}(e,e),\ \ldots,\ E_{k_n}(e,e)] \mid \texttt{raise}\ e \\
& \mid & \texttt{try}\ e\ \texttt{with}\ E\ x.e \\
& \mid & \texttt{try}\ e\ \texttt{with}\ E\ x\ y.e\ \texttt{on other}\ z.e\ \texttt{otherwise}\ e
\end{array}
$$

Figure 5.1: Syntax

A **locally raised exceptions context** (LREC) is an object of the form $[\Omega_1, \Omega_2, \ldots, \Omega_n]$ where $\Omega_1 = \omega_0,\ \omega_1,\ \ldots,\ \omega_{k_1}$ and $\Omega_{i+1} = \omega_{k_i+1}, \omega_{k_i+2}, \ldots, \omega_{k_{(i+1)}}$ with $k_n = p - 1$ and ω_m is either a BSλ_p^{exn}-expression either the symbol $\top$.

We use the following shorthands :

- $\Omega^\top = \omega_k,\ \omega_{k+1},\ \ldots,\ \omega_{k+m}$, where $\omega_{k+i} = \top$ for all $i \in \{0, \ldots, m\}$.
- $\Omega^\bot = \omega_k,\ \omega_{k+1},\ \ldots,\ \omega_{k+m}$, where there is at least one $i \in \{0, \ldots, m\}$ such as ω_{k+i} is an expression.
- $[\Omega]_i = \omega_i$ if $\Omega = [\omega_0, \omega_1, \ldots, \omega_i, \ldots, \omega_{p-1}]$.

A **BSλ_p^{exn} term** is an object of the form $e\ [\Omega_1, \Omega_2, \ldots, \Omega_n]$ where e is a BSλ_p^{exn}-expression and $[\Omega_1, \Omega_2, \ldots, \Omega_n]$ is a locally raised exceptions context.

The calculus introduced in this section, respects the BSP spirit. In BSP, data exchanged between processes are achievable in the BSP superstep following the call to the communication operation. It then seems legitime that an exception raised locally by a process (and not caught by this process during a superstep) be recoverable only at the next superstep (in the sense that this exception handling mechanism doesn't add any synchronisation operation). If one process raises locally an exception during a BSP superstep, it has to be freezed until the end of this superstep while keeping

a indication about the raised exception. This job is done by the locally raised exception context which keeps "in memory" exceptions raised during a BSP-superstep. Intuitively, a locally raised exceptions context $[\Omega_1, \Omega_2, \ldots, \Omega_n]$ represents the state of processes. $[\Omega_1, \Omega_2, \ldots, \Omega_n]_i{=}\top$ means that the evaluation is going on at process i where as $[\Omega_1, \Omega_2, \ldots, \Omega_n]_i = e$, means that the process i has locally raised an exception and is waiting for the end of the current superstep to let the other processes (and itself) recover globally this exception. We'll say that a process i is **active** in a locally raised exceptions context $[\Omega]$ if $[\Omega]_i = \top$.

The construction $[E_{k_1}(e, e),\ E_{k_2}(e, e),\ \ldots,\ E_{k_n}(e, e)]$ denotes a set of exceptions locally raised on processes during a BSP superstep. An exception is said to be **locally raised** if it has been raised within a parallel value, i.e. locally by one or more processes. It is said to be **globally raised** otherwhise, i.e. if it has been raised by all (actives) processes. At the end of a BSP superstep, all exceptions locally raised during this superstep must be transmitted to the global level by the mean of such a set of exceptions.

For each locally raised exceptions context $[\Omega]$, we define the expression $\Delta_\Omega = [E_{k_1}(f_1, g_1),\ E_{k_2}(f_2, g_2),\ \ldots,\ E_{k_n}(f_n, e_n)]$ where $E_{k_i} \in \Delta_\Omega$ iff $E_{k_i} \in \Omega$. f_i and g_i are terms denoting functions which maps the process name i, respectively to e_i and `true` if $[\Omega]_i = E_{k_i} e_i$, `raise` Not_Raised and `false` otherwhise. An expression Δ_Ω is build during evaluation, and is not supposed to be written directly by the user. In this section, we will always suppose that an expression of this form is well typed and compatible with the current locally raised exceptions context.

The meaning of the construction

$$\texttt{try}\ e\ \texttt{with}\ E\ x.e$$

is the same as usual, i.e. that of the handler of the exception constructor E while the meaning of the construction

$$\texttt{try}\ e\ \texttt{with}\ E\ x\ y.e\ \texttt{on other}\ z.e\ \texttt{otherwhise}\ e$$

is that of the handler of a set of locally raised exceptions. Others constructions have the same meaning as in the original BSλ_p-calculus.

We define respectively $FV(e)$, in fig 5.2, the free variables of the expression e and $e[e'/x]$, in fig 5.3, the substition of the variable x by the expression e' in e. We define the **compatibility** relation $\mathcal{C}$ on expression and locally raised exceptions context in fig 5.4. This relation is used to check that a process evaluates local terms iff it is active.

$$
\begin{array}{lcllcl}
FV(x) & = & x & FV(e_1 \# e_2) & = & FV(e_1) \cup FV(e_2) \\
FV(\lambda x.e) & = & FV(e) \backslash \{x\} & FV(e_1 e ?_2) & = & FV(e_1) \cup FV(e_2) \\
FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) & FV(E\ e) & = & FV(e) \\
FV(\pi\ e) & = & FV(e) & FV(\texttt{raise}\ e) & = & FV(e) \\
FV(\langle e_1, e_2, \ldots, e_{p-1} \rangle) & = & \emptyset & FV([S]) & = & \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
FV(e_1 \rightarrow^{e_2} e3,\ e_4) & = & FV(e_1) \cup FV(e_2) \cup FV(e_3) \cup FV(e_4) \\
FV(\texttt{try}\ e_1\ \texttt{with}\ E\ x.e_2) & = & FV(e_1) \cup (FV(e_2) \backslash \{x\})
\end{array}
$$

$$
\begin{array}{lcl}
FV(\texttt{try}\ e_1\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3) & = & \begin{array}{l} FV(e_1) \\ \cup (FV(e_2) \backslash \{x, y, z\}) \\ \cup (FV(e_3) \backslash \{x, y\}) \end{array}
\end{array}
$$

Figure 5.2: Free variables

$$
\begin{array}{lcllcl}
x[e'/x] & = & e' & y[e'/x] & = & y \text{ if } y \neq x \\
(\lambda x.e)[e'/x] & = & \lambda x.e & \lambda y.e[e'/x] & = & \lambda y.e[e'/x] \text{ if } y \neq x \\
(e_1\ e_2)[e'/x] & = & e_1[e'/x]\ e_2[e'x] & (\ )[e'/x] & = & (\ ) \\
(\pi\ e)[e'/x] & = & \pi\ e[e'/x] & \langle e_0, e_1, \ldots, e_{p-1} \rangle [e'/x] & = & \langle e_0, e_1, \ldots, e_{p-1} \rangle \\
(e_1 \# e_2)[e'/x] & = & e_1[e'/x] \# e_2[e'x] & (e_1\ ?\ e_2)[e'/x] & = & e_1[e'/x] e ?_2 [e'/x] \\
(E\ e)[e'/x] & = & E\ e[e'/x] & \texttt{raise}\ e[e'/x] & = & \texttt{raise}\ e[e'/x]
\end{array}
$$

$$
\begin{array}{lcl}
(e_1 \rightarrow^{e_2} e_3, e_4)[e'/x] & = & (e_1[e'/x] \rightarrow^{e_2[e'/x]} e_3[e'/x],\ e_4[e'/x]) \\
(\texttt{try}\ e_1\ \texttt{with}\ E\ x.e_2)[e'/x] & = & (\texttt{try}\ e_1[e'/x]\ \texttt{with}\ E\ x.e_2) \\
(\texttt{try}\ e_1\ \texttt{with}\ E\ y.e_2)[e'/x] & = & (\texttt{try}\ e_1[e'/x]\ \texttt{with}\ E\ y.e_2[e'/x]) \\
 & & \text{if } y \neq x
\end{array}
$$

$$
\begin{array}{l}
(\texttt{try}\ e_1\ \texttt{with}\ E\ y\ z\ \texttt{on other}\ t.e_2\ \texttt{otherwhise}\ e_3)[e'/x] = \\
\texttt{try}\ e_1[e'/x]\ \texttt{with}\ E\ y\ z\ \texttt{on other}\ t.e_2[e'/x]\ \texttt{otherwhise}\ e_3[e'/x] \quad \text{if } x \notin \{y, z, t\}
\end{array}
$$

$$
\begin{array}{l}
(\texttt{try}\ e_1\ \texttt{with}\ E\ y\ z\ \texttt{on other}\ t.e_2\ \texttt{otherwhise}\ e_3)[e'/x] = \\
\texttt{try}\ e_1[e'/x]\ \texttt{with}\ E\ y\ z\ \texttt{on other}\ t.e_2\ \texttt{otherwhise}\ e_3[e'/x] \qquad \text{if } x = z
\end{array}
$$

$$
\begin{array}{l}
(\texttt{try}\ e_1\ \texttt{with}\ E\ y\ z\ \texttt{on other}\ t.e_2\ \texttt{otherwhise}\ e_3)[e'/x] = \\
\texttt{try}\ e_1[e'/x]\ \texttt{with}\ E\ y\ z\ \texttt{on other}\ t.e_2\ \texttt{otherwhise}\ e_3 \qquad \text{if } x \in \{x, y\}
\end{array}
$$

Figure 5.3: Substitution

- $x\ \mathcal{C}\ [\Omega]$
- $\lambda x.e\ \mathcal{C}\ [\Omega]$ if for all e' such as $e'\ \mathcal{C}\ [\Omega]$ we have $e[e'/x]\ \mathcal{C}\ [\Omega]$
- $(e_1\ e_2)\ \mathcal{C}\ [\Omega] \Leftrightarrow e_1\ \mathcal{C}\ [\Omega]$ *and* $e_2\ \mathcal{C}\ [\Omega]$
- $(\pi\ e)\ \mathcal{C}\ [\Omega] \Leftrightarrow e\ \mathcal{C}\ [\Omega]$
- $\langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle\ \mathcal{C}\ [\Omega] \Leftrightarrow$
  - $[\Omega]_i \neq \bot \Rightarrow w_i$ is a value or $w_i = \bot$.
  - $w_i = \bot \Rightarrow [\Omega]_i \neq \top$
- $E\ e\ \mathcal{C}\ [\Omega] \Leftrightarrow e\ \mathcal{C}\ [\Omega]$
- $[S]\ \mathcal{C}\ [\Omega]$
- `raise` $e\ \mathcal{C}\ [\Omega] \Leftrightarrow e\ \mathcal{C}\ [\Omega]$
- `try` e_1 `with` $Ex.e_2\ \mathcal{C}\ [\Omega] \Leftrightarrow$ $e_1\ \mathcal{C}\ [\Omega]$ *for all* e *such as* $e\ \mathcal{C}\ [\Omega]$ *we have* $e_2[e/x]\ \mathcal{C}\ [\Omega]$
- `try` e `with` $E\ x\ y$ `on other` e_2 `otherwhise` $e_3 \mathcal{C}[\Omega] \Leftrightarrow$ $e_1\mathcal{C}[\Omega]$ *for all* f, g, h *compatible with* $[\Omega]$ $e_2[f/x,\ g/y, h/z]\ \mathcal{C}\ [\Omega]$ $e_3[f/x,\ g/y]\ \mathcal{C}\ [\Omega]$

Figure 5.4: Compatibility

5.3 Static Semantics

The type system must reject global terms nesting. Thus we introduce a relation $\mathcal{R}$ on types which is used to restrict the use of typing rules. We define types τ of the BSλ_p^{exn}-calculus by

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \\ & \mid & exn_L \mid exn_G \mid Mexn \\ & \mid & \tau\ \texttt{par} \\ & \mid & \tau \rightarrow \tau \end{array}$$

and the relation $\mathcal{R}$ on types by

$$\begin{array}{rcl} \texttt{int} & \mathcal{R} & \tau \\ \texttt{bool} & \mathcal{R} & \tau \\ \texttt{unit} & \mathcal{R} & \tau \\ \tau_1 & \mathcal{R} & \tau_2\ par \end{array} \qquad \begin{array}{rcl} Mexn & \mathcal{R} & \tau \\ exn_L & \mathcal{R} & \tau \\ \tau_1 & \mathcal{R} & exn_G \end{array}$$

where τ_1 is a non functional type and

$$\begin{array}{rcl} \tau\ \mathcal{R}\ \tau_1 \rightarrow \tau_2 & \Leftrightarrow & \tau\ \mathcal{R}\ \tau_2 \\ \tau_1 \rightarrow \tau_2\ \mathcal{R}\ \tau & \Leftrightarrow & \tau_1\ \mathcal{R} \end{array}$$

To each constructor E we associate $\begin{cases} T(E) \in \{exn_L, exn_G\} \\ Arg(E) = \tau,\ \tau\ \mathcal{R}\ T(E) \end{cases}$

We use Arg(E) in the relation $\mathcal{R}$.

A type τ is said to be local, noted $\tau \in \mathcal{L}$, if τ in `int`, `bool`, `unit` or if $\tau \mathcal{R} \tau'$ for $\tau' \in \mathcal{L}$.

A typing judgment has the form $\Gamma \vdash e : \tau$ where, Γ ,called a pattern assignment, is a set of elements of the form $p : B$. A pattern assignement Γ is said to be linear, if for all $p : B \in \Gamma$, p is linear.

5.4 Dynamic Semantics

In this section, we introduce a small-step call-by-value semantics for our calculus. Firstly, we introduce the reduction rules corresponding to usual BSλ_p terms. In each rule, $[\Omega]$ is supposed to be compatible with the left member. One can see that if a process is active in the left member of these rules then it is active in the right member. This property will allow us to prove that our semantics is conservative w.r.t the BSλ_p semantics.

We now give the semantics of globally raised exceptions handlers. The construction

$$\texttt{try}\ e\ \texttt{with}\ E\ x.e$$

$$\frac{}{x_1 : \tau_1,\ x_2 : \tau_2,\ \ldots,\ x_n : \tau_n \vdash x_i : \tau_i} \quad \frac{}{\Gamma \vdash () : \texttt{unit}} \quad \frac{}{\Gamma \vdash E : Arg(E) \to T(E)}$$

$$\frac{\Gamma,\ x : \tau_1 \vdash e : \tau_2 \quad \tau_1\ \mathcal{R}\ \tau_2}{\Gamma \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash e_1\ e_2 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \texttt{int} \to \tau \quad \tau \in \mathcal{L}}{\Gamma \vdash \pi\ e : \tau\ \texttt{par}} \quad \frac{\forall i \in \mathcal{N}\ .\ \Gamma \vdash e_i : \tau}{\Gamma \vdash \langle e_0,\ e_1,\ \ldots,\ e_{p-1} \rangle : \tau\ \texttt{par}}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1 \to \tau_2)\ \texttt{par} \quad \Gamma \vdash e_2 : \tau_1\ \texttt{par}}{\Gamma \vdash e_1 \,\#\, e_2 : \tau_2\ par} \quad \frac{\Gamma \vdash e_1 : \tau\ \texttt{par} \quad \Gamma \vdash e_2 : \texttt{int par}}{\Gamma \vdash e_1 e\,?_2 : \tau\ \texttt{par}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool par} \quad \Gamma \vdash e_2 : \texttt{bool} \quad \Gamma \vdash e_3 : \tau \quad \Gamma \vdash e_4 : \tau}{\Gamma \vdash (e_1 \to^{e_2} e_3,\ e_4) : \tau}$$

$$\frac{\Gamma \vdash e : exn_L}{\Gamma \vdash \texttt{raise}\ e : \tau} \quad \frac{\Gamma \vdash e : exn_G \quad exn_G\ \mathcal{R}\ \tau}{\Gamma \vdash \texttt{raise}\ e : \tau} \quad \frac{\Gamma \vdash e : Mexn}{\Gamma \vdash \texttt{raise}\ e : \tau}$$

$$\frac{\Gamma \vdash E : \tau_1 \to \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash E\ e : \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma,\ x : Arg(E) \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{with}\ E\ x.e_2 : \tau}$$

$$\frac{\forall i \in \{1,\ldots,n\}.\Gamma \vdash e_i : \texttt{int} \to Arg(E_i) \quad \forall i \in \{1,\ldots,n\}.\Gamma \vdash e'_i : \texttt{int} \to \texttt{bool}}{\Gamma \vdash [E_{k_1}(e_1, e'_1), E_{k_2}(e_2, e'_2), \ldots, E_{k_n}(e_n, e'_n)] : Mexn}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma' \vdash e_2 : \tau \quad \Gamma' \vdash e_3 : \tau \quad \Gamma' = \Gamma,\ x : \texttt{int} \to Arg(E),\ y : \texttt{int} \to \texttt{bool}}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3 : \tau}$$

Figure 5.5: Static Semantics

$$
\begin{array}{lcll}
(\lambda x.e)v\ [\Omega] & \rightarrow^{\epsilon} & e[v/x]\ [\Omega] & \\
(\pi\ v)[\Omega] & \rightarrow^{\epsilon} & \langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle[\Omega] & \\
 & & w_i = \begin{cases} (v\ i)\ \mathit{if}\ [\Omega]_i = \top \\ \bot\ \texttt{otherwhise} \end{cases} & \\
\langle f_0, f_1, \ldots, f_{p-1}\rangle\ \#\ \langle e_0, e_1, \ldots, e_{p-1}\rangle\ [\Omega] & \rightarrow^{\epsilon} & \langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle\ [\Omega] & \\
 & & w_i = \begin{cases} (f_i\ e_i)\ \mathit{if}\ [\Omega]_i = \top \\ \bot\ \texttt{otherwhise} \end{cases} & \\
\langle v_0, v_1, \ldots, v_{p-1}\rangle\ ?\ \langle m_0, m_1, \ldots, m_{p-1}\rangle\ [\Omega^{\top}] & \rightarrow^{\epsilon} & \langle v_{m_0}, v_{m_1}, \ldots, v_{m_{p-1}}\rangle\ [\Omega^{\top}] & (?_{\top}) \\
\langle v_0, v_1, \ldots, v_{p-1}\rangle\ ?\ \langle m_0, m_1, \ldots, m_{p-1}\rangle\ [\Omega^{\bot}] & \rightarrow^{\epsilon} & \texttt{raise}\ \Delta_{\Omega^{\bot}}\ [\Omega^{\top}] & (?_{\bot}) \\
\langle v_0, v_1, \ldots, S, \ldots, v_{p-1}\rangle \rightarrow^{n} e_1, e_1\ [\Omega^{\top}] & \rightarrow^{\epsilon} & T\ [\Omega^{\top}] & (\rightarrow^{n}_{\top}) \\
 & & \mathit{where}\ T = \begin{cases} e_1\ \mathit{if}\ S = \texttt{true} \\ e_2\ \mathit{if}\ S = \texttt{false} \end{cases} & \\
\langle v_0, v_1, \ldots, S, \ldots, v_{p-1}\rangle \rightarrow^{n} e_1, e_1\ [\Omega^{\bot}] & \rightarrow^{\epsilon} & \texttt{raise}\ \Delta_{\Omega^{\bot}}\ [\Omega^{\top}] & (\rightarrow^{n}_{\bot})
\end{array}
$$

$$
\frac{e\ [\Omega] \rightarrow^{\epsilon} e'\ [\Omega]}{\langle e_0,\ e_1,\ \ldots, e, e_{p-1}\rangle\ [\Omega] \rightarrow^{\epsilon} \langle e_0,\ e_1,\ \ldots, e, e_{p-1}\rangle\ [\Omega]}
$$

has the same behaviour as usual, except that now it must let throw up sets of local exceptions. Its semantics is given by the following rules :

$$
\begin{array}{lcl}
\texttt{try}\ v\ \texttt{with}\ E\ x.e[\Omega] & \rightarrow^{\epsilon} & v[\Omega] \\
\texttt{try raise}\ E\ v\ \texttt{with}\ E\ x.e\ [\Omega] & \rightarrow^{\epsilon} & e[v/x]\ [\Omega]
\end{array}
$$

We also introduce the one-step reduction rules for locally raised exceptions handlers.

$$
\begin{array}{l}
\texttt{try}\ v\ \texttt{with} \\
\quad E\ x\ y\ \texttt{on other}\ z.e_1\ \texttt{otherwhise}\ e_2\ [\Omega] \rightarrow^{\epsilon} v\ [\Omega]
\end{array}
$$

$$
\begin{array}{l}
\texttt{try raise}\ [\mathcal{S}, E(f,g), \mathcal{S}']\ \texttt{with} \\
\quad E\ x\ y\ \texttt{on other}\ z.e_1\ \texttt{otherwhise}\ e_2\ [\Omega] \rightarrow^{\epsilon} e_1[f/x,\ g/y,\ [\mathcal{S}, \mathcal{S}']/z]\ [\Omega]
\end{array}
$$

$$
\begin{array}{l}
\texttt{try raise}\ [E(f,g)]\ \texttt{with} \\
\quad E\ x\ y\ \texttt{on other}\ z.e_1\ \texttt{otherwhise}\ e_2\ [\Omega] \rightarrow^{\epsilon} e_2[f/x,\ g/y]\ [\Omega]
\end{array}
$$

A raised exception must propagate throught terms but all terms do not have the same behavior.

- terms with no synchronisation simply let propagate exceptions as usual.
- the behavior of terms with synchronisation depends on the locally raised exceptions context. If no local exception has been raised then the raised exception propagates. If some local exceptions have been raised then they are transmit to the global level and the exception raised globally is thrown.

- when a component of a parallel vector evaluates to a raised exception then it must transmit it to the locally raised context, waiting for the end of the current BSP-superstep.

$$
\begin{array}{rcl}
(\texttt{raise}\ v)\ e_2\ [\Omega] & \rightarrow^{\epsilon} & \texttt{raise}\ v\ [\Omega] \\
v\ (\texttt{raise}\ v)\ [\Omega] & \rightarrow^{\epsilon} & \texttt{raise}\ v\ [\Omega] \\
(\texttt{raise}\ v)\ \#\ e_2\ [\Omega] & \rightarrow^{\epsilon} & \texttt{raise}\ v\ [\Omega] \\
v\ \#\ (\texttt{raise}\ v)\ [\Omega] & \rightarrow^{\epsilon} & \texttt{raise}\ v\ [\Omega] \\
E(\texttt{raise}\ v)\ [\Omega] & \rightarrow^{\epsilon} & \texttt{raise}\ v\ [\Omega] \\
\texttt{raise}\ (\texttt{raise}\ \texttt{v})\ [\Omega] & \rightarrow^{\epsilon} & \texttt{raise}\ v\ [\Omega]
\end{array}
$$

$$
\begin{array}{rcl}
(\texttt{raise}\ v)\ ?\ e_2\ [\Omega^{\top}] & \rightarrow^{\epsilon} & \texttt{raise}\ v[\Omega^{\top}] \\
(\texttt{raise}\ v)\ ?\ e_2\ [\Omega^{\bot}] & \rightarrow^{\epsilon} & \texttt{raise}\ \Delta_{\Omega^{\bot}}\ [\Omega^{\top}] \\
v\ ?\ (\texttt{raise}\ v)\ [\Omega^{\top}] & \rightarrow^{\epsilon} & \texttt{raise}\ v\ [\Omega^{\top}] \\
v\ ?\ (\texttt{raise}\ v)\ [\Omega^{\bot}] & \rightarrow^{\epsilon} & \texttt{raise}\ \Delta_{\Omega^{\bot}}\ [\Omega^{\top}] \\
(\texttt{raise}\ v \rightarrow^{e_2} e_3,\ e4))\ [\Omega^{\top}] & \rightarrow^{\epsilon} & \texttt{raise}\ v\ [\Omega^{\top}] \\
(e_1 \rightarrow^{\texttt{raise}\ v} e_3,\ e4))\ [\Omega^{\top}] & \rightarrow^{\epsilon} & \texttt{raise}\ v\ [\Omega^{\top}] \\
(\texttt{raise}\ v \rightarrow^{e_2} e_3,\ e4))\ [\Omega^{\bot}] & \rightarrow^{\epsilon} & \texttt{raise}\ \Delta_{\Omega^{\bot}}\ [\Omega^{\top}] \\
(e_1 \rightarrow^{\texttt{raise}\ v} e_3,\ e4))\ [\Omega^{\bot}] & \rightarrow^{\epsilon} & \texttt{raise}\ \Delta_{\Omega^{\bot}}\ [\Omega^{\top}] \\
\texttt{try raise}\ E'\ v\ \texttt{with}\ E\ x.e\ [\Omega] & \rightarrow^{\epsilon} & \texttt{raise}\ E'\ v\ [\Omega] \\
\texttt{try raise}\ [S]\ \texttt{with}\ E\ x.e\ [\Omega] & \rightarrow^{\epsilon} & \texttt{raise}\ [S]\ [\Omega] \\
\texttt{try raise}\ [S^{\neg E}]\ \texttt{with} & & \\
E\ x\ y\ \texttt{on other}\ z.e_1\ \texttt{otherwhise}\ e_2\ [\Omega] & \rightarrow^{\epsilon} & \texttt{raise}\ [S^{\neg E}]\ [\Omega]
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{try raise}\ E'v\ \texttt{with} & & \\
E\ x\ y\ \texttt{on other}\ z.e_1\ \texttt{otherwhise}\ e_2\ [\Omega] & \rightarrow^{\epsilon} & \texttt{raise}\ E'\ v\ [\Omega] \\
\multicolumn{3}{c}{\langle e_0, e_1, \ldots, \overbrace{\texttt{raise}\ v}^{i}, \ldots, e_{p-1}\rangle\ [\Omega_1, \overbrace{\top}^{i}, \Omega_2]} \\
\multicolumn{3}{c}{\rightarrow^{\epsilon} \quad \langle e_0, e_1, \ldots, \overbrace{\bot}^{i}, \ldots, e_{p-1}\rangle\ [\Omega_1, \overbrace{v}^{i}, \Omega_2]}
\end{array}
$$

The evaluation order is strongly important when exceptions may be raised, so we have to define evaluation contexts to specify this order, we write C to denote one of these contexts.

$$
\begin{array}{rcl}
C & ::= & [] \mid C\ e \mid v\ C \\
& \mid & \pi\ C \mid C\ \#\ e \mid v\ \#\ C \mid C\ ?\ e \mid v\ ?\ C \mid (C \rightarrow^{e_1} e_2, e_3) \\
& \mid & E\ C \mid \texttt{raise}\ C \\
& \mid & \texttt{try}\ C\ \texttt{with}\ E\ x.e \\
& \mid & \texttt{try}\ C\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e\ \texttt{otherwhise}\ e
\end{array}
$$

The reduction rule of the BSλ_p^{exn}-calculus is then defined by

$$\frac{e\ [\Omega] \rightarrow^{\epsilon} e'\ [\Omega']}{C[e]\ [\Omega] \rightarrow_{BS\lambda_p^{exn}}\ C[e'][\Omega']}$$

We also specify special rules for the empty context which is use if some local exceptions have been raised but no synchronisation has occured before the end of the execution.

$$[v][\Omega^{\perp}] \rightarrow_{BS\lambda_p^{exn}} \texttt{raise}\ \Delta_{\Omega^{\perp}}$$

$$[v][\Omega^{\perp}] \rightarrow_{BS\lambda_p^{exn}} \texttt{raise}\ \Delta_{\Omega^{\perp}}$$

Proposition 4 *If $\Gamma \vdash e : \tau$, (where e is a closed term) then one of the following propositions holds*

- *e is a value.*
- *$e =$* `raise` *v, where v is a value.*
- *$e[\Omega]$ is reductible for any $[\Omega]$ compatible with e.*

Proposition 5 *If $\Gamma \vdash e : \tau$,$[\Omega]$ compatible with e and $e[\Omega] \rightarrow^{\epsilon} e'[\Omega']$ then $\Gamma \vdash e' : \tau$.*

Proposition 6 *If e is compatible with $[\Omega]$ and $e[\Omega] \rightarrow e'[\Omega']$ then e' is compatible with $[\Omega']$.*

proof, proposition 4 : By induction on the structure of e.

- if $e = \lambda x.e'$, then e is a value.
- if $e = (e_1\ e_2)$, then if $\Gamma \vdash e_1\ e_2 : \tau$ we have $\Gamma \vdash e_1 : \tau' \rightarrow \tau$ and $\Gamma \vdash e_2 : \tau'$. By I.H., e_1 and $e2$ are values, `raise` v or reductibles for any compatible LREC.
  - If e_1 is a value, then e_1 is of the form $\lambda x.e_1'$ and we have
    - If e_2 is a value, then $(e_1\ e_2)\ [\Omega] \rightarrow^{\epsilon} e_1'[e_2/x]\ [\Omega]$ for any $[\Omega]$ compatible with $(e_1\ e_2)$.
    - If $e_2 =$ `raise` v then $(e_1\ e_2)\ [\Omega] \rightarrow^{\epsilon}$ `raise` $v\ [\Omega]$ for any $[\Omega]$ compatible with $(e_1\ e_2)$
    - If e_2 is reductible for any compatible LREC, then for any LREC $[\Omega]$ compatible with $(e_1\ e_2)$, $[\Omega]$ is compatible with e_2 by definition and if $e_2\ [\Omega] \rightarrow^{\epsilon} e_2'\ [\Omega]$ then $(e_1\ e_2)\ [\Omega] \rightarrow (e_1\ e_2')\ [\Omega]$
  - If $e_1 =$ `raise` v, then $(e_1\ e_2)\ [\Omega] \rightarrow^{\epsilon}$ `raise v` $[\Omega]$ for any compatible $[\Omega]$.
  - If e_1 is reductible for any compatible LREC, then for any LREC $[\Omega]$ compatible with $(e_1\ e_2)$, $[\Omega]$ is compatible with e_1 by definition and if $e_1\ [\Omega] \rightarrow^{\epsilon} e_1'\ [\Omega']$ then $e_1\ e_2\ [\Omega] \rightarrow (e_1'\ e_2)\ [\Omega']$

- if $e = (\ )$, then e is a value.
- if $e = \langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle$, then if $\Gamma \vdash e : A$, $A = \tau$ `par` and for all $i \in \mathcal{N}$ we have $\Gamma \vdash w_i : \tau$ or $w_i = \bot$. By I.H. each w_i is $\bot$, a value, `raise` v or reductible for any compatible LREC.
  - if each w_i is a value or $\bot$, then e is a value.
  - if exists i such that $w_i =$ `raise` v, then $[\Omega]$ compatible with e implies $[\Omega] = [\Omega_1, \top, \Omega_2]$ and

$$\begin{array}{rl} & \langle w_0,\ w_1,\ \ldots,\ \texttt{raise}\, v,\ \ldots,\ w_{p-1}\rangle\ [\Omega_1,\ \top,\ \Omega_2] \\ \rightarrow^{\epsilon} & \langle w_0,\ w_1,\ \ldots,\ \bot,\ \ldots,\ w_{p-1}\rangle\ [\Omega_1,\ e,\ \Omega_2] \end{array}$$

  - if exists i such that e_i is reductible for any compatible LREC, then for any LREC $[\Omega]$ compatible with e, we have $[\Omega]$ compatible with e_i since by the abscence of imbrication of parallel values, it can be easily checked that e_i is compatible with any LREC , then if $e_i\ [\Omega] \rightarrow^{\epsilon} e_i'\ [\Omega']$

$$\langle e_0, e_1, \ldots, e_i, \ldots, e_{p-1}\rangle\ [\Omega] \rightarrow^{\epsilon} \langle e_0, e_1, \ldots, e_i', \ldots, e_{p-1}\rangle\ [\Omega']$$

- if $e = e_1\ \#\ e_2$, then if $\Gamma \vdash e : A$, $A = \tau$ `par`, $\Gamma \vdash e_1 : (\tau' \rightarrow \tau)$ `par`, $\Gamma \vdash e_2 : \tau'$ `par`. By I.H., e_1 and e_2 are values, `raise` v or reductibles for any compatible LREC.
  - if e_1 is a value, then $e_1 = \langle w_0, w_1, \ldots, w_{p-1}\rangle$
    - if e_2 is a value then $e_2 = \langle w_0', w_1', \ldots, w_{p-1}'\rangle$ and

$$e_1\ \#\ e_2\ [\Omega] \rightarrow^{\epsilon} \langle w_0'', w_1'', \ldots, w_{p-1}''\rangle\ [\Omega] \quad w_i'' = \begin{cases} (w_i\ w_i') \text{ if } [\Omega]_i = \top \\ \bot\ otherwhise \end{cases}$$

      for any compatible LREC $[\Omega]$.
      Since for all i such as $[\Omega]_i = \top$, w_i and w_i' are not $\bot$ by definition of the compatibility relation, w_i'' is defined for all $i \in \mathcal{N}$.
    - if $e_2 =$ `raise` v, then $e_1\ \#\ e_2\ [\Omega] \rightarrow^{\epsilon}$ `raise` $v\ [\Omega]$ for any compatible $[\Omega]$.
    - if e_2 is reductible for any compatible LREC then for any LREC $[\Omega]$ compatible with $e_1\ \#\ e_2$, we have $[\Omega]$ compatible with e_2 by definition and if $e_2\ [\Omega] \rightarrow^{\epsilon} e_2'[\Omega']$, then $e_1\ \#\ e_2\ [\Omega] \rightarrow e_1\ e_2'[\Omega']$
  - if $e_1 =$ `raise` v, then $e_1\ \#\ e_2\ [\Omega] \rightarrow^{\epsilon}$ `raise` $v\ [\Omega]$ for any compatible $[\Omega]$.
  - if e_1 is reductible for any compatible LREC then for any LREC $[\Omega]$ compatible with $e_1\ \#\ e_2$, we have $[\Omega]$ compatible with e_1 by definition and if $e_1\ [\Omega] \rightarrow^{\epsilon} e_1'\ [\Omega']$, then $e_1\ \#\ e_2\ [\Omega] \rightarrow e_1'\ \#\ e_2\ [\Omega']$

- if $e = e_1 \ ?\ e_2$, then if $\Gamma \vdash e : A$, we have $A = \tau$ `par`, $\Gamma \vdash e_1 : \tau$ `par`, $\Gamma \vdash e_2 :$ `int par`. By I.H., e_1 and e_2 are values, `raise` v or reductibles for any compatible LREC.
  - if e_1 is a value, then $e_1 = \langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle$
    - if e_2 is a value, then $e_2 = \langle m_0,\ m_1,\ \ldots,\ m_{p-1}\rangle$

      $If\ [\Omega] = [\Omega^\top],$ then by compatibility $[\Omega]$ is *compatible with* e_1 *and* e_2 *and then each* $w_i,\ m_i$ is *a value.*
      $e_1\ ?\ e_2\ [\Omega^\top] \rightarrow \langle w_{m_0}, w_{m_1}, \ldots, w_{m_{p-1}}\rangle\ [\Omega^\top]$

      $If\ [\Omega] = [\Omega^\perp],\ then\ \ \Delta_{\Omega^\perp}$ is $defined$ *and*
      $e_1\ ?\ e_2\ [\Omega^\perp] \rightarrow$ `raise` $\Delta_{\Omega^\perp}\ [\Omega^\top]$

    - if $e_2 =$ `raise` v, then

      $e_1\ ?\ e_2\ [\Omega^\top] \rightarrow$ `raise` $v\ [\Omega^\top]$

      $If\ [\Omega] = [\Omega^\perp],\ then\ \ \Delta_{\Omega^\perp}$ is $defined$ *and*
      $e_1\ ?\ e_2\ [\Omega^\perp] \rightarrow$ `raise` $\Delta_{\Omega^\perp}\ [\Omega^\top]$

    - if e_2 is reductible for any compatible LREC then for any LREC $[\Omega]$ compatible with e, we have $[\Omega]$ compatible with e_2 by definition and if $e_2 \rightarrow e_2'$, then $e_1\ ?\ e_2\ [\Omega] \rightarrow e_1\ ?\ e_2'\ [\Omega^\top]$
  - if $e_1 =$ `raise` v, then

    $e_1\ ?\ e_2\ [\Omega^\top] \rightarrow$ `raise` $v\ [\Omega^\top]$

    $If\ [\Omega] = [\Omega^\perp],\ then\ \ \Delta_{\Omega^\perp}$ is $defined$ *and*
    $e_1\ ?\ e_2\ [\Omega^\perp] \rightarrow$ `raise` $\Delta_{\Omega^\perp}\ [\Omega^\top]$

  - if e_1 is reductible for any compatible LREC, then for any LREC $[\Omega]$ compatible with e, we have $[\Omega]$ compatible with e_1 by definition and if $e_1\ [\Omega] \rightarrow e_1'\ [\Omega']$, then $e_1\ ?\ e_2\ [\Omega] \rightarrow e_1'\ ?\ e_2\ [\Omega']$
- if $e = E\ e_0$, then if $\Gamma \vdash E\ e_0 : A$, $A \in \{exn_L, exn_G\}$ and $\Gamma \vdash e_0 : Arg(E)$. By I.H e_0 is a value, `raise` v or reductible for any compatible LREC.
  - if e_0 is a value, then $E\ e_0$ is a value.
  - if $e_0 =$ `raise` v, then $E\ e_0\ [\Omega] \rightarrow^\epsilon$ `raise` $v\ [\Omega]$ for any compatible $[\Omega]$
  - if e_0 is reductible for any compatible LREC then for any LREC $[\Omega]$ compatible with e, we have $[\Omega]$ compatible with e_0 and if $e_0\ [\Omega] \rightarrow^\epsilon e_0'[\Omega']$, then $E\ e_0\ [\Omega] \rightarrow E\ e_0'\ [\Omega']$
- if $e = [S]$ then e is a value.

- if $e = \texttt{try}\ e_1\ \texttt{with}\ E\ x.e_2$, then if $\Gamma \vdash e : \tau$ we have $\Gamma \vdash e_1 : \tau$. By I.H. e_1 is a value, $\texttt{raise}\ v$ or reductible for any compatible LREC.
  - if e_1 is a value, then $e\ [\Omega] \to^{\epsilon} e_1\ [\Omega]$ for any compatible $[\Omega]$.
  - if $e_1 = \texttt{raise}\ v$,
    - if $v = E\ v$ then $e\ [\Omega] \to^{\epsilon} e_2[v/x]\ [\Omega]$
    - if $v = E_{k_i}\ v,\ E_{k_i} \neq E$ then $e\ [\Omega] \to^{\epsilon} \texttt{raise}\ v\ [\Omega]$
    - if $v = [S]$ then $e\ [\Omega] \to^{\epsilon} \texttt{raise}\ v\ [\Omega]$

    for any compatible $[\Omega]$
  - if e_1 is reductible for any compatible LREC, then for any $LREC\ [\Omega]$ compatible with e, we have $[\Omega]$ compatible with e_1 and if $e_1\ [\Omega] \to^{\epsilon} e_1'\ [\Omega']$, then
$$\texttt{try}\ e_1\ \texttt{with}\ E\ x.e_2\ [\Omega] \to \texttt{try}\ e_1'\ \texttt{with}\ E\ x.e_2\ [\Omega']$$
- if $e = \texttt{try}\ e_1\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3$, then if $\Gamma \vdash e : \tau$ then $\Gamma \vdash e_1 : \tau$. By I.H, e_1 is a value, raise v or reductible for any compatible LREC.
  - if e_1 is a value, then $e\ [\Omega] \to^{\epsilon} e_1\ [\Omega]$ for any compatible $[\Omega]$
  - if $e_1 = \texttt{raise}\ v$,
    - if $v = [\mathcal{S},\ E(f,g),\ \mathcal{S}']$, then
$$\begin{array}{rl} & \texttt{try}\ e_1\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3\ [\Omega] \\ \to^{\epsilon} & e_2[f/x, g/y, [\mathcal{S}, \mathcal{S}']/z]\ [\Omega] \end{array}$$
    - if $v = [E(f,g)]$, then
$$\begin{array}{rl} & \texttt{try}\ e_1\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3\ [\Omega] \\ \to^{\epsilon} & e_3[f/x, g/y]\ [\Omega] \end{array}$$
    - if $v = [\mathcal{S}^{\neg\mathcal{E}}]$, then
$$\begin{array}{rl} & \texttt{try}\ e_1\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3\ [\Omega] \\ \to^{\epsilon} & \texttt{raise}\ v\ [\Omega] \end{array}$$
    - if $v = E_{k_i}$, then $e\ [\Omega] \to^{\epsilon} \texttt{raise}\ v\ [\Omega]$

    for any compatible $[\Omega]$.
  - if e_1 is reductible for any compatible LREC, then for any LREC $[\Omega]$ compatible with e, we have $[\Omega]$ compatible with e_1 and if $e_1\ [\Omega] \to^{\epsilon} e_1'[\Omega']$, then
$$\begin{array}{rl} & \texttt{try}\ e_1\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3\ [\Omega] \\ \to & \texttt{try}\ e_1'\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3\ [\Omega'] \end{array}$$

proof, proposition 5 : by case on the rule $\rightarrow^{\epsilon}$ used.

- $(\lambda x : \tau.e)\ v\ [\Omega] \rightarrow^{\epsilon} e[v/x]\ [\Omega]$
  If $\Gamma \vdash (\lambda x : \tau.e)\ v : \tau'$, then $\Gamma \vdash (\lambda x; \tau.e) : \tau \rightarrow \tau'$ and $\Gamma \vdash v : \tau$.
  We have $\Gamma, x : \tau \vdash e : \tau'$, and then $\Gamma \vdash e[v/x]; \tau'$.

- $(\pi\ v)\ [\Omega] \rightarrow^{\epsilon} \langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle\ [\Omega]$
  where $w_i = \begin{cases} (v\ i)\ \mathrm{if}\ [\Omega]_i = \top \\ \bot\ otherwhise \end{cases}$
  If $\Gamma \vdash (\pi\ v) : A$, then $A = \tau$ `par`.
  We have $\Gamma \vdash v :$ `int` $\rightarrow \tau$ and then for all $i \in \mathcal{N}$ $\Gamma \vdash vi : \tau$.
  Finally, $\Gamma \vdash \langle w_0,\ w_1,\ \ldots,\ w_1\rangle : \tau$ `par`

- $\langle f_0,\ f_1,\ \ldots,\ f_{p-1}\rangle\ \#\ \langle e_0,\ e_1,\ \ldots,\ e_{p-1}\rangle\ [\Omega] \rightarrow^{\epsilon} \langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle\ [\Omega]$
  where $w_i = \begin{cases} (f\ i)\ (e\ i)\ \mathrm{if}\ [\Omega]_i = \top \\ \bot\ otherwhise \end{cases}$
  If $\Gamma \vdash \langle f_0,\ f_1,\ \ldots,\ f_{p-1}\rangle\ \#\ \langle e_0,\ e_1,\ \ldots,\ e_{p-1}\rangle : \tau'$ `par` then
  $\Gamma \vdash \langle f_0,\ f_1,\ \ldots,\ f_{p-1}\rangle : (\tau \rightarrow \tau')$ `par` *and* $\Gamma \vdash \langle f_0,\ f_1,\ \ldots,\ f_{p-1}\rangle : \tau$ `par`
  We have for all $i \in \mathcal{N}$, $\Gamma \vdash f_i : \tau \rightarrow \tau'$ or $f_i = \bot$ and $\Gamma \vdash e_i : \tau'$ or $e_i = \bot$. By compatibility if $f_i = \bot$ or $e_i = \bot$ then $[\Omega]_i \neq \top$.
  Finally for all $i \in \mathcal{N}$, $[\Omega]_i \neq \top$, i.e. $w_i = \bot$ or $\Gamma \vdash f_i\ e_i : \tau'$ and then
  $\Gamma \vdash \langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle : \tau'$ `par`

- $\langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle\ ?\ \langle m_0,\ m_1,\ \ldots,\ m_{p-1}\rangle\ [\Omega^{\top}] \rightarrow^{\epsilon} \langle v_{m_0},\ v_{m_1},\ \ldots,\ v_{p-1}\rangle [\Omega^{\top}]$
  If $\Gamma \vdash \langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle\ ?\ \langle m_0,\ m_1,\ \ldots,\ m_{p-1}\rangle\ : \tau\ par$, then
  $\Gamma \vdash \langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle : \tau$ `par` and $\Gamma \vdash \langle m_0,\ m_1,\ \ldots,\ m_{p-1}\rangle :$ `int par`
  We have for all $i \in \mathcal{N}$, $\Gamma \vdash v_i : \tau$ or $v_i = \bot$ and for all $i \in \mathcal{N}$, $\Gamma \vdash m_i :$ `int` or $m_i = \bot$. Since for all $i \in \mathcal{N}$, $[\Omega]_i = \top$, then by compatibility, we have $v_i \neq \bot$ and $m_i \neq \bot$ and then $\Gamma \vdash \langle v_{m_0},\ v_{m_1},\ \ldots,\ v_{p-1}\rangle : \tau$ `par`

- $\langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle\ ?\ \langle m_0,\ m_1, \ldots,\ v_{p-1}\rangle\ [\Omega^{\perp}] \rightarrow^{\epsilon}$ `raise` $\Delta_{\Omega^{\perp}}$
  If $\Gamma \vdash \langle v_0, v_1, \ldots, v_{p-1}\rangle\ ?\ \langle m_0, m_1, \ldots, v_{p-1}\rangle : A$, then $A = \tau$ `par`. By definition, $\Gamma \vdash \Delta_{\Omega^{\perp}} : Mexn$ and then $\Gamma \vdash$ `raise` $\Delta_{\Omega^{\perp}} : \tau$ `par`

- $\langle v_0,\ v_1,\ \ldots, S,\ \ldots,\ v_{p-1}\rangle \rightarrow^{n} e_1, e_2\ [\Omega^{\top}] \rightarrow^{\epsilon} T[\Omega^{\top}]$
  where $T = \begin{cases} e_1\ \mathrm{if}\ S =$ `true` $\\ e_2\ \mathrm{if}\ S =$ `false` $\end{cases}$
  If $\Gamma \vdash \langle v_0,\ v_1,\ \ldots, S,\ \ldots,\ v_{p-1}\rangle \rightarrow^{n} e_1, e_2 : \tau$, then $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$.

- $\langle v_0,\ v_1,\ \ldots,\ v_{p-1}\rangle \to^n e_1, e_2\ [\Omega^\perp] \to^\epsilon \texttt{raise}\ \Delta_{\Omega^\top}[\Omega^\top]$

  If $\Gamma \vdash \langle v_0,\ v_1,\ \ldots, S,\ \ldots,\ v_{p-1}\rangle \to^n e_1, e_2 : \tau$, then by definition $\Gamma \vdash \Delta_{\Omega^\perp} : Mexn$, and we have $\Gamma \vdash \texttt{raise}\ \Delta_{\Omega^\perp}\ [\Omega^\top] : \tau$

- $\texttt{try}\ v\ \texttt{with}\ E\ x.e\ [\Omega] \to^\epsilon v\ [\Omega]$

  If $\Gamma \vdash \texttt{try}\ v\ \texttt{with}\ E\ x.e : \tau$, then $\Gamma \vdash v : \tau$.

- $\texttt{try raise}\ E\ v\ \texttt{with}\ E\ x.e\ [\Omega] \to^\epsilon e[v/x]\ [\Omega]$

  If $\Gamma \vdash \texttt{try raise}\ E\ v\ \texttt{with}\ E\ x.e : \tau$, then $\Gamma,\ x : Arg(E) \vdash e : \tau$ and $\Gamma \vdash \texttt{raise}\ E\ v : \tau$, i.e. $\Gamma \vdash E\ v : exn_L$ or $\Gamma \vdash E\ v : exn_G$.

  Finally $\Gamma \vdash v : Arg(E)$. Thus we have $\Gamma \vdash e[v/x] : \tau$.

- $\texttt{try}\ v\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3\ [\Omega] \to^\epsilon v\ [\Omega]$

  If $\Gamma \vdash \texttt{try}\ v\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3 : \tau$, then $\Gamma \vdash v : \tau$

- $$\begin{array}{ll} & \texttt{try raise}\ [S,\ E(f,g),\ S']\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3\ [\Omega] \\ \to^\epsilon & e_2[f/x, g/y, [S,S']/z]\ [\Omega] \end{array}$$

  If $\Gamma \vdash \texttt{try raise}\ [S,\ E(f,g),\ S']\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3 : \tau$, then

  $$\Gamma, x : \texttt{int} \to Arg(E), y : \texttt{int} \to bool,\ z : Mexn \vdash e_2 : \tau$$

  $\Gamma \vdash \texttt{raise}\ [S, E(f,g), S'] : \tau$, i.e $\Gamma \vdash [S, E(f,g), S'] : Mexn$ and

  $\Gamma \vdash f : \texttt{int} \to Arg(E)$
  $\Gamma \vdash g : \texttt{int} \to bool$
  $\Gamma \vdash [S, S'] : Mexn$

  Finally $\Gamma \vdash e_2[f/x,\ g/y,\ [S,S']/z] : \tau$.

- $$\begin{array}{ll} & \texttt{try raise}\ [S,\ E(f,g),\ S']\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3\ [\Omega] \\ \to^\epsilon & e_3[f/x, g/y]\ [\Omega] \end{array}$$

  If $\Gamma \vdash \texttt{try raise}\ [E(f,g)]\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_2\ \texttt{otherwhise}\ e_3 : \tau$, then

  $$\Gamma, x : \texttt{int} \to Arg(E), y : \texttt{int} \to bool \vdash e_3 : \tau$$

  $\Gamma \vdash \texttt{raise}\ [E(f,g)] : \tau$, i.e $\Gamma \vdash [E(f,g)] : Mexn$ and

  $\Gamma \vdash f : \texttt{int} \to Arg(E)$
  $\Gamma \vdash g : \texttt{int} \to bool$

  Finally $\Gamma \vdash e_3[f/x,\ g/y] : \tau$.

- $$\begin{aligned} & \langle e_0,\ e_1,\ \ldots,\ \texttt{raise}\, e,\ \ldots,\ e_{p-1}\rangle\ [\Omega_1, \overbrace{\top}^{i}, \Omega_2] \\ \rightarrow^{\epsilon}\ & \langle e_0,\ e_1,\ \ldots,\ \bot,\ \ldots,\ e_{p-1}\rangle\ [\Omega_1, \overbrace{e}^{i}, \Omega_2] \end{aligned}$$

  If $\Gamma \vdash \langle e_0,\ e_1,\ \ldots,\ \texttt{raise}\, e,\ \ldots,\ e_{p-1}\rangle : A$, then $A = \tau\ \texttt{par}$ and for all $i \in \mathcal{N}$ $\Gamma \vdash e_i : \tau$. Then we have $\Gamma \vdash \langle e_0,\ e_1,\ \ldots,\ \texttt{raise}\, e,\ \ldots,\ e_{p-1}\rangle : \tau\ \texttt{par}$

- The result is trivial for all other rules since they all evaluates to an expression of the form $\texttt{raise}\, v.$ where $\texttt{raise}\, v$ is either a subterm of the left member of the rule (and then well typed) or of the form raise Δ_Ω wich is well typed by definition.

proof, proposition 6 :

- $(\lambda x.e)v \rightarrow^{\epsilon} e[v/x]$

  If $[\Omega]$ is compatible with $(\lambda x.e)\ v$, then $[\Omega]$ is compatible with $\lambda x.e$ and v by definition and $[\Omega]$ is compatible with $e[e'/x]$ for any e' compatible with $[\Omega]$. Finally, $e[v/x]$ is compatible with $[\Omega]$.

- $(\pi\ v)\ [\Omega] \rightarrow^{\epsilon} \langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle\ [\Omega]$

  If $[\Omega] = [\Omega^{\top}]$ then for all $i \in \mathcal{N}, w_i = (v\ i)$ and $\langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle$ is compatible with $[\Omega]$.

  If $[\Omega] = [\Omega^{\bot}]$ then for all i such as $[\Omega]_i \neq \top$, we have $w_i = \bot$ and $(v\ j)$ for all other $j \in \mathcal{N}$. Thus $w_i = \bot \Rightarrow [\Omega]_i \neq \top$ and $\langle w_0,\ w_1,\ \ldots,\ w_{p-1}\rangle$ is compatible with $[\Omega]$.

- $\langle f_0,\ f_1, \ldots, f_{p-1}\rangle\ \#\ \langle e_0,\ e_1, \ldots, e_{p-1}\rangle \rightarrow^{\epsilon} \langle w_0,\ w_1, \ldots, w_{p-1}\rangle$

  where $w_i = \begin{cases} (f_i\ e_i) if [\Omega]_i = \top \\ \bot otherwhise \end{cases}$

  for all $i \in \mathcal{N}$ such as $[\Omega]_i \neq \top$, we have $w_i = \bot$ and $w_j = (f_j\ e_j)$ for all other $j \in \mathcal{N}$. Thus $\langle w_0,\ w_1, \ldots, w_{p-1}\rangle$ is compatible with $[\Omega]$.

- $\langle v_0,\ v_1, \ldots, v_{p-1}\rangle\ ?\ \langle m_0,\ m_1, \ldots, m_{p-1}\rangle\ [\Omega^{\top}] \rightarrow^{\epsilon} \langle v_0,\ v_1, \ldots, v_{p-1}\rangle\ [\Omega^{\top}]$

  Since for all $i \in \mathcal{N}$, v_i is a value, $\langle v_0,\ v_1, \ldots, v_{p-1}\rangle$ is compatible with $[\Omega^{\top}]$.

- $\langle v_0,\ v_1, \ldots, v_{p-1}\rangle\ ?\ \langle m_0,\ m_1, \ldots, m_{p-1}\rangle\ [\Omega^{\bot}] \rightarrow^{\epsilon} \texttt{raise}\ \Delta_{\Omega^{\bot}}\ [\Omega^{\top}]$

  $\Delta_{\Omega^{\bot}}$ is compatible with $[\Omega^{\top}]$

- $(\langle v_0,\ v_1, \ldots, v_{p-1}\rangle \rightarrow^{n} e_1, e_2)\ [\Omega^{\top}] \rightarrow^{\epsilon} T[\Omega^{\top}]$ where $T = \begin{cases} e_1 if S = \texttt{true} \\ e_2 if S = \texttt{false} \end{cases}$

  e_1 and e_2 are compatibles with $[\Omega^{\top}]$ by definition.

- $(\langle v_0,\ v_1, \ldots, v_{p-1}\rangle \rightarrow^n e_1, e_2)\ [\Omega^\bot] \rightarrow^\epsilon \texttt{raise}\ \Delta_{\Omega^\top}\ \Omega^\bot$
  $\Delta_{\Omega^\bot}$ is compatible with $[\Omega^\top]$

- $\texttt{try}\ v\ \texttt{with}\ E\ x.e\ [\Omega] \rightarrow v\ [\Omega]$
  If $[\Omega]$ is compatible with $\texttt{try}\ v\ \texttt{with}\ E\ x.e$ then $[\Omega]$ is compatible with v, by definition.

- $\texttt{try raise}\ E\ v\ \texttt{with}\ E\ x.e\ [\Omega] \rightarrow^\epsilon e[v/x]\ [\Omega]$ By definition, if $[\Omega]$ is compatible with $\texttt{try raise}\ E\ v\ \texttt{with}\ E\ x.e$ then $[\Omega]$ is compatible with v and $[\Omega]$ is compatible with e[e'/x] for all e' compatible with $[\Omega]$.

- $\texttt{try raise}\ E'\ v\ \texttt{with}\ E\ x.e\ [\Omega] \rightarrow^\epsilon \texttt{raise}\ E'\ v\ [\Omega]$
  If $[\Omega]$ is compatible with $\texttt{try raise}\ E'\ v\ \texttt{with}\ E\ x.e$ then $[\Omega]$ is compatible with $\texttt{raise}\ E'\ v$.

- $\texttt{try raise}\ [S]\ \texttt{with}\ E\ x.e\ [\Omega] \rightarrow^\epsilon \texttt{raise}\ [S]\ [\Omega]$
  If $[\Omega]$ is compatible with $\texttt{try raise}\ [S]\ \texttt{with}\ E\ x.e$ and then $[\Omega]$ is compatible with $\texttt{raise}\ [S]$.

- $\texttt{try}\ v\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_1\ \texttt{otherwhise}\ e_2\ [\Omega] \rightarrow^\epsilon v[\Omega]$
  If $[\Omega]$ is compatible with $\texttt{try}\ v\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_1\ \texttt{otherwhise}\ e_2$, then $[\Omega]$ is compatible with v.

- $\texttt{try raise}\ [S,\ E(f,g), S']\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_1\ \texttt{otherwhise}\ e_2\ [\Omega] \rightarrow^\epsilon e_1[f/x,\ g/y,\ [S, S']/z][\Omega]$
  If $[\Omega]$ is compatible with $\texttt{try}\ v\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_1\ \texttt{otherwhise}\ e_2$, then $[\Omega]$ is compatible with $\texttt{raise}\ [S,\ E(f,g),\ S']$ i.e $[\Omega]$ is compatible with $[S,\ E(f,g),\ S']$ and then with f, g and $[S,\ S']$. Thus $[\Omega]$ is compatible with $e_2[e'_1/x,\ e'_2/y,\ e'_3/z]$ for any $e'_1,\ e'_2,\ e'_3$ compatibles with $[\Omega]$.

- $\texttt{try raise}\ [E(f,g)]\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_1\ \texttt{otherwhise}\ e_2\ [\Omega] \rightarrow^\epsilon e_2[f/x,\ g/y][\Omega]$
  If $[\Omega]$ is compatible with $\texttt{try raise}\ [E(f,g)]\ \texttt{with}\ E\ x\ y\ \texttt{on other}\ z.e_1\ \texttt{otherwhise}\ e_2$ then $[\Omega]$ is compatible with $\texttt{raise}\ [E(f,g)]$ i.e $[\Omega]$ is compatible with f and g. Thus $[\Omega]$ is compatible with $e_2[f/x,\ g/y]$.

- $$\begin{array}{rl} & \langle e_0,\ e_1,\ \ldots,\ \texttt{raise}\ e,\ \ldots,\ e_{p-1}\rangle\ [\Omega_1,\ \top,\ \Omega_2] \\ \rightarrow^n & \langle e_0,\ e_1,\ \ldots,\ \bot,\ \ldots,\ e_{p-1}\rangle\ [\Omega_1,\ e,\ \Omega_2] \end{array}$$
  If $[\Omega_1,\ \top,\ \Omega_2]$ is compatible with $\langle e_0,\ e_1,\ \ldots,\ \texttt{raise}\ e,\ \ldots,\ e_{p-1}\rangle$ then $[\Omega_1, e, \Omega_2]$ is compatible with $\langle e_0, e_1, \ldots, \bot, \ldots, e_{p-1}\rangle$ since for all $j \neq i$, $[\Omega_1, \top, \Omega_2]_j = [\Omega_1, e, \Omega_2]_j$ and e_j is unchanged by the reduction rule. $[\Omega_1, e, \Omega_2]_i \neq \top$ and $w_i = \top$

Chapter 6

Conclusion and Future Work

In the current `BSMLlib`, it is possible to use Objective Caml exceptions but we have seen that it raises some safety issues. We have then introduced a new BSλ_p-calculus that recovers usual BSLM safety properties. This calculus stays in the BSP algorithms spirit and uses BSP superstep as atomic steps for local exceptions propagation what permits to handle such exceptions at the BSP superstep which follows the one in which it has been raised (no synchronization is add to the original calculus). This allows a safe implementation of BSML with exceptions in which dead-locks are avoided. The implementation of this new possibility has not yet been done, we are actually writing a distributed version of our semantics which will be used for the implementation. Moreover, we have before to investigate some other good properties of the calculus and this one is probably going to be modified. Actually, The use of the new `try` construction is not totally satisfying since the user could "forget" that an exception raised locally cannot be caught before the next superstep. He could then write a program where the `try` construct is not used in a proper way. Thus, we are thinking of a simple modification which would avoid this kind of programming mistakes. The `try` construction could disappear and would be replaced by a similar construction introduced as a extension of synchronization operations. Handlers would then be declared in the code where one writes such an operation. This semantics will have to be proved equivalent to the semantics introduced here and an implementation close to it will be realized.

We have also made a simple modification to the BSλ-calculus by introducing pattern matching facilities for parallel vectors. The global control construction `if at` has been replaced by a `match at` construction that permits to express more complex global control conditions. An implementation of this new property has been realized using the camlp4 pre-processor functionality. The usual Objective Caml pattern-matching mechanism is extended to take into account or new constructions on BSML parallel vectors.

Bibliography

[BBH99] M. Bamha, F. Bentayeb, and G. Hains. An efficient scalable parallel view maintenance algorithm for shared nothing multi-processor machines. In T. Bench-Capon, G. Soda, and A. Min Tjoa, editors, *10th International Conference on Database and Expert Systems Applications, DEXA'99*, number 1677 in LNCS, pages 616–625. Springer-Verlag, August 30 – September 3 1999.

[BCKL03] Gilles Barthe, Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Pure patterns type systems. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-03)*, pages 250–261, New York, 2003. ACM Press.

[BLH00] O. Ballereau, F. Loulergue, and G. Hains. High-level BSP Programming: BSML and BSλ. In G. Michaelson and Ph. Trinder, editors, *Trends in Functional Programming*, pages 29–38. Intellect Books, 2000.

[BV99] A. Braud and C. Vrain. A parallel genetic algorithm based on the BSP model. In *Evolutionary Computation and Parallel Processing GECCO & AAAI Workshop*, Orlando (Florida), USA, 1999.

[CK99] Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. In *Logic in Computer Science*, pages 98–108, 1999.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[de 95] Philippe de Groote. A simple calculus of exception handling. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings 2nd Int. Conf. on Typed Lambda Calculi and Applications, TLCA'95, Edinburgh, UK, 10–12 Apr. 1995*, volume 902, pages 201–215. Springer-Verlag, Berlin, 1995.

[DK96] D. C. Dracopoulos and S. Kent. Speeding up genetic programming: A parallel BSP implementation. In *First Annual Conference on Genetic Programming*. MIT Press, July 1996.

[DL94] J. W. Demmel and X. Li. Faster numerical algorithms via exception handling. *IEEE Transactions on Computers*, 43(8):983–992, August 1994.

[Edi99] F. Dehne (Guest Editor). Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14:173–421, 1999.

[GHMR98] L. Granvilliers, G. Hains, Q. Miller, and N. Romero. A system for the high-level parallelization and cooperation of constraint solvers. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 596–601, Las Vegas, USA, 1998. IASTED/ACTA Press.

[HF93] G. Hains and C. Foisy. The Data-Parallel Categorical Abstract Machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93*, number 694 in LNCS, pages 56–67. Springer, 1993.

[HL02] G. Hains and F. Loulergue. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, pages 165–178. Nova Science Publishers, august 2002.

[HMa98] J.M.D. Hill, W.F. McColl, and al. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.

[Iss01] Valérie Issarny. Concurrent exception handling. *Lecture Notes in Computer Science*, 2022:111+, 2001.

[KO02] Aaron W. Keen and Ronald A. Olsson. Exception handling during asynchronous method invocation. In *Euro-Par 2002*, LNCS. Springer Verlag, 2002.

[KPT96] Delia Kesner, Laurence Puel, and Val Tannen. A typed pattern calculus. *Information and Computation*, 124(1):32–61, 1996.

[Ler02] Xavier Leroy. The Objective Caml System 3.06, 2002. web pages at www.ocaml.org.

[LHF00] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.

[Lou01] F. Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, (4):423–437, 2001.

[Lou02] F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In 14^{th} *IASTED International Conference on Parallel and Distributed Computing Systems*, pages 452–457. ACTA Press, 2002.

[McC96] W. F. McColl. Universal computing. In L. Bouge and al., editors, *Proc. Euro-Par '96*, volume 1123 of *LNCS*, pages 25–36. Springer-Verlag, 1996.

[MJMR01] Simon Marlow, Simon L. Peyton Jones, Andrew Moran, and John H. Reppy. Asynchronous exceptions in haskell. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 274–285, 2001.

[RS98] R. O. Rogers and D. B. Skillicorn. Using the BSP cost model to optimise parallel neural network training. *Future Generation Computer Systems*, 14(5-6):409–424, 1998.

[SG98] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.

[SHM97] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3), 1997.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[VH] Pimpen Vejjajiva and Mark E. Hall. A lambda-calculus with patterns.