A Polymorphic Type System for BSML

Frédéric Gava

Technical Report TR-2002-12

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

Presentation

Le projet CARAML

Description

Certains problèmes comme la simulation de phénomènes physiques ou chimiques ou la gestion de bases de données de grande taille nécessitent des performances que seules les machines massivement parallèles peuvent offrir. Leur programmation demeure néanmoins plus difficile que celle des machines séquentielles. La conception de langages adaptés est un sujet de recherche actif. Le but du projet CARAML (CoordinAtion et Répartition des Applications Multiprocesseurs en objective camL) est le développement de bibliothèques pour le calcul haute-performance et globalisé autour du langage CAML de l'INRIA, dans son dialecte Objective Caml (OCaml). Ceci peut être effectué par la création de bibliothèques de primitives parallèles et globalisées, bibliothèques applicatives orientées SGBD et calcul numérique et enfin d'exemples d'applications à la simulation moléculaire.

Ce projet est commun aux universités de Paris VII, Paris XII, université d'Orléans et de l'INRIA. Il est dirigé par M. Hains de l'université d'Orléans.

Motivation

CAML est un langage de programmation de haut niveau qui offre de nombreux avantages pour le développement de logiciel : programmation fonctionnelle et impérative, typage fort implicite, gestion dynamique de la mémoire, implantations efficaces et nombreuses bibliothèques. Le dialecte OCaml y ajoute un système de programmation orientée objet ainsi que plusieurs nouveautés. Le développement du langage par l'INRIA et son utilisation répandue en recherche et en enseignement en font un vecteur idéal pour l'avancement et la popularisation des technologies du logiciel parallèle et réparti. Le paradigme GRID (ACI Globalisation des ressources informatiques et des données), fondé sur la fusion de ces deux domaines de l'informatique, se diffusera rapidement dans l'enseignement et la recherche en informatique si on intègre à OCaml les fonctionnalités appropriées : multi-programmation de systèmes data-parallèles, algorithmes en mémoire répartie et disques répartis, équilibrage de charge, répartition géographique des applications, tolérance aux pannes. Le projet CARAML propose de développer ces fonctionnalités sous forme de bibliothèques simples d'utilisation et maximisant les performances du matériel. L'utilité du projet est d'alimenter l'enseignement et la recherche par des outils logiciels pertinents, simples d'utilisation et bien modélisés. Les fonctionnalités proposées par CARAML, bien qu'individuellement présentes dans la littérature scientifique, ne sont actuellement réunies par aucun système. Il existe par exemple : des systèmes parallèles tolérants aux pannes mais restreints aux calculs asynchrones sans communications internes, des systèmes de programmation parallèle/répartie fonctionnelle sans modèle de performance ou sans globalisation, etc. L'originalité du projet est de proposer une intégration de la programmation déclarative, parallèle et concurrente dans un cadre de gestion dynamique des performances, d'extensibilité aux grands systèmes et de tolérance aux pannes.

Contexte

Le parallélisme de données est un paradigme de programmation parallèle dans lequel un programme décrit une séquence d'actions sur des tableaux à accès parallèle. Le modèle BSP vise à maximiser la portabilité des performances en ajoutant une notion de processus explicite au parallélisme de données. Un programme BSP est écrit en fonction du nombre de processeurs de l'architecture sur laquelle il s'exécute. Le modèle d'exécution BSP sépare synchronisation et communication et oblige les deux à être des opérations collectives. Il propose un modèle de coût fiable et simple permettant de prévoir les performances de façon réaliste et portable.

Le projet BSlambda/BSML a deux objectifs principaux : parvenir à des langages universels et dans lesquels le programmeur peut se faire une idée du coût à partir du code source. Cette dernière exigence nécessite que soient explicites dans les programmes les lieux du réseau statique de processeurs de la machine.

Le BSlambda-calcul est un lambda-calcul étendu par des opérations parallèles BSP qui s'avère confluent et universel pour les algorithmes BSP. La BSMLlib est une implantation partielle de ces opérations sous forme d'une bibliothèque pour le langage Objective CAML. Cette bibliothèque permet d'écrire des programmes parallèles BSP portable sur une grande variété d'architectures allant du PC à deux processeurs au système massivement parallèle Cray comprenant plusieurs centaines de processeurs, en passant par des clusters de PC.

Description

Le BSlambda-calcul est un calcul non-typé. Il faut alors concevoir un système de typage polymorphe compatible avec les deux niveaux de termes du BSlambda-calcul. Un tel système est indispensable à un langage BSML complet ayant Objective Caml comme cas particulier.

La première partie de ce stage a donc pour objectifs de concevoir un système de typage polymorphe le plus expressif possible et de prouver la correction du typage par rapport à l'évaluation.

La seconde partie consiste en la création d'un algorithme de synthèse de types, de prouver sa correction et sa complétude puis de l'implanter. Pourront être ajoutées, quelques extensions comme les "nuples", les types concrets et enfin l'utilisation de traits impératifs.

Le projet CARAML promeut l'utilisation de la BSMLlib comme base pour le calcul par grille (Grid). Ce stage s'intègre donc dans le projet dans le sens qu'un système de types est le premier pas vers la sûreté (puis de la sécurité) demandé à un langage de haut niveau comme BSML.

Déroulement

Ce texte est à l'origine un mémoire de DEA (DEA Programmation de l'Université de Paris VII année 2002). Il a été écrit au cours d'un stage d'initiation à la recherche au LACL (Laboratoire d'algorithmique, complexité, logique de l'Université de Paris XII) sous la direction de Frédéric Loulergue qui est membre du projet CARAML. Le titre du mémoire était: *Un système de types polymorphes pour le langage BSML.*

Contents

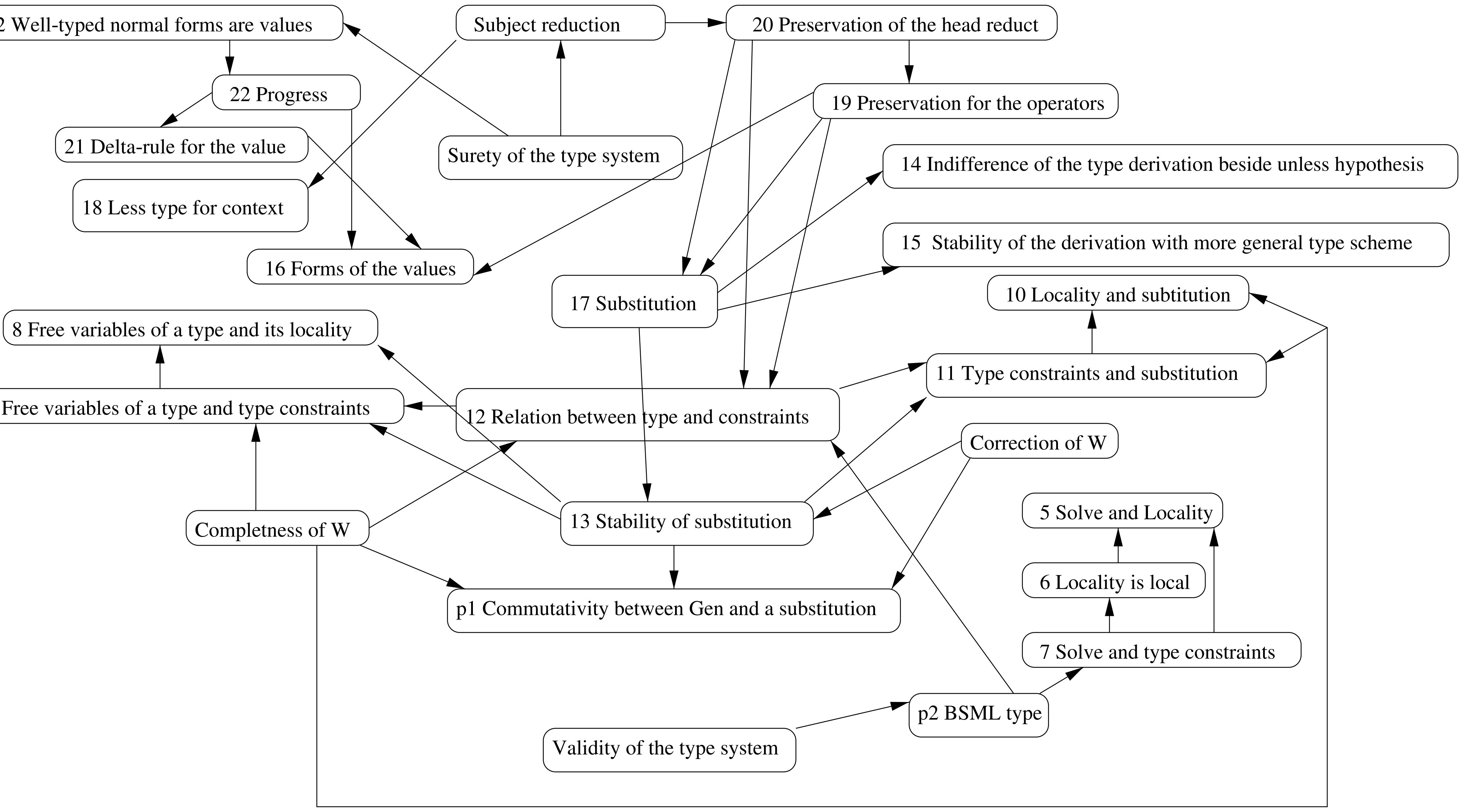
A little Help

To facilitate the reading of this report, we give a sketch of the interactions of the lemmas in the next page. From a box, the point of an arrow means the use of another box. In the box, we note the number of the lemma, "p" for a proposition and "c" for a corollary.

Thanks

I thank Pierre Valarcher who told me of the existence of the DEA Programmation and Frédéric Loulergue for his training course's subject and his encouragements.

This work is supported by the ACI Grid program from the French Ministry of Research, under the project CARAML.


2 Well-typed normal forms are values
Subject reduction
20 Preservation of the head reduct
22 Progress
19 Preservation for the operators
21 Delta-rule for the value
Surety of the type system
14 Indifference of the type derivation beside unless hypothesis
18 Less type for context
15 Stability of the derivation with more general type scheme
16 Forms of the values
17 Substitution
10 Locality and subtitution
8 Free variables of a type and its locality
11 Type constraints and substitution
Free variables of a type and type constraints
12 Relation between type and constraints
Correction of W
13 Stability of substitution
5 Solve and Locality
Completness of W
6 Locality is local
p1 Commutativity between Gen and a substitution
7 Solve and type constraints
p2 BSML type
Validity of the type system

Chapter 1

Introduction

1.1 The subject

Some problems, like the simulation of physical or chemistry problems require performance that only massively parallel computers offer whose programming is still difficult. The BSP execution model represents a parallel computation on p processors as an alternating sequence of computation *superstep* (p asynchronous computations) and communications (data exchanges between processors) with global synchronisation. These two operations are collective. A BSP program is written, according to the number of processors in the computer on which it is computed. The BSP cost model facilitates performance prediction by a simple formula for the execution times.

An algorithm is in *direct mode* when its physical process structure is made explicit. This makes it less convenient to express but more efficient in many cases. This report outlines a type system for BSML, a functional programming language for direct mode BSP algorithms.

In the first chapter, we introduce the theoritical calculus. In the second chapter, we introduce our language, its syntax, its executions and the type system. Next, we give some technical lemmas of the following elements. In the fourth chapter, we introduce a new dynamic semantics and prove some properties of the type system. The fifth chapter gives an algorithm for the type reconstruction and we prove its correction and completeness. In the next chapter we describe the implementation that has been done and in the last, we introduce some possible extensions and further works.

1.2 Bulk-Synchronous Parallelism

BSP computing is a parallel programming model introduced by Valiant [Valiant, 1990] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP computer contains a 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. Its performance is characterized by 3 parameters expressed as multiples of the local processing speed: the number of processor-memory pairs p, the time l required for a global synchronization and 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 in time gh for any arity h.

A BSP program is executed as a sequence of *supersteps*, each one divided into (at most) three successive and logically disjoint phases (figure 1.1).

In the first phase each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes. In the second phase the network delivers the requested data transfers and in the third phase a global synchronization barrier occurs, making the transferred data available for the next superstep. The execution time of a superstep s is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

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

where $w_i^{(s)}$ = local processing time on processor i during superstep s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where

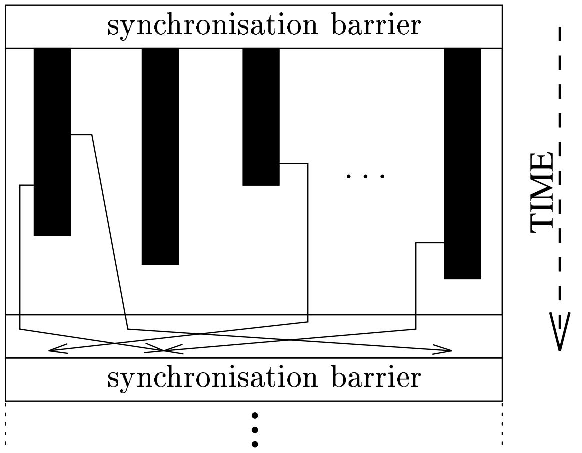


Figure 1.1: A BSP superstep

$h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during superstep s. The execution time $\sum_s \mathrm{Time}(s)$ of a BSP program composed of S supersteps is therefore a sum of 3 terms:$W + H * g + S * l$ where $W = \sum_s \max_i w_i^{(s)}$ and $H = \sum_s \max_i h_i^{(s)}$. 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 and histogram sizes. To minimize execution time the BSP algorithm design must jointly minimize the number S of supersteps and the total volume h (resp. W) and imbalance $h^{(s)}$ (resp. $W^{(s)}$) of communication (resp. local computation).

1.3 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.

1.3.1 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:

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

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

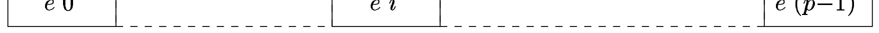


Figure 1.2: The term $\pi\ e$

set of processors to values. The set $\bar{\mathcal{T}}$ of global terms is given by the following grammar:

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

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 $\to$ global function (respectively a global argument to a local $\to$ 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 1.2). 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 1.2). 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:

$$
\begin{array}{ll}
(B) & (\lambda\bar{x}.E)\ E' \to E[\bar{x} \leftarrow E'] \\
(B') & (\lambda\dot{x}.E)\ e' \to E[\dot{x} \leftarrow e']
\end{array}
$$

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.

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

$$(?\pi) \quad (\pi\ e_1)?(\pi\ e_2) \to \pi(\lambda\dot{x}.e_1(e_2\ \dot{x})) \quad \text{where } \forall i \in \mathcal{N},\ (e_2\ i) \to n \in \mathcal{N} \quad \text{(figure 1.3)}$$

$$(\#\pi) \quad (\pi\ e_1)\#(\pi\ e_2) \to \pi(\lambda\dot{x}.(e_1\ \dot{x})(e_2\ \dot{x})) \qquad \text{(figure 1.4)}$$

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.

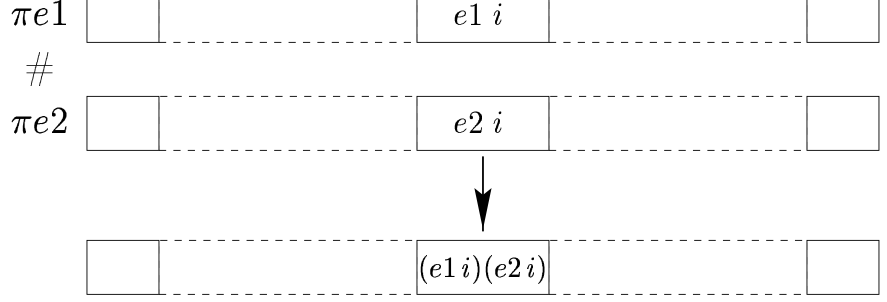


Figure 1.3: The rule ($\#\pi$)

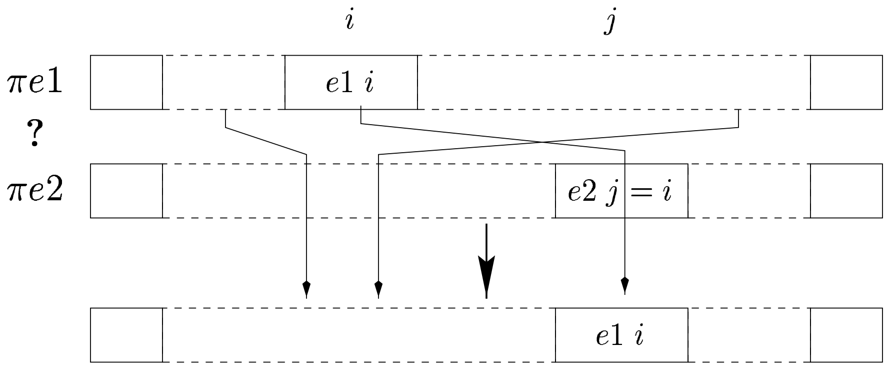


Figure 1.4: The rule ($?\pi$)

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:

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

where e' belongs to $\mathcal{N}$ ((Figure 1.5).

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. [Loulergue et al., 2000]*

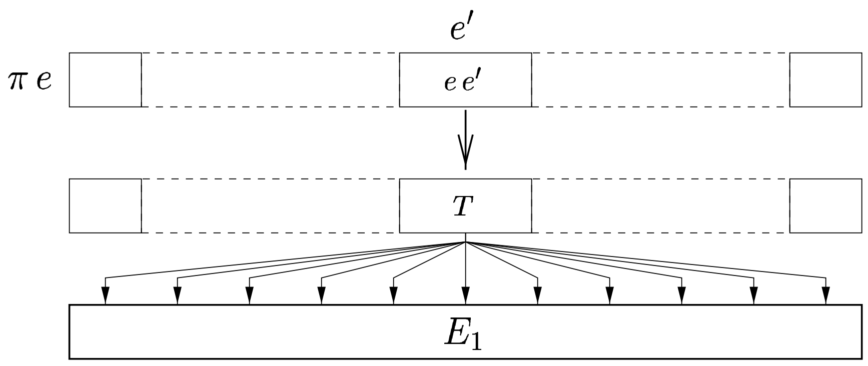


Figure 1.5: The rule ($\overset{e}{\rightarrow}$)

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

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:

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

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

1.4 The BSMLlib library

BSML is a data parallel funtional language for programming BSP algorithms. BSMLlib is based on the following elements ([Loulergue, 2000]). 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 [Hains and Foisy, 1993] 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
```

tores (i-1) (j-1) on process i. Neither the implementation of BSMLLIB, nor its semantics [Loulergue, 2001b] 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:

$$\texttt{put(mkpar(fun i->fs}_i\texttt{))} \tag{1.1}$$

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

Expression (1.1) evaluates to a parallel vector containing a function $\texttt{fd}_i$ of delivered messages on every process. At process `i`, $(\texttt{fd}_i\ \texttt{j})$ 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)`.

Chapter 2

A Type System for mini-BSML

Reasoning on the complete definition of a functional and parallel language such as BSML, would have been complex and tedious. In order to simplify the presentation and to ease the formal reasoning, this chapter introduces a core language. It is an attempt to trade between integrating the principal features of functional and BSP language, and being simple. This chapter introduces its syntax, its dynamic and static semantics together with some conventions, definitions and notation that are used in this technical report.

2.1 Definition of our language

2.1.1 The mini-BSML language

For the sake of conciseness, we limit our study to a subpart of the BSML language. The expressions of mini-BSML, written e possibly with a prime or subscript, have the following abstract syntax:

$$\begin{array}{lll}
e ::= & x & \text{variables} \\
& \mid\ c & \text{constants} \\
& \mid\ op & \text{primitive operations} \\
& \mid\ \mathbf{fun}\ x \to e & \text{function abstraction} \\
& \mid\ (e\ e) & \text{application} \\
& \mid\ \mathbf{let}\ x = e\ \mathbf{in}\ e & \text{local binding} \\
& \mid\ (e, e) & \text{couple}
\end{array}$$

In this grammar, x ranges over a countable set of identifiers. The form $(e\ e')$ stands for the application of a function or an operator e, to an argument e'. The form **fun** $x \to e$ is the so-called and well-known lambda-abstraction that defines the first-class function whose parameter is x and whose result is the value of e. Constants c are the integers 1, 2, the booleans and we assume having a unique value: **()**. The set of primitive operations op contains arithmetic operations, fixpoint operator **fix**, conditional, test function **isnc** of the **nc** constant (which plays the role of the `None` constructor in Objective Caml) and our parallel operations (**mkpar**, **apply**, **put**, **ifat**).

We note $\mathcal{F}(e)$, the set of free variables of an expression e. **let** and **fun** are the binding operators. The formal definition is:

$$\begin{array}{lcl}
\mathcal{F}(c) & = & \emptyset \\
\mathcal{F}(op) & = & \emptyset \\
\mathcal{F}(x) & = & \{x\} \\
\mathcal{F}((e_1\ e_2)) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \\
\mathcal{F}((e_1, e_2)) & = & \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \\
\mathcal{F}(\mathbf{fun}\ x \to e_1) & = & \mathcal{F}(e_1) \setminus \{x\} \\
\mathcal{F}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) & = & \mathcal{F}(e_1) \cup (\mathcal{F}(e_2) \setminus \{x\})
\end{array}$$

Before typing these expressions, we present the dynamic semantics of the language, i.e., how the expressions of mini-BSML are computed to *values*. There is one semantics per value of p, the number of processes of the parallel machine. In the following, $\forall i$ means $\forall i \in \{0, \ldots, p-1\}$. The values of mini-BSML are defined

$v ::=$	$\mathbf{fun}\ x\ \to e$	functional value
	$\mid\ c$	constant
	$\mid\ op$	primitive
	$\mid\ (v, v)$	couple values
	$\mid\ \langle v, \dots, v\rangle$	p-wide parallel vector value

Remark: values are not a sub-set of the expressions. So, we have to define the free variables of a parallel vector: $\mathcal{F}(\langle e_0, \dots, e_{p-1}\rangle) = \bigcup\limits_{i=0}^{p-1} \mathcal{F}(e_i)$

2.1.2 Evaluation rules

The dynamic semantics is defined by an evaluation mechanism that relates expressions to values. To express this relation, we use the formalism of relational semantics (or natural semantics). It consists of a predicate between expressions and values defined by a set of axioms and inference rules called evaluation judgments. An evaluation judgment tells whether an expression evaluates to a given result. There are two kinds of inductive rules, the first for the abstract syntax and the second for primitive operators. We wrote $e_1[x \leftarrow e_2]$ the expression by substituting all the free occurences of x in e_1 by e_2. Now we give the first inductive rules:

$$c \rhd c\ (1) \qquad op \rhd op\ (2) \qquad (\mathbf{fun}\ x \to e) \rhd (\mathbf{fun}\ x \to e)\ (3)$$

$$\frac{e_1 \rhd (\mathbf{fun}\ x \to e) \quad e_2 \rhd v_2 \quad e[x \leftarrow v_2] \rhd v}{(e_1\ e_2) \rhd v}\ (4) \qquad \frac{e_1 \rhd v_1 \quad e_2[x \leftarrow v_1] \rhd v}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rhd v}\ (5) \qquad \frac{e_1 \rhd v_1 \quad e_2 \rhd v_2}{(e_1, e_2) \rhd (v_1, v_2)}\ (6)$$

For addition, conditional, projection, fixpoint operator with an extended syntax, the rules are:

$$\frac{e_1 \rhd + \quad e_2 \rhd n_1 \quad e_3 \rhd n_2 \quad n_1 \text{ and } n_2 \text{ integer and } n = n_1 + n_2}{e_1\ (e_2, e_3)}$$

$$\frac{e_1 \rhd (\mathbf{fun}\ x \to e_2) \quad e_2[x \leftarrow \mathbf{fix}(e_1)] \rhd v}{\mathbf{fix}(e_1) \rhd v} \qquad \mathbf{fix}(op) \rhd op$$

$$\frac{e_1 \rhd \mathbf{true} \quad e_2 \rhd v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd v} \qquad \frac{e_1 \rhd \mathbf{false} \quad e_3 \rhd v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd v}$$

$$\frac{a_1 \rhd \mathbf{fst} \quad a_2 \rhd (v_1, v_2)}{(a_1\ a_2) \rhd v_1} \qquad \frac{a_1 \rhd \mathbf{snd} \quad a_2 \rhd (v_1, v_2)}{(a_1\ a_2) \rhd v_2}$$

and for parallel operations, the rules are:

$$\frac{e_1 \rhd (\mathbf{fun}\ x \to e) \quad \forall i (e[x \leftarrow i] \rhd v_i)}{\mathbf{mkpar}\ e_1 \rhd \langle v_0, ..., v_{p-1}\rangle}$$

$$\frac{e_1 \rhd \langle \mathbf{fun}\ x \to e'_0, \dots, \mathbf{fun}\ x \to e'_{p-1}\rangle \quad e_2 \rhd \langle v'_0, \dots, v'_{p-1}\rangle \quad \forall i (e'_i[x \leftarrow v'_i] \rhd v_i)}{\mathbf{apply}\ e_1\ e_2 \rhd \langle v_0, \dots, v_{p-1}\rangle}$$

$$\frac{f \rhd \langle \mathbf{fun}\ dst \to e'_0, \dots, \mathbf{fun}\ dst \to e'_{p-1}\rangle \quad \forall j \forall i (e'_j[dst \leftarrow\ i] \rhd v^i_j)}{\mathbf{put}\ f \rhd \langle f'_0, \dots, f'_{p-1}\rangle}$$

where $\forall i (f'_i = \mathbf{fun}\ x \to \mathbf{if}\ x = 0\ \mathbf{then}\ v^i_0\ \mathbf{else} \dots \mathbf{if}\ x = (p-1)\ \mathbf{then}\ v^i_{p-1}\ \mathbf{else}\ \mathbf{nc}())$.

$$\frac{e_1 \rhd \langle \dots, \overbrace{\mathbf{true}}^{n}, \dots \rangle \quad e_2 \rhd n \quad e_3 \rhd v_3}{\mathbf{if}\ e_1\ \mathbf{at}\ e_2\ \mathbf{else}\ e_3\ \mathbf{then}\ e_4 \rhd v_3} \qquad \frac{e_1 \rhd \langle \dots, \overbrace{\mathbf{false}}^{n}, \dots \rangle \quad e_2 \rhd n \quad e_4 \rhd v_4}{\mathbf{if}\ e_1\ \mathbf{at}\ e_2\ \mathbf{else}\ e_3\ \mathbf{then}\ e_4 \rhd v_4}$$

$$\frac{e \rhd v \quad v \neq \mathbf{nc}()}{\mathbf{isnc}\ e \rhd\ \mathbf{False}} \qquad \frac{e \rhd \mathbf{nc}()}{\mathbf{isnc}\ e \rhd\ \mathbf{True}}$$

We have not written the evaluation rules that make an error, ie, return an special values **err** (too much rules). For the sequentiel part, it is the classical rules and for the parallel operator, those rules are easy to guess. This problem is not very important because, we will develop a type system that will protect us again those incorrect expressions.

In this section, we present the static semantics of our language. The object of static typing is to detect absurd programs like (**fun** $x \to x) + 1$. The rules of the static semantics associate the expressions of our language with their type in the same way as the dynamic semantics associate expressions with values. The technic of typing ML programs is well-known [Milner, 1978], but is not suited for our language. For example, consider the following expression:

```
let replicate = fun x->mkpar(fun i->x)
   in replicate(mkpar(fun i->5))
```

Its type given by the Objective Caml system is (*int par*) *par*. If this expression was accepted by our system, this would imply that one process could evaluate a whole parallel object. In this case, the cost model would be very complex and the evaluation cost of an expression would depend on its context, making the cost semantics non-compositional. Another example is for the projections (here **fst'**). We can write our projection:

```
let fst' = (fun x -> fun y -> x) in ...
```

We give four cases of the application of the **fst'** to different kinds of objects.

1. (two local objects): `(fst 1 2)`
2. (two parallel objects): `(fst (mkpar (fun i -> i)) (mkpar (fun i -> i)))`
3. (first combination): `(fst (mkpar (fun i -> i)) 1)`
4. (second combination): `(fst 1 (mkpar (fun i -> i)))`

The problem is for the second combination. Its type given by the Objective Caml system is *int*. If this expression was accepted by our type system, we would "hide" the fact that we have a whole parallel object because during the evaluation, we will evaluate the **mkpar** whereas the final result is 1.

The goal of our type system is to reject such expressions. We are first going to equip the language with a type system, then we will give the inference rules of the static semantics and give some examples.

2.2.1 Types for BSML

Type algebra

We begin by defining the term algebra for the basic kinds of semantic objects: the simple types. Simple types are defined by the following grammar:

$\tau ::=$	κ	base type (bool, int, unit etc.)
$\mid$	α	type variable
$\mid$	$\tau_1 \to \tau_2$	type of function from τ_1 to τ_2
$\mid$	$\tau_1 * \tau_2$	type for couple
$\mid$	$(\tau\ par)$	parallel vector type

We want to distinguish between three subsets of simple types. The set of local types L, which represent usual Objective Caml types, the variable types V for polymorphic types and global types G, for parallel objects. The local types (written $\dot{\tau}$) are:

$$\dot{\tau} ::= \kappa \mid \dot{\tau}_1 \to \dot{\tau}_2 \mid \dot{\tau}_1 * \dot{\tau}_2$$

the variable types are (written $\breve{\tau}$):

$$\breve{\tau} ::= \alpha \mid \dot{\tau} \mid \breve{\tau}_1 \to \breve{\tau}_2 \mid \breve{\tau}_1 * \breve{\tau}_2$$

and the global types (written $\bar{\tau}$) are:

$$\bar{\tau} ::= (\breve{\tau}\ par) \mid \breve{\tau}_1 \to \bar{\tau}_2 \mid \bar{\tau}_1 \to \bar{\tau}_2 \mid \bar{\tau}_1 * \bar{\tau}_2 \mid \breve{\tau}_1 * \bar{\tau}_2 \mid \bar{\tau}_1 * \breve{\tau}_2$$

To attain this goal, we will use constraints to say which variable is local or not. For a simple type τ, $\mathcal{L}(\tau)$ says that the simple type is local. For a polymorphic type system, with this kind of constraints, we introduce a Milner's style type scheme with constraints to generically represent the different types of an expression:

$$\boxed{\sigma ::= \forall \alpha_1 ... \alpha_n . [\tau / C]}$$

Where τ is a simple type and C is a constraint of classical propositional calculus given by the following grammar:

$$\boxed{\begin{array}{lll} C ::= & \textbf{False} & \text{the false constraint} \\ \quad \mid & \textbf{True} & \text{the true constraint} \\ \quad \mid & \mathcal{L}(\alpha) & \text{locality of variable of type} \\ \quad \mid & C_1 \wedge C_2 & \text{conjonction of 2 constraints} \\ \quad \mid & C_1 \Rightarrow C_2 & \text{implication of 2 constraints} \end{array}}$$

When the set of variables is empty, we simply write $[\tau/C]$ (or τ/C) and do not write the constraints when they are equal to **True**. A type scheme, consists of a simple type which is universally quantified over a sequence of type variables. Next, we present how to have an instance of a type scheme, i.e., a simple type.

Locality and contraints

Now, when we say that a simple type is **local**, we distinguish its expression from parallel vector of the global networks objects. To affine this idea, we define rules to transform the locality of a type to constraints:

$$\boxed{\begin{array}{cc} \dfrac{\forall \alpha \in \kappa \quad \mathcal{L}(\alpha)}{\textbf{True}} (Kappa) & \dfrac{\forall \tau \quad \mathcal{L}(\tau \ par)}{\textbf{False} \wedge \mathcal{L}(\tau)} (Par) \\ \\ \dfrac{\mathcal{L}(\tau_1 \to \tau_2)}{\mathcal{L}(\tau_1) \wedge \mathcal{L}(\tau_2)} (Arrow) & \dfrac{\mathcal{L}(\tau_1 * \tau_2)}{\mathcal{L}(\tau_1) \wedge \mathcal{L}(\tau_2)} (Couple) \end{array}}$$

In the type system and for the substitution of a type scheme we will use rules to construct constraints from a simple type. We note C_τ for this construction from the simple type τ with the following rules:

$$\boxed{\begin{array}{ccc} \dfrac{\tau \text{ atomic}}{\tau \rightsquigarrow \textbf{True}} (CAtom) & \dfrac{\tau \rightsquigarrow C_1}{(\tau \ par) \rightsquigarrow \mathcal{L}(\tau) \wedge C_1} (CPar) & \dfrac{\tau_1 \rightsquigarrow C_1 \quad \tau_2 \rightsquigarrow C_2}{(\tau_1, \tau_2) \rightsquigarrow C_1 \wedge C_2} (CCouple) \\ \\ \multicolumn{3}{c}{\dfrac{\tau_1 \rightsquigarrow C_1 \qquad \tau_2 \rightsquigarrow C_2}{(\tau_1 \to \tau_2) \rightsquigarrow C_1 \wedge C_2 \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))} (CArrow)} \end{array}}$$

The last rule ($CArrow$) says that if τ_2 is local then τ_1 must be also local.
Remark: this inductive construct rules can also be read like:

$$\begin{array}{lcll} C_\tau & = & \textbf{True} \text{ if } \tau \text{ atomic} & (CAtom) \\ C_{(\tau \ par)} & = & \mathcal{L}(\tau) \wedge C_\tau & (CPar) \\ C_{(\tau_1 * \tau_2)} & = & C_{\tau_1} \wedge C_{\tau_2} & (CCouple) \\ C_{(\tau_1 \to \tau_2)} & = & C_{\tau_1} \wedge C_{\tau_2} \wedge \mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1) & (CArrow) \end{array}$$

(this is the notation we will generally use in the following)

Substitution

The set of free variables ($\mathcal{F}$) of a simple type, a type scheme and a constraint is calculated by:

$$\begin{array}{rcl|lcl} \mathcal{F}(\kappa) & = & \emptyset & \mathcal{F}(\textbf{False}) & = & \emptyset \\ \mathcal{F}(\alpha) & = & \{\alpha\} & \mathcal{F}(\textbf{True}) & = & \emptyset \\ \mathcal{F}(\tau_1 \to \tau_2) & = & \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) & \mathcal{F}(\mathcal{L}(\alpha)) & = & \{\alpha\} \\ \mathcal{F}(\tau_1 * \tau_2) & = & \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) & \mathcal{F}(C_1 \Rightarrow C_2) & = & \mathcal{F}(C_1) \cup \mathcal{F}(C_2) \\ \mathcal{F}(\tau \ par) & = & \mathcal{F}(\tau) & \mathcal{F}(C_1 \wedge C_2) & = & \mathcal{F}(C_1) \cup \mathcal{F}(C_2) \\ \mathcal{F}(\forall \alpha_1 ... \alpha_n . [\tau / \mathcal{C}]) & = & (\mathcal{F}(\tau) \cup \mathcal{F}(C)) \setminus \{\alpha_1, ..., \alpha_n\} & & & \end{array}$$

$$\varphi ::= [\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n]$$

with $\{\alpha_1, \ldots, \alpha_n\}$ *is the domain (Dom) of* φ.

We generally note a substitution: $\varphi ::= [\alpha_i \leftarrow \tau_i]$. We write $\varphi \circ \varphi'$ for the composition of the substitution φ and φ'. The identity is written *id*. The application of a substitution φ to simple types and to constraints is defined by

$$\begin{array}{lcl|lcl}
\varphi(\kappa) & = & \kappa & \varphi(\textbf{True}) & = & \textbf{True} \\
\varphi(\alpha_i) & = & \tau_i \text{ if } \alpha_i \in Dom(\varphi) & \varphi(\textbf{False}) & = & \textbf{False} \\
\varphi(\gamma) & = & \gamma \text{ if } \gamma \notin Dom(\varphi) & \varphi(C_1 \wedge C_2) & = & \varphi(C_1) \wedge \varphi(C_2) \\
\varphi(\tau_a \to \tau_b) & = & \varphi(\tau_a) \to \varphi(\tau_b) & \varphi(C_1 \Rightarrow C_2) & = & \varphi(C_1) \Rightarrow \varphi(C_2) \\
\varphi(\tau_a * \tau_b) & = & \varphi(\tau_a) * \varphi(\tau_b) & \varphi(\mathcal{L}(\alpha_i)) & = & \mathcal{L}(\tau_i) \text{ if } \alpha_i \in Dom(\varphi) \\
\varphi(\tau\ par) & = & (\varphi(\tau)\ par) & \varphi(\mathcal{L}(\gamma)) & = & \mathcal{L}(\gamma) \text{ if } \gamma \notin Dom(\varphi)
\end{array}$$

With this definition we can define a substitution on a type scheme.

Definition 2 (Substitution on a type scheme)

$$\boxed{\varphi(\forall \alpha_1 \ldots \alpha_n.[\tau/C]) = \forall \alpha_1 \ldots \alpha_n.[\varphi(\tau)/\varphi(C) \bigwedge_{\beta_i \in Dom(\varphi) \cap \mathcal{F}(\tau/C)} C_{\varphi(\beta_i)}] \textit{ if } \alpha_1 \ldots \alpha_n \textit{ are out of reach of } \varphi}$$

We said that a variable α is out of reach of a substitution φ if

1. $\varphi(\alpha) = \alpha$, i.e φ don't modify α (or α is not in the domain of φ)
2. α is not free in $[\tau/C]$, then α is not free in $\varphi([\tau/C])$, i.e, φ do not introduce α in its result.

Remark: the condition that $\alpha_1 \ldots \alpha_n$ are out of reach of φ can always be validated by renaming first $\alpha_1 \ldots \alpha_n$ with fresh variables (we suppose that we have an infinite set of variables).

Remark: on the following, it is a misuse of language if we apply a substitution φ on types, on type schemes, on environments (see next section) and also on contraints. This is possible because the domain (variables of type) and the co-domain are the same (simple type).

2.2.2 Inductive rules of the type system

Instantiation and generalisation

A type scheme can be seen like the set of types given by instantiation of the quantifier variables. We introduce the notion of instance of a type scheme with contraints.

Definition 3 (Instantiation of a type scheme) $[\tau/C] \leq \forall \alpha_1 ... \alpha_n.[\tau'/C']$ *if and only if, there exists a substitution* φ *of domain* $\alpha_1, \ldots, \alpha_n$ *where:*

$$\boxed{\tau = \varphi(\tau') \quad \textit{and} \quad C = \varphi(C') \bigwedge_{\beta_i \in Dom(\varphi)} C_{\varphi(\beta_i)}}$$

We write E for an environment which associates type schemes to free variables of an expression. It is an application from free variables (identifiers) of expressions to type scheme:

$$E ::= \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$$

and $Dom(E) = \{x_1, \ldots, x_n\}$ for its domain, i.e the set of variables associated. We assume that all the identifiers are distinct. The empty mapping is written $\emptyset$ and $E(x)$ for the type scheme associated with x in E. The substitution φ on E is a point to point substitution on the domain of E:

$$\varphi(\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}) = \{x_1 : \varphi(\sigma_1), \ldots, x_n : \varphi(\sigma_n)\}$$

$$\mathcal{F}(E) = \bigcup_{x \in Dom(E)} \mathcal{F}(E(x))$$

Finally, we write $E + \{x : \sigma\}$ for the extension of E to the mapping of x to σ. If, before this operation, we have $x \in Dom(E)$, we can replace the range by the new type scheme for x. To continue with the introduction of the type system, we define how to construct a type scheme. Yet, type schemes have universal quantified variables, but not all the variables of a type scheme can.

Definition 4 (Generalisation of a type scheme) *Given an environment E, a type scheme τ without universal quantification and constraints, we define an operator Gen to introduce universal quantification:*

$$\boxed{Gen([\tau/C], E) = \forall \alpha_1 ... \alpha_n . [\tau/C] \text{ where } \{\alpha_1, ..., \alpha_n\} = \mathcal{F}(\tau) \setminus \mathcal{F}(E)}$$

With this definition, we have introduced polymorphism. The universal quantification gives the choice for the system to take the good type from a type scheme.

Inductive rules

We note TC (figure 2.2) the function which associates a type scheme to the constants and to the primitive operations. We formulate type inference by a deductive proof system that assigns a type to an expression of the language. The context in which an expression is associated with a type is represented by an environment which maps type scheme to identifiers. Deductions produce conclusions of the form $E \vdash e : [\tau/C]$ which are called typing judgments, they could be read as: "in the type environment E, the expression e has the type $[\tau/C]$". The static semantics manipulates type schemes by using the mechanism of generalisation and instantiation specified in the previous sections. Now the inductive rules of the type system are given in the Milner's style. In all the inductives rules if a constraint C is such that $Solve(C) = \textbf{False}$ then the inductive

$$\frac{[\tau/C] \leq E(x)}{E \vdash x : [\tau/C]}(Var) \qquad \frac{[\tau/C] \leq TC(c)}{E \vdash c : [\tau/C]}(Const) \qquad \frac{[\tau/C] \leq TC(op)}{E \vdash op : [\tau/C]}(Op)$$

$$\frac{E + \{x : [\tau_1/C_{\tau_1}]\} \vdash e : [\tau_2/C_2]}{E \vdash (\textbf{fun } x \rightarrow e) : [\tau_1 \rightarrow \tau_2/C_{(\tau_1 \rightarrow \tau_2)} \wedge C_2]}(Fun)$$

$$\frac{E \vdash e_1 : [\tau' \rightarrow \tau/C_1] \qquad E \vdash e_2 : [\tau'/C_2]}{E \vdash (e_1\ e_2) : [\tau/C_1 \wedge C_2]}(App)$$

$$\frac{E \vdash e_1 : [\tau_1/C_1] \qquad E + \{x : Gen([\tau_1/C_1], E)\} \vdash e_2 : [\tau_2/C_2]}{E \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : [\tau_2/C_1 \wedge C_2 \wedge \mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1)]}(Let)$$

$$\frac{E \vdash e_1 : [\tau_1/C_1] \qquad E \vdash e_2 : [\tau_2/C_2]}{E \vdash (e_1, e_2) : [\tau_1 * \tau_2/C_1 \wedge C_2]}(Couple)$$

Figure 2.1: The inductive rules

rule cannot be applied and then the expression is not well typed. To *Solve* the constraints we use the classical boolean reduct rules of propositional calculus and the rules to transform the locality of type to constraints. Our constraints are a sub part of the propositional calculus, so *Solve* is a decidable function (quine method, truth table etc.)

For the proof, we suppose that we work modulos these following equations that are natural for the $\wedge$ operators: $\textbf{True} \wedge C = C$, $C \wedge C = C$ and the commutativity of the $\wedge$ operator.

(elims tautologies with classical reduce rules like for example ($\mathbf{True} \wedge \varphi$) = φ) and gives more readable constraints like ($\mathcal{L}(\alpha) \Rightarrow$ **False**) $= \neg\mathcal{L}(\alpha)$. *Simple* does not change the semantics but it is used to give constraints more readable to users.

Afterwards, we need to know when a constraint is *Solved* to **True**, i.e., when a constraint could not be an incorrect constraint. It will be important, notably for the validity of the type system.

$TC(i)$	$=$	$int \quad i = 0, 1, \ldots$	
$TC(b)$	$=$	$bool \quad b =$ **true** or $b =$ **false**	
$TC(())$	$=$	$unit$	
$TC(+)$	$=$	$(int * int) \to int$	
$TC(\mathbf{fix})$	$=$	$\forall\alpha.[(\alpha \to \alpha) \to \alpha / \mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\alpha)]$	
$TC(\mathbf{fst})$	$=$	$\forall\alpha\beta.[(\alpha * \beta) \to \alpha / \mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]$	and its dual
$TC(\mathbf{snd})$	$=$	$\forall\alpha\beta.[(\alpha * \beta) \to \beta / \mathcal{L}(\beta) \Rightarrow \mathcal{L}(\alpha)]$	
$TC(\mathbf{ifthenelse})$	$=$	$\forall\alpha.[(bool * \alpha * \alpha) \to \alpha / \mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\alpha)]$	
$TC(\mathbf{mkpar})$	$=$	$\forall\alpha.[(int \to \alpha) \to (\alpha\ par) / \mathcal{L}(\alpha)]$	
$TC(\mathbf{apply})$	$=$	$\forall\alpha\beta.[((\alpha \to \beta)\ par * (\alpha\ par)) \to (\beta\ par) / \mathcal{L}(\alpha) \wedge \mathcal{L}(\beta)]$	
$TC(\mathbf{ifat})$	$=$	$\forall\alpha.[(int * (bool\ par) * \alpha * \alpha) \to \alpha / \mathcal{L}(\alpha) \Rightarrow \mathbf{False}]$	
$TC(\mathbf{put})$	$=$	$\forall\alpha.[(int \to \alpha)\ par \to (int \to \alpha)\ par / \mathcal{L}(\alpha)]$	
$TC(\mathbf{nc})$	$=$	$\forall\alpha.[unit \to \alpha / \mathcal{L}(\alpha) \Rightarrow \mathbf{True}]$	
$TC(\mathbf{isnc})$	$=$	$\forall\alpha.[\alpha \to bool / \mathcal{L}(\alpha)]$	

Figure 2.2: Definition of TC

Definition 5 (The trues constraints) *We wrote $\varphi \models C$, if the substitution φ on the free variables of C have the two following properties*

1. $\mathcal{F}(\varphi(C)) = \emptyset$.
2. $Solve(\varphi(C)) =$ ***True***.

We also wrote the set $\phi_C = \{\varphi \mid \varphi \models C\}$, to designe all the substitution that have these properties and ϕ_C^L (respectively ϕ_C^G) for the set of substitution where their co-domains are in L (respectively G). Example:

$$[\alpha \leftarrow int, \beta \leftarrow (int \to int)] \models \mathcal{L}(\alpha) \wedge \mathcal{L}(\beta)$$

Definition 6 (Local and global expressions) *Give $E \vdash e : [\tau / C]$. e is said:*

1. *local when $\tau \in L$ or ($\tau \in V$ and $\phi_C^L \neq \emptyset$)*
2. *global when $\tau \in G$ or ($\tau \in V$ and $\phi_C^G \neq \emptyset$)*

and totally polymorph when the two cases are possible.

Examples:

- (**fun** $f \to$ **mkpar** f) is global.
- (**fun** $x \to x$) is totally polymorph.

Remark: Those definitions are decidable (propositional calculus)

Examples

For the example given at the beginning of this section, the type system calculates that `replicate` has the type scheme: $\forall\alpha.[\alpha \to (\alpha\ par) / \mathcal{L}(\alpha)]$ and (**mkpar** (**fun** $i \to 5$)) has type ($int\ par$). So after (*App*) and (*Op*) rules, the substitution is $\alpha = (int\ par)$ and the constraint is *Solved* to **False**, so this expression is not well-typed.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{int \leq\ int}{\{a : int, b : int\ par\} \vdash a : int}}{\{b : int\ par\} \vdash \textbf{fun}\ b \to a : [(int\ par) \to int / C_{(int\ par) \to int}]}}{\vdash \textbf{fun}\ a \to \textbf{fun}\ b \to a : ?} \quad \dfrac{int \leq TC(1)}{\vdash 1 : int}}{\vdash (\textbf{fun}\ a \to \textbf{fun}\ b \to a)\ 1 : ?} \quad \dfrac{\ldots}{\vdash: (mkpar\ (\textbf{fun}\ i \to 5)) : int\ par}}{\vdash ((\textbf{fun}\ a \to \textbf{fun}\ b \to a)\ 1)(mkpar\ (\textbf{fun}\ i \to 5)) : ?}$$

Effectively, $C = C_{(int\ par) \to int} = \mathcal{L}(int\ par) \wedge \ldots = \textbf{False} \wedge \ldots$ so $Solve(C) = \textbf{False}$ and this expression is not well-typed.

Remark: the importance in (Let), (App) to have $C_1 \wedge C_2$ comes that in C_1 we can have, for example, that $\mathcal{L}(\alpha)$ and $\neg\mathcal{L}(\alpha)$ in C_2. If we forget one of these constraints, we don't secure that a variable can not be either global and local. Two examples:

$$\begin{array}{l} \textbf{fun}\ x \to \textbf{let}\ y = (\textbf{mkpar}\ (\textbf{fun}\ a \to x))\ \textbf{in}\ (\textbf{fun}\ b \to x)(\textbf{mkpar}\ (\textbf{fun}\ a \to 5)) \\ \textbf{fun}\ x \to (\textbf{fun}\ y \to (\textbf{mkpar}\ (\textbf{fun}\ a \to x)))\ ((\textbf{fun}\ b \to x)(\textbf{mkpar}\ (\textbf{fun}\ a \to 5))) \end{array}$$

In these two cases, we have x that has the type *int par*. For the two cases, we have the sub-expression: **mkpar** (**fun** $a \to x$). But the nesting of `par` types is prohibited so it is not well-typed.

Remark: in (Let), we introduce the fact that $\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1)$ because an expression like **let** $x = e_1$ **in** e_2 can be seen like (**fun** $x \to e_2$) e_1. So we have to protect our type system against expression from global values to local values like:

$$\textbf{let}\ x = \textbf{mkpar}(\textbf{fun}\ i \to 5)\ \textbf{in}\ 8$$

that have the type *int* for Objective Caml, but is rejected by our type system. We will formalise these intuitions in the next sections.

Chapter 3

Technical lemmas

In this Chapter, we present and prove some technical lemmas that will serve in further chapters. In the first section, we present some technical lemmas of the constraints for the *Solve* operation and lemmas about free variables on type and constraints. After, we give lemmas about the substitution on constraints and a proposition on the generalisation. In the third section, we give a relationship between our objects and we end with a very important technical lemma: the stability of substitution that will use next, particulary in the inference algorithm.

3.1 Construction of the constraints

3.1.1 The correct constraints

Lemma 1 (*Solve* and $\wedge$) *If $Solve(C_1 \wedge C_2) \neq$ **False** then $Solve(C_1) \neq$ **False** and $Solve(C_2) \neq$ **False**.*

Proof: *Solve* uses classical boolean reduction rules of propositional calculus so to have in this calculus $v_1 \wedge v_2 \neq$ **False** you need $v_1 \neq$ **False** and $v_2 \neq$ **False** (see truth table) hence the result. $\square$

Lemma 2 (typing and $\models$) *if $E \vdash e : [\tau/C]$ then there exists $\varphi \models C$.*

Proof: by definition of the constraints (propositional calculus), we have $Solve(C) \neq$ **False** and the result. $\square$

Lemma 3 ($\models$ and *Solve*) *There exists $\varphi \models C$ if and only if $Solve(C) \neq$ **False**.*

Lemma 4 ($\models$ and $\wedge$) *if $\varphi \models C$ and $\varphi \models C'$ then $\varphi \models C \wedge C'$.*

Proof (of the 2 lemmas): by definition of the propositional calculus. $\square$

Lemma 5 (Solve and locality) *if $Solve(\mathcal{L}(\tau)) \neq$ **False** then $\tau \notin G$ (i.e. $\tau \in V$)*

Proof: by structural induction on τ.

Case $\tau = \kappa$. We have $\mathcal{L}(\tau) =$ **True** with $\tau \in V$ and the result.

Case $\tau = \alpha$. We have $Solve(\mathcal{L}(\tau)) \neq$ **False** with $\tau \in V$ and the result.

Case $\tau = \tau_1 \to \tau_2$. We have $Solve(\mathcal{L}(\tau)) = Solve(\mathcal{L}(\tau_1) \wedge \mathcal{L}(\tau_2))$. So by the Lemma 1 we have that $Solve(\mathcal{L}(\tau_1)) \neq$ **False** (idem for τ_2). So by induction we have $\tau_1 \in V$ and $\tau_2 \in V$ and the result.

Case $\tau = \tau_1 * \tau_2$. Idem.

Case $\tau = \tau_1\ par$. We have $Solve(\mathcal{L}(\tau)) = Solve(\textbf{False} \wedge \mathcal{L}(\tau_1)) =$ **False** and the result. $\square$

Lemma 6 (Locality is local) *We have:*

- *if $\tau \in L$ then $Solve(\mathcal{L}(\tau)) =$ **True**.*
- *if $\tau \in G$ then $Solve(\mathcal{L}(\tau)) =$ **False**.*

Lemma 7 (Solve and Type Constraints) *If $Solve(C_\tau) \neq$ **False** then $\tau \in G \cup V$.*

Proof: by structural induction on τ.

Case $\tau = \kappa$. We have $C_\tau =$ **True** and $\tau \in L$.

Case $\tau = \alpha$. We have $C_\tau =$ **True** and $\tau \in L$.

Case $\tau = \tau_1 \rightarrow \tau_2$. We have $C_\tau = C_{\tau_1} \wedge C_{\tau_2} \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))$. By induction, we have $\tau_1 \in G \cup V$ and $\tau_2 \in G \cup V$. To have that $\tau \in G \cup V$, we have to proof that the case $\tau_2 \in L$ and $\tau_1 \in G$ is impossible by contradiction that $Solve((\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))) \neq$ **False** (Lemma 1). By Lemma 6, if $\tau_2 \in L$ and $\tau_1 \in G$ then we have $Solve(\mathcal{L}(\tau_2)) =$ **True** and $Solve(\mathcal{L}(\tau_1)) =$ **False**. So we have $Solve((\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))) =$ **False**, the contradiction and the result.

Case $\tau = (\tau_1\ par)$. We have $C_\tau = C_{\tau_1} \wedge \mathcal{L}(\tau_1)$. By induction, we have that $\tau_1 \in G \cup V$. By the Lemma 1, we have $Solve(\mathcal{L}(\tau_1)) \neq$ **False** and by the Lemma 5, we get $\tau_1 \in V$ and the result.

Case $\tau = \tau_1 * \tau_2$. We have $C_\tau = C_{\tau_1} \wedge C_{\tau_2}$. By induction, we have $C_{\tau_1} \in G \cup V$ (idem for τ_2) and the result.

$\square$

3.1.2 Relationship between free variables on constraints and types

Lemma 8 (Free variables of a type and its locality) $\mathcal{F}(\mathcal{L}(\tau)) = \mathcal{F}(\tau)$

Proof: by structural induction on τ.

Case (base type): we have $\tau \in \kappa$, so $\mathcal{F}(\tau) = \emptyset$ and $\mathcal{F}(\mathcal{L}(\tau)) = \mathcal{F}(\textbf{True}) = \emptyset$ hence the result.

Case (variable): we have $\tau = \alpha$, so $\mathcal{F}(\tau) = \{\alpha\} = \mathcal{F}(\mathcal{L}(\alpha))$ hence the result.

Case (type of function): we have $\tau = \tau_a \rightarrow \tau_b$ so $\mathcal{F}(\tau) = \mathcal{F}(\tau_a) \cup \mathcal{F}(\tau_b)$ and $\mathcal{F}(\mathcal{L}(\tau)) = \mathcal{F}(\mathcal{L}(\tau_a) \wedge \mathcal{L}(\tau_b)) = \mathcal{F}(\mathcal{L}(\tau_a)) \cup \mathcal{F}(\mathcal{L}(\tau_b))$. So by induction, we obtain that $\mathcal{F}(\mathcal{L}(\tau_a)) = \mathcal{F}(\tau_a)$ and $\mathcal{F}(\mathcal{L}(\tau_b)) = \mathcal{F}(\tau_b)$. By the definition of the $\cup$ operator on sets we deduce the result.

Case (Couple): we have $\tau = \tau_a * \tau_b$ so $\mathcal{F}(\tau) = \mathcal{F}(\tau_a) \cup \mathcal{F}(\tau_b)$ and $\mathcal{F}(\mathcal{L}(\tau)) = \mathcal{F}(\mathcal{L}(\tau_a) \ \wedge \mathcal{L}(\tau_b)) = \mathcal{F}(\mathcal{L}(\tau_a)) \cup \mathcal{F}(\mathcal{L}(\tau_b))$. So by induction, we have $\mathcal{F}(\mathcal{L}(\tau_a)) = \mathcal{F}(\tau_a)$ and $\mathcal{F}(\mathcal{L}(\tau_b)) = \mathcal{F}(\tau_b)$ hence the result.

Case (parallel vector): we have $\tau = (\tau_1\ par)$ so $\mathcal{F}(\tau) = \mathcal{F}(\tau_1)$ and
$\mathcal{F}(\mathcal{L}(\tau)) = \mathcal{F}(\mathcal{L}(\tau_1\ par)) = \mathcal{F}(\textbf{False} \wedge \mathcal{L}(\tau_1)) = \mathcal{F}(\mathcal{L}(\tau_1))$ so by induction we deduce the result.

$\square$

Lemma 9 (Free variables of a type and type constraints) $\mathcal{F}(C_\tau) \subseteq \mathcal{F}(\tau)$

Example: $\tau = (\alpha, \beta)$ and $C_\tau =$ **True** so $\mathcal{F}(\tau) = \{\alpha, \beta\}$ and $\mathcal{F}(C_\tau) = \emptyset$.

Proof: by strutural induction on τ.

Case (base type): we have $\tau \in \kappa$ so $C_\tau = C_\kappa =$ **True**, so $\mathcal{F}(C_\tau) = \emptyset = \mathcal{F}(\tau)$ and the result.

Case (variable): we have $\tau = \alpha$ so $C_\tau = C_\alpha =$ **True**, so $\mathcal{F}(C_\tau) = \emptyset \subseteq \{\alpha\} = \mathcal{F}(\tau)$ and the result.

Case (type of function): we have $\tau = \tau_a \rightarrow \tau_b$, so $C_\tau = C_{\tau_a \rightarrow \tau_b} = C_{\tau_a} \wedge C_{\tau_b} \wedge \mathcal{L}(\tau_b) \Rightarrow \mathcal{L}(\tau_a)$ so $\mathcal{F}(C_\tau) = \mathcal{F}(C_{\tau_a}) \cup \mathcal{F}(C_{\tau_b}) \cup \mathcal{F}(\mathcal{L}(\tau_a)) \cup \mathcal{F}(\mathcal{L}(\tau_b))$ and $\mathcal{F}(\tau) = \mathcal{F}(\tau_a) \cup \mathcal{F}(\tau_b)$. By induction, we obtain that

We have the result by definition of $\mathcal{C}$.

Case (Couple): we have $\tau = \tau_a * \tau_b$ so $C_\tau = C_{\tau_a * \tau_b} = C_{\tau_a} \wedge C_{\tau_b}$ so $\mathcal{F}(C_\tau) = \mathcal{F}(C_{\tau_a}) \cup \mathcal{F}(C_{\tau_b})$ and $\mathcal{F}(\tau) = \mathcal{F}(\tau_a) \cup \mathcal{F}(\tau_b)$. By induction, we obtain that $\mathcal{F}(C_{\tau_a}) \subseteq \mathcal{F}(\tau_a)$ and also $\mathcal{F}(C_{\tau_b}) \subseteq \mathcal{F}(\tau_b)$ and the result.

Case (parallel vector): we have $\tau = (\tau_1\ par)$ so $C_\tau = C_{(\tau_1\ par)} = \mathcal{L}(\tau_1) \wedge C_{\tau_1}$ so $\mathcal{F}(C_\tau) = \mathcal{F}(\mathcal{L}(\tau_1)) \cup \mathcal{F}(C_{\tau_1})$ and $\mathcal{F}(\tau) = \mathcal{F}(\tau_1)$. By the Lemma 8, we get $\mathcal{F}(\mathcal{L}(\tau_1)) = \mathcal{F}(\tau_1)$ and by induction $\mathcal{F}(C_{\tau_1}) \subseteq \mathcal{F}(\tau_1)$, hence the result.

□

3.2 Substitution

3.2.1 Substitution and *Gen*

Proposition 1 (Commutativity between *Gen* and a substitution) *Given an environment E, a type scheme without universal quantification $[\tau/C]$ and a substitution φ such as all the generalisable variables of $\mathcal{F}([\tau/C]) \setminus \mathcal{F}(E)$ are all out of reach of φ. Then $Gen(\varphi([\tau/C]), \varphi(E)) = \varphi(Gen([\tau/C], E))$.*

<u>Proof</u>: we have $\mathcal{F}(C_{\varphi(\beta_i)}) \subseteq \mathcal{F}(\varphi(\beta_i))$ (Lemma 9) for all $\beta_i \in Dom(\varphi)$ so it is easy to see that a variable α out of reach of φ is free in $\varphi([\tau/C])$ if and only if it is free in $[\tau/C]$. Thus $\mathcal{F}(\varphi([\tau/C]))\setminus\mathcal{F}(\varphi(E)) = \mathcal{F}([\tau/C])\setminus\mathcal{F}(E)$ and the result.□

3.2.2 Relationship between types, constraints and substitution

Lemma 10 (Locality and substitution) *Given a substitution $\varphi = [\alpha_i \leftarrow \tau_i]$, a type τ, then $\varphi(\mathcal{L}(\tau)) = \mathcal{L}(\varphi(\tau))$.*

<u>Proof</u>: by structural induction on τ.

Case (base type): we have $\tau \in \kappa$. $\mathcal{L}(\tau) = \textbf{True}$ so $\varphi(\mathcal{L}(\tau)) = \textbf{True}$ and $\varphi(\tau) = \tau$ so $\mathcal{L}(\varphi(\tau)) = \mathcal{L}(\tau) = \textbf{True}$ hence the result.

Case (variable): we have two cases:

1. $\tau = \gamma$ with $\gamma \neq \alpha_i$, we have $\varphi(\mathcal{L}(\tau)) = \mathcal{L}(\tau)$ and $\mathcal{L}(\varphi(\tau)) = \mathcal{L}(\tau)$.
2. $\tau = \alpha_i$, so $\varphi(\mathcal{L}(\alpha_i)) = \mathcal{L}(\tau_i)$ and $\mathcal{L}(\varphi(\alpha_i)) = \mathcal{L}(\tau_i)$

hence the result.

Case (type of function): we have $\tau = \tau_a \rightarrow \tau_b$ so

$$\begin{array}{rcl} \varphi(\mathcal{L}(\tau_a \rightarrow \tau_b)) & = & \varphi(\mathcal{L}(\tau_a) \wedge \mathcal{L}(\tau_b)) \\ & & \text{(by definition)} \\ & = & \varphi(\mathcal{L}(\tau_a)) \wedge \varphi(\mathcal{L}(\tau_b)) \\ & & \text{(by induction)} \\ & = & \mathcal{L}(\varphi(\tau_a)) \wedge \mathcal{L}(\varphi(\tau_b)) \end{array}$$

$$\begin{array}{rcl} \mathcal{L}(\varphi(\tau_a \rightarrow \tau_b)) & = & \mathcal{L}(\varphi(\tau_a) \rightarrow \varphi(\tau_b)) \\ & & \text{(by definition)} \\ & = & \mathcal{L}(\varphi(\tau_a)) \wedge \mathcal{L}(\varphi(\tau_b)) \end{array}$$

hence the result

Case (Couple): we have $\tau = \tau_a * \tau_b$ so

$$\begin{array}{rcl} \varphi(\mathcal{L}(\tau_a * \tau_b)) & = & \varphi(\mathcal{L}(\tau_a) \wedge \mathcal{L}(\tau_b)) \\ & & \text{(by definition)} \\ & = & \varphi(\mathcal{L}(\tau_a)) \wedge \varphi(\mathcal{L}(\tau_b)) \\ & & \text{(by induction)} \\ & = & \mathcal{L}(\varphi(\tau_a)) \wedge \mathcal{L}(\varphi(\tau_b)) \end{array}$$

$$\begin{array}{rcl} \mathcal{L}(\varphi(\tau_a * \tau_b)) & = & \mathcal{L}(\varphi(\tau_a) * \varphi(\tau_b)) \\ & & \text{(by definition)} \\ & = & \mathcal{L}(\varphi(\tau_a)) \wedge \mathcal{L}(\varphi(\tau_b)) \end{array}$$

hence the result.

and $\mathcal{L}(\varphi(\tau_1\ par)) = \mathcal{L}(\varphi(\tau_1)\ par) = \mathbf{False} \wedge \mathcal{L}(\varphi(\tau_1))$, by induction, we have $\varphi(\mathcal{L}(\tau_1)) = \mathcal{L}(\varphi(\tau_1))$ hence the result.

□

Lemma 11 (Constraints and substitution) *Given a type τ and a substitution $\varphi = [\alpha_i \leftarrow \tau_i]$ then*

$$C_{\varphi(\tau)} = \varphi(C_\tau) \bigwedge_{\alpha_i \in \mathcal{F}(\tau)\cap Dom(\varphi)} C_{\varphi(\alpha_i)}$$

Case (base type): we have $\tau \in \kappa$, $C_\tau = \mathbf{True}$ so $\varphi(C_\tau) = \mathbf{True}$ and $\varphi(\tau) = \tau$ so $C_{\varphi(\tau)} = C_\tau = \mathbf{True}$. Because $\mathcal{F}(\tau) = \emptyset$ we deduce the result.

Case (variable): we have two cases:

1. $\tau = \gamma$ $(\gamma \neq \alpha_i)$. We have $\varphi(C_\tau) = \varphi(\mathbf{True}) = \mathbf{True}$ and $C_{\varphi(\tau)} = C_\tau = \mathbf{True}$. We have $\mathcal{F}(\tau) = \{\gamma\}$ so $\mathcal{F}(\tau) \cap Dom(\varphi) = \emptyset$ and we deduct the result.
2. $\tau = \alpha_i$. We have $C_\tau = \mathbf{True}$ so $\varphi(C_\tau) = \mathbf{True}$ and $C_{\varphi(\alpha_i)} = C_{\tau_i}$. We have $\mathcal{F}(\tau) \cap Dom(\varphi) = \{\alpha_i\}$ and we deduce the result.

Case (functional) $\tau = \tau_a \to \tau_b$: we have

$$\begin{array}{rcll}
\varphi(C_\tau) & = & \varphi(C_{\tau_a} \wedge C_{\tau_b} \wedge (\mathcal{L}(\tau_b) \Rightarrow \mathcal{L}(\tau_a))) & \text{by definition of a substitution} \\
& = & \varphi(C_{\tau_a}) \wedge \varphi(C_{\tau_b}) \wedge \varphi(\mathcal{L}(\tau_b)) \Rightarrow \varphi(\mathcal{L}(\tau_a)) & \text{by the Lemma 8} \\
& = & \varphi(C_{\tau_a}) \wedge \varphi(C_{\tau_b}) \wedge (\mathcal{L}(\varphi(\tau_b)) \Rightarrow \mathcal{L}(\varphi(\tau_a))) & \text{by induction} \\
& = & C_{\varphi(\tau_a)} \bigwedge_{\alpha_i \in \mathcal{F}(\tau_a)\cap Dom(\varphi)} C_{\varphi(\alpha_i)} \wedge C_{\varphi(\tau_b)} \wedge \bigwedge_{\alpha_i \in \mathcal{F}(\tau_b)\cap Dom(\varphi)} C_{\varphi(\alpha_i)} & \\
& & \wedge(\mathcal{L}(\varphi(\tau_b)) \Rightarrow \mathcal{L}(\varphi(\tau_a))) & \text{property of } \wedge \text{ and } \cap \\
& = & C_{\varphi(\tau_a)} \bigwedge_{\alpha_i \in (\mathcal{F}(\tau_a)\cup\mathcal{F}(\tau_b))\cap Dom(\varphi)} C_{\varphi(\alpha_i)} \wedge & \\
& & C_{\varphi(\tau_b)} \wedge (\mathcal{L}(\varphi(\tau_b)) \Rightarrow \mathcal{L}(\varphi(\tau_a))) & \text{by definition} \\
& = & C_{\varphi(\tau)} \bigwedge_{\alpha_i \in ((\mathcal{F}(\tau_a)\cup\mathcal{F}(\tau_b))\cap Dom(\varphi))} C_{\varphi(\alpha_i)} &
\end{array}$$

By definition, we have $\mathcal{F}(\tau_a \to \tau_b) = \mathcal{F}(\tau_a) \cup \mathcal{F}(\tau_b)$ and the result.

Case (parallel vector). $\tau = (\tau_1\ par)$: we have $C_\tau = \mathcal{L}(\tau_1) \wedge C_{\tau_1}$ so $\varphi(C_\tau) = \varphi(\mathcal{L}(\tau_1)) \wedge \varphi(C_{\tau_1})$ and $C_{\varphi(\tau)} = C_{\varphi(\tau_1\ par)} = C_{\varphi(\tau_1)\ par} = \mathcal{L}(\varphi(\tau_1)) \wedge C_{\varphi(\tau_1)}$. With the Lemma 10, we have $\varphi(\mathcal{L}(\tau_1)) = \mathcal{L}(\varphi(\tau_1))$. By definition, we have $\mathcal{F}(\tau) = \mathcal{F}(\tau_1)$ so by induction, we get $C_{\varphi(\tau_1)} = \varphi(C_{\tau_1}) \bigwedge_{\beta_i \in \mathcal{F}(\tau_1)\cap Dom(\varphi)} C_{\varphi(\beta_i)}$ and the result.

Case (Couple). $\tau = (\tau_a * \tau_b)$: we have $\varphi(C_\tau) = \varphi(C_{\tau_a} \wedge C_{\tau_b}) = \varphi(C_{\tau_a}) \wedge \varphi(C_{\tau_b})$ and $C_{\varphi(\tau)} = C_{\varphi(\tau_a)} \wedge C_{\varphi(\tau_b)}$. By induction we get $C_{\varphi(\tau_a)} = \varphi(C_{\tau_a}) \bigwedge_{\alpha_i \in \mathcal{F}(\tau_a)\cap Dom(\varphi)} C_{\varphi(\alpha_i)}$ and $C_{\varphi(\tau_b)} = \varphi(C_{\tau_b}) \bigwedge_{\alpha_i \in \mathcal{F}(\tau_b)\cap Dom(\varphi)} C_{\varphi(\alpha_i)}$. By definition $\mathcal{F}(\tau) = \mathcal{F}(\tau_a) \cup \mathcal{F}(\tau_b)$ so like in case (functional) we have the result.

□

3.3 Relation between our objects

Lemma 12 (Relationship between contraints and types) *Given an environment E such as $\forall x \in Dom(E)$, $x : \forall \alpha_1, \ldots, \alpha_n.[\tau_0/C_0]$ then $C_0 = C_{\tau_0} \wedge C'_0$. Given an expression e such as $E \vdash e : [\tau/C]$ then $C = C_\tau \wedge C'$.*

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\ldots}{E \vdash (\mathbf{fun}\ a \to \mathbf{fun}\ b \to a) : [\alpha \to (\beta \to \alpha)/\mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]} \quad \dfrac{\alpha \leq \alpha}{E \vdash x : \alpha}}{E \vdash (\mathbf{fun}\ a \to \mathbf{fun}\ b \to a)x : [\beta \to \alpha/\mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]}}{E \vdash (\mathbf{fun}\ a \to \mathbf{fun}\ b \to a)x\ y : [\beta \to \alpha/\mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]} \quad \dfrac{\beta \leq \beta}{E \vdash y : \beta}}{x : \alpha, y : \beta \vdash (\mathbf{fun}\ a \to \mathbf{fun}\ b \to a)x\ y : [\alpha/\mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]}}{x : \alpha \vdash \mathbf{fun}\ y \to\ (\mathbf{fun}\ a \to \mathbf{fun}\ b \to a)x\ y : [\beta \to \alpha/\mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]}}{\vdash \mathbf{fun}\ x \to \mathbf{fun}\ y \to\ (\mathbf{fun}\ a \to \mathbf{fun}\ b \to a)x\ y : [\alpha \to (\beta \to \alpha)/\mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]}$$

Case (Var). We have the following derivation:

$$\frac{[\tau/C] \leq E(x)}{E \vdash x : [\tau/C]}(Var)$$

By hypothesis $E(x) = \forall\alpha_1, \ldots, \alpha_n.[\tau_0/C_{\tau_0} \wedge C'_0]$ and there exists a substitution ψ as $\tau = \psi(\tau_0)$ and $C = \psi(C_{\tau_0}) \wedge \psi(C'_0) \bigwedge_{\beta_i \in Dom(\psi)} C_{\psi(\beta_i)}$ with $Dom(\psi) = \{\alpha_1, \ldots, \alpha_n\}$. By the Lemma 9 we have $\mathcal{F}(C_{\tau_0}) \in \mathcal{F}(\tau)$ and so by the Lemma 11, we have

$$\begin{aligned} \psi(C_{\tau_0}) \bigwedge_{\beta_i \in \mathcal{F}(\tau_0/C_{\tau_0}) \cap Dom(\psi)} C_{\psi(\beta_i)} &= \psi(C_{\tau_0}) \bigwedge_{\beta_i \in Dom(\psi)} C_{\psi(\beta_i)} \\ &= C_{\psi(\tau_0)} \end{aligned}$$

hence the result.

Case $(Const)$, (Op). We have the following derivation:

$$\frac{[\tau/C] \leq TC(c)}{E \vdash c : [\tau/C]}(Const) \qquad \frac{[\tau/C] \leq TC(op)}{E \vdash op : [\tau/C]}(Op)$$

We verify trivially that for all typed τ operator, C_τ are the constraints given by TC and by application of the Lemma 11 we have the result. It is easy to see that the constants have the type, $\tau \in \kappa$ (for example int, bool, unit, ...), So, for the constants, we have $C_\tau = \textbf{True}$ and the result.

Case (App). We have the following derivation:

$$\frac{E \vdash e_1 : [\tau' \rightarrow \tau/C_1] \quad E \vdash e_2 : [\tau'/C_2]}{E \vdash (e_1\ e_2) : [\tau/C_1 \wedge C_2]}(App)$$

Apply the inductive hypothesis to the first premise, we obtain that $C_1 = C_{\tau' \rightarrow \tau} \wedge C'$. By construction, $C_{\tau' \rightarrow \tau} = C_\tau \wedge C_{\tau'} \wedge (\mathcal{L}(\tau) \Rightarrow \mathcal{L}(\tau'))$ hence the result.

Case (Let). We have the following derivation:

$$\frac{E \vdash e_1 : [\tau_1/C_1] \qquad E + \{x : Gen([\tau_1/C_1], E)\} \vdash e_2 : [\tau_2/C_2]}{E \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : [\tau_2/C_1 \wedge C_2 \wedge C_1 \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))]}(Let)$$

Like in (App), the result comes of the application of the induction hypothesis to the premises and we introduce constraints as we want.

Case (Fun). We have the following derivation:

$$\frac{E + \{x : [\tau_1/C_{\tau_1}]\} \vdash e_1 : [\tau_2/C_2]}{E \vdash (\textbf{fun}\ x \rightarrow e_1) : [\tau_1 \rightarrow \tau_2/C_{(\tau_1 \rightarrow \tau_2)} \wedge C_2]}(Fun)$$

The result is trival by construction and application of the induction hypothesis to the premise and we introduce constraints as we want.

Case $(Couple)$. We have the following derivation:

$$\frac{E \vdash e_1 : [\tau_1/C_1] \qquad E \vdash e_2 : [\tau_2/C_2]}{E \vdash (e_1, e_2) : [\tau_1 * \tau_2/C_1 \wedge C_2]}(Couple)$$

We can apply the induction hypothesis to the two premises. We obtain that $C_1 = C_{\tau_1} \wedge C'$ and $C_2 = C_{\tau_2} \wedge C''$ hence the result.

$\square$

Lemma 13 (Stability of typing per substitution) *Given an expression e, an environment E such as $E \vdash e : [\tau/C]$ and $\varphi = [\alpha_i \leftarrow \tau_i]$ a substitution as $Solve(\varphi(C) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau/C)} C_{\varphi(\beta_i)}) \neq$ **False** then $\varphi(E) \vdash e : \varphi([\tau/C])$*

Example: for the type derivation $\{y : \beta, x : \alpha\} \vdash ((\textbf{fun}\ a \rightarrow \textbf{fun}\ b \rightarrow a)\ x)\ y : [\alpha/\mathcal{L}(\alpha) \rightarrow \mathcal{L}(\beta)]$, if we apply the substitution $\varphi = \{\alpha \leftarrow int, \beta \leftarrow (int\ par)\}$ to E and to the type scheme, it is easy to see that we have the constraints *Solve* to **False**, and now, it is not a correct derivation. A substitution needs some properties before being applied to a type derivation.

Proof: by structural induction on e.

Case $e = x$. We have $E \vdash x : [\tau/C]$. So $[\tau/C] \leq E(x)$ with $E(x) = \forall\alpha_1, \ldots, \alpha_n.[\tau_0/C_0]$. After renaming, if necessary, we can suppose that all the α_i are out of reach of φ. Given ψ a substitution, of domain $\{\alpha_1, \ldots, \alpha_n\}$ as $\tau = \psi(\tau_a)$, $C = \psi(C_0) \bigwedge_{\beta_i \in Dom(\psi)} C_{\psi(\beta_i)}$ by definition 2 and $Solve(C) \neq$ **False**. (by definition of the (Var) rule). We have:

$$\begin{array}{rcll}
\varphi(E(x)) & = & \varphi(\forall\alpha_1, \ldots, \alpha_n.[\tau_0/C_0]) & \varphi \text{ is out of reach so} \\
& = & \forall\alpha_1, \ldots, \alpha_n.[\varphi(\tau_0)/\varphi(C_0) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau_0/C_0)} C_{\varphi(\beta_i)}] &
\end{array}$$

Given θ a substitution, with domain $\{\alpha_1, \ldots, \alpha_n\}$ as $\theta(\alpha_i) = \varphi(\psi(\alpha_i))$, we have:

$$\begin{array}{rcccl}
\theta(\varphi(\alpha_i)) & = & \theta(\alpha_i) & = & \varphi(\psi(\alpha_i)) \quad \text{for all } \alpha_i \\
\theta(\varphi(\beta)) & = & \varphi(\beta) & = & \varphi(\psi(\beta)) \quad \text{for all } \beta \text{ distinct of } \alpha_i
\end{array}$$

and

$$\begin{array}{rcccl}
\theta(\varphi(\mathcal{L}(\alpha_i))) & = & \theta(\mathcal{L}(\alpha_i)) & = & \varphi(\psi(\mathcal{L}(\alpha_i))) \quad \text{for all } \alpha_i \\
\theta(\varphi(\mathcal{L}(\beta))) & = & \varphi(\mathcal{L}(\beta)) & = & \varphi(\psi(\mathcal{L}(\beta))) \quad \text{for all } \beta \text{ distinct of } \alpha_i
\end{array}$$

So $\theta(\varphi(\tau_0)) = \varphi(\psi(\tau_0)) = \varphi(\tau)$.
Given $[\varphi(\tau)/C'] \leq \varphi(E(x))$. We have:

$$\begin{array}{rcll}
C' & = & \theta(\varphi(C_0) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau_0/C_0)} C_{\varphi(\beta_i)}) \bigwedge_{\beta_i \in Dom(\theta)} C_{\theta(\beta_i)} & \text{by definition and } Dom(\theta) = Dom(\psi) \\
& = & \theta(\varphi(C_0)) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau_0/C_0)} \theta(C_{\varphi(\beta_i)}) \bigwedge_{\beta_i \in Dom(\psi)} C_{\varphi(\psi(\beta_i))} & \text{like for the simple type} \\
& = & \varphi(\psi(C_0)) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau_0/C_0)} \theta(C_{\varphi(\beta_i)}) \bigwedge_{\beta_i \in Dom(\psi)} C_{\varphi(\psi(\beta_i))} & \\
& & \text{the } \alpha_i \text{ is out of reach of } \varphi \text{ and } \mathcal{F}(C_{\varphi(\beta_i)}) \subseteq \mathcal{F}(\varphi(\beta_i)) \text{ (Lemma 9)} & \\
& = & \varphi(\psi(C_0)) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau_0/C_0)} C_{\varphi(\beta_i)} \bigwedge_{\beta_i \in Dom(\psi)} C_{\varphi(\psi(\beta_i))} & \text{by the Lemma 11} \\
& = & \varphi(\psi(C_0)) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau_0/C_0)} C_{\varphi(\beta_i)} \bigwedge_{\beta_i \in Dom(\psi)} (\varphi(C_{\psi(\beta_i)}) \bigwedge_{\gamma_i \in \mathcal{F}(\psi(\beta_i))\cap Dom(\varphi)} C_{\varphi(\gamma_i)}) & \\
& = & \varphi(\psi(C_0)) \bigwedge_{\beta_i \in Dom(\psi)} \varphi(C_{\psi(\beta_i)}) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau_0/C_0)} C_{\varphi(\beta_i)} \bigwedge_{\beta_i \in Dom(\psi)} \bigwedge_{\gamma_i \in \mathcal{F}(\psi(\beta_i))\cap Dom(\varphi)} C_{\varphi(\gamma_i)} &
\end{array}$$

So by definition of C and to be more readable we have:

$$\begin{array}{rcll}
C' & = & \varphi(C) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau_0/C_0)} C_{\varphi(\beta_i)} \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\psi(\alpha_i))} C_{\varphi(\beta_i)} & \text{for all the } \alpha_i \\
& = & \varphi(C) \bigwedge_{\beta_i \in Dom(\varphi)\cap(\mathcal{F}(\tau_0/C_0)\cup\mathcal{F}(\psi(\alpha_i)))} C_{\varphi(\beta_i)} & \text{for all the } \alpha_i
\end{array}$$

(with the definition of the $\wedge$ and $\cap$ operator)
We have $\varphi([\tau/C]) = [\varphi(\tau)/\varphi(C) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau/C)} C_{\varphi(\beta_i)}$ so to conclude the result, we have to prove that C' is equal to this expression.
By definition, $Dom(\varphi) \cap \mathcal{F}([\tau/C]) = Dom(\varphi) \cap (\mathcal{F}(\psi(\tau_0)) \cup \mathcal{F}(C))$. It is easy to see that for the substitution ψ of domain $\alpha_1, \ldots, \alpha_n$, that $\mathcal{F}(\tau_0) \cup \mathcal{F}(\psi(\alpha_i)) = \mathcal{F}(\psi(\tau_0)) \cup \{\alpha_1, \ldots, \alpha_n\}$. But the α_i are out of reach of φ so $\varphi(\alpha_i) = \alpha_i$ so $C_{\varphi(\alpha_i)} =$ **True** and because we work modulo $D \wedge$ **True** $= D$, we can ignore those

Furthermore, we have $\mathcal{F}(C') = \mathcal{F}(\psi(C_0) \bigwedge_{\gamma_i\in Dom(\psi)} C_{\psi(\gamma_i)})$. Like for the simple type, it is easy to see that $\mathcal{F}(C_0)\cup\mathcal{F}(\psi(\alpha_i)) = \mathcal{F}(\psi(C_0))\cup\{\alpha_1,\dots,\alpha_n\}$ and like previously, we ignore the constraints for $\{\alpha_1,\dots,\alpha_n\}$. Moreover, it is easy to see, because $\mathcal{F}(C_\tau)\subseteq\mathcal{F}(\tau)$ (Lemma 9) that $\mathcal{F}(\bigwedge C_{\psi(\alpha_i)})\subseteq\bigcup\mathcal{F}(\psi(\alpha_i))$. But we have also these constraints on C'.
So, with those facts, and modulo elimination of the duplicates and of the **True** constraints, we have

$$\bigwedge_{\beta_i\in Dom(\varphi)\cap(\mathcal{F}(\tau_0/C_0)\cup\mathcal{F}(\psi(\alpha_i)))} C_{\varphi(\beta_i)} = \bigwedge_{\beta_i\in Dom(\varphi)\cap\mathcal{F}(\tau/C)} C_{\varphi(\beta_i)} \quad \text{for all the } \alpha_i$$

By hypothesis, we have $Solve(\varphi(C) \bigwedge_{\beta_i\in Dom(\varphi)\cap\mathcal{F}(\tau/C)} C_{\varphi(\beta_i)}) \neq$ **False**. Thus $\varphi([\tau/C])$ is an instance of $\varphi(E(x))$ and by application of the (Var) rule $E\vdash x:\varphi([\tau/C])$.

Case $e = op$ (primitive). Like (Var).

Case $e = c$ (constant). We have:

$$\frac{[\tau/C]\leq TC(c)}{E\vdash c:[\tau/C]}$$

$TC(c) = [\kappa/\mathbf{True}]$ ($\tau=\kappa$ and $C=\mathbf{True}$) so we have $\varphi([\tau/C]) = [\tau/C]$ because $\varphi(\tau)=\tau$, $\varphi(C)=C$ and $\mathcal{F}([\tau/C])=\emptyset$. So $\varphi([\tau/C])$ is also an instance of $TC(c)$. By application of the $(Const)$ rule, we have $\varphi(E)\vdash c:\varphi([\tau/C])$.

Case $e = \mathbf{fun}\ x\to e_1$. We have the following derivation:

$$\frac{E+\{x:[\tau_1/C_{\tau_1}]\}\vdash e_1:[\tau_2/C_2]}{E\vdash(\mathbf{fun}\ x\to e_1):[\tau_1\to\tau_2/C_{\tau_1\to\tau_2}\wedge C_2]}$$

By hypothesis, we have $Solve(\varphi(C_{\tau_1\to\tau_2}\wedge C_2) \bigwedge_{\beta_i\in Dom(\varphi)\cap\mathcal{F}(\tau_1\to\tau_2/C_{\tau_1\to\tau_2}\wedge C_2)} C_{\varphi(\beta_i)}) \neq$ **False**. and $C_{\tau_1\to\tau_2} = C_{\tau_1}\wedge C_{\tau_2}\wedge(\mathcal{L}(\tau_2)\Rightarrow\mathcal{L}(\tau_1))$. We have $\mathcal{F}(C_{\tau_1})\subseteq\mathcal{F}(\tau_1)$, so there is no free variable on the constraints that are not in the simple type. We can apply the Lemma 1 for the difference between $Dom(\varphi)\cap\mathcal{F}(\tau_1\to\tau_2/C_{\tau_1\to\tau_2}\wedge C_2)$ and $Dom(\varphi)\cap\mathcal{F}([\tau_1/C_{\tau_1}])$ and we get:

$$Solve(\varphi(C_{\tau_1}) \bigwedge_{\beta_i\in(\mathcal{F}(\tau_1/C_{\tau_1})\cap Dom(\varphi))} C_{\varphi(\beta_i)}) \neq \mathbf{False}$$

so $Solve(C_{\varphi(\tau_1)})\neq$ **False**. So we do not introduce an incorrect constraint on the environment.
By inductive hypothesis, we obtain:

$$\begin{aligned}
& \varphi(E+\{x:[\tau_1/C_{\tau_1}]\})\vdash e_1:\varphi([\tau_2/C_2]) && \text{a subtitution on environment is point to point}\\
= \; & \varphi(E)+\varphi(\{x:[\tau_1/C_{\tau_1}]\})\vdash e_1:\varphi([\tau_2/C_2]) && \text{by a definition of substitution}\\
= \; & \varphi(E)+\{x:[\varphi(\tau_1)/\varphi(C_{\tau_1}) \textstyle\bigwedge_{\beta_i\in\mathcal{F}(\tau_1/C_{\tau_1})\cap Dom(\varphi)} C_{\varphi(\beta_i)}]\}\vdash e_1:[\varphi(\tau_2)/\varphi(C_2) \textstyle\bigwedge_{\beta_i\in Dom(\varphi)\cap\mathcal{F}(\tau_2/C_2)} C_{\varphi(\beta_i)}]
\end{aligned}$$

By application of the hypothesis and the Lemma 1, we have that no constraint is *Solve* to **False**, so we can apply the (Fun) rule:

$$\varphi(E)\vdash\mathbf{fun}\ x\to e_1:[\varphi(\tau_1)\to\varphi(\tau_2)/C_{\varphi(\tau_1)\to\varphi(\tau_2)}\wedge\varphi(C_2) \bigwedge_{\beta_i\in Dom(\varphi)\cap\mathcal{F}(\tau_2/C_2)} C_{\varphi(\beta_i)}]$$

by application of the Lemma 11 to $C_{\varphi(\tau_1\to\tau_2)}$

$$\begin{aligned}
& \varphi(E)\vdash\mathbf{fun}\ x\to e_1:[\varphi(\tau_1\to\tau_2)/\varphi(C_{\tau_1\to\tau_2}) \textstyle\bigwedge_{\beta_i\in Dom(\varphi)\cap\mathcal{F}(\tau_1\to\tau_2)} C_{\varphi(\beta_i)}\wedge\varphi(C_2) \textstyle\bigwedge_{\beta_i\in Dom(\varphi)\cap\mathcal{F}(\tau_2/C_2)} C_{\varphi(\beta_i)}]\\
= \; & \varphi(E)\vdash\mathbf{fun}\ x\to e_1:[\varphi(\tau_1\to\tau_2)/\varphi(C_{\tau_1\to\tau_2}\wedge C_2) \textstyle\bigwedge_{\beta_i\in Dom(\varphi)\cap(\mathcal{F}(\tau_1\to\tau_2)\cup\mathcal{F}(\tau_2/C_2)))} C_{\varphi(\beta_i)}]
\end{aligned}$$

We know that $\mathcal{F}(C_{\tau_1\to\tau_2})\subseteq\mathcal{F}(\tau_1\to\tau_2)$ (Lemma 9), so we have by definition:

$$\begin{aligned}
& \varphi(E)\vdash\mathbf{fun}\ x\to e_1:[\varphi(\tau_1\to\tau_2)/\varphi(C_{\tau_1\to\tau_2}\wedge C_2) \textstyle\bigwedge_{\beta_i\in Dom(\varphi)\cap(\mathcal{F}(\tau_1\to\tau_2/C_{\tau_1\to\tau_2})\cup\mathcal{F}(\tau_2/C_2)))} C_{\varphi(\beta_i)}]\\
= \; & \varphi(E)\vdash\mathbf{fun}\ x\to e_1:\varphi([\tau_1\to\tau_2/C_{\tau_1\to\tau_2}\wedge C_2])
\end{aligned}$$

Case e=$(e_1\ e_2)$. We have the following derivation:

$$\frac{E \vdash e_1 : [\tau' \to \tau/C_1] \qquad E \vdash e_2 : [\tau'/C_2]}{E \vdash (e_1\ e_2) : [\tau/C_1 \wedge C_2]}(App)$$

By induction hypothesis applied to the first premise, we obtain $\varphi(E) \vdash e_1 : \varphi([\tau' \to \tau)/C_1])$, and with the definition of a substitution, we get:

$$\varphi(E) \vdash e_1 : [\varphi(\tau') \to \varphi(\tau)/\varphi(C_1) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau'\to\tau/C_1)} C_{\varphi(\beta_i)}]$$

By induction hypothesis applied to the second premise and by definition, we obtain:

$$\varphi(E) \vdash e_2 : [\varphi(\tau')/\varphi(C_2) \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau'/C_2)} C_{\varphi(\beta_i)}]$$

By hypothesis, we have $Solve(C_1 \wedge C_2 \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau/C_1\wedge C_2)} C_{\varphi(\beta_i)}) \neq$ **False**. By the Lemma 12, we have $C_1 = C_{\tau_1\to\tau'} \wedge C_1'$ so we have $\mathcal{F}(C_{\tau_1} \wedge C_{\tau_2} \wedge \mathcal{L}(\tau') \Rightarrow \mathcal{L}(\tau)) \subseteq \mathcal{F}(C_1)$. By the Lemma 8, we have $\mathcal{F}(\mathcal{L}(\tau')) = \mathcal{F}(\tau')$, so $\mathcal{F}(\tau') \subseteq \mathcal{F}(C_1)$ (idem for τ). So by the Lemma 1, we have $Solve(C_1 \bigwedge_{\beta_i \in Dom(\varphi)\cap\mathcal{F}(\tau'\to\tau/C_1)} C_{\varphi(\beta_i)}) \neq$ **False** (idem for $[\tau'/C_2]$) and no constraint is *Solve* to **False** So we can apply the (App) rule and we get, with the definition of the $\wedge$ and $\cap$ operators:

$$\begin{aligned} & \varphi(E) \vdash e_1\ e_2 : [\varphi(\tau)/\varphi(C_1) \wedge \varphi(C_2) \bigwedge_{\beta_i \in Dom(\varphi)\cap(\mathcal{F}(\tau'/C_2)\cup\mathcal{F}(\tau'\to\tau/C_1))} C_{\varphi(\beta_i)}] \quad \text{by definition} \\ = \quad & \varphi(E) \vdash e_1\ e_2 : [\varphi(\tau)/\varphi(C_1) \wedge \varphi(C_2) \bigwedge_{\beta_i \in Dom(\varphi)\cap(\mathcal{F}(\tau)\cup\mathcal{F}(C_2)\cup\mathcal{F}(C_1)\cup\mathcal{F}(\tau'))} C_{\varphi(\beta_i)}] \end{aligned}$$

We have $\mathcal{F}(\tau') \subseteq \mathcal{F}(C_1)$ and so by elimination of the duplicates we have:

$$\begin{aligned} & \varphi(E) \vdash e_1\ e_2 : [\varphi(\tau)/\varphi(C_1) \wedge \varphi(C_2) \bigwedge_{\beta_i \in Dom(\varphi)\cap(\mathcal{F}(\tau)\cup\mathcal{F}(C_2)\cup\mathcal{F}(C_1))} C_{\varphi(\beta_i)}] \quad \text{by definition} \\ = \quad & \varphi(E) \vdash e_1\ e_2 : \varphi([\tau/C_1 \wedge C_2]) \end{aligned}$$

hence the result.

Case $e = (e_1, e_2)$. By induction hypothesis like *App*.

Case $e =$ (**let** $x = e_1$ **in** e_2). We have the following derivation:

$$\frac{E \vdash e_1 : [\tau_1/C_1] \quad E + \{x : [Gen([\tau_1/C_1], E)]\} \vdash e_2 : [\tau_2/C_2]}{E \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : [\tau_2/C_2 \wedge C_1 \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))]}$$

By definition:

$$Gen([\tau_1/C_1], E) = \forall\alpha_1, \ldots, \alpha_n.[\tau_1/C_1]$$

with $\{\alpha_1, \ldots, \alpha_n\} = \mathcal{F}(\tau_1) \setminus \mathcal{F}(E)$. Given a set of binding variables $\beta_1, \ldots, \beta_n$ in E and out of reach of φ and we define the substitution $\psi = \varphi \circ [\alpha_i \leftarrow \beta_i]$. By inductive hypothesis applied to the first premise with the substitution φ, we have:

$$\varphi(E) \vdash e_1 : \varphi([\tau_1/C_1])$$

The α_i are not free in E (and $C_{\beta_i} =$ **True** so we can forget them), so we have $E[\alpha_i \leftarrow \beta_i] = E$ and also $\psi(E) = \varphi(E)$. Because the β_i are out of reach of φ and so of ψ, we can apply the inductive hypothesis and we obtain:

$$\psi(E) \vdash e_1 : \psi([\tau_1/C_1])$$

So

$$\varphi(E) \vdash e_1 : \psi([\tau_1/C_1])$$

$[\tau_1/C_1]$) and for all the constraints, so we do no introduce incorrect constraints).
We apply the inductive hypothesis to the second premise with the substitution φ and we obtain a derivation:

$$\varphi(E) + \{x : [\varphi(Gen([\tau_1/C_1], E))]\} \vdash e_1 : \varphi([\tau_2/C_2])$$

To conclude by application of the inductive rule (*Let*), we must prove that
$\varphi(Gen([\tau_1/C_1], E)) = Gen(\psi([\tau_1/C_1]), \varphi(E))$. We have:

$$Gen(\psi([\tau_1/C_1]), \varphi(E)) = Gen(\varphi([\tau_1/C_1][\alpha_i \leftarrow \beta_i]), \varphi(E))$$

So, because the β_i are out of reach of φ , with the Proposition 1, we get:

$$Gen(\psi([\tau_1/C_1]), \varphi(E)) = \varphi(Gen([\tau_1/C_1][\alpha_i \leftarrow \beta_i]), E)$$

Yet the variables $\{\beta_1, \dots, \beta_n\}$ are not free in E and out of reach of φ (and $C_{\beta_i} =$ **True** so we can forget these constraints), so we have $\varphi(Gen([\tau_1/C_1], E)) = Gen(\psi([\tau_1/C_1]), \varphi(E))$ apart from rename variables.
Now, we can apply the rule (*Let*) (like before), and we have:

$$\begin{aligned} & \varphi(E) \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : [\varphi(\tau_2)/\varphi(C_2) \wedge \varphi(C_1) \wedge (\mathcal{L}(\varphi(\tau_2)) \Rightarrow \mathcal{L}(\varphi(\tau_1))) \bigwedge_{\beta_i \in Dom(\varphi) \cap (\mathcal{F}(\tau_1/C_1) \cup \mathcal{F}(\tau_2/C_2))} C_{\varphi(\beta_i)}] \\ = \ & \varphi(E) \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : [\varphi(\tau_2)/\varphi(C_2) \wedge \varphi(C_1) \wedge (\varphi(\mathcal{L}(\tau_2)) \Rightarrow \varphi(\mathcal{L}(\tau_1))) \bigwedge_{\beta_i \in Dom(\varphi) \cap (\mathcal{F}(\tau_1/C_1) \cup \mathcal{F}(\tau_2/C_2))} C_{\varphi(\beta_i)}] \end{aligned}$$

By application of the Lemma 8, it is easy to see that we have

$$\mathcal{F}(\tau_2/C_2 \wedge C_1 \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))) = \mathcal{F}([\tau_1/C_1]) \cup \mathcal{F}([\tau_2/C_2]))$$

so with the definition of a substitution, we have:

$$\begin{aligned} & \varphi(E) \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : [\varphi(\tau_2)/\varphi(C_2) \wedge \varphi(C_1) \wedge (\varphi(\mathcal{L}(\tau_2)) \Rightarrow \varphi(\mathcal{L}(\tau_1))) \bigwedge_{\beta_i \in Dom(\varphi) \cap \mathcal{F}(\tau_2/C_2 \wedge C_1 \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1)))} C_{\varphi(\beta_i)}] \\ = \ & \varphi(E) \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : \varphi([\tau_2/C_2 \wedge C_1 \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))]) \end{aligned}$$

hence the result.

□

Chapter 4

Typing safety

4.1 Tools

4.1.1 Lemma of indifference

Lemma 14 (Indifference of the type derivation beside useless hypothesis) *Given two environments E_1 and E_2 such as for all free variables x in the expression e, $E_1(x) = E_2(x)$. If $E_1 \vdash e : [\tau/C]$ then $E_2 \vdash e : [\tau/C]$.*

Proof: by induction on the type derivation of e.

Case (*Const*), (*Op*). There is no free variable in e and in the inductive rule, the type scheme do not depend on the environment, hence the result.

Case (Var). The only free variable is x. The inductive rule (Var) needs $E_1(x) = E_2(x)$ (and nothing else) hence the result.

Case (*App*), (*Couple*), (*Fun*), (*Let*). By the inductive hypothesis applied to the premises.

□

Definition 7 (More general type scheme) *We say that a type scheme σ' is more general than a type scheme σ, noted $\sigma' \geq \sigma$ if all the instances of σ are also instances of σ'*

By the Lemma 12, the type scheme that could be instanciated are like $\forall \alpha_1 \ldots \alpha_n.[\tau/C]$. It is easy to see that $\sigma' \geq \forall \alpha_1 \ldots \alpha_n.[\tau/C]$ if and only if $[\tau/C] \leq \sigma'$ (where the α_i are not free in σ').

Lemma 15 (Stability of the type derivation with more general type scheme) *Given two environments, E_1 and E_2 with the same domain such as $E_1(x) \geq E_2(x)$ for all $x \in Dom(E_1)$. If $E_2 \vdash e : [\tau/C]$ then $E_1 \vdash e : [\tau/C]$.*

Proof: by induction on the type derivation of e.

Case (Var): trivial by hypothesis on E_1 and E_2.

Case (*Const*), (*Op*): the inductive rules do not depend on the environment, hence the result.

Case (Fun), (*App*), (*Couple*): by inductive hypothesis on the premises.

Case (*Let*): We have the following derivation:

$$\frac{E_1 \vdash e_1 : [\tau_1/C_1] \qquad E_1 + \{x : Gen([\tau_1/C_1], E_1)\} \vdash e_2 : [\tau_2/C_2]}{E_1 \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : [\tau_2/C_1 \wedge C_2 \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))]}$$

It is easy to see that $\mathcal{F}(E_1) \subseteq \mathcal{F}(E_2)$ (because E_1 is more general). So we have $Gen([\tau_1/C_1], E_1) \geq Gen([\tau_1/C_1], E_2)$. We can apply the inductive hypothesis and we have the result.

□

To prove some property of the typing, we introduce a new typing rule for the parallel vectors, because they are values and we never type them. Thus, if an expression is evaluated to a parallel vector value, we want to prove that they have the same type.

$$\frac{\forall i \quad E \vdash e_i : [\tau/C_i]}{E \vdash \langle e_0, \ldots, e_{p-1} \rangle : [\tau\ par/\mathcal{L}(\tau) \bigwedge_{i=0}^{p-1} C_i]}(Vect)$$

Lemma 16 (Form of the values) *Give $\emptyset \vdash v : [\tau/C]$,*

1. *if $\tau = \tau_1 \to \tau_2$ then v is an abstract function or an operator.*
2. *if $\tau = \tau_1 * \tau_2$ then v is a couple (v_1, v_2).*
3. *if $\tau = (\tau_1\ par)$ then v is a parallel vector $\langle v_0, \ldots, v_{p-1} \rangle$.*
4. *if τ is a type base then v is a constant c.*

Proof: by a trivial induction on the inductive type rules. □

Remark: It is easy to see that if v is a parallel vector, the last used rule is $(Vect)$.

4.1.3 Lemma of substitution

In this section, we prove the most important lemma for substitution on type derivation. This lemma will be used to prove the safety of the type system.

Lemma 17 (Lemma of substitution) *Given*

$$\begin{array}{rcll} E & \vdash & e' : [\tau_1/C_1 \wedge C_1'] & \textit{(a)} \\ E + \{x : \forall \alpha_1, \ldots, \alpha_n.[\tau_1/C_1]\} & \vdash & e : [\tau_2/C] & \textit{(b)} \end{array}$$

with

1. *$\alpha_1, \ldots, \alpha_n$ not free variables in E*
2. *$Solve(C_1 \wedge C_1' \wedge C) \neq$ **False***
3. *no variable x bound in e is free in e' (to not have capture)*

We note $C_n = C_1 \wedge C_1' \wedge C$.
There exists C' such that $\forall \varphi \in \phi_{C_n}$ then $\varphi \models C'$ and $E \vdash e[x \leftarrow e'] : [\tau_2/C']$

Remark: why C' and not C ? The constraints of a type scheme, depending of the sub-expressions (typed in the type derivation). So when you substitute a variable by an expression, you need to add its constraints if these variables are not binding variables. $\forall \varphi \in \phi_{C_n}$ then $\varphi \models C'$ says that C' is less constrainted as C.

Proof: by structural induction on e. We wrote E_x for the environment $E + \{x : \forall \alpha_1, \ldots, \alpha_n.[\tau_1/C_1]\}$.

Case $e = c$ (constant). $c[x \leftarrow e'] = c$. x is not free in e. By application of the Lemma 14 we deduce the result.

Case $e = op$ (operator). Idem.

Case $e = y$ (y≠x). $y[x \leftarrow e'] = y$. Idem.

Case $e =$ **fun** $x \to a_1$. We have $e[x \leftarrow e'] = e$. Idem.

Case $e = x$. $e[x \leftarrow e'] = e'$. By hypothesis, we have, $E_x \vdash x : [\tau_2/C]$, so $[\tau_2/C] \leq E(x)$, i.e, for a substitution ψ on the α_i, $\tau_2 = \psi(\tau_1)$ and, $C = \psi(C_1) \bigwedge_{\alpha_1,\ldots,\alpha_n} C_{\psi(\alpha_i)}$. Because the α_i are not free

By application of the Lemma 13 apply to **(a)**, we have:

$$\begin{array}{rcll}
\psi(E) & \vdash & e' : [\psi(\tau_1)/\psi(C_1) \wedge \psi(C'_1) \bigwedge_{\beta_i \in \mathcal{F}([\tau_1/C_1 \wedge C'_1]) \cap Dom(\psi)} C_{\psi(\beta_i)} & \text{by commutativity and by the remark} \\
E & \vdash & e' : [\tau_2/\psi(C_1) \displaystyle\bigwedge_{\alpha_1,\ldots,\alpha_n} C_{\psi(\alpha_i)} \wedge \psi(C'_1)] & \text{by definition} \\
E & \vdash & e' : [\tau_2/C \wedge \psi(C'_1)] & \\
E & \vdash & e' : [\tau_2/C'] &
\end{array}$$

It is easy to see that we have $\psi \in \phi_{C_n}$, so by the Lemma 3 we have $Solve(C') \neq$ **False** and the result.

Case $e =$ **fun** $y \to e_1$ with $y \neq x$. Take $\varphi \in \phi_{C_n}$. If the α_i appear in $[\tau_2/C]$, we can rename then to some fresh variables β_i not free in E and distincts of the α_i with the substitution $\theta = [\alpha_i \leftarrow \beta_i]$. If the α_i do not appear in $[\tau_2/C]$, we take the identity for θ.
Now we can apply the Lemma 13 for θ (it is only a simple renaming on binding variables or the identity) to **(b)**. We have naturelly $Solve(\theta(C)) \neq$ **False** and $\theta(E_x) = E_x$. Now, we have, $E_x \vdash e : [\theta(\tau_2)/\theta(C)]$ with $\theta(\tau_2) = \tau_a \to \tau_b$:

$$\frac{E_x + \{y : [\tau_a/C_{\tau_a}]\} \vdash e_1 : [\tau_1/C_2]}{E_x \vdash \textbf{fun } y \to e_1 : [\tau_a \to \tau_b/C_{\tau_a \to \tau_b} \wedge C_2]}$$

and $\theta(C) = C_{\tau_1 \to \tau_2} \wedge C_2$. It is easy to see that the α_i not free in $E_x + \{y : [\tau_a/C_{\tau_a}]\}$, $Solve(C_3) \neq$ **False** so by application of the inductive hypothesis to the premise, we obtain:

$$E_x + \{y : [\tau_a/C_{\tau_a}]\} \vdash e_1[x \leftarrow e'] : [\tau_a/C'_2]$$

with $Solve(C'_2) \neq$ **False** and $\varphi \models C'_2$. By application of the (Fun) rule, we get $E \vdash fun\ y \to e_1[x \leftarrow e'] : [\tau_a \to \tau_b/C_{\tau_a \to \tau_b} \wedge C'_2]$, i.e, $E \vdash e : [\theta(\tau)/\theta(C')]$. We conclude $E \vdash e : [\tau/C']$ by application of the Lemma 13 for θ^{-1}, $\varphi \models C'$ by the Lemma 4 and the result.

Case $e = (e_1\ e_2)$. Take $\varphi \in \phi_{C_n}$. By hypothesis $E_x \vdash e : [\tau_2/C]$, we have:

$$\frac{E_x \vdash e_1 : [\tau_a \to \tau_2/C_a] \quad E_x \vdash e_2 : [\tau_a/C_b]}{E_x \vdash (e_1\ e_2) : [\tau_2/C_a \wedge C_b]}$$

with $Solve(C_a \wedge C_b) \neq$ **False**, so with the Lemma 1, we get $Solve(C_a) \neq$ **False** (idem for C_b). We apply the inductive hypothesis to the two premises and we obtain the following derivations:

$$\begin{array}{rcl}
E_x & \vdash & e_1[x \leftarrow e'] : [\tau_a \to \tau_2/C'_a] \\
E_x & \vdash & e_2[x \leftarrow e'] : [\tau_a/C'_b]
\end{array}$$

With $Solve(C'_a) \neq$ **False**, $Solve(C'_b) \neq$ **False** and $\varphi \models C'_a$ and $\varphi \models C'_b$. By application of the (App) rule, we get:

$$E_x \ \vdash \ e[x \leftarrow e'] : [\tau_2/C']$$

with $C' = C'_a \wedge C'_b$. By the Lemma 4, we have $\varphi \models C'_a \wedge C'_b$ and by the Lemma 3, $Solve(C') \neq$ **False** and the result.

Case $e = (e_1, e_2)$. Idem by induction on the two premises.

Case $e = \langle e_0, \ldots, e_{p-1} \rangle$. Idem by induction on the $p-1$ premises.

.

Case $e =$ **let** $y = e_1$ **in** e_2. Take $\varphi \in \phi_{C_n}$. By hypothesis $E_x \vdash e : [\tau_2/C]$, we have:

$$\frac{E_x \vdash e_1 : [\tau_a/C_a] \quad E_x + \{y : [Gen([\tau_a/C_a], E_x)]\} \vdash e_2 : [\tau_2/C_b]}{E_x \vdash \textbf{let } y = e_1 \textbf{ in } e_2 : [\tau_2/C = C_a \wedge C_b \wedge (\mathcal{L}(\tau_b) \Rightarrow \mathcal{L}(\tau_a))]}$$

We apply the inductive hypothesis to the first premise and we get $E \vdash e_1[x \leftarrow e'] : [\tau_a/C'_a]$ so $Solve(C'_a) \neq$ **False** and $\varphi \models C'_a$. We have two cases:

$E + \{x : [Gen([\tau_a/C'_a], E_x)]\}$ (there is no duplicate in the environment). We can remark that $[Gen([\tau_a/C'_a], E)] \geq$ $[Gen([\tau_a/C'_a], E_x)]$ because $(E) \subseteq (E_x)$. With the Lemma 15 and by inductive hypothesis we get:

$$E + \{x : [Gen([\tau_a/C'_a], E)]\} \vdash e_2 : [\tau/C'_b]$$

By hypothesis, we have $Solve(C'_b) \neq \textbf{False}$. We note $C' = C'_a \wedge C'_b \wedge (\mathcal{L}(\tau_a) \Rightarrow \mathcal{L}(\tau_b))$. Like in (*App*) we get $Solve(C') \neq \textbf{False}$ and $\varphi \models C'$. By application of the (*Let*) rule, we have the following derivation:

$$\frac{E \vdash e_1[x \leftarrow e'] : [\tau_a/C'_a] \quad E + \{y : [Gen([\tau_a/C'_a], E)]\} \vdash e_2 : [\tau_2/C'_b]}{E \vdash \textbf{let } x = e_1[x \leftarrow e'] \textbf{ in } e_2 : [\tau_2/C']}$$

and the result.

- If $y \neq x$, then we have $e[x \leftarrow e'] = \textbf{let } y = e_1[x \leftarrow e'] \textbf{ in } e_2[x \leftarrow e']$ and $E_x + \{y : [Gen([\tau_a/C'_a], E_x)]\} = E + \{y : [Gen([\tau_a/C'_a], E_x)]\}$. Apply the inductive hypothesis to second premise, we obtain:

$$E + \{y : [Gen([\tau_a/C'_a], E_x)]\} \vdash e_2[x \leftarrow e'] : [\tau_2/C'_b]$$

with $Solve(C'_b) \neq \textbf{False}$ and $\varphi \models C'_b$. Like previously, we have $[Gen([\tau_a/C'_a], E)] \geq [Gen([\tau_a/C'_a], E_x)]$, so with the Lemma 15, we get:

$$E + \{y : [Gen([\tau_a/C'_a], E)]\} \vdash e_2[x \leftarrow e'] : [\tau_2/C'_b]$$

$C' = C'_a \wedge C'_b \wedge (\mathcal{L}(\tau_a) \Rightarrow \mathcal{L}(\tau_b))$. Like before (by inductive hypothesis and Lemma 4), we have $Solve(C') \neq$ **False** and $\varphi \models C'$. We can apply the (*Let*) rule and we have:

$$\frac{E \vdash e_1[x \leftarrow e'] : [\tau_a/C'_a] \quad E + \{y : [Gen([\tau_a/C'_a], E)]\} \vdash e_2[x \leftarrow e'] : [\tau_2/C'_b]}{E \vdash \textbf{let } y = e_1[x \leftarrow e'] \textbf{ in } e_2[x \leftarrow e'] : [\tau_2/C']}$$

and the result.

$\square$

4.2 Typing safety

Now, we prove some technical lemmas to express that a well-typed expression is a correct expression for the BSML language and the BSλ_p-calculus. We first recall some basic tranformations of the mini-BSML expressions to BSλ_p expressions. Next, we present a property of the type system and we finish with a result of correction.

4.2.1 From mini-BSML to BSλ_p

We present that the mini-BSML language is based on the BSλ_p-calculus by some basic equivalences with an extended syntax.

We wrote E or X (do not confuse with environment) for the global expressions and e or x for the local one. We note $\mathcal{T}$, for the transformation of BSML expressions to BSλ_p-calculus expressions. It is a relation on such expressions. The couple, projections, **fix** and the arithmetic operations are encoded with the classical λ-term. The formal definition is:

$\textbf{nprocs}\ \mathcal{T}\ p \qquad c\ \mathcal{T}\ c \qquad x\ \mathcal{T}\ \dot{x} \qquad \textbf{fix}\ \mathcal{T}\ Y \qquad X\ \mathcal{T}\ \bar{x} \qquad \textbf{apply}\ \mathcal{T}\ \# \qquad \textbf{put}\ \mathcal{T}\ !$

$\textbf{mkpar}\ \ \mathcal{T}\ \ \lambda\dot{f}.\langle(\dot{f}\ 0), \ldots, (\dot{f}\ (p-1))\rangle$

$(E_1\ E_2)$	$\mathcal{T}$	$(E'_1\ E'_2)$	if $(E_1\ \mathcal{T}\ E'_1)$ and $(E_2\ \mathcal{T}\ E'_2)$
$(E\ e)$	$\mathcal{T}$	$(E'\ e')$	if $(E\ \mathcal{T}\ E')$ and $(e\ \mathcal{T}\ e')$
$(e_1\ e_2)$	$\mathcal{T}$	$(e'_1\ e'_2)$	if $(e_1\ \mathcal{T}\ e'_1)$ and $(e_2\ \mathcal{T}\ e'_2)$
$(\textbf{fun}\ X \to E)$	$\mathcal{T}$	$(\lambda\bar{x}.E')$	if $E\ \mathcal{T}\ E'$
$(\textbf{fun}\ x \to E)$	$\mathcal{T}$	$(\lambda\dot{x}.E')$	if $E\ \mathcal{T}\ E'$
$(\textbf{fun}\ x \to e)$	$\mathcal{T}$	$(\lambda\dot{x}.e')$	if $e\ \mathcal{T}\ e'$
$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$	$\mathcal{T}$	$(\lambda\dot{x}.e'_2)\ e'_1$	if $(e_1\mathcal{T}e'_1)$ and $(e_2\mathcal{T}e'_2)$
$\textbf{let}\ x = e_1\ \textbf{in}\ E_2$	$\mathcal{T}$	$(\lambda\dot{x}.E'_2)\ e'_1$	if $(e_1\mathcal{T}e'_1)$ and $(E_2\mathcal{T}E'_2)$
$\textbf{let}\ x = E_1\ \textbf{in}\ E_2$	$\mathcal{T}$	$(\lambda\bar{x}.E'_2)\ E'_1$	if $(E_1\mathcal{T}E'_1)$ and $(E_2\mathcal{T}E'_2)$
$\textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	$\mathcal{T}$	$(e' \to e'_1, e'_2)$	if $(e\ \mathcal{T}\ e')$ and $(e_1\ \mathcal{T}\ e'_1)$ and $(e_2\ \mathcal{T}\ e'_2)$
$\textbf{if}\ e\ \textbf{then}\ E_1\ \textbf{else}\ E_2$	$\mathcal{T}$	$(e' \to E'_1, E'_2)$	if $(e\ \mathcal{T}\ e')$ and $(E_1\ \mathcal{T}\ E'_1)$ and $(E_2\ \mathcal{T}\ E'_2)$
$\textbf{if}\ E\ \textbf{at}\ e\ \textbf{then}\ E_1\ \textbf{else}\ E_2$	$\mathcal{T}$	$(E' \overset{e'}{\to} E'_1, E'_2)$	if $(e\ \mathcal{T}\ e')$ and $(E\ \mathcal{T}\ E')$ and $(E_1\ \mathcal{T}\ E'_1)$ and $(E_2\ \mathcal{T}\ E'_2)$

Remark: this transformation is not deterministic because a totally polymorphic expression, like the identity, could have different transformations. It is why, we use a relation and not a function.

4.2.2 Validity of the type system

Proposition 2 (BSML type) *Given an environment E and an expression e such that $E \vdash e : [\tau/C]$ then $\tau \in G \cup V$.*

Proof: by the Lemma 12, we have that $C = C_\tau \wedge C'$. By the Lemma 1, we have $Solve(C_\tau) \neq$ **False**, so by application of the Lemma 7, we deduce the result. □

But this proposition is not sufficient, to transform our well-typed expresssions to BSλ_P-calculus expressions. Take:

```
(fun x -> ifat ((mkpar (fun i-> true)),(0,(x,x))));;
```

The type derivation given, after simplification (see chapter 6), $[\alpha \to \alpha/Loc(\alpha) \Rightarrow \textbf{False}]$ (i.e., α could not be local). Thus, this not a totaly polymorphic expression. So we could not says with only the type if an expression is local or global (see definition) and transforms it.

Lemma 18 (sub-expressions of local expressions are locals) *Given an environment E and an expression such that $E \vdash e : [\tau/C]$ (1) with $\tau \in L$ then for all judgement $E'' \vdash e' : [\tau'/C']$ of the derivation tree (cf (1)), we have $E'' = E + E'$ and $\tau \in V$.*

Proof: by structural induction on e.
Case $e = c$. The only sub-expression is e hence the result.

Case $e = op$. Idem.

Case $e = x$. Idem.

Case $e = \langle \dots \rangle$. Impossible by hypothesis hence the result.

Case $e = (e_1\ e_2)$, (e_1, e_2). By trivial application of the inductive hypothesis.

Case $e =$ **fun** $x \to e_1$. We have the following derivation:

$$\frac{E + \{x : [\tau_1/C_{\tau_1}]\} \vdash e_1 : [\tau_2/C_2]}{E \vdash (\textbf{fun}\ x \to e_1) : [\tau_1 \to \tau_2/C_{(\tau_1 \to \tau_2)} \wedge C_2]}$$

By definition of L, we have $\tau_1 \in L$. It is easy to see that $C_{\tau_1} = \textbf{True}$. We take $E' = \{x : [\tau_1/C_{\tau_1}]\}$ and we have the result by application of the inductive hypothesis.

Case $e =$ **let** $x = e_1$ **in** e_2. We have the following derivation:

$$\frac{E \vdash a_1 : [\tau_1/C_1] \qquad E + \{x : Gen([\tau_1/C_1], E)\} \vdash e_2 : [\tau_2/C_2]}{E \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : [\tau_2/C_1 \wedge C_2 \wedge C_1 \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))]}(Let)$$

By inductive hypothesis, we have $\tau_1 \in V$. By hypothesis, we have $Solve(C_1 \wedge C_2 \wedge C_1 \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))) \neq$ **False**. By application of the Lemma 1, we have $Solve(\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1)) \neq$ **False**. By definition of $\mathcal{L}$, we get $Solve(\mathcal{L}(\tau_2)) =$ **True**. So by definition of the $\Rightarrow$ operator, we obtain $Solve(\tau_1) \neq$ **False** and by application of the Lemma 5, we have $\tau_2 \in V$. By application of the inductive hypothesis with $E' = \{x : Gen([\tau_1/C_1], E)\}$ we deduce the result. □

Corollary 1 (Instance) *Given $E \vdash e : [\tau/C]$ where $\tau \in V \setminus L$ there exists a substitution φ such that $\varphi(E) \vdash \varphi([\tau/C])$ and $\varphi(\tau) \in G$ or $\varphi(\tau) \in L$.*

Proof: by application of the Lemma 13 and Lemma 18. □

Proof: by structural induction of e and by using the Proposition 2 to distinguish the global and local expression and the Corollary 1 to transform the totally polymorph expressions to a global or local expression.

□

4.3 The small step semantics

We could prove by induction and by cases, the classical following theorem:

Theorem 3 (Safety of the type system for the natural semantics) *If $E \vdash e : [\tau/C]$ and $e \triangleright e'$ then e' is a value and $E \vdash e' : [\tau/C']$.*

But, the natural semantics does not give the step of the calculus (only the result). So we do not have a result for C'. Also, many expressions cannot be evaluated on a value; they have an infinity evaluation (because we can use the fixpoint operator). Those expressions are very important for programming languages and to develop big applications. Those expressions do not do incorrect operations and can't be evaluated on a value. So, we do not have a result for those expressions. Thus, this semantics and this theorem are not sufficient for us. In order not to have those problems, we introduce a new semantics, that describes all the steps of the calculus and we proved the safety and some properties of C' with a lemma of subject reduction more easily.

4.3.1 Definition

The small step semantics has the form

$$\begin{array}{ll} e \rightharpoonup e' & \text{for one step} \\ e_0 \rightharpoonup e_1 \rightharpoonup e_2 \rightharpoonup \ldots \rightharpoonup v & \text{for all the steps of the calculus} \end{array}$$

We note $\overset{*}{\rightharpoonup}$, for the transitive closure of $\rightharpoonup$ and note $e_0 \overset{*}{\rightharpoonup} v$ for $e_0 \rightharpoonup e_1 \rightharpoonup e_2 \rightharpoonup \ldots \rightharpoonup v$. To define the relation, $\rightharpoonup$, we begin with some axioms for the relation, $e \overset{\varepsilon}{\rightharpoonup} e'$, of the head reduction:

$$\begin{array}{lcll} (\mathbf{fun}\ x \to e)\ v & \overset{\varepsilon}{\rightharpoonup} & e[x \leftarrow v] & (\beta_{fun}) \\ (\mathbf{let}\ x = v\ \mathbf{in}\ e) & \overset{\varepsilon}{\rightharpoonup} & e[x \leftarrow v] & (\beta_{let}) \end{array}$$

We give some axioms, the δ-rules, i.e., for primitive operators:

$$\begin{array}{lcll} +(n_1, n_2) & \overset{\varepsilon}{\rightharpoonup} & n \text{ with } n = n_1 + n_2 & (\delta_+) \\ \mathbf{fst}(v_1, v_2) & \overset{\varepsilon}{\rightharpoonup} & v_1 & (\delta_{fst}) \\ \mathbf{snd}(v_1, v_2) & \overset{\varepsilon}{\rightharpoonup} & v_2 & (\delta_{snd}) \\ \mathbf{fix}(\mathbf{fun}\ x \to\ e) & \overset{\varepsilon}{\rightharpoonup} & e[x \leftarrow \mathbf{fix}(\mathbf{fun}\ x \to\ e)] & (\delta_{fix}) \\ \mathbf{fix}(\mathbf{op}) & \overset{\varepsilon}{\rightharpoonup} & \mathbf{op} & \\ \mathbf{ifthenelse}(\mathbf{true}, (v_1, v_2)) & \overset{\varepsilon}{\rightharpoonup} & v_1 & (\delta_{ifthenelse}) \\ \mathbf{ifthenelse}(\mathbf{false}, (v_1, v_2)) & \overset{\varepsilon}{\rightharpoonup} & v_2 & \end{array}$$

and for parallel operators:

$$\begin{array}{lcll} \mathbf{isnc}(v) & \overset{\varepsilon}{\rightharpoonup} & \mathbf{false} \text{ if } v \neq \mathbf{nc()} & (\delta_{isnc}) \\ \mathbf{isnc}(\mathbf{nc()}) & \overset{\varepsilon}{\rightharpoonup} & \mathbf{true} & (\delta_{isnc}) \\ \mathbf{mkpar}(\mathbf{fun}\ x \to\ e) & \overset{\varepsilon}{\rightharpoonup} & \langle e[x \leftarrow 0], \ldots, e[x \leftarrow (p-1)] \rangle & (\delta_{mkpar}) \\ \mathbf{apply}(\langle \mathbf{fun}\ x \to\ e_0, \ldots, \mathbf{fun}\ x \to\ e_{p-1} \rangle, \langle v_0, \ldots, v_{p-1} \rangle) & \overset{\varepsilon}{\rightharpoonup} & \langle e_0[x \leftarrow v_0], \ldots, e_{p-1}[x \leftarrow v_{p-1}] \rangle & (\delta_{apply}) \\ \mathbf{ifat}(v, (\langle \ldots, \overbrace{\mathbf{true}}^{n}, \ldots \rangle, (v_1, v_2))) & \overset{\varepsilon}{\rightharpoonup} & v_1 \text{ if } v = n & (\delta_{ifat}) \\ \mathbf{ifat}(v, (\langle \ldots, \overbrace{\mathbf{false}}^{n}, \ldots \rangle, (v_1, v_2))) & \overset{\varepsilon}{\rightharpoonup} & v_2 \text{ if } v = n & \\ \mathbf{put}(\langle \mathbf{fun}\ dst \to e_0, \ldots, \mathbf{fun}\ dst \to e_{p-1} \rangle) & \overset{\varepsilon}{\rightharpoonup} & \langle e'_0, \ldots, e'_{p-1} \rangle & (\delta_{put}) \end{array}$$

and where $\forall i\ f_i = \textbf{fun}\ x \rightarrow \textbf{if}\ x = 0\ \textbf{then}\ v_0\ \textbf{else} \ldots \textbf{if}\ x = (p-1)\ \textbf{then}\ v_{p-1}\ \textbf{else}\ \textbf{nc}$

But, it is easy to see that we cannot always make a head reduction. We have to reduce in depth in the sub-expression. To define this deep reduction, we use the following inference rule:

$$\frac{e \overset{\varepsilon}{\rightharpoonup} e'}{\Gamma(e) \rightharpoonup \Gamma(e')}(\textit{context rule})$$

In this rule, Γ is an evaluation contex, i.e., an expression with a hole and has the following abstract syntax:

$$\begin{array}{rcll}
\Gamma & ::= & [] & \text{head evaluation} \\
 & | & \Gamma\ e & \text{right application evaluation} \\
 & | & v\ \Gamma & \text{left application evaluation} \\
 & | & \textbf{let}\ x = \Gamma\ \textbf{in}\ e & \text{let evaluation} \\
 & | & (\Gamma, e) & \text{left evaluation couple} \\
 & | & (v, \Gamma) & \text{right evaluation couple} \\
 & | & \langle \Gamma, e, \ldots, e \rangle & \text{parallel vector, first component} \\
 & \vdots & & \\
 & | & \langle e, \ldots, \overbrace{\Gamma}^{i}, e, \ldots, e \rangle & \text{parallel vector, i}^{\text{th}}\text{ component} \\
 & \vdots & & \\
 & | & \langle e, \ldots, e, \Gamma \rangle & \text{parallel vector, last component}
\end{array}$$

Remark: The non-deterministic of the evaluation of the vectors came from the parallel model.

Theorem 4 *$e \triangleright v$ if and only if $e \overset{*}{\rightharpoonup} v$.*

Proof: by induction (proof in ([Leroy, 2002])). The only difference which is not trivial is for the evaluation of the put. Informally, like in the natural semantics, we first compute the v_i^j by using the (*Let*) construction (in a **let** $x = e_1$ **in** e_2 we first evaluated in e_1). So we have simulated the calculus on the premise and we have computed the expression as in the natural semantics.

4.3.2 Subject reduction

Next, we want to prove that for each step of the calculus of an expression, the type is kept by a property called the subject reduction.

To be less typable

Definition 8 (Less typable) *e_1 is less typable than e_2, write $e_1 \sqsubseteq e_2$, if for all environment E, type $[\tau/C]$, such that $E \vdash e_1 : [\tau/C]$ then there exists C' such that $\forall \varphi \in \phi_C$ then $\varphi \models C'$ and $E \vdash e_2 : [\tau/C']$*

Remark: now for the Lemma of substitution, by using this new notation, we have that $e \sqsubseteq e[x \leftarrow e']$.

Lemma 19 (Less typed for context) *For all evaluation context Γ, if $e_1 \sqsubseteq e_2$ then $\Gamma(e_1) \sqsubseteq \Gamma(e_2)$.*

Proof: given $E \vdash \Gamma(e_1) : [\tau/C]$ and $\varphi \models C$. We prove $E \vdash \Gamma(e_2) : [\tau/C']$ and $\varphi \models C'$ by structural induction on Γ.

Case $\Gamma = []$. Trivial by definition.

Case $\Gamma = \Gamma'\ e$. We have:

$$\frac{E \vdash \Gamma'(e_1) : [\tau' \rightarrow \tau/C_1] \quad E \vdash e : [\tau'/C_2]}{E \vdash \Gamma'(e_1)e : [\tau/C_1 \wedge C_2]}$$

By application of the Lemma 2, we have $\varphi \vdash C_2$ and by application of the Lemma 1, we have $\varphi \vdash C'_1 \wedge C_2$ and by Lemma 3, we have $Solve(C'_1 \wedge C_2) \neq \textbf{False}$, so by application of the (App) rule, we obtain:

$$\frac{E \vdash \Gamma'(e_2) : [\tau' \to \tau / C'_1] \quad E \vdash e : [\tau'/C_2]}{E \vdash \Gamma'(e_2)e : [\tau / C'_1 \wedge C_2]}$$

and the result.

Case $\Gamma = v\ \Gamma'$, $\Gamma = \textbf{let}\ x = \Gamma'\ \textbf{in}\ e$, $\Gamma = (\Gamma', e)$ and $\Gamma = (v, \Gamma')$. Idem.

Case $\Gamma = \langle \Gamma'(e_1), a_1, \ldots, a_{p-1} \rangle$. We have:

$$\frac{E \vdash \Gamma'(e_1) : [\tau / C_e] \quad \forall i \in \{1, \ldots, p-1\}\ E \vdash a_i : [\tau / C_i]}{E \vdash \langle \Gamma'(e_1), a_1, \ldots, a_{p-1} \rangle : [\tau\ par / \mathcal{L}(\tau) \wedge C_e \bigwedge_{i=1}^{p-1} C_i]}$$

by application of the inductive hypothesis on the first premise, we have $E \vdash \Gamma'(e_1) : [\tau / C'_e]$ and by application of our lemmas like in (App), we have $Solve(\mathcal{L}(\tau) \wedge C'_e \bigwedge_{i=1}^{p-1} C_i]) \neq \textbf{False}$, so by application of the $(Vect)$ rule, we obtain:

$$\frac{E \vdash \Gamma'(e_2) : [\tau / C'_e] \quad \forall i \in \{1, \ldots, p-1\}\ E \vdash a_i : [\tau / C_i]}{E \vdash \langle \Gamma'(e_2), a_1, \ldots, a_{p-1} \rangle : [\tau\ par / \mathcal{L}(\tau) \wedge C'_e \bigwedge_{i=1}^{p-1} C_i]}$$

and the result.

Case $\langle e, \Gamma'(e_1), e, \ldots, e \rangle$, ..., $\langle e, \ldots, e, \Gamma'(e_1) \rangle$. Idem.

□

Preservation of the reduction

First, we prove that the head reduction keeps the typing and with this result, we will have our lemma of subject reduction.

Lemma 20 (Preservation for the operators) *If $e \overset{\varepsilon}{\rightharpoonup} e'$ by a δ-rule, then $e \sqsubseteq e'$.*

Proof: by case on the δ-rules. For each δ-rules, we study the type derivation of e and e'.

Remark: in all cases, we have for the operators, the constraints C_a. These constraints came from the calculus of the instance (substitution on the type schemes could introduce new constraints).

Case $+$. We have the following derivation:

$$\frac{\dfrac{}{E \vdash + : (int * int) \to int} \quad \dfrac{E \vdash n_1 : int \quad E \vdash n_2 : int}{E \vdash (n_1, n_2) : int * int}}{E \vdash +\ (n_1, n_2) : int}$$

and $E \vdash n : int$ where $n = n_1 + n_2$. So it is easy to see that we have the result.

Case fst. We have the following derivation:

$$\frac{\dfrac{}{E \vdash \textbf{fst} : [\tau_1 * \tau_2 \to \tau_1 / \mathcal{L}(\tau_1) \Rightarrow \mathcal{L}(\tau_2) \wedge C_a]} \quad \dfrac{E \vdash v_1 : [\tau_1 / C_1] \quad E \vdash v_2 : [\tau_2 / C_2]}{E \vdash (v_1, v_2) : [\tau_1 * \tau_2 / C_1 \wedge C_2]}}{E \vdash \textbf{fst}\ (v_1, v_2) : [\tau_1 / C_a \wedge C_1 \wedge C_2 \wedge \mathcal{L}(\tau_1) \Rightarrow \mathcal{L}(\tau_2)]}$$

and $E \vdash v1 : [\tau_1 / C_1]$. So it is easy to see that we have the result.

Case snd. Idem.

Case fix. We have two cases. First, we have the following derivation:

$$\dfrac{\dfrac{}{E \vdash \mathbf{fix} : [(\tau \to \tau) \to \tau / C_a]} \quad \dfrac{E + \{x : [\tau/C_\tau]\} \vdash e_1 : [\tau/C_1]}{E \vdash (\mathbf{fun}\ x \to e_1) : [\tau \to \tau / C_{\tau\to\tau} \wedge C_1]}}{\vdash \mathbf{fix}(\mathbf{fun}\ x \to e_1) : [\tau / C_a \wedge C_{\tau\to\tau} \wedge C_1]}$$

and $E \vdash e_1[x \leftarrow \mathbf{fix}(\mathbf{fun}\ x \to e_1)] : [\tau/C']$. By application of the Lemma 12, we have:

$$\begin{array}{rcl} C_a & = & C_{(\tau\to\tau)\to\tau} \wedge C'_a \\ & = & C_\tau \wedge C_{\tau\to\tau} \wedge (\mathcal{L}(\tau) \Rightarrow \mathcal{L}(\tau)) \wedge C'_a \\ & = & C_\tau \wedge C''_a \end{array}$$

So by application of the Lemma 17 to:

1. $E \vdash \mathbf{fix} : [(\tau \to \tau) \to \tau / C_\tau \wedge C'''_a]$
2. $E + \{x : [\tau/C_\tau]\} \vdash e_1 : [\tau/C_1]$

we deduce the result. For the second case, we have trivially $E \vdash \mathbf{fix}(op) : [\tau/C]$ and $E \vdash op : [\tau/C']$ and the result.

Case mkpar. We have the following derivation:

$$\dfrac{\dfrac{}{E \vdash \mathbf{mkpar} : [(int \to \tau) \to (\tau\ par)/C_a]} \quad \dfrac{E + \{x : int\} \vdash e_1 : [\tau/C_1]}{E \vdash (\mathbf{fun}\ x \to e_1) : [int \to \tau / C_{int\to\tau} \wedge C_1]}}{E \vdash \mathbf{mkpar}\ (\mathbf{fun}\ x \to e_1) : [(\tau\ par)/C_a \wedge C_1 \wedge C_{int\to\tau} \wedge C_1]}$$

and the following one:

$$\dfrac{\forall i \quad E \vdash e_i[x \leftarrow i] : [\tau/C_i]}{E \vdash \langle e_0[x \leftarrow 0], \ldots, e_{p-1}[x \leftarrow (p-1)]\rangle : [(\tau\ par)/\mathcal{L}(\tau) \bigwedge_{i=0}^{p-1} C_i]}$$

We have naturally $\forall i$, $E \vdash i : int$. By application, $\forall i$, of the Lemma 17 to:

1. $E \vdash i : int$
2. $E + \{x : int\} \vdash e_1 : [\tau/C_1]$

we get that $E \vdash e_i[x \leftarrow i] : [\tau/C_i]$ where $e_1 \sqsubseteq e_i[x \leftarrow i]$. By application of the Lemma 12, we have:

$$\begin{array}{rcl} C_a & = & C_{(int\to\tau)\to(\tau\ par)} \wedge C'_a \\ & = & \mathcal{L}(\tau) \wedge C''_a \end{array}$$

By application, $\forall i$, of the Lemma 4, and the Lemma 3, we have that $Solve((\tau\ par)/\mathcal{L}(\tau) \bigwedge_{i=0}^{p-1} C_i) \neq \mathbf{False}$ and the result.

Case apply. We have the following derivation:

$$\dfrac{\dfrac{}{E \vdash \mathbf{apply} : [((\tau_1 \to \tau_2)\ par * (\tau_1\ par)) \to (\tau_2\ par)/C_a]} \quad \dfrac{\dfrac{\ldots}{E \vdash v_1 : [(\tau_1 \to \tau_2)\ par/C_1]} \quad \dfrac{\ldots}{E \vdash v_2 : [(\tau_2\ par)/C_2]}}{E \vdash (v_1, v_2) : [(\tau_1 \to \tau_2)\ par * \tau_2\ par/C_1 \wedge C_2]}}{E \vdash \mathbf{apply}\ (v_1, v_2) : [\tau_2\ par/C_a \wedge C_1 \wedge C_2]}$$

where

- $v_1 = \langle(\mathbf{fun}\ x \to e_0), \ldots, (\mathbf{fun}\ x \to e_{p-1})\rangle$
- $v_2 = \langle v_0, \ldots, v_{p-1}\rangle$

$$\frac{\forall i \quad E \vdash e_i[x \leftarrow v_i] : [\tau_2/C_i]}{E \vdash \langle e_0[x \leftarrow v_0], \ldots, e_{p-1}[x \leftarrow v_{p-1}]\rangle : [\tau_2\ par/\mathcal{L}(\tau_2) \bigwedge_i^{p-1} C_i]}$$

By the Lemma 12, we have that $C_a = \mathcal{L}(\tau_2) \wedge C'_a$. The type derivation of v_1 (remark of Lemma 16) is:

$$\frac{\forall i \quad \dfrac{E + \{x : [\tau_1/C_{\tau_1}\} \vdash e_i : [\tau_2/C_{e_i}]}{E \vdash (\mathbf{fun}\ x \rightarrow e_i) : [(\tau_1 \rightarrow \tau_2)/C_{e_i} \wedge C_{(\tau_1 \rightarrow \tau_2)}]}}{E \vdash v_1 : [(\tau_1 \rightarrow \tau_2)\ par/C_1]}$$

and for v_2:

$$\frac{\forall i \quad E \vdash v_i : [\tau_1/C_{v_i}]}{E \vdash v_2 : [(\tau_1\ par)/C_2]}$$

$\forall i$, by the Lemma 12, we have that $C_{v_i} = C_{\tau_1} \wedge C'_{v_i}$. We can apply, $\forall i$, the Lemma 17 to:

1. $E + \{x : [\tau_1/C_{\tau_1}]\} \vdash e_i : [\tau_2/C_{e_i}]$

2. $E \vdash v_i : [\tau_1/C_{v_i}]$

and we get $\forall i \quad E \vdash e_i[x \leftarrow v_i] : [\tau_2/C_i]$ and the result.

Case ifat. For the two cases (**True** and **False**), the proof is the same as **fst**.

Case put. The proof is like for **apply**, i.e., by application of the Lemma 17 to the $v^i_j = e_j[dst \leftarrow i]$. and by construction of the f_i (the list of **if then else**).

□

Lemma 21 (Preservation of the head reduction) *If* $e \overset{\varepsilon}{\rightharpoonup} e'$ *then* $e \sqsubseteq e'$.

Proof: by case on the rules.

Case of a δ-rules. Trivial by application of the Lemma 20.

Case of the β_{fun} rule. We have $e = ((\mathbf{fun}\ x \rightarrow e_1)\ v)$ and $e' = e_1[x \leftarrow v]$. Give $E \vdash e : [\tau/C]$. We have the following derivation:

$$\frac{\dfrac{\dfrac{\ldots}{E + \{x : [\tau_1/C_{\tau_1}]\} \vdash e_1 : [\tau/C_a]}}{E \vdash (\mathbf{fun}\ x \rightarrow e_1) : [\tau_1 \rightarrow \tau/C_{\tau_1 \rightarrow \tau} \wedge C_a]} \quad \dfrac{\ldots}{E \vdash v : [\tau_1/C_b]}}{E \vdash (\mathbf{fun}\ x \rightarrow e_1)\ v : [\tau/C]}$$

with $C = C_{\tau_1 \rightarrow \tau} \wedge C_a \wedge C_b$ and $Solve(C) \neq$ **False**. So we can take $\varphi \models C$. By application of the Lemma 12, we have $C_b = C_{\tau_1} \wedge C'_b$. We can apply the Lemma 17 to:

1. $E + \{x : [\tau_1/C_{\tau_1}]\} \vdash e_1 : [\tau/C_a]$

2. $E \vdash v : [\tau_1/C_b]$

We get $E \vdash e_1[x \leftarrow v] : [\tau/C']$ with $Solve(C') \neq$ **False**, $\varphi \models C'$ and the result.

Case of the δ_{let} rule. We have $e = (\mathbf{let}\ x = v\ \mathbf{in}\ e_1)$ and $e' = e_1[x \leftarrow v]$. Give $E \vdash e : [\tau/C]$. We have the following derivation:

$$\frac{\dfrac{\ldots}{E \vdash v : [\tau_1/C_1]} \quad \dfrac{\ldots}{E + \{x : Gen([\tau_1/C_1], E)\} \vdash e_1 : [\tau/C_2]}}{E \vdash e : [\tau/C]}$$

with $C = C_1 \wedge C_2 \wedge (\mathcal{L}(\tau) \Rightarrow \mathcal{L}(\tau_1))$ and $Solve(C) \neq$ **False**. So we can take $\varphi \models C$. By application of the Lemma 12 on the two premises (take $C_1 \wedge$**True** for the lemma), we get $E \vdash e' : [\tau/C']$ with $Solve(C') \neq$ **False**, $\varphi \models C'$ and the result. □

Proof: we have two cases.

- If we have $e \overset{\varepsilon}{\rightharpoonup} e'$, we deduce the result by application of the Lemma 21.

- Else, it is a *context rule*, so we have the following reduction:

$$\frac{e_1 \overset{\varepsilon}{\rightharpoonup} e'_1}{\Gamma(e_1) \rightharpoonup \Gamma(e'_1)}$$

with $e = \Gamma(e_1)$ and $e' = \Gamma(e'_1)$. Apply the Lemma 21 to the premise, we get $e_1 \sqsubseteq e'_1$. So by application of the Lemma 19, we deduce the result.

□

Corollary 2 (The subject reduction) *If $e \overset{*}{\rightharpoonup} e'$ then $e \sqsubseteq e'$.*

Proof: by trivial applications of the Proposition 3. □

If an expression is well-typed, the evaluation of this expression keeps the type.

4.3.3 Safety

Now, we have all the elements to prove the safety of our type system versus the evaluation of the expressions. We will prove that if an expression is well-typed and could not be reduced then it is a value. Thus, if an expression is evaluated to a value, with the subject reduction, we can prove that it is a well-typed value.

Normal Form

Definition 9 (Normal form) *We say that an expression e is in normal form if and only if $e \not\rightharpoonup$, i.e., there is no rule which could be applicated to e.*

We give a technical lemma for the reduction, that says there exists a δ-rule for all the operator applied to a value.

Lemma 22 (δ-rule for the value) *If $E \vdash (op\ v) : [\tau/C]$ then there exists $(op\ v) \overset{\varepsilon}{\rightharpoonup} e$ by a δ-rule.*

Proof: by trivial case on *op* and v and by using the Lemma 16. □

Lemma 23 (Progress) *If $\emptyset \vdash e : [\tau/C]$ then, e is a value or there exists e' such that $e \rightharpoonup e'$.*

Proof: by structural induction on e.

Case $e = c$ (constant) or $e = op$ (operator) or $e =$ (**fun** $x \rightarrow e_1$). e is a value, so the result.

Case $e = (e_1\ e_2)$. If e_1 is not a value then by inductive hypothesis we get that e_1 could be reduced so e could be reduced by a context rule (idem for e_2). If e_1 and e_2 are values then, by the Lemma 16, e_1 is an operator or a function. In the first case, we apply the Lemma 22 and e could be reduced by a δ-rule. In the second case, e could be reduced by a β_{fun} rule.

Case $e =$ **let** $x = e_1$ **in** e_2. Idem by inductive hypothesis and with the β_{let} rule.

Case $e = (e_1, e_2)$. If e_1 is not a value, then by inductive hypothesis, e could be reduced (idem for e_2). If e_1 and e_2 are values then e is a value and we have the result.

Case $e = \langle e_0, \ldots, e_{p-1} \rangle$. If there exists e_i which is not a value then e could be reduced (context rule). Else, like before, e is a value and we have the result. □

Corollary 3 (The Well-typed normal form are values) *If $\emptyset \vdash e : [\tau/C]$ and e is in normal form then e is a value.*

Proof: by trivial application of the last lemma and of the Lemma 16. □

Theorem 5 (Typing safety) *If $\emptyset \vdash e : [\tau/C]$ and $e \xrightarrow{*} e'$ and e' is in normal form, then e' is a value and there exists C' such that $\forall\varphi \in \phi_C$ then $\varphi \models C'$ and $\emptyset \vdash e' : [\tau/C']$.*

Remark: why C' and not C ? Because with the Lemma 12, we have $C = C_\tau \wedge C_1$ where C_1 are constraints on the sub-expression of e. After evaluation, some of this sub-expression could be reduced. Example: **let** f = (**fun** $a \to$ **fun** $b \to a$) **in** 1 have the type $[int/\mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]$. This expression reduced to 1 has the type *int*. Also, we have C' is less constrainted than C.

Proof: by the Corollary 2, we have $e \sqsubseteq e'$. So e' is a well-typed normal form, so by the Corollary 3, e' is value v and the result. □

Chapter 5

An Algorithm for type inference

In the second chapter, we have seen that, if a close expression has the type $[\tau/C]$, then it also has the type less general $[\tau'/C']$, ie, $[\tau'/C'] = \varphi([\tau/C])$ (stability by a substitution). Now, we will introduce the fact that there exists a type which generalizes all the other types for an expression. This calculus is known as type inference (or reconstruction of types). The existence of an algorithm for type inference is the necessary condition for a BSML compiler to guess the type of the expression without the help of the user. This algorithm is well-known for ML expression, by the name of "Damas-Milner" (note W). We will adapt this algorithm to our type system.

5.1 Presentation of algorithm W

5.1.1 Tools

Trivial instance

To begin, we define the notion of trivial instance $Inst(\sigma, V)$ of a type scheme with a set of fresh variables V (an infinite set of variables):

$$Inst(\forall\alpha_1, ..., \alpha_n.[\tau/C], V) = [\tau[\alpha_i \leftarrow \beta_i]/\mathcal{C}[\alpha_i \leftarrow \beta_i]], V \setminus \{\beta_1, ..., \beta_n\})$$

where $\beta_1, ..., \beta_n$ are n distinct variables of V.

Type unification

We say that a substitution φ is a unifier of two types τ_1, τ_2, if we have $\varphi(\tau_1) = \varphi(\tau_2)$. Two types could be unified if there exists an unifier for these two types. We say that an unifier φ of τ_1, τ_2 is more general, if any other unifier ψ of τ_1, τ_2, can be decomposed to $\theta \circ \varphi$ for a substitution θ. The main unifier, if is exists, represents the minimal modification to unify these two types. Any other unifier must make at least this modification. Later, the algorithm W needs a first-order unification between types.

Proposition 4 *If two types τ_1 and τ_2 could be unified, then there exists only one main unifier (apart from rename variables), noted $mgu(\tau_1, \tau_2)$.*

Justification: the set of the type is a free algebra of terms with the signature $T \cup \{*, \rightarrow, par)$ where T is the set of the free variables and the base types. An algorithm which could be used to calculate this main unifier is the Robinson's one.

The algorithm must have these three behaviours (this the case for the Robinson's one):

1. if $mgu(F)$ fails, then there does not exist a solution
2. $mgu(F)$ gives a substitution φ which is a solution and any other solution ψ could be written: $\psi = \theta \circ \varphi$ for a substitution θ (φ is the main)
3. the unifier do not introduce new variables, i.e, all the binding variables in τ_1, or in τ_2 are out of reach of the unifier.

types and gives our wanted unifier.

$$\begin{aligned}
Rob(\emptyset) &= id\\
Rob(\{\alpha \overset{?}{=} \alpha\} \cup R) &= Rob(R)\\
Rob(\{\alpha \overset{?}{=} \tau\} \cup R) &= Rob(R[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau]\\
Rob(\{\tau \overset{?}{=} \alpha\} \cup R) &= Rob(R[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau]\\
Rob(\{(\tau_1 \rightarrow \tau_2) \overset{?}{=} (\tau_1' \rightarrow \tau_2')\} \cup R) &= Rob(\{\tau_1 \overset{?}{=} \tau_1'\} \cup \{\tau_2 \overset{?}{=} \tau_2'\} \cup R)\\
Rob(\{(\tau_1 * \tau_2) \overset{?}{=} (\tau_1' * \tau_2')\} \cup R) &= Rob(\{\tau_1 \overset{?}{=} \tau_1'\} \cup \{\tau_2 \overset{?}{=} \tau_2'\} \cup R)\\
Rob(\{(\tau_1\ par) \overset{?}{=} (\tau_1'\ par)\} \cup R) &= Rob(\{\tau_1 \overset{?}{=} \tau_1'\} \cup R)
\end{aligned}$$

All the other cases, *Rob* fails and there is no unifier. Now $mgu(\tau_1, \tau_2) = Rob(\{\tau_1 \overset{?}{=} \tau_2\})$

5.1.2 The algorithm W

The W's algorithm takes a syntaxic valid expression of our language, and gives a type for this expression with its constraints and a substitution for the environment. This type will be the most general type for the expression.

- **Input**: an environment E, an expression e, a set V of fresh variables.
- **Output**: $([\tau/C], \varphi, V')$ where τ is the type infered, φ is the substitution to do on the free variables of E, V' is V without the variables that W uses and $Solve(C) \neq$ **False**.

Variable: if e is a variable x with $x \in Dom(E)$ then $[\tau/C] = Inst(E(x))$, $V' = V$ and $\varphi = id$.

Constant/Operator: if e is a constant c or an operator *op* then
$[\tau/C] = Inst(TC(e))$, $V' = V$ and $\varphi = id$.

Function: if e is **fun** $x \rightarrow e_1$ then given $\alpha \in V$ and
$([\tau_1/C_1], \varphi_1, V_1) = W(E + \{x : \alpha\}, e_1, V \setminus \{\alpha\})$ then
$C = C_{\varphi_1(\alpha)\rightarrow\tau_1} \wedge C_1$, if $Solve(C) \neq$ **False** then $\tau = \varphi_1(\alpha) \rightarrow \tau_1$, and $\varphi = \varphi_1$ and $V' = V_1$.

Application: if e is an application $(e_1\ e_2)$ then given $([\tau_1/C_1], \varphi_1, V_1) = W(E, e_1, V)$ and
$([\tau_2/C_2], \varphi_2, V_2) = W(\varphi_1(E), e_2, V1)$ and given $\alpha \in V_2$ and
given $\mu = mgu\{\varphi_2(\tau_1) = \tau_2 \rightarrow \alpha\}$ and
$[\tau_1'/C_1'] = \mu \circ \varphi_2([\tau_1/C_1])$ and $[\tau_2'/C_2'] = \mu([\tau_2/C_2])$ and
$C = C_1' \wedge C_2'$ and If $Solve(C) \neq$ **False** then
$\tau = \mu(\alpha)$ and $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ and $V' = V_2 \setminus \{\alpha\}$.

Couple: if e is a couple (e_1, e_2) then given $([\tau_1/C_1], \varphi_1, V_1) = W(E, e_1, V)$ and
$([\tau_2/C_2], \varphi_2, V_2) = W(\varphi_1(E), e_2, V_1)$ and
$[\tau_1'/C_1'] = \varphi_2([\tau_1/C_1])$ and $C = C_1' \wedge C_2$ then
If $Solve(C) \neq$ **False** then $\tau = \tau_1' * \tau_2$, $\varphi = \varphi_2 \circ \varphi_1$ and $V = V_2$.

Let: if e is **let** $x = e_1$ **in** e_2 then given $([\tau_1/C_1], \varphi_1, V_1) = W(E, e_1, V)$ and
$([\tau_2/C_2], \varphi_2, V_2) = W(\varphi_1(E) + \{x : Gen([\tau_1/C_1], \varphi_1(E))\}, e_2, V_1)$ and
$C_1' = \varphi_2([\tau_1/C_1])$ and $C = C_1' \wedge C_2 \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1'))$ then
If $Solve(C) \neq$ **False** then $\tau = \tau_2$, and $\varphi = \varphi_2 \circ \varphi_1$ and $V' = V_2$.

Other: All the other cases are errors, in particular when *mgu* fails or $Solve(C) =$ **False**.

Remark: at the end, we can apply a function *Simple* (notably in the implementation) to simplify the constraints for the users.

Accepted

We run the following expression e:

```
fun x -> mkpar(fun i -> x)
```

$$\begin{array}{rcl}
W(\{x:\alpha, i:\gamma\}, x, V\setminus\{\alpha,\beta,\gamma\}) & = & (\alpha, id, V\setminus\{\alpha,\beta,\gamma\}) \\
W(\{x:\alpha\}, \mathbf{fun}\ i \to x, V\setminus\{\alpha,\beta\}) & = & ([\gamma \to \alpha/\mathcal{L}(\gamma) \Rightarrow \mathcal{L}(\alpha)], id, V\setminus\{\alpha,\beta,\gamma\}) \\
W(\{x:\alpha\}, mkpar, V\setminus\{\alpha\}) & = & ([(int \to \beta) \to (\beta\ par)/\mathcal{L}(\beta)], id, V\setminus\{\alpha,\beta\}) \\
W(\{x:\alpha\}, mkpar(\mathbf{fun}\ i \to x), V\setminus\{\alpha\}) & = & ([\alpha/\mathcal{L}(\alpha)], [\beta \leftarrow \alpha, \gamma \leftarrow int], V\setminus\{\alpha,\beta,\gamma\}) \\
W(\emptyset, e, V) & = & ([\alpha \to (\alpha\ par)/\mathcal{L}(\alpha)], [\beta \leftarrow \alpha, \gamma \leftarrow int], V\setminus\{\alpha,\beta,\gamma\})
\end{array}$$

Rejected

We run the following expression e:

```
replicate(replicate 2)
```

with replicate, like before.

$$\begin{array}{rcl}
W(\emptyset, 2, V\setminus\{\alpha,\beta\}) & = & (int, id, V\setminus\{\alpha,\beta\}) \\
W(\emptyset, replicate, V\setminus\{\alpha,\beta\}) & = & ([\beta \to (\beta\ par)/\mathcal{L}(\beta)], id, V\setminus\{\alpha,\beta\}) \\
W(\emptyset, replicate\ 2, V\setminus\{\alpha,\beta\}) & = & ((int\ par), [\beta \leftarrow int], V\setminus\{\alpha,\beta\}) \\
W(\emptyset, replicate, V\setminus\{\alpha\}) & = & ([\alpha \to (\alpha\ par)/\mathcal{L}(\alpha)], id, V\setminus\{\alpha,\beta\}) \\
W(\emptyset, e, V) & = & \text{error}
\end{array}$$

There is an error because the *mgu* calculate that $\mu = [\alpha \leftarrow (int\ par)]$ and we have the constraint $\mathcal{L}(\alpha)$ *Solve* to **False**.

Other rejected

We run the following expression e:

```
(fun a -> fun b -> a) 1 (mkpar (fun i -> 5))
```

There is an error because the *mgu* calculates that we have the constraints *Solve* to **False**. For all those examples, see the complete derivation in chapter 6 (implementation).

5.2 Correction

Theorem 6 (Correction of W) *Given an expression e, an environment E and a set of type variable V. If* $([\tau/C], \varphi, V') = W(e, E, V)$ *(so without error) then we have* $\varphi(E) \vdash e : [\tau/C]$.

Remark: we use the same notation as in the algorithm.

Remark: we will prove that W do not introduce constraints that *Solve* to **False**.

Proof: by structural induction on e.

Case *Var*. We have $[\tau/C] = Inst(E(x))$, $V' = V$ and $\varphi = id$. With the definition of *Inst*, it is easy to see that we have $[\tau/C] \leq E(x)$ for a substitution ψ, with $\{\alpha_1, \ldots, \alpha_n\} = Dom(\psi)$, $C_{\psi(\alpha_i)} = C_{\beta_i} = \mathbf{True}$, so we can ignore these constraints with the definition of a substitution on a type scheme (we work modulo $D \wedge \mathbf{True} = D$). Because in the environment there is no constraint *Solve* to **False** and ψ is a renaming that introduce fresh variables, it is easy to see that we have $Solve(C) \neq \mathbf{False}$. In the same way, (because $\varphi = id$) we have $\varphi(E) = E$, so we can apply the (Var) rule and we have $\varphi(E) \vdash x : [\tau/C]$

Case *Const*. We have $[\tau/C] = Inst(TC(c))$, $V' = V$ and $\varphi = id$, like in *Var*, we have $[\tau/C] \leq TC(c)$ for a substitution ψ and $C = \mathbf{True}$ (TC of the constants have no constraint). We have $\varphi(E) = E$, so we

Case *Op*. We have $[\tau/C] = \mathit{Inst}(TC(\mathit{op}))$, $V' = V$ and $\varphi = \mathit{id}$, like in Var, we have $[\tau/C] \leq TC(\mathit{op})$ for a substitution ψ and $\mathit{Solve}(C) \neq$ **False** (see the TC of the operator). We have $\varphi(E) = E$, so we can apply the (Op) rule and we have $\varphi(E) \vdash \mathit{op} : [\tau/C]$

Case *Fun*. We have $e =$ **fun** $x \rightarrow e_1$. By inductive hypothesis apply to e_1, we have:

$$\varphi_1(E + \{x : \alpha\}) \vdash e_1 : [\tau_1/C_1]$$

with $\mathit{Solve}(C_1) \neq$ **False**. The algorithm calculates that

$$\begin{array}{rcll} \mathit{Solve}(C_1 \wedge C_{\varphi_1(\alpha)\rightarrow\tau_1}) & \neq & \textbf{False} & \text{by the Lemma 1 we have} \\ \mathit{Solve}(C_{\varphi_1(\alpha)\rightarrow\tau_1}) & \neq & \textbf{False} & \text{by definition of the rule of construct we have} \\ \mathit{Solve}(C_{\varphi_1(\alpha)} \wedge C_{\tau_1} \wedge (\mathcal{L}(\varphi_1(\alpha)) \Rightarrow \mathcal{L}(\tau_1))) & \neq & \textbf{False} & \text{by the Lemma 1 we have} \\ \mathit{Solve}(C_{\varphi_1(\alpha)}) & \neq & \textbf{False} & \end{array}$$

So we do not introduce incorrect constraints on the environment. A substitution on an environment is point to point so we have:

$$\begin{array}{rll} & \varphi_1(E) + \varphi_1(\{x : \alpha\}) \vdash e_1 : [\tau_1/C_1] & \text{by definition} \\ = & \varphi_1(E) + \{x : [\varphi_1(\alpha)/C_{\varphi_1(\alpha)}]\} \vdash e_1 : [\tau_1/C_1] & \text{apply the } (\mathit{Fun}) \text{ rule} \\ = & \varphi_1(E) \vdash a : [\varphi_1(\alpha) \rightarrow \tau_1/C_{\varphi_1(\alpha)\rightarrow\tau_1} \wedge C_1] & \end{array}$$

and the result.

Case *App*. We have $e = (e_1\ e_2)$. By inductive hypothesis on e_1 and e_2, we have:

1. $\varphi_1(E) \vdash e_1 : [\tau_1/C_1]$
2. $\varphi_2 \circ \varphi_1(E) \vdash e_2 : [\tau_2/C_2]$

With $\mathit{Solve}(C_1) \neq$ **False** and $\mathit{Solve}(C_2) \neq$ **False**. The algorithm calculates μ is a main unifier of $\{\varphi_2(\tau_1) = \tau_2 \rightarrow \alpha\}$, $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ and $[\tau_1'/C_1'] = \mu \circ \varphi_2([\tau_1/C_1])$ and $[\tau_2'/C_2'] = \mu([\tau_2/C_2])$ and $C = C_1' \wedge C_2'$ with $\mathit{Solve}(C) \neq$ **False**. By the Lemma 1 we have $\mathit{Solve}(C_1') \neq$ **False** (idem for C_2' and for all the constraints).
By application of the substitution $\mu \circ \varphi_2$ to 1) and the Lemma 13, we have:

$$\begin{array}{rll} & \varphi(E) \vdash e_1 : \mu \circ \varphi_2([\tau_1/C_1]) & \text{by definition} \\ = & \varphi(E) \vdash e_1 : [\mu \circ \varphi_2(\tau_1)/C_1'] & \end{array}$$

and by application of the substitution μ to 2) and the Lemma 13, we have:

$$\begin{array}{rll} & \varphi(E) \vdash e_2 : \mu([\tau_2/C_2]) & \text{by definition} \\ = & \varphi(E) \vdash e_2 : [\mu(\tau_2)/C_2'] & \end{array}$$

Because μ is a main unifier, we have $\mu \circ \varphi_2(\tau_1) = \mu(\tau_2) \rightarrow \mu(\alpha)$
So we can apply the *App* type inductive rule and we obtain:

$$\varphi(E) \vdash (e_1\ e_2) : [\mu(\alpha)/C]$$

and the result.

Case *Couple*. We have $e = (e_1, e_2)$. By induction hypothesis on e_1 and e_2, we have:

1. $\varphi_1(E) \vdash e_1 : [\tau_1/C_1]$
2. $\varphi_2 \circ \varphi_1(E) \vdash e_2 : [\tau_2/C_2]$

not *Solve* to **False**). By the application of φ_2 on 1) and the Lemma 13, we have:

$$\begin{array}{rl} & \varphi(E) \vdash e_1 : \varphi_2([\tau_1/C_1]) \quad \text{by definition} \\ = & \varphi(E) \vdash e_1 : [\tau_1'/C_1'] \end{array}$$

By application of the type inductive rule (*Couple*) we have:

$$\varphi(E) \vdash: (e_1, e_2) : [\tau_1' * \tau_2/C]$$

and the result.

Case *Let*. We have $e = \textbf{let } x = e_1 \textbf{ in } e_2$. By inductive hypothesis on e_1 and e_2, we have:

1. $\varphi_1(E) \vdash e_1 : [\tau_1/C_1]$
2. $\varphi_2(\varphi_1(E) + \{x : Gen([\tau_1/C_1], \varphi_1(E))\}) \vdash e_2 : [\tau_2/C_2]$

with $Solve(C_1) \neq$ **False** and $Solve(C_2) \neq$ **False**. The algorithm calculates that $\varphi = \varphi_2 \circ \varphi_1$ and $[\tau_1'/C_1'] = \varphi_2([\tau_1/C_1])$ and $\tau = \tau_2$ and

$$C = C_2 \wedge C_1' \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1'))$$

and $Solve(C) \neq$ **False** (and all the constraints are not *Solve* to **False**). If necessary, we rename all the generalised variables. Now they are out of reach of φ_2 With the Proposition 1 we have:

$$\begin{array}{rcl} \varphi_2(Gen([\tau_1/C_1], \varphi_1(E))) & = & Gen(\varphi_2([\tau_1/C_1]), \varphi_2(\varphi_1(E))) \\ & = & Gen([\tau_1'/C_1'], \varphi(E)) \end{array}$$

a substitution is point to point so we have:

$$\varphi(E) + \{x : Gen([\tau_1'/C_1'], \varphi(E))\} \vdash e_2 : [\tau_2/C_2]$$

By application of the substitution φ_2 to 1) and the Lemma 13, we have:

$$\begin{array}{rl} & \varphi(E) \vdash e_1 : \varphi_2([\tau_1/C_1]) \quad \text{by definition} \\ = & \varphi(E) \vdash e_1 : [\tau_1'/C_1'] \end{array}$$

By application of the type inductive rule *Let*, we have:

$$\varphi(E) \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : [\tau_2/C]$$

and the result.

□

5.3 Completeness

Theorem 7 *Give an expression e, an environment E and V an infinite set of variables as $V \cap \mathcal{F}(E) = \emptyset$. If there exists a type scheme $[\tau_n/C_n]$ and a subsitution φ' as $\varphi'(E) \vdash e : [\tau_n/C_n]$ then $([\tau/C], \varphi, V') = W(e, E, V)$ is define and there exists a substitution ψ as*

$$[\tau_n/C_n] = \psi([\tau/C]), \quad \textit{and} \quad \varphi' = \psi \circ \varphi \textit{ out of } V.$$

Remark: we said that $\varphi' = \psi \circ \varphi$ is out of V, if $\forall \alpha \notin V\ \varphi'(\alpha) = \psi(\varphi(\alpha))$. The condition $\varphi' = \psi \circ \varphi$ out of V means that the two substitutions behave in the same manner with the initial typing problem.

Remark: with the hypothesis, if $([\tau/C], \varphi, V') = W(e, E, V)$ then $V' \subseteq V$ and the variables of V' are not free in $[\tau/C]$ and are out of reach of φ. So, we get $V' \cap \mathcal{F}(\varphi(E)) = \emptyset$.

Remark: it is easy to see that, if $\mathcal{F}([\tau/C]) \cap V = \emptyset$ and if $\varphi = \psi$ out of V then $\varphi([\tau/C]) = \psi([\tau/C])$.

Proof: by a stuctural induction on e.

Case $e = x$. By hypothesis, we have $\varphi'(E) \vdash x : [\tau_n/C_n]$, $x \in Dom(\varphi'(E))$ and $[\tau_n/C_n] \leq \varphi'(E)(x)$ with $Solve(C_n) \neq$ **False**. Trivially, we get, $x \in Dom(E)$. Note $E(x) = \forall\alpha_1, \dots, \alpha_n.[\tau_x/C_x]$ with the α_i choose in V' and out of reach of φ'. We have $W(E, x, V)$ is define like:

$$[\tau/C] = [\tau_x[\alpha_i \leftarrow \beta_i]/C_x[\alpha_i \leftarrow \beta_i]] \qquad \varphi = id \qquad V' = V \setminus \{\beta_1, \dots, \beta_n\}$$

for $\beta_1, \dots, \beta_n \in V$. The α_i are out of reach of φ' so we have:

$$\varphi'(E(x)) = \forall\alpha_1, \dots, \alpha_n.[\varphi'(\tau_x)/\varphi'(C_x) \bigwedge_{\gamma_i \in \mathcal{F}(\tau_x/C_x) \cap Dom(\varphi')} C_{\varphi'(\gamma_i)}]$$

Note ρ, the substitution on the α_i as $\tau_n = \rho(\varphi'(\tau_x))$. We take: $\psi = \rho \circ \varphi' \circ [\beta_i \leftarrow \alpha_i]$ We have by definition:

$$\begin{array}{rcll}
\psi([\tau/C]) & = & \rho(\varphi'(\tau_x))/\psi(C) \bigwedge_{\omega_i \in \mathcal{F}(\tau/C) \cap Dom(\psi)} C_{\psi(\omega_i)} & \text{by definition} \\
& = & \tau_n/\rho(\varphi'(C_x)) \bigwedge_{\omega_i \in \mathcal{F}(\tau/C) \cap Dom(\psi)} C_{\psi(\omega_i)} & \text{by definition} \\
& = & \tau_n/\rho(\varphi'(C_x)) \bigwedge_{\omega_i \in \mathcal{F}(\tau_x[\beta_i \leftarrow \alpha_i]/C_x[\beta_i \leftarrow \alpha_i]) \cap Dom(\rho \circ \varphi' \circ [\beta_i \leftarrow \alpha_i])} C_{\rho \circ \varphi' \circ [\beta_i \leftarrow \alpha_i](\omega_i)} & \text{by the Lemma 11}
\end{array}$$

$$= \tau_n/\rho(\varphi'(C_x)) \bigwedge_{\omega_i \in \mathcal{F}(\tau_x[\beta_i \leftarrow \alpha_i]/C_x[\beta_i \leftarrow \alpha_i]) \cap Dom(\rho \circ \varphi' \circ [\beta_i \leftarrow \alpha_i])} \rho(C_{\varphi' \circ [\beta_i \leftarrow \alpha_i](\omega_i)}) \bigwedge_{\gamma_i \in \mathcal{F}(\varphi' \circ [\beta_i \leftarrow \alpha_i](\omega_i)) \cap Dom(\rho)} C_{\rho(\gamma_i)}$$

By definition, we have:

$$C_n = \rho(\varphi'(C_x)) \bigwedge_{\omega_i \in \mathcal{F}(\tau_x/C_x) \cap Dom(\varphi')} \rho(C_{\varphi'(\omega_i)}) \bigwedge_{\alpha_1, \dots, \alpha_n} C_{\rho(\alpha_i)}$$

It is easy to see that:

$$\bigwedge_{\omega_i \in \mathcal{F}(\tau_x/C_x) \cap Dom(\varphi')} \rho(C_{\varphi'(\omega_i)}) = \bigwedge_{\omega_i \in \mathcal{F}(\tau_x[\beta_i \leftarrow \alpha_i]/C_x[\beta_i \leftarrow \alpha_i]) \cap Dom(\rho \circ \varphi' \circ [\beta_i \leftarrow \alpha_i])} \rho(C_{\varphi' \circ [\beta_i \leftarrow \alpha_i](\omega_i)})$$

(because, if $x_i \notin Dom(\varphi')$ or x_i out of reach of φ', $C_{\varphi'(x_i)} =$ **True** and we ignore this constraint)
We have $Dom(\rho) = \alpha_1, \dots, \alpha_n$ and $\alpha_1, \dots, \alpha_n \subseteq \mathcal{F}(\varphi' \circ [\beta_i \leftarrow \alpha_i](\omega_i))$ (because the α_i are in τ_x and are out of reach of φ' like for the α_i) and with the reason we have:

$$\bigwedge_{\alpha_1, \dots, \alpha_n} C_{\rho(\alpha_i)} = \bigwedge_{\gamma_i \in \mathcal{F}(\varphi' \circ [\beta_i \leftarrow \alpha_i](\omega_i)) \cap Dom(\rho)} C_{\rho(\gamma_i)}$$

So we have $[tau_n/C_n] = \psi([\tau/C])$.
All the variables $\alpha \notin V$ are not a α_i or a β_i, so $\psi(\alpha) = \rho(\varphi'(\alpha)) = \varphi'(\alpha)$ so $\psi(\alpha) = \rho(\varphi'(\alpha)) = \varphi'(\alpha)$. We have the result because $\varphi = id$.

Case $e = op$. Like (Var).

Case $e = c$. Trivial with $\psi = id$.

Case $e =$ **fun** $x \to e_1$. We have the following derivation:

$$\frac{\varphi'(E) + \{x : [\tau_{n2}/C_{\tau_{n2}}]\} \vdash e_1 : [\tau_{n1}/C_{n1}]}{\varphi'(E) \vdash (\textbf{fun } x \to e_1) : [\tau_{n2} \to \tau_{n1}/C_{(\tau_{n2} \to \tau_{n1})} \wedge C_{n1}]}$$

with $Solve(C_{(\tau_{n2} \to \tau_{n1})} \wedge C_{n1}) \neq$ **False**. Take $\alpha \in V$, like in the algorithm. We note:

$$E_1 = E + \{x : \alpha\} \qquad \text{and} \qquad \varphi'_1 = \varphi' \cup \{\alpha \leftarrow \tau_{n2}\}$$

We have:

$$\begin{array}{rcll}
\varphi'_1(E_1) & = & \text{by definition} & \\
& = & \varphi'_1(E) + \{x : \tau_{n2}/C_{\tau_{n2}}\} & \text{because } \alpha \notin \mathcal{F}(E) \\
& = & \varphi'(E) + \{x : \tau_{n2}/C_{\tau_{n2}}\} &
\end{array}$$

$$([\tau_1/C_1], \varphi_1, V_1) = W(e_1, E_1, V \setminus \{\alpha\}) \qquad [\tau_{n1}/C_{n1}] = \psi_1([\tau_1/C_1]) \qquad \varphi_1' = \psi_1 \circ \varphi_1 \text{ out of } V \setminus \{\alpha\}$$

So it is easy to see that we have $W(e, E, V)$ is well defined. We take $\psi = \psi_1$. By inductive hypothesis, we have:

$$C_{n1} = \psi_1(C_1) \bigwedge_{\gamma_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\psi_1)} C_{\psi_1(\gamma_i)}$$

For $\psi([\tau/C_{\varphi_1(\alpha)\to\tau_1} \wedge C_1])$ we have by definition:

$$\psi([\tau/C_{\varphi_1(\alpha)\to\tau_1} \wedge C_1] \quad = \quad \psi(\tau)/\psi(C_1) \wedge \psi(C_{\varphi_1(\alpha)\to\tau_1}) \bigwedge_{\gamma_i \in \mathcal{F}(\tau/C_{\varphi_1(\alpha)\to\tau_1} \wedge C_1) \cap Dom(\psi)} C_{\psi(\gamma_i)}$$

We have:

$$\begin{array}{rcll}
\psi(\tau) & = & \psi(\varphi_1(\alpha) \to \tau_1) & \text{by definition of } \tau \text{ in the algorithm} \\
& = & \psi_1(\varphi_1(\alpha) \to \tau_1) & \text{by definition of } \psi \\
& = & \varphi_1'(\alpha) \to \psi_1(\tau_1) & \text{because } \alpha \notin V \setminus \{\alpha\} \\
& = & \tau_{n2} \to \psi_1(\tau_1) & \text{by construction of } \varphi_1' \\
& = & \tau_{n2} \to \tau_{n1} & \text{by construction of } \psi_1
\end{array}$$

We have $Dom(\psi) = Dom(\psi_1)$, $\tau = \varphi_1(\alpha) \to \tau_1$ and by a the Lemma 9, we have $\mathcal{F}(C_{\varphi_1(\alpha)\to\tau_1}) \subseteq \mathcal{F}(\varphi_1(\alpha) \to \tau_1)$. We work modulo the commutativity of the $\wedge$ operator and $D \wedge D = D$, so we can have by distribution of the free variables and by definition on free variables on a type schemes:

$$\begin{array}{rll}
= & \psi(C_1) \wedge \psi(C_{\varphi_1(\alpha)\to\tau_1}) \bigwedge_{\gamma_i \in \mathcal{F}(\tau/C_{\varphi_1(\alpha)\to\tau_1} \wedge C_1) \cap Dom(\psi)} C_{\psi(\gamma_i)} & \\
= & \psi(C_1) \bigwedge_{\gamma_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\psi)} C_{\psi(\gamma_i)} \wedge \psi(C_{\varphi_1(\alpha)\to\tau_1}) \bigwedge_{\gamma_i \in \mathcal{F}(\varphi_1(\alpha)\to\tau_1) \cap Dom(\psi)} C_{\psi(\gamma_i)} & \text{by inductive hypothesis} \\
= & C_{n1} \wedge \psi(C_{\varphi_1(\alpha)\to\tau_1}) \bigwedge_{\gamma_i \in \mathcal{F}(\varphi_1(\alpha)\to\tau_1) \cap Dom(\psi)} C_{\psi(\gamma_i)} & \text{by the Lemma 11} \\
= & C_{n1} \wedge C_{\psi(\varphi_1(\alpha)\to\tau_1)} & \text{by definition} \\
= & C_{n1} \wedge C_{\tau_{n2}\to\tau_{n1}} &
\end{array}$$

So we have $[\tau_{n2} \to \tau_{n1}/C_{\tau_{n2}\to\tau_{n1}} \wedge C_{n1}] = \psi([\tau/C_\tau \wedge C_1])$.
Furthermore, for all variable γ out of V, we have:

$$\begin{array}{rcll}
\psi(\varphi(\gamma)) & = & \psi_1(\varphi_1(\gamma)) & \text{by definition of } \psi \text{ and } \varphi \\
& = & \varphi_1'(\gamma) & \text{because } \gamma \notin V \\
& = & \varphi_1(\gamma) & \text{because } \gamma \notin V \text{ give } \gamma \neq \alpha
\end{array}$$

and the result.

Case $e = (e_1\ e_2)$. We have the following derivation:

$$\frac{\varphi'(E) \vdash e_1 : [\tau_{n2} \to \tau_{n1}/C_{n1}] \quad \varphi'(E) \vdash e_2 : [\tau_{n2}/C_{n2}]}{\varphi'(E) \vdash (e_1\ e_2) : [\tau_{n1}/C_{n1} \wedge C_{n2}]}$$

Apply the inductive hypothesis to the first premise, we obtain:

$$([\tau_1/C_1], \varphi_1, V_1) = W(e_1, E, V) \quad \text{and} \quad [\tau_{n2} \to \tau_{n1}/C_{n1}] = \psi_1([\tau_1/C_1]) \quad \text{and} \quad \varphi' = \psi_1 \circ \varphi_1 \text{ out of } V$$

In particular, $\varphi'(E) = \psi_1 \circ \varphi_1(E)$. We apply the inductive hypothesis to $e_2, \varphi_1(E), V_1, \tau_{n2}$ and ψ_1. We have $\mathcal{F}(\varphi_1(E)) \cap V_1 = \emptyset$ with the remark of the beginning. We obtain:

$$([\tau_2/C_2], \varphi_2, V_2) = W(e_2, \varphi_1(E), V_1) \quad \text{and} \quad [\tau_{n2}/C_{n2}] = \psi_2([\tau_2/C_2]) \quad \text{and} \quad \psi_1 = \psi_2 \circ \varphi_2 \text{ out of } V_1$$

We have $\mathcal{F}([\tau_1/C_1]) \cap V_1 = \emptyset$, so $\psi_1([\tau_1/C_1]) = \psi_2 \circ \varphi_2([\tau_1/C_1])$. Write $\psi_3 = \psi_2 + \{\alpha \leftarrow \tau_{n1}\}$ (α is a variable of V_2 and out of reach of ψ_2) so ψ_3 prolongs ψ_2. We have:

$$\begin{array}{rcccccc}
\psi_3(\varphi_2(\tau_1)) & = & \psi_2(\varphi_2(\tau_1)) & = & \psi_1(\tau_1) & = & \tau_{n2} \to \tau_{n1} \\
\psi_3(\tau_2 \to \alpha) & = & \psi_2(\tau_2) \to \tau_{n1} & = & \tau_{n2} \to \tau_{n1} & &
\end{array}$$

We take $\psi = \psi_4$. The algorithm calculates:

$$\begin{array}{rcl}
[\tau_1'/C_1'] & = & \mu \circ \varphi_2([\tau_1/C_1]) \quad \text{by definition} \\
& = & \mu([\varphi_2(\tau_1)/\varphi_2(C_1) \displaystyle\bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\varphi_2)} C_{\varphi_2(\beta_i)}] \quad \text{by definition} \\
& = & [\mu \circ \varphi_2(\tau_1)/\mu \circ \varphi_2(C_1) \displaystyle\bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\varphi_2)} \mu(C_{\varphi_2(\beta_i)}) \bigwedge_{\gamma_i \in \ \varphi_2([\tau_1/C_1]) \cap Dom(\mu)} C_{\mu(\gamma_i)}]
\end{array}$$

and

$$\begin{array}{rcl}
[\tau_2'/C_2'] & = & \mu([\tau_2/C_2]) \\
& = & [\mu(\tau_2)/\mu(C_2) \displaystyle\bigwedge_{\gamma_i \in \mathcal{F}(\tau_2/C_2) \cap Dom(\mu)} C_{\mu(\gamma_i)}]
\end{array}$$

and $[\tau/C] = [\mu(\alpha)/C_1' \wedge C_2']$. So for $\psi([\tau/C])$ we have:

$$\psi(\tau) \quad = \quad \psi_4(\mu(\alpha)) \quad = \quad \psi_3(\alpha) \quad = \quad \tau_{n1} \quad = \quad \tau_n$$

and for the constraints we have:

$$\begin{array}{rl}
= & \psi(C) \displaystyle\bigwedge_{\gamma_i \in \mathcal{F}(\tau/C) \cap Dom(\psi)} C_{\psi(\gamma_i)} \quad \text{by definition} \\
= & \psi_4(C_1') \wedge \psi_4(C_2') \displaystyle\bigwedge_{\gamma_i \in \mathcal{F}(\tau/C) \cap Dom(\psi)} C_{\psi(\gamma_i)} \quad \text{by definition} \\
= & \psi_4 \circ \mu \circ \varphi_2(C_1) \wedge \psi_4 \circ \mu(C_2) \displaystyle\bigwedge_{\gamma_i \in \mathcal{F}(\tau/C) \cap Dom(\psi)} C_{\psi(\gamma_i)} \\
& \displaystyle\bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\varphi_2)} \psi_4 \circ \mu(C_{\varphi_2(\beta_i)}) \bigwedge_{\gamma_i \in \ \varphi_2([\tau_1/C_1]) \cap Dom(\mu)} \psi_4(C_{\mu(\gamma_i)}) \bigwedge_{\gamma_i \in \mathcal{F}(\tau_2/C_2) \cap Dom(\mu)} \psi_4(C_{\mu(\gamma_i)})
\end{array}$$

By definition of $\psi_4 \circ \mu$ and the fact that $\alpha \notin \mathcal{F}(C_2)$ (because $\alpha \in V_2$), we have that $\psi_4 \circ \mu(C_2) = \psi_2(C_2)$. For the same reasons ($\alpha$ out of reach of ψ_2) we have $\psi_4 \circ \mu \circ \varphi_2(C_1) = \psi_1(C_1)$. So we have:

$$\begin{array}{rl}
= & \psi_1(C_1) \wedge \psi_2(C_2) \displaystyle\bigwedge_{\gamma_i \in \mathcal{F}(\tau/C) \cap Dom(\psi)} C_{\psi_4(\gamma_i)} \\
& \displaystyle\bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\varphi_2)} \psi_4 \circ \mu(C_{\varphi_2(\beta_i)}) \bigwedge_{\gamma_i \in \ \varphi_2([\tau_1/C_1]) \cap Dom(\mu)} \psi_4(C_{\mu(\gamma_i)}) \bigwedge_{\gamma_i \in \mathcal{F}(\tau_2/C_2) \cap Dom(\mu)} \psi_4(C_{\mu(\gamma_i)})
\end{array}$$

By the Lemma 11, we have:

$$\begin{array}{rcl}
C_{\psi_4 \circ \mu(\varphi_2(\beta_i))} & = & \psi_4 \circ \mu(C_{\varphi_2(\beta_i)}) \displaystyle\bigwedge_{\gamma_i \in \mathcal{F}(\varphi_2(\beta_i)) \cap Dom(\psi_4 \circ \mu)} C_{\psi_4 \circ \mu(\gamma_i)} \\
C_{\psi_4 \circ \mu(\gamma_i)} & = & \psi_4(C_{\mu(\beta_i)}) \displaystyle\bigwedge_{\omega_i \in \mathcal{F}(\mu(\beta_i)) \cap Dom(\psi_4)} C_{\psi_4(\omega_i)}
\end{array}$$

We work modulo $D \wedge D = D$ and $D \wedge \textbf{True} = D$. For the variables γ_i that are out of reach of ψ_4 we have $C_{\psi_4(\gamma_i)} = \textbf{True}$ and for all the variables $\beta_i \in (\mathcal{F}([\tau_2/C_2]) \cap Dom(\mu)) \cup (\varphi_2([\tau_1/C_1]) \cap Dom(\mu)) \cup (\mathcal{F}([\tau_1/C_1]) \cap Dom(\varphi_2))$ we have $\beta_i \in \mathcal{F}([\tau/C])$ (and vice-versa modulo β_i out of reach of a substitution). We have by some applications of the Lemma 11 and Lemma 9 (we transform the $\psi_4(D_\omega)$ to $D_{\psi_4(\omega)}$ and also for μ):

$$= \quad \psi_1(C_1) \wedge \psi_2(C_2) \bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\psi_4 \circ \mu \circ \varphi_2)} C_{\psi_4 \circ \mu \circ \varphi_2(\beta_i)} \bigwedge_{\beta_i \in \mathcal{F}(\tau_2/C_2) \cap Dom(\psi_4 \circ \mu)} C_{\psi_4 \circ \mu(\beta_i)}$$

by inductive hypothesis , we have:

$$\begin{array}{rcl}
C_{n1} & = & \psi_1(C_1) \displaystyle\bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\psi_1)} C_{\psi_1(\beta_i)} \\
C_{n2} & = & \psi_2(C_1) \displaystyle\bigwedge_{\beta_i \in \mathcal{F}(\tau_2/C_2) \cap Dom(\psi_2)} C_{\psi_2(\beta_i)}
\end{array}$$

$\beta_i \notin V_1$ and $\beta_i \notin V_2$ so we have, by definition, that $\psi_4 \circ \mu \circ \varphi_2(\beta_i) = \psi_1(\beta_i)$ and $\psi_4 \circ \mu(\beta_i) = \psi_2(\beta_i)$. So we get:

$$\begin{array}{rl}
= & \psi_1(C_1) \wedge \psi_2(C_2) \displaystyle\bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\psi_1)} C_{\psi_1(\beta_i)} \bigwedge_{\beta_i \in \mathcal{F}(\tau_2/C_2) \cap Dom(\psi_2)} C_{\psi_2(\beta_i)} \quad \text{by inductive hypothesis} \\
= & C_{n1} \wedge C_{n2} \\
= & C_n
\end{array}$$

$$\begin{array}{rcll} \psi(\varphi(\beta)) & = & \psi_4(\mu(\varphi_2(\varphi_1(\beta)))) & \text{by definition of } \varphi \\ & = & \psi_3(\varphi_2(\varphi_1(\beta))) & \text{by definition of } \psi_4 \\ & = & \psi_2(\varphi_2(\varphi_1(\beta))) & \text{because } \beta \neq \alpha \text{ and } \alpha \text{ out of reach of } \varphi_1 \text{ and } \varphi_2 \\ & = & \psi_1(\varphi_1(\beta)) & \text{because } \varphi_1(\beta) \notin V_1 \\ & = & \varphi'(\beta) & \text{becase } \beta \notin V \end{array}$$

and the result.

Case $e = (e_1, e_2)$. Like (App) without the mgu μ.

Case $e = \textbf{let } x = e_1 \textbf{ in } e_2$. We have the following derivation:

$$\frac{\varphi'(E) \vdash e_1 : [\tau_{n1}/C_{n1}] \qquad \varphi'(E) + \{x : Gen([\tau_{n1}/C_{n1}], \varphi'(E))\} \vdash e_2 : [\tau_{n2}/C_{n2}]}{\varphi'(E) \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : [\tau_{n2}/C_{n1} \wedge C_{n2} \wedge \mathcal{L}(\tau_{n2}) \Rightarrow \mathcal{L}(\tau_{1n})]}$$

By application of the inductive hypothesis on the first premise, we have:

$$([\tau_1/C_1], \varphi_1, V_1) = W(e_1, E, V) \qquad \text{and } [\tau_{n1}/C_{n1}] = \psi_1([\tau_1/C_1]) \qquad \text{and } \varphi' = \psi_1 \circ \varphi_1 \text{ out of } V$$

In particular, $\varphi'(E) = \psi_1(\varphi_1(E))$. We verify that $\psi_1(Gen([\tau_1/C_1], \varphi_1(E)))$ is more general than $Gen(\psi_1([\tau_1/C_1]), \psi_1(\varphi_1(E)))$, i.e, $Gen([\tau_{n1}/C_{n1}], \varphi'(E))$. By hypothesis, we have:

$$\varphi'(E) + \{x : Gen([\tau_{n1}/C_{n1}], \varphi'(E))\} \vdash e_2 : [\tau_{n2}/C_{n2}]$$

With the Proposition 1, we get:

$$\varphi'(E) + \{x : \psi_1(Gen([\tau_1/C_1], \varphi_1(E)))\} \vdash e_2 : [\tau_{n2}/C_{n2}]$$

i.e.,

$$\psi_1(\varphi_1(E) + \{x : Gen([\tau_1/C_1], \varphi_1(E))\} \vdash a_2 : [\tau_{n2}/C_{n2}]$$

By application of the inductive hypothesis to the second premise, for the environment $\varphi_1(E) + \{x : Gen([\tau_1/C_1], \varphi_1(E))\}$ with the set of variables V_1, the type τ_{n2} and the substitution ψ_1, we get:

$$([\tau_2/C_2], \varphi_2, V_2) = W(e_2, \varphi_1(E) + \{x : Gen([\tau_1/C_1], \varphi_1(E))\}, V_1)$$

and $[\tau_{n2}/C_{n2}] = \psi_2([\tau_2/C_2])$ and $\psi_1 = \psi_2 \circ \varphi_2$ out of V_1. The algorithm takes $[\tau_1'/C_1'] = \varphi_2([\tau_1/C_1])$ and

$$\begin{array}{rcl} [\tau/C] & = & [\tau_2/C_2 \wedge C_1' \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1))] \\ & = & [\tau_2/C_2 \wedge \varphi_2(C_1) \bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\varphi_2)} C_{\varphi_2(\beta_i)} \wedge (\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\varphi_2(\tau_1)))] \end{array}$$

and $\varphi = \varphi_2 \circ \varphi_1$ and $V' = V_2$. Take $\psi = \psi_2$. We have $\psi([\tau/C]) = [\tau_{n2}/C']$ and

$$C' = \psi_2(C_2) \wedge \psi_2(\varphi_2(C_1)) \bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\varphi_2)} \psi_2(C_{\varphi_2(\beta_i)}) \wedge \psi_2((\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\varphi_2(\tau_1)))) \bigwedge_{\gamma_i \in \mathcal{F}(\tau/C) \cap Dom(\psi_2)} C_{\psi_2(\gamma_i)}$$

By the Lemma 10, we get

$$C' = \psi_2(C_2) \wedge \psi_2(\varphi_2(C_1)) \bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\varphi_2)} \psi_2(C_{\varphi_2(\beta_i)}) \wedge (\mathcal{L}(\tau_{n2}) \Rightarrow \mathcal{L}(\tau_{n1})) \bigwedge_{\gamma_i \in \mathcal{F}(\tau/C) \cap Dom(\psi_2)} C_{\psi_2(\gamma_i)}$$

By inductive hypothesis, we have:

$$\begin{array}{rcl} C_{n2} & = & \psi_2(C_2) \bigwedge_{\beta_i \in \mathcal{F}(\tau_2/C_2) \cap Dom(\psi_2)} C_{\psi_2(\beta_i)} \\ C_{n1} & = & \psi_1(C_1) \bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\psi_1)} C_{\psi_1(\beta_i)} \end{array}$$

All the free variables of $[\tau_1/C_1]$ are not free in V_1 and $\psi_1 = \psi_2 \circ \varphi_2$ out of V_1, we have $\psi_2(\varphi_2(C_1)) = \psi_1(C_1)$. Like in (App), by some applications of the Lemma 11 and by ignoring **True** constraints, we have by hypothesis and by induction

$$\begin{array}{rl} = & \psi_2(C_2) \wedge \psi_1(C_1) \bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\psi_1)} C_{\psi_2(\varphi_2(\beta_i))} \wedge (\mathcal{L}(\tau_{n2}) \Rightarrow \mathcal{L}(\tau_{n1})) \bigwedge_{\gamma_i \in \mathcal{F}(\tau_2/C_2) \cap Dom(\psi_2)} C_{\psi_2(\gamma_i)} \\ = & \psi_2(C_2) \wedge \psi_1(C_1) \bigwedge_{\beta_i \in \mathcal{F}(\tau_1/C_1) \cap Dom(\psi_1)} C_{\psi_1(\beta_i)} \wedge (\mathcal{L}(\tau_{n2}) \Rightarrow \mathcal{L}(\tau_{n1})) \bigwedge_{\gamma_i \in \mathcal{F}(\tau_2/C_2) \cap Dom(\psi_2)} C_{\psi_2(\gamma_i)} \\ = & C_{n1} \wedge C_{n2} \wedge (\mathcal{L}(\tau_{n2}) \Rightarrow \mathcal{L}(\tau_{n1})) \end{array}$$

$$\begin{array}{rcll}
\psi(\varphi(\alpha)) & = & \psi_2(\varphi_2(\varphi_1(\alpha))) & \text{by definition of } \varphi \\
 & = & \psi_1(\varphi_1(\alpha)) & \text{because } \varphi_1(\alpha) \notin V \text{ because } \alpha \text{is out of reach of } \varphi_1 \\
 & = & \varphi'(\alpha) & \text{because } \alpha \notin V
\end{array}$$

hence $\varphi' = \psi \circ \varphi$ out of reach of V and the result.

□

Chapter 6

Implementation

In this chapter, we describe an implementation of our mini-BSML. We use Objective Caml to evaluate our expressions. So we only present the first implementation of the algorithm W. To attain this goal, we need to implement substitution and the type scheme. We first describe our abstract structure and the given definitions, the technical tools like a trivial instance, *Solve*, and we finish by explaining the implementation of W algorithm. The implementation has been made in Objective Caml.

6.1 Definition

6.1.1 Description of the data structures

In this section, we describe the abstract structures of our implementation. The expressions of our language came from a **syntactic analysis** (camlyacc and camlexx) and like a lot of our structures, they are **sum types**:

```
type expression =
    Var of string
  | Const_int of int
  | Const_bool of bool
  | Op of string
  | Fun of string * expression
  | App of expression * expression
  | Pair of expression * expression
  | Let of string * expression * expression;;
```

```
type expr_type =
    Base of string
  | Type_var of string
  | Arrow of expr_type * expr_type
  | Par of expr_type
  | Product of expr_type * expr_type;;

type formula =
    Loc of string
  | True | False
  | Or of formula*formula
  | Not of formula
  | And of formula*formula
  | Imp of formula*formula;;
```

```
type type_scheme = { quantif: string list; body: expr_type; constraints: formula};;
```

The "simple type" and the constraints of our language are **trivial inductive structure**. We use "string" to define the variables of type (idem for the constraints). For mini-BSML, we have two kinds of constants: the boolean and the integer. So the string base of the "expr_type" are equal to "int" or "bool". It is possible to add new base types, but for our study, it is not. For the constraints, we have introduced the "Not" and the "Or" operators to simplify our constraints (see the description of the *Simple* operator) for the users. Now we write "Imp" for $\Rightarrow$, "And" for $\wedge$, "Or" for $\vee$ and "Not" for $\neg$ to be close by the implementation.

The **type schemes** are **records** of a list of string for the quantification, a simple type and a constraint. The binding variables are the variables on the quantification list. The **environments** are implementing by using a **list of type schemes**.

6.1.2 Definitions

We give the implementation of the definitions of the chapter 2.

The set of free variables can be calculated by induction on the abstract structure and the result is a list of the free variables. We use a union (resp intersection, difference) function that computes the union without dublicates on the list (resp intersection, difference). We give for example the free variables of the "expr_type" (it is the same manner for the constraints):

```
let type_free_var =
  let rec f_rec t = match t with
      Base b -> []
    | Type_var v -> [v]
    | Arrow(a, b) | Product(a,b) -> union (f_rec a) (f_rec b)
    | Par v -> (f_rec v)
 in f_rec;;
```

The free variables of a type scheme is computed with the union of the free variables of the constraints, the type and with the difference of the quantification.

```
let type_scheme_free_var sigma = difference
 (union (type_free_var sigma.body) (constraints_free_var sigma.constraints)) sigma.quantif;;
```

The free variables of an environment is trivially the application of this function to a list (the environment). In the same manner, we give the *Gen* operators:

```
let generalisation sigma env =
{   quantif = difference (type_free_var sigma.body) (environment_free_var env);
    body = sigma.body; constraints = sigma.constraints };;
```

Substitution

The substitutions are applications from variables of type to simple types. We need, for the type scheme substitution, to calculate their domains. So we have implemented the substitution by a **(string * expr_type) list**. To calculate our substitutions, when we have a variable, we have to look if it is in the domain of our substitution (i.e, looking in the list) and if it is, to replace the variable. For the constraints we use a function **loc** to convert a common type to a constraint (see next):

```
let constraints_substitution alpha_tau_list =
  let rec f_rec contr = match contr with
      True | False -> contr
    | Or(a,b) -> Or(f_rec a, f_rec b)
    | Not(a) -> Not(f_rec a)
    | And(a,b) -> And(f_rec a, f_rec b)
    | Imp(a,b) -> Imp(f_rec a, f_rec b)
    | Loc alpha -> begin
                     try loc(List.assoc alpha alpha_tau_list) with Not_found -> contr
                    end in f_rec;;
```

The substitution on a type scheme is more difficult. First we have to rename the binding variables (i.e, create fresh variables and apply this trivial substitution). After, we have to add the constraints calculated from the domain of the substitution (see chapter 2):

```
let generation_big_and alpha_tau_list sigma =
 let var_list = intersection (union (type_free_var sigma.body)
               (constraints_free_var sigma.constraints)) (domain alpha_tau_list)
 in List.fold_right (fun var -> fun next -> And( construction_of_constraints
      (type_substitution alpha_tau_list (Type_var var)), next)) var_list True;;
```

and finally, apply the substitution:

```
let type_scheme_substitution alpha_tau_list sigma =
     (* we generate fresh variables beta_1...beta_N *)
     let quantif' = List.map (fun alpha -> new_variable()) sigma.quantif in
      (* we generate a renamme to have our substitution out of reach *)
let renamme = (List.map2 (fun alpha beta -> (alpha, Type_var beta)) sigma.quantif quantif')
   in
{
 quantif=quantif';
 sigma.constraints), generation_big_and alpha_tau_list sigma)
};;
```

The substitution on an environment is the application of this function to all the elements of the environment (the type scheme list). But, it is easy to see that in the W algorithm, we use **composition of substitution** that creates a new substitution. To implement this, we use a list of substitutions (i.e. **(string * expr_type) list list**). Thus, the composition of substitutions is trivially the **list concatenation** and to apply a "substitution composition", it is only the application of the substitution contained in the composition. We illustrate with the substitution on an environment:

```
let environment_substitution_composition env =
 let rec f_rec composition = match composition with
      [] -> env
    | hd::tl -> environment_substitution hd (f_rec tl) in f_rec;;
```

Construction of the constraints

In the following, we see that we need to transform an "expr_type" and how to construct the constraints from a common type. The rules are given in the section 2.2.1. It is easy to translate those inductive rules in a language such as Objective Caml:

```
 let loc =
  let rec f_rec ty = match ty with
   Base(x) -> True
 | Type_var(x) -> Loc(x)
 | Par(x) -> And(False,f_rec(x))
 | Arrow(x,y) | Product(x,y) ->
          And(f_rec(x),f_rec(y))
  in f_rec;;
```

```
let construction_of_constraints =
  let rec f_rec ty = match ty with
| Par(x) -> And(loc(x),f_rec(x))
| Arrow(x,y) ->
    And(And(f_rec(x),f_rec(y)),Imp(loc(y),loc(x)))
| Product(x,y) -> And(f_rec(x),f_rec(y))
| _ -> True
 in f_rec;;
```

6.2 Tools on constraints

In this section, we present some important tools for the constraints: *Solve* and *Simple*. *Solve* is a function that calculates when a constraint is always **False**. Our constraints are a sub part of the propositional calculus, so *Solve* is a decidable function. We can apply classical technic like the **quine method**. *Simple* is the application of technicals to have more readable constraints for the users.

6.2.1 Solve

The "quine method" is the simpliest technic to resolve propositionnal calculus. The method consists in taking a variable on the constraints and replace all its occurences by the **True** (respectively **False**) value then reduce the constraints with some knowing rules. Example:

```
let rec reduct f = match f with
 Not(a) -> reduct_not(a)
| Or(a,b) -> (reduct_or a b)
| And(a,b) -> (reduct_and a b)
| Imp(a,b) -> (reduct_imp a b)
| _ -> f
and (* reduction of the not *)
  reduct_not f = let g = (reduct f) in match g with
     True -> False
| False -> True
| _ -> Not(g)
etc...
```

This is for the rules: $Not(\mathbf{True}) = \mathbf{False}$ and $Not(\mathbf{False}) = \mathbf{True}$. We have a lot of reduction rules. We give some of them:

$$\begin{array}{rcl|rcl} D \wedge \mathbf{True} & = & D & D \vee \mathbf{True} & = & \mathbf{True} \\ D \wedge \mathbf{False} & = & \mathbf{False} & D \vee \mathbf{False} & = & D \end{array} \quad \text{etc} \ldots$$

This operation repeated until there is no free variable. If, in one of these cases, a constraint is *Solved* to **True**, then the constraint is not an incorrect constraint:

```
  let x=first_variable contr in
   if x=[] then ((reduct contr)=True) else
        (solve(reduct (substi (List.hd x) False contr)))
     || (solve(reduct (substi (List.hd x) True contr)));;
```

where first_variable is a function that gives the first free variable on the constraints.

6.2.2 Simple

With the *Simple* function, we want to have the constraints such as:

$$\bigwedge_{i=1}^{n} \mathcal{L}(\alpha_i) \bigwedge_{j=1}^{m} \neg\mathcal{L}(\alpha'_j) \bigwedge_{k=1}^{p} (\mathcal{L}(w_{1k}) \wedge ... \wedge \mathcal{L}(w_{uk})) \Rightarrow \mathcal{L}(\beta_{1k}) \wedge ... \wedge \mathcal{L}(\beta_{vk}))$$

Remark: we do not prove that we will have this form. We only want to have more readable constraints. We take $\alpha \wedge (\alpha \Rightarrow (\beta \vee \gamma))$ for example.

To attain this goal, we first apply the reducting function (see in section 6.2.1) to remove the constraints **True** and **False**. Then, we pass our constraints to a **conjonctive normal form**. So we have to take off the "Imp" operator and replace it by "Or" and "And" operators:

```
let rec vir_implication f = match f with
  Imp(a,b) -> Or((Not (vir_implication a)),(vir_implication b))
        | Or(a,b) -> Or((vir_implication a),(vir_implication b))
| ...
```

Next, we "push" the "Not" operator on the literals and apply the De Morgan rules:

```
let rec pousse_negation f = match f with
 Not(a) -> begin match a with
                    And(x,y) -> Or(pousse_negation(Not(x)),pousse_negation(Not(y)))
                  | Or(x,y) -> And(pousse_negation(Not(x)),pousse_negation(Not(y)))
                  | Not(x) -> x
  end
| Or(a,b) -> Or((pousse_negation a),(pousse_negation b))
| ...
```

And finally, distribute the "And" on the "Or":

```
let rec distribution_et_ou f = match f with
  And(a,b) -> And((distribution_et_ou a ),(distribution_et_ou b))
| Or(a,b) -> begin match (a,b) with
    (x,And(u,v)) -> And((distribution_et_ou(Or(x,u))),( distribution_et_ou(Or(x,v))))
  | (And(u,v),y) -> And((distribution_et_ou(Or(u,y))),(distribution_et_ou(Or(v,y))))
  | _ -> f end
|_ -> f;;

let forme_normale_conjonctive f = distribution_et_ou (pousse_negation (vir_implication f));;
```

Thus, we have our constraints such as:

$$\bigwedge_{i=1}^{n} (\bigvee_{j=1}^{m} \mathcal{L}(\alpha_i^j))$$

for the example, we have $\alpha \wedge (\neg\alpha \vee (\beta \vee \gamma))$.

Now, we can pass our constraints to a set of clauses, i.e., a **constraints list list**. With this kind of list, it will be easier to remove duplicates:

```
let rec expFCL_vers_liste_de_ou f= match f with
  And(a,b) -> (expFCL_vers_liste_de_ou a)@(expFCL_vers_liste_de_ou b)
| _ -> f::[];;
| _ -> f::[];;

let forme_clausale f = List.map suite_ou_vers_liste
                         (expFCL_vers_liste_de_ou(forme_normale_conjonctive f));;
```

For the example, we have: $[[\alpha]; [\neg\alpha; \beta; \gamma]]$.

Now, we can sort our clauses and our lists of clauses. To do that, we use the **List.sort** function of Objective Caml with arbitrary function to compare the clauses and the variables (alphabetic order in the implementation). After these operations, we can remove dublicates. In the clauses, we can take off the literals like $[\ldots; P; \neg P; \ldots]$ and replace it by $[\ldots;$ **True**$; \ldots]$. We do not give the code source because, it is long and not interesting.

After simplifying our clauses, we can transform these sets of clauses to result form. We apply the inverse rules that have taken off the "Imp" operators in the clause and we gather together our clauses in our constraints. Intuitively, this method works well because in the original constraint have only "Imp", "Not" and "And" operators.

6.3 The W algorithm

In this section, we give the implementation of the W algorithm that has been made. We first give the unification of two simple types and the trivial instance. We finish by some cases of the W algorithm.

6.3.1 Unification and trivial instance

Unification

The Robinson's algorithm works on type equations. We represent our equations by a list of couples of simple types. This algorithm needs two simple functions: "occurence" that tests if a variable is on the free variables of a type and a function that made an application to all the elements of our equations. The function "occurrence" has been written by induction instead of using the function of free variables. We remind that the substitution are list of "expr_type" and the composition is the concatenation of list. We give some cases of the algorithm:

```
let mgu tau1 tau2 =
 let rec robinson c = match c with
  [] -> identity
| (Base b1, Base b2) :: c' when b1 = b2 -> robinson c'
| (Type_var alpha, Type_var beta) :: c' when alpha = beta -> robinson c'

| (Type_var alpha, tau) :: c' when not(occurrence alpha tau) ->
    let new_composante = [(alpha,tau)] in
      let new_R = robinson (apply_list_couple (type_substitution new_composante) c') in
        (new_R@[new_composante])

| (Arrow(tau1,tau2), Arrow(tau1',tau2'))::c' -> robinson((tau1,tau1')::(tau2,tau2')::c')
| (Par(tau1), Par(tau2)) :: c' -> robinson((tau1,tau2)::c')
| ...
| _ -> raise Not_unifiable
  in robinson [(tau1,tau2)];;
```

Trivial instance

As for the substitution on a type scheme, we rename the variables. To do this, we create a substitution with a set of fresh variables:

```
let trivial_instance sigma =
  (* we generate fresh variables beta1... betaN *)
  let new_var = List.map (fun alpha -> new_variable()) sigma.quantif in
  (* We construct the  renamme alphai <- betai *)
```

```
    {
     quantif=[];
     body=type_substitution renamme sigma.body;
     constraints= constraints_substitution renamme sigma.constraints
    };;
```

6.3.2 The W algorithm

This part uses all the functions that have been described in the last sections, particularly the type scheme and environment substitutions. We present the implementation of the algorithm cases by cases. The W algorithm, for result, gives a type scheme without quantification and a substitution (a composition).

The cases for the constants are trivial. We do no describe them. Idem for the case operator, by using a trivial instance of the type scheme of operator (see section 5.1.1). Idem for the case variable by searching in the environment the type scheme of the variable and take a trivial instance on its. In all these cases, the substitution is the identity. Now we describe two cases: functional and application. The other cases are identical.

Functional

We have an expression that has the form: **fun** $x \to e_1$. First, we have to create a new variable α and its type scheme. Next we apply recursively the W algorithm to e_1. Thus we can create the simple type of the expression by application of the substitution given recursively to the variable α and create the type scheme of the expression with the function of construction of constraints (see section 6.1.2).

```
| Fun(x, a1) ->
        let alpha = Type_var(new_variable()) in
         let schema_alpha = {quantif = []; body = alpha; constraints=True} in
          let (sigma_1, phi1) = www ((x, schema_alpha) :: env) a1 in
           let new_type=Arrow(type_substitution_composition alpha phi1, sigma_1.body) in
            let sigma={
             quantif=[];
                     body=new_type;
                     constraints=And(construction_of_constraints(new_type),sigma_1.constraints)
  }
in if solve(sigma.constraints) then
 begin
    if !verbose then print_string(" \" FUN: "^expression_to_string(expr)^" : "
         ^(type_scheme_to_string(simpl sigma))^" \" ;;\n")
      else print_string "";
    (sigma,phi1)
   end else raise Pb_constraints
```

After creating the type scheme (so the constraints), we can verify if those constraints are correct or not. If not, we raise an exception to signal that the expression is not valid. For the users (and in all the cases), we have created a verbose mode that write the type scheme of the expression (here a sub-expression). Thus, the user can see all the running steps of the algorithm, i.e, the type derivation.

Application

We have an expression that has the form: (e_1 e_2). Like in functional case, we apply the algorithm recursively to the sub-expressions e_1 and e_2. For e_2, like the definition of the algorithm, we apply the substitution, given by the first recursive step, to the environment. Next we create a new variable by using the function: "new_variable" and we use the type unification define before.

```
| App(a1, a2) ->
        let (sigma_1, phi1) = www env a1 in
          let (sigma_2, phi2) = www (environment_substitution_composition env phi1) a2 in
           let alpha = Type_var(new_variable()) in
             let mu = mgu (type_substitution_composition sigma_1.body phi2)
                  (Arrow(sigma_2.body, alpha)) in
```

```
      let sigma_1'= type_scheme_substitution_composition sigma_1 first_compo in
       let sigma= {
            quantif=[];
     body = type_substitution_composition alpha mu;
     constraints = And(sigma_1'.constraints, sigma_2'.constraints)
   }
      in if solve(sigma.constraints) then
       begin
  if !verbose then print_string(" \" APP: "^expression_to_string(expr)^" : "
       ^(type_scheme_to_string(simpl sigma))^" \" ;;\n")
    else print_string "";
  (sigma,first_compo@phi1)
end else raise Pb_constraints;;
```

Now, like in the definition of the algorithm, we can create the new type scheme of the sub expression by using the recursive substitution and the substitution create by the Robinson's algorithm and then we construct the type scheme of the expression. Like in functional case, we verify the constraints with the solve function and we raise an exception if the constraints are not correct. For the substitution, we use, like before, the concatenation to define the substitution composition.

In this case (as for functionnal), it is easy to implement the W algorithm with the functions that have been implemented. In the last section, we present how mini-BSML works and give some examples.

6.4 Mini-BSML

The implementation of mini-BSML, could be downloaded in: *http://www.bsml.fr.st*

6.4.1 Operating

The compilation of the W algorithm in Objective Caml (OCaml) gives an executable byte-code file. This implementation takes in entry the user's expression and prints to the standard output, an OCaml's string for the errors or the type scheme (and if verbose mode, the step of the type derivation) and the expression for OCaml. Thus, OCaml could compute the expression and give the result. But to avoid the type given by OCaml, and have only the mini-BSML's type schemes, we have implemented a "reader" that screens the output of Ocaml. So we have a script to run those programs and redirect the ouput and the input with shell pipes:

```
cat bsmlib.ml; bsml_typing | ocaml | reader
```

For this, we need a function to transform mini-BSML's expression to OCaml ones. We can transform the **fix** operator to a **let rec** expression and the arithmetic's operations like, $+(e_1, e_2)$ to their suffixe versions $(1 + 2)$. We transform the conditional **ifthenelse**(t_1, e_1, e_2) to have the classical OCaml's expression: **if** t_1 **then** e_1 **else** e_2.

For the synchronous conditional operation, we apply the transformation of the BSMLlib (see introduction) like for the sequential conditional:

```
let expression_to_string =
  let rec f_rec exp = match exp with
      Var a -> a
    | Const_int a -> string_of_int a
    | Const_bool a -> string_of_bool a
    | App(Op "ifat",Pair(vect,Pair(pid,Pair(e1,e2)))) ->
       "if (at "^(f_rec vect)^" "^(f_rec pid)^") then "^(f_rec e1)^" else "^(f_rec e2)
    | App(Op "ifthenelse",Pair(b,Pair(e1,e2))) ->
        "if ("^(f_rec b)^") then "^(f_rec e1)^" else "^(f_rec e2)
    | App(Op "+",Pair(e1,e2)) -> "("^(f_rec e1)^"+"^(f_rec e2)^")"
    | App(Op "?",Pair(e1,e2)) -> "("^(f_rec e1)^"="^(f_rec e2)^")"
    | ...
```

```
   | Fun(a,b) -> "(fun "^a^" -> "^(f_rec b)^")"
   | App(a,b) -> "("^(f_rec a)^" "^(f_rec b)^")"
   | Pair(a,b) -> "("^(f_rec a)^","^(f_rec b)^")"
   | Let(a,b,c) -> "let "^a^" = "^(f_rec b)^" in "^(f_rec c)
 in f_rec;;
```

A sequential implementation of the parallel operations was made by Frédéric Loulergue for teaching. To test our type system, it is sufficient:

```
type 'a par = Par of 'a array;;
let bsp_p () = 16;;
let mkpar f = Par(Array.init (bsp_p()) f);;

let put (Par (vf:(int->'a option) array)) =
  Par(Array.init (bsp_p()) (fun i -> fun j -> (vf.(j)) i));;

let apply (Par vf) (Par vv) =
  Par (Array.init (bsp_p()) (fun i-> (vf.(i)) (vv.(i))));;

let at (Par (vb:bool array)) n = vb.(n);;

let isnc v = match v with None -> true | _ -> false;;
```

A new version of the BSMLlib, using the MPI library, is described in [Loulergue, 2002] and can be found at *http://www.bsml.fr.st*

6.4.2 Examples

We give the running of our implementation for valid expressions (or not) and for the incorrect examples given in this report.

Incorect replicate

Run with verbose mode. We test the examples given in chapter 4:

```
 "          mini-BSML version 0.1-alpha ";;
let replicate = fun x -> mkpar(fun i -> x) in replicate(replicate 2);;
 " OP: mkpar : [((int -> a3) -> (a3 par)) | L(a3)] " ;;
 " VAR: x : a1 " ;;
 " FUN: (fun i -> x) : [(a4 -> a1) | (L(a1) => L(a4))] " ;;
 " APP: (mkpar (fun i -> x)) : [(a1 par) | L(a1)] " ;;
 " FUN: (fun x -> (mkpar (fun i -> x))) : [(a1 -> (a1 par)) | L(a1)] " ;;
 " VAR: replicate : [(a6 -> (a6 par)) | L(a6)] " ;;
 " VAR: replicate : [(a7 -> (a7 par)) | L(a7)] " ;;
 " CONST_INT: 2 : int " ;;
 " APP: (replicate 2) : (int par) " ;;
 " A constraint is solved to false ";;

fun x -> mkpar(fun i -> x);;
 " OP: mkpar : [((int -> a12) -> (a12 par)) | L(a12)] " ;;
 " VAR: x : a10 " ;;
 " FUN: (fun i -> x) : [(a13 -> a10) | (L(a10) => L(a13))] " ;;
 " APP: (mkpar (fun i -> x)) : [(a10 par) | L(a10)] " ;;
 " FUN: (fun x -> (mkpar (fun i -> x))) : [(a10 -> (a10 par)) | L(a10)] " ;;
 " - : [(a10 -> (a10 par)) | L(a10)] ";;
```

In the chapter 2, we have given two examples to illustrate the importance to have all the constraints of the sub-expressions. Now, we give their type inference:

```
fun x -> (let y (mkpar (fun a -> x)) in (fun b -> x)(mkpar (fun a -> 5)));;
 " OP: mkpar : [((int -> a3) -> (a3 par)) | L(a3)] " ;;
 " VAR: x : a1 " ;;
 " FUN: (fun a -> x) : [(a4 -> a1) | (L(a1) => L(a4))] " ;;
 " APP: (mkpar (fun a -> x)) : [(a1 par) | L(a1)] " ;;
 " VAR: x : a1 " ;;
 " FUN: (fun b -> x) : [(a6 -> a1) | (L(a1) => L(a6))] " ;;
 " OP: mkpar : [((int -> a8) -> (a8 par)) | L(a8)] " ;;
 " CONST_INT: 5 : int " ;;
 " FUN: (fun a -> 5) : [(a9 -> int) | L(a9)] " ;;
 " APP: (mkpar (fun a -> 5)) : (int par) " ;;
 " APP: ((fun b -> x) (mkpar (fun a -> 5))) : [a11 | ~(L(a11))] " ;;
 " A constraint is solved to false ";;

fun x -> ( (fun y -> (mkpar (fun a -> x))) ((fun b -> x) (mkpar (fun a -> 5))));;
 " OP: mkpar : [((int -> a15) -> (a15 par)) | L(a15)] " ;;
 " VAR: x : a12 " ;;
 " FUN: (fun a -> x) : [(a16 -> a12) | (L(a12) => L(a16))] " ;;
 " APP: (mkpar (fun a -> x)) : [(a12 par) | L(a12)] " ;;
 " FUN: (fun y -> (mkpar (fun a -> x))) : [(a13 -> (a12 par)) | L(a12)] " ;;
 " VAR: x : a12 " ;;
 " FUN: (fun b -> x) : [(a18 -> a12) | (L(a12) => L(a18))] " ;;
 " OP: mkpar : [((int -> a20) -> (a20 par)) | L(a20)] " ;;
" CONST_INT: 5 : int " ;;
 " FUN: (fun a -> 5) : [(a21 -> int) | L(a21)] " ;;
 " APP: (mkpar (fun a -> 5)) : (int par) " ;;
 " APP: ((fun b -> x) (mkpar (fun a -> 5))) : [a23 | ~(L(a23))] " ;;
 " A constraint is solved to false ";;
```

Incorrect projection

```
 "         mini-BSML version 0.1-alpha ";;
( ( (fun x -> fun y -> x) 1) (mkpar (fun i->4)));;
 " VAR: x : a1 " ;;
 " FUN: (fun y -> x) : [(a2 -> a1) | (L(a1) => L(a2))] " ;;
 " FUN: (fun x -> (fun y -> x)) : [(a1 -> (a2 -> a1)) | (L(a1) => L(a2))] " ;;
 " CONST_INT: 1 : int " ;;
 " APP: ((fun x -> (fun y -> x)) 1) : [(a2 -> int) | L(a2)] " ;;
 " OP: mkpar : [((int -> a5) -> (a5 par)) | L(a5)] " ;;
 " CONST_INT: 4 : int " ;;
 " FUN: (fun i -> 4) : [(a6 -> int) | L(a6)] " ;;
 " APP: (mkpar (fun i -> 4)) : (int par) " ;;
 " A constraint is solved to false ";;
```

A special identity

This expression is a function of identity to parallel vector:

```
 "         mini-BSML version 0.1-alpha ";;
(fun x -> ifat ((mkpar (fun i-> true)),(0,(x,x))));;
 " OP: ifat : [(((bool par) * (int * (a3 * a3))) -> a3) | ~(L(a3))] " ;;
 " OP: mkpar : [((int -> a5) -> (a5 par)) | L(a5)] " ;;
 " CONST_BOOL: true : bool " ;;
 " FUN: (fun i -> true) : [(a6 -> bool) | L(a6)] " ;;
 " APP: (mkpar (fun i -> true)) : (bool par) " ;;
 " CONST_INT: 0 : int " ;;
 " COUPLE:(x,x) : (a1 * a1) " ;;
 " COUPLE:(0,(x,x)) : (int * (a1 * a1)) " ;;
 " COUPLE:((mkpar (fun i -> true)),(0,(x,x))) : ((bool par) * (int * (a1 * a1))) " ;;
 " APP: if (at (mkpar (fun i -> true)) 0) then x else x : [a8 | ~(L(a8))] " ;;
 " FUN: (fun x -> if (at (mkpar (fun i -> true)) 0) then x else x)
          : [(a8 -> a8) | ~(L(a8))] " ;;
 " - : [(a8 -> a8) | ~(L(a8))] ";;
```

So, if we apply this function to a local expression (1 for example), the constraints are *Solved* to **False**.

Examples with evaluation

The number of processes is 16.

```
 "         mini-BSML version 0.1-alpha "
# let fact = fix(fun fact -> fun n -> if ?(n,0) then 1 else *(n,fact(-(n,1)))) in fact(5);;
 " - : int "
= 120
# ((mkpar (fun i->4)),(mkpar (fun i->5)));;
 " - : ((int par) * (int par)) "
=
Par [|4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4|],
Par [|5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5|]

# ( ( (fun x -> fun y -> x) (mkpar (fun i->4)) ) 1);;
 " - : (int par) "
= Par [|4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4|]

# let apply2 = fun f -> fun x -> fun y -> (apply (apply f x) y) in apply2;;
" - : [(((a24 -> (a25 -> a26)) par) -> ((a24 par) -> ((a25 par) -> (a26 par))))
| (L(a14) & (L(a15) & (L(a18) & (L(a24) & (L(a25) & (L(a26) &
((L(a26) => L(a25)) & (((L(a25) & L(a26)) => L(a24)) &
((L(a15) => L(a14)) & (((L(a14) & L(a15)) => L(a18)) & ))] "
= <fun>

# let id=fun x -> x in (id id, id (mkpar (fun i -> 5)));;
 " - : ((a29 -> a29) * (int par)) "
= <fun>, Par [|5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5|]

# let f= fun x -> x (mkpar (fun i ->4)) in f (fun x ->x);;
 " - : [(int par) | ~(L(a44))] "
= Par [|4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4|]
#(mkpar (fun j -> (fun i -> 18)));;
 " - : [((a4 -> int) par) | L(a4)] "
=
Par
 [|<fun>; <fun>; <fun>; <fun>; <fun>; <fun>; <fun>; <fun>; <fun>; <fun>;
   <fun>; <fun>; <fun>; <fun>; <fun>; <fun>|]
```

In the "apply2" example, we have free variables from the derivation of the sub-expressions. We can see that in further works, we can reduct these constraints by replace for example "L(a26) & (L(a26) => L(a25))" by a more readable form: "Loc(a26) & Loc(a25)".

A last example

This example, came from the BSMLlib 0.1 to illustrate the problem of nesting in BSML and the difficulty of detecting them (the objective of our report).

```
let vec1 = (mkpar (fun pid -> pid)) in
 let vec2 = put (mkpar (fun pid -> fun from -> +(1,from))) in
   let couple1=(vec1,3) in
    let couple2=(vec2,4) in
     mkpar (fun pid -> ifthenelse( ((pid,nproc)), ((snd couple1),(snd couple2))));;
```

```
 " OP: mkpar : [((int -> a2) -> (a2 par)) | L(a2)] " ;;
 " VAR: pid : a3 " ;;
 " FUN: (fun pid -> pid) : (a3 -> a3) " ;;
 " APP: (mkpar (fun pid -> pid)) : (int par) " ;;
 " OP: put : [(((int -> a6) par) -> ((int -> a6) par)) | L(a6)] " ;;
 " OP: mkpar : [((int -> a8) -> (a8 par)) | L(a8)] " ;;
 " OP: + : ((int * int) -> int) " ;;
 " CONST_INT: 1 : int " ;;
 " VAR: from : a10 " ;;
 " COUPLE:(1,from) : (int * a10) " ;;
 " APP: (1+from) : int " ;;
 " FUN: (fun from -> (1+from)) : (int -> int) " ;;
 " FUN: (fun pid -> (fun from -> (1+from))) : [(a9 -> (int -> int)) | L(a9)] " ;;
 " APP: (mkpar (fun pid -> (fun from -> (1+from)))) : ((int -> int) par) " ;;
 " APP: (put (mkpar (fun pid -> (fun from -> (1+from))))) : ((int -> int) par) " ;;
 " VAR: vec1 : (int par) " ;;
 " CONST_INT: 3 : int " ;;
 " COUPLE:(vec1,3) : ((int par) * int) " ;;
 " VAR: vec2 : ((int -> int) par) " ;;
 " CONST_INT: 4 : int " ;;
 " COUPLE:(vec2,4) : (((int -> int) par) * int) " ;;
 " OP: mkpar : [((int -> a15) -> (a15 par)) | L(a15)] " ;;
 " OP: ifthenelse : ((bool * (a18 * a18)) -> a18) " ;;
 " OP: < : ((int * int) -> bool) " ;;
 " VAR: pid : a16 " ;;
 " OP: nproc : int " ;;
 " COUPLE:(pid,bsp_p()) : (a16 * int) " ;;
 " APP: (pid<bsp_p()) : bool " ;;
 " OP: snd : [((a22 * a23) -> a23) | (L(a23) => L(a22))] " ;;
 " VAR: couple1 : ((int par) * int) " ;;
 " A constraint is solved to false ";;
```

Objective Caml (and BSML) being a strict language, the evaluation of the last expression would imply the evaluation of "vec1" on the first half of the networks and "vect2" on the second half of the networks. But **put** implies a synchronization barrier and a **mkpar** implies no synchronisation barrier. So this will lead to mismatched barriers and the behaviour of the program will be unpredictable. The programm is rejected by our type system in "snd couple1" because snd : $[\alpha * \beta \to \beta / \mathcal{L}(\beta) \Rightarrow \mathcal{L}(\alpha)]$ and after an unification, $\alpha = (int\ par)$ and $\beta = int$ and the constraints are *Solved* to **False**.

Chapter 7

Further extensions

In this chapter, we introduce three extensions of the mini-BSML language. The first, the nuples, are a trivial extension of the couple. Next, we introduce a very important extension for a functionnal language: the sum types. We want to construct "infinite" expressions, like the list or the tree. We finish by the description of a new dynamic semantics for imperative treats.

7.1 A Trivial extension: the Nuples

The nuple, is the natural extension of the couple. With a serie of couples, we can simulate the nuples, but it is not natural and it is interesting to add this extension to our language to be more expressive.

7.1.1 Definition

Expression, value and evaluation

We add this new kind of expressions and values:

$$\begin{array}{rll} e ::= & \ldots & \text{like before} \\ & \mid (e_1, e_2, \ldots, e_n) & \text{Nuple with } n \geq 2 \end{array}$$

$$\begin{array}{rll} v ::= & \ldots & \text{like before} \\ & \mid (v_1, v_2, \ldots, v_n) & \text{Nuple value with } n \geq 2 \end{array}$$

Remark: the fact that in the following, we need $n \geq 2$, comes that a "1-uple" is a variable or a constant. With this problem, we could apply differents inductive rules for this expression and it is not deterministic. We can solve this problem by a syntactic analysis. From now, we supose that for the nuples, $n \geq 2$.

To calculate the free variables of our expressions, we need this new rule: $\mathcal{F}((e_1, e_2, \ldots, e_n)) = \bigcup_{i=1}^{n} \mathcal{F}(e_i)$.

For the evaluation, like the couple, we need to calculate all the sub-expressions before has the result. We give the natural semantics rule:

$$\frac{a_1 \triangleright v_1 \quad a_2 \triangleright v_2 \quad \ldots \quad a_n \triangleright v_n}{(a_1, a_2, \ldots, a_n) \triangleright (v_1, v_2, \ldots, v_n)}$$

Remark: for the small step semantics, the context grammar is extended as the couple.

Type and inference rule

We have to modify our type algebra and add a new inference rule:

$$\begin{array}{rll} \tau ::= & \ldots & \text{like before} \\ & \mid (\tau_1 * \tau_2 * \ldots * \tau_n) & \text{nuples} \end{array}$$

$$\frac{E \vdash a_1 : [\tau_1 / C_1] \quad E \vdash a_2 : [\tau_2 / C_2] \quad \ldots \quad E \vdash a_n : [\tau_n / C_n]}{E \vdash (a_1, a_2, \ldots, a_n) : [\tau_1 * \tau_2 * \ldots * \tau_n / \bigwedge_{i=1}^{n} C_i)} (Nuple)$$

$$\mathcal{F}((\tau_1 * \tau_2 * \ldots * \tau_n)) = \bigcup_{i=1}^{n} \mathcal{F}(\tau_i)$$

$$\varphi((\tau_1 * \tau_2 * \ldots * \tau_n)) = (\varphi(\tau_1) * \varphi(\tau_2) * \ldots * \varphi(\tau_n))$$

For the constraints, we add a new rule for the construction of the constraints and also for the transformation from the locality to constraints:

$$\frac{\mathcal{L}(\tau_1 * \tau_2 * \ldots * \tau_n)}{\bigwedge_{i=1}^{n} \mathcal{L}(\tau_i)}(Nuple)$$

$$\frac{\tau_1 \leadsto C_1 \quad \tau_2 \leadsto C_2 \quad \ldots \tau_n \leadsto C_n}{(\tau_1 * \tau_2 * \ldots * \tau_n) \leadsto \bigwedge_{i=1}^{n} C_i}(CNuple)$$

7.1.2 Operators

To do that, we give to our language, a family of new operators of projections (like **fst** and **snd** for the couple), $proj_{i,n}$ with $(n \geq i \geq 1)$, to have the i^{th} component of a nuple. So we have the following evalution rules and type scheme:

$$\frac{f \triangleright \mathbf{proj}_{i,n} \quad (a_1, \ldots, a_n) \triangleright (v_1, \ldots, v_n)}{f(a_1, \ldots, a_n) \to v_i} \qquad \mathbf{proj}_{i,n}(v_1, \ldots, v_n) \xrightarrow{\varepsilon} v_i$$

$$TC(\mathbf{proj}_{i,n}) = \forall \alpha_1, \ldots, \alpha_n.[(\alpha_1 * \ldots * \alpha_n) \to \alpha_i / \mathcal{L}(\alpha_i) \Rightarrow (\bigwedge_{j=1}^{n} \mathcal{L}(\alpha_j))]$$

Remark: with all these definitions and these new operators, we can remove the couple of our definitions (because the 2-uples are like a couple). The two operators, **fst** and **snd**, are now a synonym for respectively $\mathbf{proj}_{1,2}$ and $\mathbf{proj}_{2,2}$.

7.1.3 Examples

Our first example in the following expression:

```
(fun f -> f (1,replicate 0));;
```

with *replicate* is the function defined in the chapter 2. We can give the following derivation for this expression:

$$\dfrac{\dfrac{\dfrac{\ldots \leq \ldots}{\vdash f : [int * (int\ par) \to \omega / (\mathcal{L}(\omega) \Rightarrow \mathbf{False})]} \quad \dfrac{\dfrac{int \leq int}{\vdash 1 : int} \quad \dfrac{\ldots}{\vdash replicate\ 0 : (int\ par)}}{\vdash (1, replicate\ 0) : int * (int\ par)}}{f : [int * (int\ par) \to \omega / (\mathcal{L}(\omega) \Rightarrow \mathbf{False})] \vdash f\ (1, replicate\ 0) : [\omega / (\mathcal{L}(\omega) \Rightarrow \mathbf{False})]}}{\vdash (\mathbf{fun}\ f \to f\ (1, replicate\ 0)) : [(int * (int\ par) \to \omega) \to \omega / (\mathcal{L}(\omega) \Rightarrow \mathbf{False})]}$$

To continue, here an incorrect example:

```
(fun f -> f (1,replicate 0)) fst;;
```

because **fst** : $[(\alpha * \beta) \to \alpha / \mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\beta)]$ and after the subsitution $\alpha = int, \beta = (int\, par)$ and $\omega = int = \alpha$. The constraints are *Solved* to **False**. But not for this expression:

```
(fun f -> f (1, replicate 0)) snd;;
```

because **snd** : $[(\alpha * \beta) \to \beta / \mathcal{L}(\beta) \Rightarrow \mathcal{L}(\alpha)]$ and after the subsitution $\alpha = int, \beta = (int\ par)$ et $\omega = (int\ par) = \alpha$. The constraints are *Solved* to **True**.

The Nuples work as the couples but with $n \geq 2$. So for the proof, we work with the same manner (not two premises but n premises).

For the algorithm, the case couple is replaced by a case nuple. This case works in the same way but we have not to apply W to two premises but to n premises and the constraints are constructed in the same maner but generalised to n.

7.2 Sum types

7.2.1 Declaration

The sum types, or concrete types, are very importants for functional languages. They allow to construct inductive structures (like tree, list). The concrets types, are represented by special operators, name contructors. The sum types, for our study, must be declared at the beginning of the programs and not be mutually recursive. Examples:

```
type 'a list = Nil of unit | Cons of 'a * 'a list
type expr= Const of int | Var of string | Add of expr*expr | Mul of expr*epxr
```

The general form of a declaration of a sum type is:

$$\textbf{type } (\alpha_1, \ldots, \alpha_n)\ t = C_1 \textbf{ of } \tau_1 \mid \ldots \mid C_n \textbf{ of } \tau_n$$

where

- $\alpha_1, \ldots, \alpha_n$ are the parameters of the sum type t.
- $C_1, \ldots, C_n$ are the constructors
- $\tau_1, \ldots, \tau_n$ are the constructors's arguments.
- We need that, $Solve(\tau_i) \neq \textbf{False}$.

Remark: to represents the constructors that have no arguments, we add a type "unit" with a only value: "()". Example: "Nil" of the list is "Nil of unit".

Remark: we impose, for trivial reasons, that $\mathcal{F}(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_i\}$.

7.2.2 Definition

Expression, value, evaluation

For each sum types t, we add to the constructors $C_1, \ldots, C_p$ and the matching operator F_t. The matching operator discriminate a value of type t according to the head constructor of the value. In Objective Caml, the matching operator is:

```
match e with
    C_1(x1) -> e1
  | C_2(x2) -> e2
  | ...
  | C_p(xp) -> ep ;;
```

For our language and for our study, we could note this like the following application of the F_t operator:

$$F_t(e, (\textbf{fun } x_1 \to e_1), (\textbf{fun } x_1 \to e_2), \ldots, (\textbf{fun } x_p \to e_p))$$

We impose that in the expressions, each constructors C_i and F_t are applied to their arguments.

forms of values and rules for the context in the small step semantics:

$$\begin{array}{rcll} op & ::= & \ldots \mid C_1 \mid \ldots \mid C_p \mid F_t & \text{operators} \\ \tau & ::= & \ldots \mid (\tau_1, \ldots, \tau_n)\ t & \text{type algebra} \\ v & ::= & \ldots \mid C_1(v) \mid \ldots \mid C_p(v) & \text{values} \\ \Gamma & ::= & \ldots \mid C_1(\Gamma) \mid \ldots \mid C_p(\Gamma) & \text{contexts} \end{array}$$

and for the natural semantics, the dynamic rule for the constructors is:

$$\frac{e \rhd v}{C_k(e) \rhd C_k(v)} \quad \text{if } C_k \text{ is a constructor of } t$$

The reduction rule of the F_t operator is (for our two kinds of semantics):

$$F_t(C_k(v), v_1, \ldots, v_n) \overset{\varepsilon}{\rightharpoonup} (v_k\ v) \qquad \text{if } C_k \text{ is a constructor of } t$$

$$\frac{C_k(e) \rhd C_k(v) \quad e_1 \rhd v_1 \quad \ldots \quad e_n \rhd v_n}{F_t(C_k(e), e_1, \ldots, e_n) \rhd (v_k\ v)} \qquad \text{if } C_k \text{ is a constructor of } t$$

We naturally extend our definition of free variables by

$$\begin{array}{rcl} \mathcal{F}((\tau_1, \ldots, \tau_n)\ t) & = & \bigcup_{i=1}^{n} \mathcal{F}(\tau_i) \\ \mathcal{F}(C_k(e)) & = & \mathcal{F}(e) \end{array}$$

7.2.3 Type system

Definitions

We give the natural type scheme for our new operators and constructors:

$$\begin{array}{rcl} TC(C_k) & = & \forall \alpha_1, \ldots, \alpha_n.[\tau_k \rightarrow (\alpha_1, \ldots, \alpha_n)\ t\ /\ C_{\tau_k} \wedge \mathcal{L}((\alpha_1, \ldots, \alpha_n)\ t) \Rightarrow \mathcal{L}(\tau_k)] \\ TC(F_t) & = & \forall \alpha_1, \ldots, \alpha_n, \beta.[((\alpha_1, \ldots, \alpha_n)\ t\ * (\tau_1 \rightarrow \beta)\ * \ldots\ * (\tau_1 \rightarrow \beta)) \rightarrow \beta \\ & & /\ C_{\tau_1} \wedge \ldots \wedge C_{\tau_n} \wedge \mathcal{L}(\beta) \Rightarrow (\mathcal{L}((\alpha_1, \ldots, \alpha_n)\ t) \wedge \mathcal{L}(\tau_1) \wedge \ldots \wedge \mathcal{L}(\tau_n))] \end{array}$$

We extend our constraints algebra with:

$$\begin{array}{rcll} C & ::= & \ldots & \text{like before} \\ & \mid & \mathcal{L}(((\tau_1, \ldots, \tau_n)t)^{\perp}) & \text{sum types constraints} \end{array}$$

And we have to define new inductive rules for the definition of $\mathcal{L}$ and for the construction of the constraints:

$$\frac{\mathcal{L}((\tau_1, \ldots, \tau_n)\ t)}{\mathcal{L}((\tau_1, \ldots, \tau_n)) \wedge \mathcal{L}(((\tau_1, \ldots, \tau_n)\ t)^{\perp})}(Sum) \qquad \Bigg| \qquad \frac{(\tau_1, \ldots, \tau_n) \rightsquigarrow C_n}{((\tau_1, \ldots, \tau_n)\ t) \rightsquigarrow C_n}(CSum)$$

The $\perp$ makes the infinite recursivity calcul impossible. Thus we said that a sum type is local if all its parameters (if they exist) are locals and the sum type itself is local. We have introduce a new kind of constraints: sum type constraints. and extend naturally our constraints substitution and free variables:

$$\begin{array}{rcl} \mathcal{F}(\mathcal{L}(((\tau_1, \ldots, \tau_n)t)^{\perp})) & = & \bigcup_{i=1}^{n} \mathcal{F}(\tau_i) \\ \varphi(\mathcal{L}(((\tau_1, \ldots, \tau_n)t)^{\perp})) & = & \mathcal{L}((\varphi(\tau_1), \ldots, \varphi(\tau_n))\ t) \end{array}$$

Examples

For examples, we take an important sum type, the list (used in the BSMLlib) and a polymorphic sum type that intuivelly could be either local or global.

```
type 'a strange =  Zero of int
 | One of 'a
 | Two of 'a par;;
```

rules. We have simplify the constraints to be more readable.

- For the list we have:

$$\begin{array}{lcl} TC(Nil) & = & \forall\alpha.unit \to (\alpha\ list) \\ TC(Cons) & = & \forall\alpha.(\alpha * \alpha\ list) \to \alpha\ list. \\ TC(F_{list}) & = & \forall\alpha,\beta.[(\alpha\ list * (unit \to \beta) * ((\alpha * \alpha\ list) \to \beta)) \to \beta\ /\ \mathcal{L}(\beta) \Rightarrow (\mathcal{L}(\alpha) \wedge \mathcal{L}((\alpha\ list)^{\perp}))] \end{array}$$

$$\begin{array}{lcl} F_{list}(Nil(v), v_1, v_2) & \overset{\varepsilon}{\rightharpoonup} & (v_1\ v) \\ F_{list}(Cons(v), v_1, v_2) & \overset{\varepsilon}{\rightharpoonup} & (v_1\ v) \end{array}$$

For the following expression

```
let tl lis = match lis with
   Nil(x) -> lis
 | Cons(x,xl) -> xl;;
```

We traduce this expression to our language:

$$e = \textbf{let}\ tl = \textbf{fun}\ l \to F_{list}(l, (\textbf{fun}\ x \to l), (\textbf{fun}\ x_c \to snd(x_c)))\ \textbf{in}\ tl$$

We do no give the type induction (too long) but some results:

$$\begin{array}{rcl} \textbf{fun}\ x \to snd(x) & : & [(\alpha * \alpha\ list) \to \alpha\ list / \mathcal{L}((\alpha\ list)^{\perp}) \Rightarrow \mathcal{L}(\alpha)] \\ \textbf{fun}\ x \to l & : & unit \to \alpha\ list \\ F_{list} & : & (\alpha\ list * (unit \to \alpha\ list) * ((\alpha * \alpha\ list) \to \alpha\ list)) \to \alpha\ list \\ e & : & [\alpha\ list \to \alpha\ list / \mathcal{L}((\alpha\ list)^{\perp}) \Rightarrow \mathcal{L}(\alpha)] \end{array}$$

- For our "strange" sum type, we have:

$$\begin{array}{lcl} TC(Zero) & = & \forall\alpha.int \to \alpha\ strange \\ TC(One) & = & \forall\alpha.\alpha \to \alpha\ strange \\ TC(Two) & = & \forall\alpha.[(\alpha\ par) \to \alpha\ strange / \mathcal{L}(\alpha) \wedge (\mathcal{L}(\alpha) \wedge \mathcal{L}((\alpha\ strange)^{\perp})) \Rightarrow \textbf{False}] \\ TC(F_{strange}) & = & \forall\alpha,\beta.[(\alpha\ strange * (int \to \beta) * (\alpha \to \beta) * (\alpha\ par \to \beta)) \to \beta / \mathcal{L}(\alpha) \wedge (\mathcal{L}(\beta) \Rightarrow \textbf{False})] \end{array}$$

$$\begin{array}{lcl} F_{strange}(Zero(v), v_1, v_2, v_3) & \overset{\varepsilon}{\rightharpoonup} & (v_1\ v) \\ F_{strange}(One(v), v_1, v_2, v_3) & \overset{\varepsilon}{\rightharpoonup} & (v_2\ v) \\ F_{strange}(Two(v), v_1, v_2, v_3) & \overset{\varepsilon}{\rightharpoonup} & (v_3\ v) \end{array}$$

For the following expression

```
(Zero(1), Two(mkpar (fun i -> i)))
```

We give the type induction (with using the ¬ operator defined before):

$$\dfrac{\dfrac{\vdash Zero : int \to \beta\ strange \quad \vdash 1 : int}{\vdash Zero(1) : \beta\ strange} \quad \dfrac{\dfrac{}{\vdash Two : [int\ par \to int\ strange / \neg(\mathcal{L}((int\ strange)^{\perp}))]} \quad \dfrac{\dots}{\vdash \textbf{mkpar}\ (\textbf{fun}\ i \to i) : int\ par}}{\vdash Two(\textbf{mkpar}\ (\textbf{fun}\ i \to i)) : [int\ strange / \neg(\mathcal{L}((int\ strange)^{\perp}))]}}{\vdash (Zero(1), Two(\textbf{mkpar}\ (\textbf{fun}\ i \to i))) : [(\beta\ strange * int\ strange) / \neg(\mathcal{L}((int\ strange)^{\perp}))]}$$

and for the following one (that hide a whole parallel object):

```
Two(mkpar(fun i -> One(i)));;
```

$$\dfrac{\dfrac{\dots}{\vdash Two : \dots} \quad \dfrac{\dfrac{}{\vdash \textbf{mkpar} : [(int \to int\ strange) \to (int\ strange)\ par / \mathcal{L}((int\ strange)^{\perp})]} \quad \dfrac{\dots}{\vdash \textbf{fun}\ i \to One(i) : int \to int\ strange}}{\vdash \textbf{mkpar}(\textbf{fun}\ i \to One(i)) : [(int\ strange)\ par / \mathcal{L}((int\ strange)^{\perp})]}}{\vdash Two(\textbf{mkpar}(\textbf{fun}\ i \to One(i))) :\ (int\ strange\ strange) / \mathcal{L}((int\ strange)^{\perp}) \wedge (\mathcal{L}((int\ strange\ strange)^{\perp}) \Rightarrow \textbf{False})}$$

Thus, for example, the expression: $\textbf{mkpar}(\textbf{fun}\ pid \to Two(\textbf{mkpar}(\textbf{fun}\ i \to One(i))))$ is rejected because **mkpar** needs a local argument and it is not (so the constraints are *Solved* to **False**)and we do not have nesting.

With this new definitions, we have to re-prove all our lemmas, propositions and the important theorem of safety. Thus, for all our proofs, we add a new case. For the algorithm, we have to add a case for the sum types and have another algorithm to constructs the sum types given by the users. The implementation of this new type system and algorithms will be necessary to test programs (a lots of expressions of the BSMLlib are programs on lists) and to have a more expressive language.

Furthermore, our type system for the sum types is very strict. We have seen that the constraints for a couple is a $\wedge$ of the components. In the isomorphism of Curry-Howard, the couples are also transforming to a $\wedge$ of the types of the components. On the other hand, the sum types are represented by $\vee$ of the types of the composantes that is not do in our type system. A better type system could used this remark to type more expressions (like in the example).

7.3 Imperative treats

In this section, we add to mini-BSML, a characteristic of imperative languages: modification in the memory of variables or data structures. Indeed, a BSP computer contains a set of processor-memory pairs and a network so it is realistic that each processors could reached to its memory.

We add this modification to our language by the possibility of **affectation** and **allocation** of a variable or a data structure. The idea is adding **reference**. A reference is a cell of the memory which could be modified by the program. We create a reference with the allocation's **ref**(e) construction which gives a new reference initialised to e. To used the value kept on the reference, we need a operation to extract this value written **!**. We can modify our reference e_1 by replacing the kept value by another. This operation is called affectation and written: $e_1 := e_2$. We use the same notations than Objective Caml. Example:

```
let r=ref(fun a -> a*a) in let tmp= !r in r:=(fun b -> tmp (tmp b)); !r 2;;
```

Naturally, this expression is evaluated to 16. The construction $a; b$ evaluates a, next b and returns the value of b. It is an abbreviation for **let** $x = a$ **in** b where $x \notin \mathcal{F}(b)$. A binding reference is the same thing as a variable in an imperative language. We give for example the variables generator of the implementation of the W algorithm:

```
let compteur = ref 0;;
let new_variable() = compteur:=!compteur+1; "a" ^ string_of_int !compteur;;
```

We assume having a unique value: () that have the type *unit*. This is the result type of the affectation (like in Objective Caml).

Each processor has a name or pid (see chapter 1). To extend our dynamic semantics to the reference, we formalize the memory's locations and the memory's store for each processors. For each processor, we give a infinite set of memory's address written ℓ. A store, written s, is a partial function from location to values. A reference is evaluated to a location. We had this new kind of expressions and values:

$$\begin{array}{rcl} e ::= & \dots & \text{like before} \\ & \mid \ \ell & \text{location} \end{array}$$

$$\begin{array}{rcl} v ::= & \dots & \text{like before} \\ & \mid \ \ell & \text{location} \end{array}$$

and we had **!**, **:=** and **ref** to our operators.

Remark: at the beginning, the expression do not contain locations and the stores are empty.

Remark: there are no free variable of a location, $\mathcal{F}(\ell) = \emptyset$.

Small steps

We assume a finite set $\mathcal{N} = \{0, \dots, p-1\}$ which represents the set of processors names and we write i for these names and $\bowtie$ for all the network. We write s_i for the store of processor i with $i \in \mathcal{N}$. We assume that each processor has a store. The small step semantics has the form:

$$\begin{array}{ll} e/s \rightharpoonup e'/s' & \text{for one step} \\ e_0/s_a \rightharpoonup e_1/s_b \rightharpoonup e_2/s_c \rightharpoonup \dots \rightharpoonup v/s & \text{for all the steps of the calculus} \end{array}$$

for all the processors (i.e. $\forall i \in \mathcal{N}$). We write $S = [s_0, \dots, s_{p-1}]$ for all the stores of our network. With this notation, we could, for the sake of simplicity give the rules of reduction for all the network. Now, the small step semantics has the form:

$$\begin{array}{ll} e/S \rightharpoonup e'/S' & \text{for one step} \\ e_0/S_0 \rightharpoonup e_1/S_1 \rightharpoonup e_2/S_2 \rightharpoonup \dots \rightharpoonup v/S & \text{for all the steps of the calculus} \end{array}$$

Remark: with this new notation and imperative treats, we have to take care that the global expressions could be differents on each processors (side effects, see above).

We note $\overset{*}{\rightharpoonup}$, for the transitive closure of $\rightharpoonup$ and note $e_0/S_0 \overset{*}{\rightharpoonup} v/S$ for $e_0/S_0 \rightharpoonup e_1/S_1 \rightharpoonup e_2/S_2 \rightharpoonup \dots \rightharpoonup v/S$. We begin the reduction with a set of empty stores $\{\emptyset_0, \dots, \emptyset_{p-1}\}$ noted $\emptyset_\bowtie$.

Now we have two kinds of reductions:

1. $e/s_i \overset{i}{\rightharpoonup} e'/s'_i$ which could be read has "in the initial store s_i, at processor i, the expression e is reduce to e' in the store s'_i".
2. $e/S \overset{\bowtie}{\rightharpoonup} e'/S'$ which could be read has "in the initial network store S, the expression e is reduce to e' in the network store S'".

As in the definiton of the environment, we write $s + \{\ell \leftarrow v\}$ for the extension of s to the mapping of ℓ to v. If, before this operation, we have $\ell \in Dom(s)$, we can replace the range by the new value for the location ℓ.

To define this two new relations, we begin with some axioms for the relation of head reduction:

$$\begin{array}{llll||llll} (\mathbf{fun}\ x \to e)\ v\ /\ s_i & \overset{\varepsilon}{\underset{i}{\rightharpoonup}} & e[x \leftarrow v]\ /\ s_i & (\beta_{fun_i}) & (\mathbf{fun}\ x \to e)\ v\ /\ S & \overset{\varepsilon}{\underset{i}{\rightharpoonup}} & e[x \leftarrow v]\ /\ S & (\beta_{fun_\bowtie}) \\ (\mathbf{let}\ x = v\ \mathbf{in}\ e)\ /\ s_i & \overset{\varepsilon}{\underset{i}{\rightharpoonup}} & e[x \leftarrow v]\ /\ s_i & (\beta_{let_i}) & (\mathbf{let}\ x = v\ \mathbf{in}\ e)\ /\ S & \overset{\varepsilon}{\underset{i}{\rightharpoonup}} & e[x \leftarrow v]\ /\ S & (\beta_{let_\bowtie}) \end{array}$$

For our operators, each δ-rules $e \overset{\varepsilon}{\rightharpoonup} e'$ of the classical small steps semantics give two new reduction rules: $e\ /\ s_i \overset{\varepsilon}{\underset{i}{\rightharpoonup}} e'\ /\ s'_i$ and $e\ /\ S \overset{\varepsilon}{\underset{i}{\rightharpoonup}} e'\ /\ S'$. Indeed, these reductions do not change the stores and do not depend of the stores. We do not have a δ-rule on a single processor for the parallel operators δ-rules. Examples:

$$\begin{array}{llll} +(n_1, n_2)\ /\ s_i & \overset{\varepsilon}{\underset{i}{\rightharpoonup}} & n\ /\ s_i \text{ with } n = n_1 + n_2 & (\delta_{+_i}) \\ \mathbf{fst}(v_1, v_2)\ /\ s_i & \overset{\varepsilon}{\underset{i}{\rightharpoonup}} & v_1\ /\ s_i & (\delta_{fst_i}) \\ \dots & & & \end{array}$$

and for the network:

$$\begin{array}{llll} +(n_1, n_2)\ /\ S & \overset{\varepsilon}{\underset{\bowtie}{\rightharpoonup}} & n\ /\ S \text{ with } n = n_1 + n_2 & (\delta_{+_\bowtie}) \\ \mathbf{fst}(v_1, v_2)\ /\ S & \overset{\varepsilon}{\underset{\bowtie}{\rightharpoonup}} & v_1\ /\ S & (\delta_{fst_\bowtie}) \\ \mathbf{mkpar}(\mathbf{fun}\ x \to\ e)\ /\ S & \overset{\varepsilon}{\underset{\bowtie}{\rightharpoonup}} & \langle e[x \leftarrow 0], \dots, e[x \leftarrow (p-1)] \rangle\ /\ S & (\delta_{mkpar_\bowtie}) \\ \dots & & & \end{array}$$

Now we complete our semantics by giving the δ-rules of the operators on references. Like before, we need two kinds of reductions. First for a single processor:

$$\begin{array}{llll} \mathbf{ref}(v)\ /\ s_i & \overset{\varepsilon}{\underset{i}{\rightharpoonup}} & \ell\ /\ s_i + \{\ell \leftarrow v\} \quad \text{if } \ell \notin Dom(s_i) & (\delta_{ref_i}) \\ \mathbf{!}(\ell)\ /\ s_i & \overset{\varepsilon}{\underset{i}{\rightharpoonup}} & s_i(\ell)\ /\ s_i \quad \text{if } \ell \in Dom(s_i) & (\delta_{deref_i}) \\ \mathbf{:=}(\ell, v)\ /\ s_i & \overset{\varepsilon}{\underset{i}{\rightharpoonup}} & ()\ /\ s_i + \{\ell \leftarrow v\} & (\delta_{aff_i}) \end{array}$$

For the network, when a global expression creates a new allocation, every processors create a new allocation (an address) on its store. We add the constructor $\ell_{(i)}$ to design the location ℓ at processor i. Now, a first and natural definition of the δ-rules are:

$$\begin{array}{llll}
\mathbf{ref}(v) \;/\; S & \overset{\varepsilon}{\underset{\bowtie}{\rightharpoonup}} & \ell_{(i)} \;/\; S' \text{ where } S' = [s_0 + \{\ell_0 \leftarrow v\}, \ldots, s_{p-1} + \{\ell_{p-1} \leftarrow v\}] \;\text{ if } \ell_i \notin Dom(s_i) & (\delta_{ref_{\bowtie}}) \\
!(\ell_{(i)}) \;/\; S & \overset{\varepsilon}{\underset{\bowtie}{\rightharpoonup}} & \forall i \; s_i(\ell_i) \;/\; S \text{ if } \ell_i \in Dom(s_i) & (\delta_{deref_{\bowtie}}) \\
:=(\ell_{(i)}, v) \;/\; S & \overset{\varepsilon}{\underset{\bowtie}{\rightharpoonup}} & () \;/\; S' \quad \text{where } S' = [s_0 + \{\ell_0 \leftarrow v\}, \ldots, s_{p-1} + \{\ell_{p-1} \leftarrow v\}] & (\delta_{aff_{\bowtie}})
\end{array}$$

and we add: $\ell_{(i)} \;/\; s_j \;\overset{\varepsilon}{\underset{j}{\rightharpoonup}}\; \ell_j \;/\; s_j$.

Remark: for network rules (head reductions or δ-rules), the value (before the calculus) could be different on each processors because this value could depend of the store which could be different on each processors (side effects).

Contexts

Like in the classical small steps semantics, it is easy to see that we cannot always make a head reduction. We define two kinds of contexts that have the following abstract syntaxes:

$$\begin{array}{lll|lll}
\Gamma_{\bowtie} & ::= & [] & \Gamma_l^j & ::= & \Gamma_l^j \; e \\
 & | & \Gamma_{\bowtie} \; e & & | & v \; \Gamma_l^j \\
 & | & v \; \Gamma_{\bowtie} & & | & \mathbf{let}\; x = \Gamma_l^j \;\mathbf{in}\; e \\
 & | & \mathbf{let}\; x = \Gamma_{\bowtie} \;\mathbf{in}\; e & & | & (\Gamma_l^j, e) \\
 & | & (\Gamma_{\bowtie}, e) & & | & (v, \Gamma_l^j) \\
 & | & (v, \Gamma_{\bowtie}) & & | & \langle e, \ldots, \overbrace{\Gamma}^{j}, e, \ldots, e \rangle
\end{array}$$

where Γ are the contexts defined in the chapter 4.

Now we can reduce in depth in the sub-expression. To define this deep reduction, we use the following inference rules:

$$\frac{e \;/\; s_j \overset{\varepsilon}{\underset{j}{\rightharpoonup}} e' \;/\; s'_j}{\Gamma_l^j(e) \;/\; s_j \rightharpoonup \Gamma_l^j(e') \;/\; s'_j} \;(\textit{local context rule}) \qquad \Bigg| \qquad \frac{e \;/\; S \overset{\varepsilon}{\underset{\bowtie}{\rightharpoonup}} e' \;/\; S'}{\Gamma_{\bowtie}(e) \;/\; S \rightharpoonup \Gamma_{\bowtie}(e') \;/\; S'} \;(\textit{global context rule})$$

So we can reduce into the parallel vectors and the context gives the name of the processor where the expression is reduced.

Thus, we have a rule and its context to reduce global expressions and a rule to reduce local expressions (in the parallel vectors).

Examples

We run the following expressions:

```
let a=ref(1) in mkpar (fun j -> let b=ref(j) in b:=!b+!a;!b);;

let r=ref(3) in r:=!r+1;!r;;
```

$$
\begin{array}{lll}
& \textbf{let } a = \textbf{ref}(1) \textbf{ in mkpar}(\textbf{fun } j \to \textbf{let } b = \textbf{ref}(j) \textbf{ in } b := !b + !a; !b) & / \quad \emptyset_{\bowtie} \\
\rightharpoonup & \textbf{let } a = \ell_{(i)} \textbf{ in mkpar}(\textbf{fun } j \to \textbf{let } b = \textbf{ref}(j) \textbf{ in } b := !b + !a; !b) & / \quad \{\{\ell_0 \leftarrow 1\}, \{\ell_1 \leftarrow 1\}\} \\
\rightharpoonup & \textbf{mkpar}(\textbf{fun } j \to \textbf{let } b = \textbf{ref}(j) \textbf{ in } b := !b + !\ell_{(i)}; !b) & / \quad \{\{\ell_0 \leftarrow 1\}, \{\ell_1 \leftarrow 1\}\} \\
\rightharpoonup & \langle \textbf{let } b = \textbf{ref}(0) \textbf{ in } b := !b + !\ell_{(i)}; !b \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b := !b + !\ell_{(i)}; !b \rangle & / \quad \{\{\ell_0 \leftarrow 1\}, \{\ell_1 \leftarrow 1\}\} \\
\rightharpoonup & \langle \textbf{let } b = \ell' \textbf{ in } b := !b + !\ell_{(i)}; !b \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b := !b + !\ell_{(i)}; !b \rangle & / \quad \{\{\ell_0 \leftarrow 1, \ell'_0 \leftarrow 0\}, \{\ell_1 \leftarrow 1\}\} \\
\rightharpoonup & \langle \ell' := !\ell' + !\ell_{(i)}; !\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b := !b + !\ell_{(i)}; !b \rangle & / \quad \{\{\ell_0 \leftarrow 1, \ell'_0 \leftarrow 0\}, \{\ell_1 \leftarrow 1\}\} \\
\rightharpoonup & \langle \ell' := 0 + !\ell_{(i)}; !\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b := !b + !\ell_{(i)}; !b \rangle & / \quad \{\{\ell_0 \leftarrow 1, \ell'_0 \leftarrow 0\}, \{\ell_1 \leftarrow 1\}\} \\
\rightharpoonup & \langle \ell' := 0 + !\ell_0; !\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b := !b + !\ell_{(i)}; !b \rangle & / \quad \{\{\ell_0 \leftarrow 1, \ell'_0 \leftarrow 0\}, \{\ell_1 \leftarrow 1\}\} \\
\rightharpoonup & \langle \ell' := 0 + 1; !\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b := !b + !\ell_{(i)}; !b \rangle & / \quad \{\{\ell_0 \leftarrow 1, \ell'_0 \leftarrow 0\}, \{\ell_1 \leftarrow 1\}\} \\
\rightharpoonup & \langle \ell' := 1; !\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b := !b + !\ell_{(i)}; !b \rangle & / \quad \{\{\ell_0 \leftarrow 1, \ell'_0 \leftarrow 0\}, \{\ell_1 \leftarrow 1\}\} \\
\rightharpoonup & \langle !\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b := !b + !\ell_{(i)}; !b \rangle & / \quad \{\{\ell_0 \leftarrow 1, \ell'_0 \leftarrow 1\}, \{\ell_1 \leftarrow 1\}\} \\
\rightharpoonup & \langle 1 \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b := !b + !\ell_{(i)}; !b \rangle & / \quad \{\{\ell_0 \leftarrow 1, \ell'_0 \leftarrow 1\}, \{\ell_1 \leftarrow 1\}\} \\
\rightharpoonup & \ldots & \\
\rightharpoonup & \langle 1, 2 \rangle \ / \ \{\{\ell_0 \leftarrow 1, \ell'_1 \leftarrow 1\}, \{\ell_1 \leftarrow 1, \ell'_1 \leftarrow 2\}\} &
\end{array}
$$

For the second expression we have:

$$
\begin{array}{lll}
& \textbf{let } r = \textbf{ref}(3) \textbf{ in } r := !r + 1; !r & / \quad \emptyset_{\bowtie} \\
\rightharpoonup & \textbf{let } r = \ell_{(i)} \textbf{ in } r := !r + 1; !r & / \quad \{\{\ell_0 \leftarrow 3\}\{\ell_1 \leftarrow 3\}\} \\
\rightharpoonup & \ell_{(i)} := !\ell_{(i)} + 1; !\ell_{(i)} & / \quad \{\{\ell_0 \leftarrow 3\}\{\ell_1 \leftarrow 3\}\} \\
\rightharpoonup & \ell_{(i)} := 3 + 1; !\ell_{(i)} & / \quad \{\{\ell_0 \leftarrow 3\}\{\ell_1 \leftarrow 3\}\} \\
\rightharpoonup & \ell_{(i)} := 4; !\ell_{(i)} & / \quad \{\{\ell_0 \leftarrow 3\}\{\ell_1 \leftarrow 3\}\} \\
\rightharpoonup & !\ell_{(i)} & / \quad \{\{\ell_0 \leftarrow 4\}\{\ell_1 \leftarrow 4\}\} \\
\rightharpoonup & 4 & / \quad \{\{\ell_0 \leftarrow 4\}\{\ell_1 \leftarrow 4\}\}
\end{array}
$$

Natural semantics

Like in the small steps semantics, we have to distinguished two types of reductions: one for the local reductions (in each processors) and one for the global reductions (the network). Also, we have our stores and our set of stores. We do not write these rules because they are to much rules and they are easy to guess.

7.3.2 Type system

A natural extension

To type the reference, we extend our type algebra with:

$$
\begin{array}{lll}
\tau & ::= & \ldots \quad \text{like before} \\
 & | & \tau\ ref \quad \text{type reference}
\end{array}
$$

The free variables are defined with: $\mathcal{F}(\tau\ ref) = \mathcal{F}(\tau)$ and the substitution with $\varphi(\tau\ ref) = (\varphi(\tau)\ ref)$. We give the type schemes of our new operators and of the constant ():

$$
\begin{array}{lll}
TC(\textbf{ref}) & = & \forall\alpha.\ \alpha \to \alpha\ ref \\
TC(\textbf{!}) & = & \forall\alpha.\ \alpha\ ref \to \alpha \\
TC(\textbf{:=}) & = & \forall\alpha.[(\alpha\ ref * \alpha) \to unit/\mathcal{L}(\alpha)]
\end{array}
$$

Thus the affectation only works for local values. We give the natural extensions for $\mathcal{L}$ and for the construction of the constraints:

$$
\frac{\mathcal{L}(\tau\ ref)}{\mathcal{L}(\tau)}(Ref) \qquad \Bigg| \qquad \frac{\tau \leadsto C_1}{(\tau\ ref) \leadsto C_1}(CRef)
$$

Problems

But this extension of our type system is not sufe. Take for example:

```
let a=ref(fun x -> x)
  in r:=(fun x -> +(x,1)); (!r) true;;
```

expression is well-typed. But, when we compute this expression, we have $\vdash (1, \textbf{true})$ that could not be reduce.
Much type systems give new rules to not have this problem and ensure safety. One of them, give a new rule for the **let** binding expression ([Leroy, 2002] and [Leroy, 1992]):

$$\frac{E \vdash e_1 : \tau' \quad \sigma = \begin{cases} Gen(\tau', E) & \text{if } e_1 \text{ is not expansive} \\ \tau' & \text{else} \end{cases} \quad E + \{x : \sigma\} \vdash e_2 : \tau}{E \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau}$$

where the non-expansive expressions are:

$$\begin{array}{rcl} e_{ne} & ::= & x \mid c \mid op \mid \textbf{fun } x \to e \\ & & \mid (e_{ne},\ e_{ne}) \\ & & \mid \textbf{let } x = e_{ne} \textbf{ in } e_{ne} \\ & & \mid op(e_{ne}) \text{ where op is not ref} \end{array}$$

With this new inductive rule, our last expression is not well-typed because **ref** (**fun** $x \to x$) is expansive and $(!r)$**true** is now not well-typed. We can adapt this method to our type system.

But, for a BSP parallel functional language (like mini-BSML), it is not sufficient because the side effects are very dangerous for the safety of the language. Example:

```
let a=ref(0) in
 let truc=mkpar(fun pid -> a:=pid; pid mod 2=0) in
   if (at truc !a) then 1 else 2;;
```

We run, this expression with our small steps semantics for two processors:

$$\begin{array}{lll} & e \ / \ \emptyset_{\bowtie} & \\ \rightharpoonup & \textbf{let } a = \ell_{(i)} \textbf{ in let } truc = \textbf{mkpar}(\textbf{fun } pid \to \ell_{(i)} := pid; pid \ mod \ 2 = 0) & \\ & \quad \textbf{in if } truc \ \textbf{at}(!\ell_{(i)}) \textbf{ then } E_1 \textbf{ else } E_2 & / \quad \{\{l_0 \leftarrow 0\}, \{l_1 \leftarrow 0\}\} \\ \overset{*}{\rightharpoonup} & \textbf{if } \langle \textbf{true}, \textbf{false} \rangle \ \textbf{at}(!\ell_{(i)}) \textbf{ then } E_1 \textbf{ else } E_2 & / \quad \{\{l_0 \leftarrow 0\}, \{l_1 \leftarrow 1\}\} \end{array}$$

Now $(!\ell_{(i)})$ do not have the them value on each processors (0 for $i = 0$ and 1 for $i = 1$). With the first representation, intuitivelly we have **true** / $\{l_0 \leftarrow 0\}$ in processor 0 and **false** / $\{l_1 \leftarrow 1\}$ in processor 1 and we would have E_1 in first processor and E_2 in the second processor. It is a side effects.

So we have a **ifat** different in each processors. But the **ifat** is a synchronous operator and the **ifat** must be the same on each processors (see scheme of the operator).

Further work

The problem came from the $(\delta_{deref_{\bowtie}})$-rule and of the affectation in the **mkpar**: a "global" allocation is affected and is now different in all the store of the processor. So when we apply the $(\delta_{deref_{\bowtie}})$-rule, we have a different value on each processor for the **ifat**.

A solution, could be to detect programs, by a syntactic analysis, which affect some values to the global allocations. Thus, global reference would be the same in each processors. Thus, we always have the same value on the memory for the global allocation and we do not have to decompose our semantics.

But, a simple syntactic analysis is not sure and sufficient. Take for example:

```
let a=ref 0
in let truc=apply (mkpar (fun i -> fun x -> x:=i)) (mkpar (fun i -> a))
 in  if truc at !a the E1 else E2;;
```

That is why, for insure safety, we want to adapt the types and effects discipline [Talpin and Jouvelot, 1994] to our type system and thus eliminate the side effects.

allocations. But the allocation could be seen as a pointer to the memory (to a memory's adress). If we send an allocation (a local allocation) to a processor that do not has the adress for this allocation, there is no reduce rule to apply and the program stop with an error (the famous **segmentation fault**) if it makes a **!** on this allocation (if read "out" of the memory). To not have this problem, we could in the type system detected for the **put** inference rule, if the type of the send value is not a local allocation. If it is, the expression is not well-typed. With this new inductive rule, we ensure that the **!** always found the adress in the store. Another solution is to communicate the value contained by the location and to create a new location for this value (as in the Marshal module of Objective Caml).

Conclusion

In this technical report, we have described the design of a core language for BSML:

1. The theoretical bases of BSML was the BSλ-calculus and the BSλ_p-calculus. But these calculus were not typed. They have two kinds of syntactic constructions not to have the nesting of parallel vectors. For our language, these two kinds of constructions come down to indicate for each variable, if it is local or not. We have conceived a polymorphic type system, a type inference algorithm (and proved its correction and completeness) and proved its consistency with the BSλ_p syntactic constraints.
2. We have designed a weak call-by-value strategy. Our natural semantics evaluates more mini-BSML constructions than the older ones: [Loulergue, 2001a] has goal to represent a parallel composition and has neither couple nor fix point (because it was a no typed calculus); [Merlin et al., 2001] has presented a monomorphic type system for a explicitly typed language. There was no couple but they designed a BSP SECD abstract machine for this language.
3. Our small steps semantics is new and close to the distributed evaluation ([Loulergue, 2001b]). The type system has been proved correct for this semantics.
4. The extensions (nuples and sum types) make this version of mini-BSML the more expressive.
5. The small steps semantics for imperative treats and the problems of side effects are new in mini-BSML.

The implementation of the algorithm and the interaction with Objective-Caml for the operational semantics has allowed to verify for some none-trivial examples, our type system and the inference algorithms.

After adding imperative features to our language, to insure safety, the types and effects discipline [Talpin and Jouvelot, 1994] will then be necessary. We must implement those type systems and the algorithms for those extensions. Other extensions will be considered: complete matching (in particular, the matching of parallel vector expressions) and exception (we will study the adaptation of [Colard 1999] to BSλ_p-calculus).

The ultimate goal of the project is the conception of the BSML language and to adapt it to the Grid programmation (CARAML project). The resulting language will provide static analysis and profiling tools to relate source programs with performance estimates, tools to help to prove programs correction, tools to derivate programs, the derivation being driven by costs and provide safety and security for real-time application. A last step will be a study of the problems of no availability of processors and code mobility. An interaction with the *join-calculus* will be used [Gonthier, 2002].

Bibliography

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

[Chailloux1995] E. Chailloux and C. Foisy. Caml-Flight alpha: Implantation et applications. In Queinnec et al.

[Colard 1999] T Colard. A confluent $\lambda - calculus$ with a catch/throw mechanism. Journal of Functional Programming 9:6(1999):625-647

[Foisy1995] C. Foisy and E. Chailloux. Caml Flight: a portable SPMD extension of ML for distributed memory multiprocessors. In A. W. Böhm and J. T. Feo, editors, *Workshop on High Performance Functionnal Computing*, Denver, Colorado, April 1995. Lawrence Livermore National Laboratory, USA.

[Foisy1995] C. Foisy, J. Vachon, and G. Hains. DPML: de la sémantique à l'implantation. In P. Cointe, C. Queinnec, and B. Serpette, editors, *DPML: de la Sémantique à l'Implantation*, volume 11 of *Collection Didactique*, Noirmoutier, Février 1994. INRIA.

[Gerbessiotis and Valiant, 1994] Gerbessiotis, A. V. and Valiant, L. G. (1994). Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267. Long version of Gerbessiotis1992.

[Gonthier, 2002] G. Gonthier.(2002). Note course.

[Hains and Foisy, 1993] Hains, G. and Foisy, C. (1993). The data-parallel categorical abstract machine. In Bode, A., Reeve, M., and Wolf, G., editors, *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, pages 56–67, Munich. Springer.

[J.M.D.Hill et al., 1997] J.M.D.Hill, W.F.McColl, D.C.Stefanescu, Goudreau, M. W., Lang, K., Rao, S. B., T.Suel, T.Tsantilas, and Bisseling, R. (1997). BSPlib, the BSP programming library. Technical report, BSP Worldwide. URL=`http://www.bsp-worldwide.org/`.

[Leroy, 2001] X. Leroy. (2001). The Objective Caml System 3.04. http://www.caml.org.

[Leroy, 2002] Leroy, X.(2002). Note course.

[Leroy, 1992] X. Leroy. Typage polymorphe d'un langage Algorithmique *Thèse de Doctorat d'Université, Université Paris VII.* Juin 1992.

[Loulergue, 2002] F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library.

[Loulergue, 2001a] F. Loulergue,(2001). Parallel Composition and Bulk Synchronous Parallel Functional Programming.

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

[Loulergue, 2000] F. Loulergue. Conception de langages fonctionnels pour la programmation massivement parallèle. *Thèse de Doctorat d'Université, Université d'Orléans.* janvier 2000 of *International Symposium on High Performance Computing*, number 1940 in Lecture Note inComputer Science, page 355-363? Springer, Octobre 2000.

[Loulergue, 2000] F. Loulergue. G. Hains. O. Ballereau. BSMLlib version 0.1 Reference Manual.

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

[Loulergue, 1999a] F. Loulergue. Extension du BSλ-calcul. In P. Weis, editor, *JFLA'99 : Journées Francophones des Langages Applicatifs*, pages 93–112, Morzine-Avoriaz, February 1999.

[Merlin et al., 2001] Merlin, A., Hains, G., and Loulergue, F. (2001). A SPMD Environment Machine for Functional BSP Programs. In *Proceedings of the Third Scottish Functional Programming Workshop.*

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

[Talpin and Jouvelot, 1994] Talpin, J.-P. and Jouvelot, P. (1994). The type and effect discipline. *Information and Computation*, 111(2):245–296.

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