A Type-Based Algorithm for the Control-Flow Analysis of Higher-Order Concurrent Programs

M. Debbabi[†] A. Faour[†] N. Tawbi[†‡]

[†]Computer Science Department, Laval University, Quebec, G1K 7P4, Canada.
[‡]Bull Dyade, 78340 Les Clayes-Sous-Bois, France.
E-mails: {debabi,faour,tawbi}@ift.ulaval.ca

Abstract

We address, in a type-based framework, the problem of control-flow analysis for concurrent and functional languages. We present an efficient algorithm that propagates automatically types, communication effects and call graphs. The algorithm comes with a logical characterization that consists of a type proof system. The latter operates on a Concurrent ML core-syntax: a strongly typed, polymorphic kernel that supports higher-order functions and concurrency primitives. Effects are represented as algebraic terms that record communication effects resulting from channel creation, sending and receiving. Call graphs record function calls and are captured by a term algebra that is close to usual process algebras. Types are annotated with effects and call graphs. For the sake of flexibility, a subtyping relation is considered on the type algebra. We present the language syntax together with its static semantics that consists of the typing rules and an inference algorithm. The latter is proved to be consistent and complete with respect to the typing rules.

Keywords: Control-Flow Analysis, Concurrent ML, Annotated Type Systems, Call Graphs, Subtyping, Static Semantics.

1 Motivation and Background

Control-flow analysis is a traditional optimizing compiler technique. It aims to approximate, at compile time, dynamic function call graphs induced by program execution. Furthermore, control-flow analysis provides a static information that is valuable for debugging purposes. Control-flow analysis has been extensively studied for traditional imperative programming languages such as Fortran and C [1]. However, very little seems to have been done about the control-flow analysis of higher-order programming languages such as Scheme and ML.

In this paper, we are concerned in the control-flow analysis of concurrent, functional and imperative languages. More accurately, the language considered here consists of a Concurrent ML core-syntax [24, 25, 26, 27]. Consequently, the analysis will be done in the presence of first-class functions, higher-order processes, channels and references. The key idea underlying this work is to annotate types with structured approximations of call graphs. The annotated types are

then propagated by an inference system and its corresponding algorithm. As mentioned by O. Shivers [28, 30, 29, 31], the call graph information is crucial for control and data-flow optimizations.

Lately, a great deal of interest has been expressed in the use of type inference system in program analysis. Such an interest is motivated by the fact that type systems caters for a natural separation of the analysis into three parts. First, a specification of the analysis is elaborated thanks to a proof system. Second, an inference algorithm is devised. It aims to reconstruct automatically the information needed by the analysis. Finally, the information gathered is used to perform the analysis. Such a separation is valuable especially when extending the analysis to deal with new language features. Furthermore, type-based analyses are suitable to the extension in presence of modularity and separate compilation. On the other hand, abstract interpretation [10, 11, 12] provides a powerful approach for static analysis. It performs very precise analyses but does not provide a clear separation between the specification and the implementation issues.

Annotated type systems emerged first from the idea of considering side effects as part of the static evaluation. They have been firstly used within the FX project [15, 19, 17]. Afterwards, many annotated type systems have been proposed. Among them one can cite the type and effect discipline of Talpin and Jouvelot [33]. In this discipline, the static evaluation of an expression yields as a result not only its principal type, but also all the minimal side effects. Afterwards, various type systems have been devised by Bolignano and Debbabi [7, 5, 4, 6, 8, 14] as a generalization of the type and effect discipline in order to deal with concurrency. The generalization consists of an inference type system that propagates the communication effects that results from channel creation, sending and receiving. Nevertheless, Similar type systems have been advanced. In [22, 23], H. Riis-Nielson and F. Nielson presented a framework for analyzing the communication topology of CML programs. They extended a polymorphic type system for SML in order to deduce the types of CML programs together with their communication behaviours expressed as terms in a process algebra. In [36], Thomsen presented a sort and type system for a simple variant of Facile where constructs for channel creation, sending and receiving are functions of polymorphic types as opposed to syntactic constructs in the original definition of Facile.

Here is the way the rest of the paper is organized. Section 2 is devoted to the state of the art in control-flow analysis. An informal description of the language is presented in Section 3. A full presentation of the static semantics is given in Section 4. The reconstruction algorithm comes in Section 5. Section 6 is devoted to the establishment of the completeness and the soundness of the algorithm with regard to the typing rules. A few concluding remarks and a discussion of future work are ultimately sketched as a conclusion in Section 7.

2 Related Work

Control-flow analysis is a very well-known problem for compiler developers. A heavy machinery of various control-flow analyses is employed in traditional imperative compilers such as C and Fortran. These analyses contribute significantly to the efficiency of the corresponding compilers. The reader may refer to [2] for a full account of such optimizations.

On the other hand, higher-order programming languages, such as ML, Scheme, Concurrent ML, etc, are very popular mainly for their widely recognized expressiveness power. However, the most up-to-date ML or Scheme compiler is still roughly as efficient as the non-optimizing traditional imperative compilers. One of the main reasons underlying this gap is the level of optimization applied. It is well known since [28, 30, 29, 31], that in order to apply any data-flow optimization: common-subexpression elimination, loop invariant detection, induction variable elimination, etc, one has to know explicitly at compile-time an approximation of control-flow graph.

Since then, a great deal of interest has been expressed in the control-flow analysis problem usually referred to as the CFA problem. Shivers demonstrated in [28, 30, 29, 31] that one key problem in CFA is the static computation of call graphs. He proposed in [28, 30, 29, 31] many algorithms for the static reconstruction of call graphs in Scheme. These algorithms have been formulated in the framework of abstract interpretation of P. Cousot and R. Cousot [10, 11, 12]. Consequently, he adopted the following approach: first, he developed a standard denotational semantics of CPS (Continuation-Passing Style) Scheme. Second, he modified this semantics so as to compute the desired property i.e. call graphs. This semantics (exact control-flow analysis) referred to as non-standard semantics is uncomputable, which leads him to abstract it so as to obtain a computable approximation. Accordingly, he developed various abstractions of the exact control-flow analysis such as 0CFA and 1CFA [28, 30, 29, 31].

In [16], Jagannathan and Weeks described a framework for flow analysis in higher-order languages that makes explicit use of flow graphs. It aims to model control and data flow properties of untyped higher-order programs. This frame work is an extension of earlier work in this area, most notably [29, 32].

Later Boucher and Feeley, in [9] proposed a new paradigm of abstraction inspired by partial evaluation techniques. They called it abstract compilation. In fact, this approach consists in compiling the source program into a program which computes the desired analysis when it is executed instead of interpreting it. They illustrate this paradigm by applying it to control-flow analysis of functional programs.

In [35, 34], Tang and Jouvelot proved that the control-flow analysis could be expressed either as a type and effect system or abstract interpretation. They presented a new static control-flow analysis that combines both techniques in a single unified framework. This is what they called separate abstract interpretation. Furthermore they demonstrated that this new framework could be viewed as a conservative extension of abstract interpretation. In addition, the underlying subtyping effect system is built upon a monomorphic type system where the

subtype relation is induced by a subsumption rule on effects. Moreover it does not handle concurrency.

The work reported here originates from the following observation: the control information gathered by the algorithms of both Shivers and Tang-Jouvelot consists of the set of function names possibly called during the evaluation of expression. By doing so, these algorithms lose the control structure of the program. Tang and Jouvelot decided to do so in order to preserve the decidability of their type and effect system.

In this paper, we will elaborate a new algorithm that gathers a much more precise and structured approximation of the control information. This will be accomplished by putting much more structure on the control information. Actually this structure is a strong simulation relation on call graphs [20]. Moreover, our algoritm operates on a multi-paradigmatic language that incorporates higher-order concurrency extensions. Furthermore, the information gathered by our algorithm is richer since it comes with a compile-time approximation of the communication effects. The presence of communication effects control the generalization of type variables in the `let` construct. The algorithm reported here, is firstly specified with a logical characterization that consists of a type and effect system. The latter is proved to be sound and complete with respect to the logical specification.

3 Syntax

In this section we present the CML core-syntax considered in this work. We have kept the number of constructs to a bare minimum so as to facilitate a more compact and complete description of the static and the dynamic semantics. We consider:

– Literals, such as the boolean `true` and `false`, and a distinguished value `()` that belongs to the one-element type usually referred to as *unit*.

– Three binding constructs, the functional abstraction, the recursion and the let definition. The construct f `where` $f(x)$`=` e stands for the definition of a function whose body is e. The construct `rec` f `where`$f(x)$`=` e stands for the definition of a recursive function whose body is e. In fact the recursion operator `rec` is not part of the CML syntax, since recursion is implicit in CML. It has been included just to provide a semantics for recursive behaviors.

– Imperative aspects are supported in CML through the notion of reference. The latter is not included in this core-syntax since it can be simulated easily using servers that communicate through typed channels, as shown in [3, 27]. Semantically, references are treated as communication channels.

– Expressions may communicate through channels. New channels are created by applying the built-in function `channel` to the trivial value `()`. The expression `transmit`(e_1,e_2) means evaluate the expression e_1 to get a channel value and then evaluate the expression e_2. The value returned afterwards is an "event" that stands for a potential communication that results from sending the value along the channel. Similarly, `receive`(e) means evaluate the expression e to get

Exp	$\ni e$	$::= x \mid v \mid e\ e' \mid$ `rec` f `where` $f(x)$`=` $e \mid$	(Expressions)
		$\mid\ (e, e') \mid e\ ;\ e' \mid$ `let` x`=`e `in` e' `end`	
		$\mid$ `if` e `then` e' `else` e'' `end`	
Const	$\ni c$	$::=$ `()` $\mid$ `true` $\mid$ `false` $\mid$ `channel` $\mid$ `spawn` $\mid$ `sync`	(Constants)
		$\mid\ ec$	
Val	$\ni v$	$::= c \mid f$ `where` $f(x)$ `=` e	(Values)
ECon	$\ni ec$	$::=$ `receive` $\mid$ `transmit` $\mid$ `choose`	(Event Constructors)

Table 1. The Core Syntax

a channel value and then form an event that denotes the possibility of receiving a value on the channel. Events are activated using the construct `sync`.
– Parallel behaviors are supported through the unary operator `spawn`, which stands for the activation of its argument expression, meant to be executed in parallel with the rest of the program.
– We have a CCS-like choice operator, referred to as `choose` whose behavior amounts to the nondeterministic choice of some event. The reader should notice that we used a binary choice operator, as in [3, 27], for the sake of compactness. More formally, the BNF syntax of the core language is presented in Table 1. Notice that we have four syntactic categories. The category of expressions (Exp) ranged over by e, the category of constants (Const) ranged over by c, the category of values (Val) ranged over by v and the category of event constructors (ECon) ranged over by ec.

Along this paper, given two sets A and B, we will write $A \xrightarrow[m]{} B$ to denote the set of all mappings from A to B, having in mind that there is no overlap (that is, mappings are functions, not relations). A mapping (map for short) $m \in A \xrightarrow[m]{} B$ could be defined by extension as $[a_1 \mapsto b_1, ..., a_n \mapsto b_n]$ to denote the association of the elements b_i's to a_i's. We will write $dom(m)$ to denote the domain of the map m and $ran(m)$ to denote its range (co-domain). We will write $m_{x_1,x_2,...}$, the map m excluding the associations of the form $x_i \mapsto \_$. Given two maps m and m', we will write $m \dagger m'$ the overwriting of the map m by the associations of the map m' i.e. the domain of $m \dagger m'$ is $dom(m) \cup dom(m')$ and we have $(m \dagger m')(a) = m'(a)$ if $a \in dom(m')$ and $m(a)$ otherwise. We will use $\mathcal{P}_f(E)$ to denote the finite powerset of a given set E.

4 Static Semantics

The static semantics is based on an extension of the type and effect discipline. This allows us to control safely the type generalization in the presence of polymorphic channels. Actually types and effects are used to control the generalization of type and effect variables. As shown in [18, 33], effect-based type systems outperform the other type systems when typing in the presence of mutable data. Furthermore, call graphs, communication effects, together with principal types

capture a valuable static information on the dynamic behaviors of a program. The latter information may be of great interest in the static analysis of programs.

We need the following static domains:

– The domain of *regions*: regions are intended to abstract channels. Their domain consists in the disjoint union of a countable set of constants ranged over by r, and variables ranged over by ϱ. We will use ρ, ρ', etc., to represent values drawn from this domain.

– The domain of communication *effects*: it is defined inductively as follows:

$$\sigma ::= \emptyset \mid \varsigma \mid \sigma \cup \sigma' \mid create(\rho, \tau) \mid in(\rho, \tau) \mid out(\rho, \tau)$$

We use $\emptyset$ to denote an empty communication effect. We use ς to stand for a communication effect variable. The communication effect $create(\rho, \tau)$ represents the creation, in the region ρ, of a channel that is a medium for values of type τ. The term $in(\rho, \tau)$ denotes the effect resulting from receiving a value of type τ on a channel in the region ρ. The term $out(\rho, \tau)$ denotes an output of a value of type τ, on a channel in the region ρ. We introduce also a union operator $\cup$ intended to effect cumulation.

– The algebra of call *graphs*: it is defined as follows:

$$\gamma ::= nil \mid \delta \mid f.\gamma \mid \gamma + \gamma' \mid \gamma; \gamma' \mid fork(\gamma) \mid rec\ \delta.\gamma$$

The terms of this algebra stand for call graph approximation noted Call graphs in the sequel. We use nil to denote the empty call graph and δ to denote a polymorphic variable call graph. A term of the form $f.\gamma$ stands for a call graph recording an initial call of the function f and having γ as the corresponding unique subgraph. The term $\gamma + \gamma'$ is used to denote a call graph with two alternatives call graphs γ and γ'. One and only one of the two branches will be effective for some program execution. The notation $\Sigma_{i=0}^{n}\gamma_i$ will be used as an abbreviation of $\gamma_1 + \ldots + \gamma_n$. The call graph operator $+$ is associative, commutative, idempotent and has nil as a unit element. The term $\gamma; \gamma'$ denotes the sequencing of γ and γ'. The term $fork(\gamma)$ denotes the spawning of an expression having γ as a call graph. The term $rec\ \delta.\gamma$ denotes a recursively defined call graph. For instance, the term $rec\ \delta.(f.g.nil + h.t.\delta)$ denotes the call graph of a program that either calls the function f and then g, or makes a call to the function h, followed by a call to the function t, and then recurses.

– The domain of *types*: is inductively defined by:

$$\tau ::= unit \mid bool \mid \alpha \mid chan_\rho(\tau) \mid event_{\sigma,\gamma}(\tau) \mid \tau \xrightarrow{\sigma,\gamma} \tau'$$

unit is the type with only one element "`()`", *bool* the type of usual truth values `true` and `false`, α a type variable. The term $chan_\rho(\tau)$ is the type of channels in the region ρ that are intended to be media for values of type τ. The term $\tau \xrightarrow{\sigma,\gamma} \tau'$ is the type of functions that take parameters of type τ to values of type τ' with a *latent* effect σ and a latent call graph γ. By latent effect (respectively call graph), we mean the effect (respectively call graph) generated when the corresponding function expression is applied to its arguments. The type $event_{\sigma,\gamma}(\tau)$

denotes inactive processes made of potential communications (latent effect σ) and function calls (latent call graph γ) that are expected to return a value of type τ once their execution terminated.
The static semantics manipulates sequents of the form $\mathcal{E} \vdash e : \tau, \sigma, \gamma$.which state that under some typing environment $\mathcal{E}$ the expression e has type τ, effect σ and call graph γ. We also define type schemes of the form $\forall v_1, \ldots, v_n.\tau$, where v_i can be type, region, effect or call graph variable. A type τ' is an instance of $\forall v_1, \ldots, v_n.\tau$ noted $\tau' \lhd \forall v_1, \ldots, v_n.\tau$, if there exists a substitution θ defined over $v_1, \ldots, v_n$ such that $\tau' = \theta\tau$. Static environments, ranged over by $\mathcal{E}$, map identifiers to type schemes. The Table 2 presents the static semantics of our core language.

Type generalization in this type system states that a variable cannot be generalized if it is free in the type environment $\mathcal{E}$ or if it is present in the inferred effect or call graph. The first condition is classical while the other is due to the fact that types are annotated by effects and graphs. Concerning the effect part, the reader should refer to [33, 4] for a detailed explanation of this issue. The same argument apply to call graphs.

$$Gen(\mathcal{E}, \sigma, \gamma)(\tau) = \textbf{let } \{v_1, \ldots, v_n\} = fv(\tau) \backslash (fv(\mathcal{E}) \cup fv(\sigma) \cup fv(\gamma)) \\ \textbf{in } \forall v_1, \ldots, v_n.\tau \textbf{ end}$$

where $fv(\_)$ denotes the set of free variables:

$$\begin{aligned} fv(\mathcal{E}) &= \cup\{fv(\mathcal{E}(x)) \mid x \in dom(\mathcal{E})\} \\ fv(\forall v_1, ..., v_n.\tau) &= fv(\tau) \backslash \{v_1, \ldots, v_n\} \\ fv(chan_\rho(\tau)) &= fv(\rho) \cup fv(\tau) \\ fv(event_{\sigma,\gamma}(\tau)) &= fv(\sigma) \cup fv(\gamma) \cup fv(\tau) \\ fv(\tau \xrightarrow{\sigma,\gamma} \tau') &= fv(\sigma) \cup fv(\gamma) \cup fv(\tau) \cup fv(\tau') \\ fv(create(\rho, \tau)) &= fv(\rho) \cup fv(\tau) \\ fv(in(\rho, \tau)) &= fv(\rho) \cup fv(\tau) \\ fv(out(\rho, \tau)) &= fv(\rho) \cup fv(\tau) \\ fv(\sigma \cup \sigma') &= fv(\sigma) \cup fv(\sigma') \\ fv(\gamma + \gamma') &= fv(\gamma) \cup fv(\gamma') \\ fv(\gamma; \gamma') &= fv(\gamma) \cup fv(\gamma') \\ fv(rec\ \delta.\gamma) &= fv(\gamma) \backslash \{\delta\} \end{aligned}$$

$$\begin{aligned} fv(unit) &= \{\} \\ fv(bool) &= \{\} \\ fv(\emptyset) &= \{\} \\ fv(\alpha) &= \{\alpha\} \\ fv(\varsigma) &= \{\varsigma\} \\ fv(r) &= \{\} \\ fv(\delta) &= \{\delta\} \\ fv(\varrho) &= \{\varrho\} \\ fv(nil) &= \{\} \\ fv(f.\gamma) &= fv(\gamma) \\ fv(fork(\gamma)) &= fv(\gamma) \end{aligned}$$

The function $TypeOf$ allows the typing of built-in primitives as defined in Table 6. Since types may be annotated by effects and call graphs, the typing context may lead occasionally to a type mismatch between subexpressions that have similar type structure but different graph annotations. This may appear when typing applications and conditionals. Generally, two techniques may be considered to get rid of this problem:
– The first technique, usually referred to as *subeffecting* consists in computing larger annotations so as to avoid type clashes. This technique remains acceptable as far as programming is the issue. Nevertheless, in the case of static analysis, the use of sub-effecting may result in a significant loss of information that affects the analysis precision.

(cte) $$\frac{\tau \lhd TypeOf(\texttt{cte})}{\mathcal{E} \vdash \texttt{cte} : \tau, \emptyset, nil}$$

(var) $$\frac{\tau \lhd \mathcal{E}(x)}{\mathcal{E} \vdash x : \tau, \emptyset, nil}$$

(abs) $$\frac{\mathcal{E}_x \dagger [x \mapsto \tau] \vdash e : \tau', \sigma, \gamma}{\mathcal{E} \vdash \ f \ \texttt{where}\ f(x)\texttt{=}e : \tau \xrightarrow{\sigma, f.\gamma} \tau', \emptyset, nil}$$

(app) $$\frac{\mathcal{E} \vdash e : \tau \xrightarrow{\sigma,\gamma} \tau', \sigma', \gamma' \quad \mathcal{E} \vdash e' : \tau, \sigma'', \gamma''}{\mathcal{E} \vdash (e \ \ e') : \tau', \sigma \cup \sigma' \cup \sigma'', ((\gamma'; \gamma''); \gamma)}$$

(let) $$\frac{\mathcal{E} \vdash e : \tau, \sigma, \gamma \quad \mathcal{E}_x \dagger [x \mapsto Gen(\mathcal{E}, \sigma, \gamma)(\tau)] \vdash e' : \tau', \sigma', \gamma'}{\mathcal{E} \vdash \texttt{let}\ x\texttt{=}e\ \texttt{in}\ e'\ \texttt{end} : \tau', \sigma \cup \sigma', (\gamma; \gamma')}$$

(pair) $$\frac{\mathcal{E} \vdash e : \tau, \sigma, \gamma \quad \mathcal{E} \vdash e' : \tau', \sigma', \gamma'}{\mathcal{E} \vdash (e, e') : \tau \times \tau', \sigma \cup \sigma', (\gamma; \gamma')}$$

(seq) $$\frac{\mathcal{E} \vdash e : \tau, \sigma, \gamma \quad \mathcal{E} \vdash e' : \tau', \sigma', \gamma'}{\mathcal{E} \vdash e \ ; \ e' : \tau', \sigma \cup \sigma', (\gamma; \gamma')}$$

(if) $$\frac{\mathcal{E} \vdash e : bool, \sigma, \gamma \quad \mathcal{E} \vdash e' : \tau', \sigma', \gamma' \quad \mathcal{E} \vdash e'' : \tau'', \sigma'', \gamma''}{\mathcal{E} \vdash \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e''\ \texttt{end} : \tau, \sigma''', \gamma'''} \begin{cases} \tau' \preccurlyeq \tau \\ \tau'' \preccurlyeq \tau \\ \sigma \cup \sigma' \cup \sigma'' \preccurlyeq \sigma''' \\ \gamma; (\gamma' + \gamma'') \preccurlyeq \gamma''' \end{cases}$$

(rec) $$\frac{\mathcal{E}_{x,f} \dagger [x \mapsto \tau, \ f \mapsto \tau \xrightarrow{\sigma,\delta} \tau'] \vdash e : \tau', \sigma, \gamma}{\mathcal{E} \vdash \texttt{rec}\ f\ \texttt{where}\ f(x)\texttt{=}\ e : \tau \xrightarrow{\sigma, rec\ \delta.\gamma} \tau', \emptyset, nil}$$

Table 2. The Typing Rules

– The second technique consists in introducing a *subtyping* relation on the type and annotation algebras. In other words, appropriate preorders are defined on types and annotations. Consequently, different types are tolerated during the pattern-matching as far as the difference concerns only annotations. For instance, in a conditional construct we allow the types of the branches to be dissimilar and only require them to be subtypes of a common and explicitly given type. By doing so, the information gathered by the type system is kept intact and then no loss of information is reported. However, subtyping introduce a significant complexity at the level of the type system as well as the inference algorithm.

In this work, we adopt the second technique, subtyping, since the issue is static analysis. Accordingly, we begin by introducing preorder relations on types, effects and graphs. First of all, these preorders will all be written $\preccurlyeq$ ($\preccurlyeq$ is overloaded) and the corresponding kernel will be written $\equiv$ which is equivalent to $\preccurlyeq \cap \succcurlyeq$. It is defined by: $t \equiv t'$ if and only if $t \preccurlyeq t'$ and $t \succcurlyeq t'$.

(Reflexivity)	$\gamma \preccurlyeq \gamma$	**(;+ Distributivity)**	$(\gamma + \gamma'); \gamma'' \equiv (\gamma; \gamma'') + (\gamma'; \gamma'')$
(Transitivity)	$\frac{\gamma \preccurlyeq \gamma' \quad \gamma' \preccurlyeq \gamma''}{\gamma \preccurlyeq \gamma''}$	**(+ 1st law)**	$\gamma \preccurlyeq \gamma + \gamma'$
(Pre-cong. 1)	$\frac{\gamma \preccurlyeq \gamma'}{f.\gamma \preccurlyeq f.\gamma'}$	**(+ 2nd law)**	$\gamma \preccurlyeq \gamma' + \gamma$
(Pre-cong. 2)	$\frac{\gamma \preccurlyeq \gamma'}{rec\ \delta.\gamma \preccurlyeq rec\ \delta.\gamma'}$	**(+ 3nd law)**	$\gamma + nil \preccurlyeq \gamma$
(Pre-cong. 3)	$\frac{\gamma \preccurlyeq \gamma' \quad \gamma'' \preccurlyeq \gamma'''}{\gamma + \gamma'' \preccurlyeq \gamma' + \gamma'''}$	**(Idempotence)**	$\gamma + \gamma \preccurlyeq \gamma$
(Pre-cong. 4)	$\frac{\gamma \preccurlyeq \gamma' \quad \gamma'' \preccurlyeq \gamma'''}{\gamma; \gamma'' \preccurlyeq \gamma'; \gamma'''}$	**(; Associativity)**	$\gamma; (\gamma'; \gamma'') \equiv (\gamma; \gamma'); \gamma''$
(Pre-cong. 5)	$\frac{\gamma \preccurlyeq \gamma'}{fork(\gamma) \preccurlyeq fork(\gamma')}$	**(Unwinding)**	$rec\ \delta.\gamma \equiv \gamma[rec\ \delta.\gamma/\gamma]$
(EmptyGraph 1)	$\gamma \equiv nil; \gamma$	**(EmptyGraph 2)**	$\gamma; nil \equiv \gamma$
	(Renaming) $rec\ \delta.\gamma \equiv rec\ \delta'.\gamma[\delta'/\delta]$ where $\delta' \notin fv(\gamma)$		

Table 3. Subtyping on Call Graphs

(Reflexivity)	$\tau \preccurlyeq \tau$	**(Pre-cong. 2)**	$\frac{\tau \preccurlyeq \tau' \quad \tau'' \preccurlyeq \tau''' \quad \sigma \preccurlyeq \sigma' \quad \gamma \preccurlyeq \gamma'}{\tau' \xrightarrow{\sigma,\gamma} \tau'' \preccurlyeq \tau \xrightarrow{\sigma',\gamma'} \tau'''}$
(Transitivity)	$\frac{\tau \preccurlyeq \tau' \quad \tau' \preccurlyeq \tau''}{\tau \preccurlyeq \tau''}$	**(Pre-cong. 3)**	$\frac{\tau \preccurlyeq \tau'}{chan_\rho(\tau) \preccurlyeq chan_\rho(\tau')}$
(Pre-cong. 1)	$\frac{\tau \preccurlyeq \tau' \quad \tau'' \preccurlyeq \tau'''}{\tau \times \tau'' \preccurlyeq \tau' \times \tau'''}$	**(Pre-cong. 4)**	$\frac{\tau \preccurlyeq \tau' \quad \sigma \preccurlyeq \sigma' \quad \gamma \preccurlyeq \gamma'}{event_{\sigma,\gamma}(\tau) \preccurlyeq event_{\sigma',\gamma'}(\tau')}$

Table 4. Subtyping on Types

Table 3 axiomatizes the subtyping relation on the algebra of call graphs. The rules stipulate that such a relation is a preorder (reflexive and transitive) and a precongruence that is closed under the different operations on graphs. The rules also state that the operator ; is associative, commutative and idempotent and admit *nil* as a neutral element. The call graph operator ; is right-distributive with respect to the call graph operator +. We can easily realize that using idempotence with the first and second law leads to an equivalence between $\gamma + \gamma$ and γ. The unwinding property states that it is always possible to fold and unfold a recursion. Finally, the renaming rule states that it is possible to rename bound call graph variables. Notice that such an axiomatization is close to some extent to

(Reflexivity)	$\sigma \preccurlyeq \sigma$	**(Idempotence)**	$\sigma \cup \sigma \preccurlyeq \sigma$
(Transitivity)	$\dfrac{\sigma \preccurlyeq \sigma' \quad \sigma' \preccurlyeq \sigma''}{\sigma \preccurlyeq \sigma''}$	**($\cup$ 1st law)**	$\sigma \preccurlyeq \sigma \cup \sigma'$
(Precong.)	$\dfrac{\sigma \preccurlyeq \sigma' \quad \sigma'' \preccurlyeq \sigma'''}{\sigma \cup \sigma'' \preccurlyeq \sigma' \cup \sigma'''}$	**($\cup$ 2nd law)**	$\sigma \preccurlyeq \sigma' \cup \sigma$
(EmptyEff. 1)	$\sigma \equiv \emptyset \cup \sigma$	**(EmptyEff. 2)**	$\sigma \cup \emptyset \equiv \sigma$

(ElemEffe) $\dfrac{\tau \preccurlyeq \tau'}{\mu(\rho,\tau) \preccurlyeq \mu(\rho,\tau')}$ $\mu \in \{create, in, out\}$

Table 5. Subtyping on Effects

$$
\begin{array}{l}
TypeOf \;=\; [\\
\texttt{()} \mapsto unit, \\
\texttt{true} \mapsto bool, \\
\texttt{false} \mapsto bool, \\
\texttt{channel} \mapsto \forall\alpha, \varrho.\; unit \xrightarrow{create(\varrho,\alpha),channel.nil} chan_{\varrho}(\alpha), \\
\texttt{receive} \mapsto \forall\alpha, \varrho.\; chan_{\varrho}(\alpha) \xrightarrow{\emptyset,receive.nil} event_{in(\varrho,\alpha),nil}(\alpha), \\
\texttt{transmit} \mapsto \forall\alpha, \varrho.\; chan_{\varrho}(\alpha) \times \alpha \xrightarrow{\emptyset,transmit.nil} event_{out(\varrho,\alpha),nil}(unit), \\
\texttt{choose} \mapsto \forall\alpha, \varsigma, \varsigma', \delta, \delta'.\; event_{\varsigma,\delta}(\alpha) \times event_{\varsigma',\delta'}(\alpha) \xrightarrow{\emptyset,choose.nil} event_{\varsigma\cup\varsigma',\delta+\delta'}(\alpha), \\
\texttt{spawn} \mapsto \forall\varsigma, \delta.\; (unit \xrightarrow{\varsigma,\delta} unit) \xrightarrow{\emptyset,spawn.fork(\delta).nil} unit, \\
\texttt{sync} \mapsto \forall\alpha, \varsigma.\; event_{\varsigma,\delta}(\alpha) \xrightarrow{\varsigma,sync.\delta} \alpha \\
]
\end{array}
$$

Table 6. The Initial Static Basis

the strong simulation notion of Park and Milner [21, 20]). Table 4 axiomatizes subtyping on the type algebra. The rules stipulate that such a relation is a preorder (reflexive and transitive) that is monotonically closed under all the type constructors not including the arrow type constructor (function). The latter is contravariant on the first argument and covariant on the second. Table 5 axiomatizes subtyping on effects. The rules stipulate that such a relation is a preorder (reflexive and transitive) that is closed under effects union. The rules also define the $\cup$ operator as being associative, commutative, idempotent and admitting $\emptyset$ as a neutral element.

Now, let us turn to the explanation of the typing rules of Table 2. The typing of constants is standard and is dictated by the basis environment. It stipulates that any instance of the corresponding type scheme is accepted as a type together

with an empty effect and a *nil* call graph. The same explanation applies to the typing of variables. For function abstraction, the resulting type, effect and call graph indicate that neither effect is reported nor function call is performed when defining the function. Effects and function calls are latent and take place when the function is applied. For an application expression, the overall inferred effect and call graph express eager left-to-right evaluation: first, the function expression e is evaluated into a function abstraction, then the argument e' is evaluated and finally the function is applied to the argument. For the pair and sequencing rules, the corresponding inferred types, effects and call graphs are straightforward. The typing rule for recursive function definition is similar to the rule for abstraction except that we need to extend the type environment with assumptions about the recursive function. Finally, the rule for conditional allows the types of the branches to be dissimilar and only requires them to be subtypes of a common and explicitly given type. The effect inferred is the union of the three effects produced by both the boolean expression and the two branches, while the resulting call graph should report at first the boolean expression call graph γ followed by the call graph that results from evaluating either the then-branch or the else-branch. Since the inference is done at compile-time, it is undecidable to know which branch will be executed. Accordingly, both call graphs are reported using the $+$ operator on call graphs i.e. $(\gamma' + \gamma'')$.

5 Inference Algorithm

The present section is dedicated to the algorithm of type, effect and call graph inference. This algorithm is inspired by the type inference discipline of Damas-Milner [13]. However, it deviates from the Damas-Milner schema by propagating inequations (constraints) on types, effects and call graphs and also by using a constrained unification instead of a syntactic unification. The inference algorithm is presented in Table 7.

In the static semantics, type schemes are of the form $\forall v_1, \ldots, v_n.\tau$ where each v_i can be type, region, effect or call graph variable. In the algorithm, effects and call graphs are represented by variables and are subjected to sets of constraints. Consequently, type schemes will be now of the form $\forall v_1, \ldots, v_n.(\tau, \varphi)$ where φ is a set of type, effect and call graph constraints.

In order to relate the constrained type schemes and environments of the algorithm to the static semantics, we write $\overline{\forall v_1, \ldots, v_n.(\tau, \varphi)}$ to denote $\forall v_1, \ldots, v_n.\overline{\varphi}\tau$ and we define $\overline{\mathcal{E}}$ by extension as $\overline{\mathcal{E}}(x) = \overline{\mathcal{E}(x)}$ for all $x \in dom(\mathcal{E})$.

Our inference algorithm proceeds by case analysis on the structure of expression. It takes as input a 3-tuple made of a static environment, an expression and a set of type, effect and call graph constraints. The algorithm either fails or terminates successfully producing a 5-tuple whose components are: a substitution, a type, an effect, a call graph and a set of constraints. The substitution records those substitutions that take place during the various recursive calls of the inference algorithm. The type produced by the algorithm is the inferred type of the argument expression. The effect produced by the algorithm corresponds to the

$\mathrm{Infer}(\mathcal{E}, \varphi, \texttt{cte}) =$
  let $\forall v_1, ..., v_n.\ (\tau, \varphi') = TypeOf(\texttt{cte})$
  in let $v'_1, ..., v'_n$ **new**, $\theta = [v_i \mapsto v'_i | i = 1, ..., n]$ **in** $(Id, \theta\tau, \emptyset, nil, \varphi \cup \theta\varphi')$ **end**
  end
$\mathrm{Infer}(\mathcal{E}, \varphi, x) =$
  if $x \notin dom(\mathcal{E})$ **then** $fail$
  else
    let $\forall v_1, ..., v_n.\ (\tau, \varphi') = \mathcal{E}(x)$
    in let $v'_1, ..., v'_n$ **new**, $\theta = [v_i \mapsto v'_i | i = 1, ..., n]$
      in $(Id, \theta\tau, \emptyset, nil, \varphi \cup \theta\varphi')$
      end
    end
  end
$\mathrm{Infer}(\mathcal{E}, \varphi, f\ \texttt{where}\ f(x)\texttt{=}e) =$
  let $\alpha, \varsigma, \delta$ **new**
    $(\theta, \tau, \sigma, \gamma, \varphi') = \mathrm{Infer}(\mathcal{E}_x \dagger [x \mapsto (\alpha, \{\})], \varphi, e)$
  in $(\theta, \theta\alpha \xrightarrow{\varsigma, \delta} \tau, \emptyset, nil, \varphi' \cup \{\sigma \preccurlyeq \varsigma, f.\gamma \preccurlyeq \delta\})$
  end
$\mathrm{Infer}(\mathcal{E}, \varphi, e\ \ e') =$
  let $(\theta, \tau, \sigma, \gamma, \varphi') = \mathrm{Infer}(\mathcal{E}, \varphi, e)$
    $(\theta', \tau', \sigma', \gamma', \varphi'') = \mathrm{Infer}(\theta\mathcal{E}, \varphi', e')$
    $\alpha, \varsigma, \delta$ **new**
    $\theta'' = \mathcal{U}_{\varphi''}(\theta'\tau, \tau' \xrightarrow{\varsigma, \delta} \alpha)$
  in $(\theta'' \circ \theta' \circ \theta, \theta''\alpha, \theta''(\theta'\sigma \cup \sigma' \cup \varsigma), (((\theta''\theta'\gamma); \theta''\gamma'); \theta''\delta), \theta''\varphi'')$
  end
$\mathrm{Infer}(\mathcal{E}, \varphi, \texttt{let}\ x\texttt{=}e\ \texttt{in}\ e'\ \texttt{end}) =$
  let $(\theta, \tau, \sigma, \gamma, \varphi') = \mathrm{Infer}(\mathcal{E}, \varphi, e)$
    $(\forall v_1, ..., v_n.(\tau, \varphi''), \varphi''') = Gen_{\varphi'}(\theta\mathcal{E}, \sigma, \gamma)(\tau)$
    $\mathcal{E}' = \theta\mathcal{E}_x \dagger [x \mapsto \forall v_1, ..., v_n.(\tau, \varphi'')]$
    $(\theta', \tau', \sigma', \gamma', \varphi'''') = \mathrm{Infer}(\mathcal{E}', \varphi''', e')$
  in $(\theta' \circ \theta, \tau', \theta'\sigma \cup \sigma', (\theta'\gamma; \gamma'), \varphi'''')$
  end
$\mathrm{Infer}(\mathcal{E}, \varphi, e; e') =$
  let $(\theta, \tau, \sigma, \gamma, \varphi') = \mathrm{Infer}(\mathcal{E}, \varphi, e)$
    $(\theta', \tau', \sigma', \gamma', \varphi'') = \mathrm{Infer}(\theta\mathcal{E}, \varphi', e')$
  in $(\theta' \circ \theta, \tau', \theta'\sigma \cup \sigma', (\theta'\gamma; \gamma'), \varphi'')$
  end
$\mathrm{Infer}(\mathcal{E}, \varphi, \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e''\ \texttt{end}) =$
  let $(\theta, \tau, \sigma, \gamma, \varphi') = \mathrm{Infer}(\mathcal{E}, \varphi, e)$
    $\theta' = \mathcal{U}_{\varphi'}(\tau, bool)$
    $(\theta'', \tau', \sigma', \gamma', \varphi'') = \mathrm{Infer}(\theta'\theta\mathcal{E}, \theta'\varphi', e')$
    $(\theta''', \tau'', \sigma'', \gamma'', \varphi''') = \mathrm{Infer}(\theta''\theta'\theta\mathcal{E}, \varphi'', e'')$
    $\alpha, \varsigma, \delta$ **new**
    $\varphi_0 = \varphi''' \cup \{\tau' \preccurlyeq \alpha, \tau'' \preccurlyeq \alpha, (\sigma'' \cup \theta'''(\sigma' \cup \theta''\theta'\sigma)) \preccurlyeq \varsigma, ((\theta'''\theta''\theta'\gamma); (\theta'''\gamma' + \gamma'')) \preccurlyeq \delta\}$
  in $(\theta''' \circ \theta'' \circ \theta' \circ \theta, \alpha, \varsigma, \delta, \varphi_0)$
  end

Table 7. The Inference Algorithm

minimal approximation of communication effects that may be generated when the expression is evaluated. Similarly, the call graph produced by the algorithm corresponds to the minimal approximation of the call graph induced by the execution of the original expression. The constraint set produced by the algorithm corresponds to those constraints gathered during the inference process. The resolution of these constraints yields a substitution whose application to the type produced by the algorithm yields the principal type of the original expression.

In order to type a constant expression, our algorithm simply applies the function $TypeOf$ described in Table 14 (that allows the typing of built-in primitives) to that constant. Consequently, the original set of constraints is extended with those constraints reported in the basis of the type system. To type an identifier, the algorithm refers simply to a fresh instance of the type scheme reported in the static environment. When the identifier does not belong to the domain of the static environment, the inference procedure fails. The generation of the constraint set generated is similar to that of the constant case. In both the constant and identifier cases, it is obvious that neither effect nor call graph are reported.

The typing of the function abstraction proceeds as follows: first, a recursive call to the inference algorithm is made so as to type the body of the functional abstraction. The static environment used in this inference is the original one extended by the association that maps the argument of the function to a type variable and an empty constraint set. The type produced by this inference together with the substitution generated and the type variable used in the environment are combined to compute an arrow type for the whole function. The algorithm records the effect and call graph generated by the typing of the body of the function, as subtypes of respectively the latent effect and call graph of the function type. Since a function abstraction refers to a definition (no evaluation), an empty effect and an empty call graph are generated. The typing of an application expression is as follows: first, a recursive call to the inference algorithm is made to type the function expression. Second, a new call to the inference algorithm is made to type the argument expression (taking into account the substitution and the constraint set produced by the first call). Third, a call to the unification procedure is made so as to determine the type of the whole expression. For the conditional expression, the algorithm begins with the static evaluation of the first expression (condition) and then unifying the resulting type with *bool*. Then the algorithm proceeds by statically evaluating the "then" and the "else" branches respectively. Our algorithm should assure that both "then" and "else" branches have similar type structure but potential dissimilar effect and call graph annotations. This is achieved through the use of subtyping. The rest of the cases are straightforward and the corresponding intuitions can be easily reconstructed using the explanations above.

5.1 Unification

The inference algorithm uses a syntactic unification procedure modulo a set of type, effect and call graph constraints. The procedure takes as input two types and return, when succeeding, a substitution that stands for the most general

unifier of the two given types. The unification procedure also checks the well-formedness of the set of constraints. The procedure is defined in Table 8.

$$
\begin{array}{rl}
\mathcal{U}_\varphi(\tau,\tau') = & \textbf{case } (\tau,\tau') \textbf{ of} \\
(\alpha,\alpha') \Rightarrow & [\alpha \mapsto \alpha'] \\
(\alpha,\tau'') \mid (\tau'',\alpha) \Rightarrow & \textbf{if } \alpha \in fv(\overline{\varphi}\tau'') \textbf{ then } fail \textbf{ else } [\alpha \mapsto \tau''] \textbf{ end} \\
(chan_\varrho(\tau''), chan_{\varrho'}(\tau''')) \Rightarrow & \textbf{let } \theta = [\varrho \mapsto \varrho'] \textbf{ in } (\mathcal{U}_{\theta\varphi}(\theta\tau'', \theta\tau''')) \circ \theta \textbf{ end} \\
(\tau_i \xrightarrow{\varsigma,\delta} \tau_f, \tau_i' \xrightarrow{\varsigma',\delta'} \tau_f') \Rightarrow & \textbf{let } \theta_i = \mathcal{U}_\varphi(\tau_i,\tau_i') \qquad \theta_f = \mathcal{U}_{\theta_i\varphi}(\theta_i\tau_f, \theta_i\tau_f') \\
& \qquad \theta = [\theta_f(\theta_i\varsigma) \mapsto \theta_f(\theta_i\varsigma'), \theta_f(\theta_i\delta) \mapsto \theta_f(\theta_i\delta')] \circ \theta_f \circ \theta_i \\
& \textbf{in if } wf(\theta\varphi) \textbf{ then } \theta \textbf{ else } fail \textbf{ end end} \\
& \textbf{else if } t = t' \textbf{ then } Id \textbf{ else } fail \textbf{ end end}
\end{array}
$$

Table 8. Syntactic Unification Procedure

The following lemma establishes the soundness of the unification procedure.

Lemma 5.1. (Soundness of $\mathcal{U}$) *If φ is well-formed and $\mathcal{U}_\varphi(\tau,\tau') = \theta$ then $\theta\varphi$ is well-formed and $\theta\tau = \theta\tau'$.*

The following lemma establishes the completeness of the unification procedure.

Lemma 5.2. (Completeness of $\mathcal{U}$)) *Let φ be well-formed. Whenever $\theta'\tau = \theta'\tau'$ for a substitution θ' satisfying φ, then $\mathcal{U}_\varphi(\tau,\tau') = \theta$, $\theta'\varphi$ is well-formed and there exists a substitution θ'' satisfying $\theta\varphi$ such that $\theta' = \theta'' \circ \theta$.*

5.2 Generalization

The generalization function employed by the inference algorithm states that a variable cannot be generalized if it is free in the type environment $\mathcal{E}$ or if it is present in the inferred effect or call graph. The reader should refer to [14, 33, 4] for a detailed explanation of this issue. The same argument applies to call graphs. It is defined as follows:

$$
\begin{array}{rl}
Gen_\varphi(\mathcal{E},\sigma,\gamma)(\tau) = & \textbf{let } \{v_{1..n}\} = fv(\overline{\varphi}\tau) \backslash (fv(\overline{\varphi}\overline{\mathcal{E}}) \cup fv(\overline{\varphi}\sigma) \cup fv(\overline{\varphi}\gamma)) \\
& \textbf{in } (\forall v_{1..n}.(\tau, \varphi_{v_{1..n}}), \varphi \backslash \varphi_{v_{1..n}}) \textbf{ end}
\end{array}
$$

where:

$$
\begin{array}{rl}
\varphi_{v_{1..n}} = & \{\sigma \preccurlyeq \sigma' \in \varphi \mid \bigvee_{i=1}^n v_i \in (fv(\sigma) \cup fv(\sigma'))\} \cup \\
& \{\gamma \preccurlyeq \gamma' \in \varphi \mid \bigvee_{i=1}^n v_i \in (fv(\gamma) \cup fv(\gamma'))\} \cup \\
& \{\tau \preccurlyeq \tau' \in \varphi \mid \bigvee_{i=1}^n v_i \in (fv(\tau) \cup fv(\tau'))\}
\end{array}
$$

5.3 Constraint Resolution

When typing an expression, the constraints collected by the inference algorithm should be resolved in order to get a substitution. When the latter is applied to the corresponding produced type, effect an call graph, it will yield respectively, the principal type, the minimal effect and the minimal call graph of the original expression.

Actually, the constraint resolution is performed in 3 steps: splitting, flattening, and resolving. The explanations as well as the formalizations of these 3 phases are given hereafter.

Flattening: This step aims to simplify the constraints generated during the inference process. Such a simplification consists in removing trivial constraints and in decomposing complex ones into simpler constraints. This transformation is formally captured by the semantic function $\mathcal{F}$ defined in Table 9. The definition of $\mathcal{F}$ is inductive on to the structure of type expressions.

$$
\begin{array}{lll}
\mathcal{F}(\varphi) & = \cup\{\mathcal{F}(x \preccurlyeq y) | x \preccurlyeq y \in \varphi\} \\
\mathcal{F}(\sigma \preccurlyeq \sigma') & = \{\sigma \preccurlyeq \sigma'\} \\
\mathcal{F}(\gamma \preccurlyeq \gamma') & = \{\gamma \preccurlyeq \gamma'\} \\
\mathcal{F}(\tau \preccurlyeq \tau') & = \textbf{case } (\tau, \tau') \textbf{ of} \\
 & \qquad\qquad (\tau, \alpha) \Rightarrow \{\tau \preccurlyeq \alpha\} \\
 & \quad (chan_\rho(\tau), chan'_\rho(\tau')) \Rightarrow \{\rho \preccurlyeq \rho'\} \cup \mathcal{F}(\tau \preccurlyeq \tau') \\
 & \quad (\tau_i \xrightarrow{\sigma,\gamma} \tau_f, \tau'_i \xrightarrow{\sigma,\gamma} \tau'_f) \Rightarrow \mathcal{F}(\tau'_i \preccurlyeq \tau_i) \cup \mathcal{F}(\tau_f \preccurlyeq \tau'_f) \cup \{\sigma \preccurlyeq \sigma'\} \cup \{\gamma \preccurlyeq \gamma'\} \\
 & \quad \textbf{else if } \tau = \tau' \textbf{ then } \{\} \textbf{ else } fail \textbf{ end} \\
 & \quad \textbf{end}
\end{array}
$$

Table 9. Constraint Flattening

Splitting: This step aims to partition the collected constraint set into two separate sets: one that corresponds to type constraints and the other reports effect and call graph constraints. This is done by the function S described in Table 10.

Resolving: To resolve a constraint set, first, one has to simplify it thanks to the function $\mathcal{F}$ (flattening). Second, the simplified constraint set is partitioned into two different sets (splitting): one for type constraints, and the other one for effect and call graph constraints. Third, the effect and call graph constraint set is subjected to a resolution procedure and the resulting substitution is reported in the resolution of the type constraint set.

A resolved constraint set, written $\overline{\varphi}$, is defined as follows:

$$\overline{\varphi} = \textbf{let } \varphi' = \mathcal{F}(\varphi) \textbf{ in let } (\varphi_\tau, \varphi_{\sigma\gamma}) = \mathcal{S}(\varphi'),\ \theta = \widehat{\varphi}_{\sigma\gamma} \textbf{ in } (\widetilde{\theta\varphi_\tau}) \circ \theta \textbf{ end end}$$

The function that performs the resolution of effect and call graph constraints is written $\widehat{\varphi}$ and is presented in Table 11. The semantic function that performs the

$$
\begin{array}{l}
\mathcal{S}(\varphi) = \textbf{let } f \;=\; \lambda\varphi'.\lambda\varphi_\tau.\lambda\varphi_{\sigma\gamma}. \\
\qquad\qquad \textbf{case } \varphi' \textbf{ of} \\
\qquad\qquad\qquad\qquad \{\} \Rightarrow (\varphi_\tau, \varphi_{\sigma\gamma}) \\
\qquad\qquad\qquad \{\sigma \preccurlyeq \varsigma\} \cup \varphi'' \Rightarrow f(\varphi'')(\varphi_\tau)(\varphi_{\sigma\gamma} \cup \{\sigma \preccurlyeq \varsigma\}) \\
\qquad\qquad\qquad \{\gamma \preccurlyeq \delta\} \cup \varphi'' \Rightarrow f(\varphi'')(\varphi_\tau)(\varphi_{\sigma\gamma} \cup \{\gamma \preccurlyeq \delta\}) \\
\qquad\qquad\qquad \{\tau' \preccurlyeq \alpha\} \cup \varphi'' \Rightarrow f(\varphi'')(\varphi_\tau \cup \{\tau' \preccurlyeq \alpha\})(\varphi_{\sigma\gamma}) \\
\qquad\qquad \textbf{end} \\
\qquad\quad \textbf{in } f(\varphi)(\{\})(\{\}) \\
\qquad\quad \textbf{end}
\end{array}
$$

Table 10. Constraint Splitting

$$
\begin{array}{l}
\widehat{\varphi} \;=\; \textbf{case } \varphi \textbf{ of} \\
\qquad\qquad \{\} \Rightarrow Id \\
\{\sigma \preccurlyeq \varsigma\} \cup \varphi' \Rightarrow \textbf{let } \theta = \widehat{\varphi}' \\
\qquad\qquad\qquad \textbf{in if } \varsigma \mapsto \sigma' \in \theta \textbf{ then } \theta \dagger [\varsigma \mapsto \theta(\sigma \cup \sigma')] \textbf{ else } \theta \dagger [\varsigma \mapsto \theta\sigma] \textbf{ end} \\
\qquad\qquad\qquad \textbf{end} \\
\{\gamma \preccurlyeq \delta\} \cup \varphi' \Rightarrow \textbf{let } \theta = \widehat{\varphi}' \textbf{ in} \\
\qquad\qquad\qquad\quad \textbf{if } \delta \mapsto \gamma' \in \theta \textbf{ then} \\
\qquad\qquad\qquad\qquad \textbf{if } \delta \in fv(\theta\gamma + \theta\gamma') \textbf{ then } \theta \dagger [\delta \mapsto rec\, \delta.(\theta\gamma + \theta\gamma')] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \theta \dagger [\delta \mapsto (\theta\gamma + \theta\gamma')] \textbf{ end} \\
\qquad\qquad\qquad\quad \textbf{else} \\
\qquad\qquad\qquad\qquad \textbf{if } \delta \in fv(\theta\gamma) \textbf{ then } \theta \dagger [\delta \mapsto rec\, \delta.(\theta\gamma)] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \theta \dagger [\delta \mapsto (\theta\gamma)] \textbf{ end} \\
\qquad\qquad\qquad\quad \textbf{end} \\
\qquad\qquad\qquad \textbf{end} \\
\textbf{end}
\end{array}
$$

Table 11. Resolving Effect and Call Graph Constraints

resolution of type constraints is written $\widetilde{\varphi}$ and is defined as follows:

$$
\begin{array}{l}
\widetilde{\varphi} \;= \textbf{case } \varphi \textbf{ of} \\
\qquad\qquad \{\} \Rightarrow \mathrm{Id} \\
\{\tau \preccurlyeq \alpha\} \cup \varphi' \Rightarrow \textbf{let } \theta = \widetilde{\varphi}' \\
\qquad\qquad\qquad\qquad \textbf{in} \\
\qquad\qquad\qquad\qquad\quad \textbf{if } \alpha \mapsto \tau' \in \theta \\
\qquad\qquad\qquad\qquad\quad \textbf{then } \theta \dagger [\alpha \mapsto \pi_2(\bigsqcup(\theta\tau, \theta\tau'))] \\
\qquad\qquad\qquad\qquad\quad \textbf{else } \theta \dagger [\alpha \mapsto \theta\tau] \\
\qquad\qquad\qquad\qquad\quad \textbf{end} \\
\qquad\qquad\qquad\qquad \textbf{end} \\
\textbf{end}
\end{array}
$$

where $\pi_2(\bigsqcup(\theta\tau, \theta\tau'))$ stands for the type that is the second projection of the couple generated by the least upper bound function $\bigsqcup$. The Latter is presented in Table 12 and is employed by the resolution function of type constraints. It allows to combine two types that appear in a covariant position. Notice that the least upper bound of two function types is a function type annotated by the union of the effect annotations and by the sum of the call graph annotations. The function $\bigsqcap$ presented in Table 13 is employed by the resolution function of type constraints. It allows to combine two types that appear in a contravariant position. An effect (respectively a call graph) that annotates a function type that appears in a contravariant position, is necessarily an effect variable (respectively a call graph variable). The reason is that effects as well as call graphs are completely unknown since the function type is associated with a function that denotes a formal parameter in a higher-order expression. Consequently, the greatest lower bound will combine two function types (appearing in a contravariant position) by renaming annotation variables of one of the two function types into varibales of the other function type.

$$
\begin{array}{l}
\bigsqcup(\tau, \tau') = \textbf{case } (\tau, \tau') \textbf{ of} \\
\qquad\qquad\qquad\qquad (\alpha, \tau'') \Rightarrow ([\alpha \mapsto \tau''], \tau'') \\
\qquad\qquad\qquad\qquad (\tau'', \alpha) \Rightarrow ([\alpha \mapsto \tau''], \tau'') \\
\qquad\qquad (chan_{\varrho}(\tau_i), chan_{\varrho'}(\tau_i')) \Rightarrow \textbf{if } \bigsqcup(\tau_i, \tau_i') = fail \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } fail \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \theta = [\varrho \mapsto \varrho'] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\theta', \tau_f) = \bigsqcup(\theta\tau_i, \theta\tau_i') \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } (\theta' \circ \theta, chan_{\theta' \varrho}(\tau_f)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{end} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{end} \\
\qquad\qquad (\tau_i \xrightarrow{\sigma,\gamma} \tau_f, \tau_i' \xrightarrow{\sigma',\gamma'} \tau_f') \Rightarrow \textbf{if } (\bigsqcap(\tau_i, \tau_i') = fail) \textbf{ or } (\bigsqcup(\tau_f, \tau_f') = fail) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } fail \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\theta_1, \tau_1) = \bigsqcap(\tau_i, \tau_i') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\theta_2, \tau_2) = \bigsqcup(\theta_1\tau_f, \theta_1\tau_f') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \theta = \theta_2 \circ \theta_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } (\theta, \tau_1 \xrightarrow{\theta(\sigma \cup \sigma'), \theta(\gamma + \gamma')} \tau_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{end} \\
\qquad\qquad \textbf{else if} \tau = \tau' \textbf{ then } (Id, \tau) \textbf{else } fail \textbf{ end} \\
\qquad\qquad \textbf{end}
\end{array}
$$

Table 12. Least Upper Bounds on Types

$$
\begin{array}{rl}
\bigsqcap(\tau,\tau') = \textbf{case } (\tau,\tau') \textbf{ of} & \\
(\alpha,\tau'') \Rightarrow & ([\alpha \mapsto \tau''],\tau'') \\
(\tau'',\alpha) \Rightarrow & ([\alpha \mapsto \tau''],\tau'') \\
(chan_{\varrho}(\tau_i), chan_{\varrho'}(\tau_i')) \Rightarrow & \textbf{if } \bigsqcap(\tau_i,\tau_i') = fail \\
& \textbf{then } fail \\
& \textbf{else} \\
& \quad \textbf{let} \\
& \quad\quad \theta = [\varrho \mapsto \varrho'] \\
& \quad\quad (\theta',\tau_f) = \bigsqcap(\theta\tau_i, \theta\tau_i') \\
& \quad \textbf{in } (\theta' \circ \theta, chan_{\theta'\varrho}(\tau_f)) \\
& \quad \textbf{end} \\
& \textbf{end} \\
(\tau_i \xrightarrow{\varsigma,\delta} \tau_f, \tau_i' \xrightarrow{\varsigma',\delta'} \tau_f') \Rightarrow & \textbf{if } (\bigsqcup(\tau_i,\tau_i') = fail) \textbf{ or } (\bigsqcap(\tau_f,\tau_f') = fail) \\
& \textbf{then } fail \\
& \textbf{else} \\
& \quad \textbf{let} \\
& \quad\quad \theta = [\varsigma \mapsto \varsigma', \delta \mapsto \delta'] \\
& \quad\quad (\theta_1,\tau_1) = \bigsqcup(\theta\tau_i, \theta\tau_i') \\
& \quad\quad (\theta_2,\tau_2) = \bigsqcap((\theta_1 \circ \theta)\tau_f, (\theta_1 \circ \theta)\tau_f') \\
& \quad \textbf{in } (\theta_2 \circ \theta_1 \circ \theta, \tau_1 \xrightarrow{\varsigma',\delta'} \tau_2) \\
& \quad \textbf{end} \\
\textbf{else if} \tau = \tau' \textbf{ then } (Id,\tau) \textbf{else } fail \textbf{ end} & \\
\textbf{end} &
\end{array}
$$

Table 13. Greatest Lower Bounds on Types

6 Completeness and Soundness

In the following, the intention is to prove that our inference algorithm is sound and complete with respect to the typing rules.

We need the following properties of the static semantics that are used in the soundness and completeness proofs of the inference algorithm.

Lemma 6.1. (Substitution) *If* $\mathcal{E} \vdash e : \tau,\sigma,\gamma$ *then* $\theta\mathcal{E} \vdash e : \theta\tau,\theta\sigma,\theta\gamma$ *for any substitution* θ*.*

Lemma 6.2. (Strengthening) *If* $\mathcal{E} \vdash e : \tau,\sigma,\gamma$ *and if* $\tau' \lhd \mathcal{E}(x)$ *implies* $\tau' \lhd \mathcal{E}'(x)$ *for every* $x \in Dom(\mathcal{E})$ *then* $\mathcal{E}' \vdash e : \tau,\sigma,\gamma$*.*

Definition 6.3. (Model) Given a constraint set φ and a substitution θ, we say that θ is a model of φ and we write $\theta \models \varphi$, if and only if, $\forall t,t'$ such that $t \preccurlyeq t' \in \varphi$ we have $\theta t \preccurlyeq \theta t'$.

$$
\begin{array}{l}
TypeOf \;=\; [\\
\texttt{()} \mapsto (unit, \{\}), \\
\texttt{true} \mapsto (bool, \{\}), \\
\texttt{false} \mapsto (bool\{\}), \\
\texttt{channel} \mapsto (\forall \alpha, \varrho, \varsigma, \delta.\ unit \xrightarrow{\varsigma,\delta} chan_{\varrho}(\alpha), \{create(\varrho,\alpha) \preccurlyeq \varsigma, channel.nil \preccurlyeq \delta\}), \\
\texttt{receive} \mapsto (\forall \alpha, \varrho, \delta.\ chan_{\varrho}(\alpha) \xrightarrow{\emptyset,\delta} event_{in(\varrho,\alpha),nil}(\alpha), \{receive.nil \preccurlyeq \delta\}), \\
\texttt{transmit} \mapsto (\forall \alpha, \varrho, \delta. chan_{\varrho}(\alpha) \times \alpha \xrightarrow{\emptyset,\delta} event_{out(\varrho,\alpha),nil}(unit), \{transmit.nil \preccurlyeq \delta\}), \\
\texttt{choose} \mapsto \forall \alpha, \varsigma', \varsigma'', \delta', \delta'', \delta. \\
\qquad event_{\varsigma',\delta'}(\alpha) \times event_{\varsigma'',\delta''}(\alpha) \xrightarrow{\emptyset,\delta} event_{\varsigma' \cup \varsigma'', \delta' + \delta''}(\alpha), \{choose.nil \preccurlyeq \delta\}), \\
\texttt{spawn} \mapsto (\forall \varsigma, \delta, \delta'.\ (unit \xrightarrow{\varsigma,\delta'} unit) \xrightarrow{\emptyset,\delta} unit, \{spawn.fork(\delta').nil \preccurlyeq \delta\}), \\
\texttt{sync} \mapsto (\forall \alpha, \varsigma, \delta.\ event_{\varsigma,\delta'}(\alpha) \xrightarrow{\varsigma,\delta} \alpha, \{sync.\delta' \preccurlyeq \delta\}) \\
]
\end{array}
$$

Table 14. The Static Initial Basis, The $TypeOf$ Function

Since types are decorated by effect annotations, it is possible that an effect makes reference to a type annotated by that effect. As a consequence, some expressions may now have recursively defined types and effects, and shall thus be rejected by the static semantics. This phenomenon motivates the definition hereafter.

Definition 6.4. (Well-Formed Constraint Set) A constraint set φ is well formed, written $wf(\varphi)$, if and only if, for every $\sigma \preccurlyeq \varsigma$ such that $\varphi = \varphi' \cup \{\sigma \preccurlyeq \varsigma\}$ we have $\forall \mu(\rho, \tau) \in \overline{\varphi}'\sigma, \varsigma \notin fv(\tau)$ where $\mu \in \{chan, in, out\}$.

The notation $wf(\varphi)$ is extended to type schemes by $wf(\forall v_1 \ldots v_n.(\tau, \varphi))$ iff $wf(\varphi)$ and to type environments by $wf(\mathcal{E})$ iff $wf(\mathcal{E}(x))$ for every x in $dom(\mathcal{E})$.

The following lemmas state that well-formed constraint sets are solvable by finite substitutions.

Lemma 6.5. (Model Preservation) *Let φ be a set of type, effect and call graph constraints. If φ is well-formed then $\theta \models \varphi$ if and only if $\theta \models \mathcal{F}(\varphi)$.*

Lemma 6.6. (Least Upper Bound) *Let τ and τ' be two types. If $\bigsqcup(\tau, \tau')$ does not fail, then $\bigsqcup(\tau, \tau')$ is the least upper bound of τ and τ' with respect to the preorder on types.*

Lemma 6.7. (Greatest Lower Bound) *Let τ and τ' be two types. If $\bigsqcap(\tau, \tau')$ does not fail, then $\bigsqcap(\tau, \tau')$ is the greatest lower bound of τ and τ' with respect to the preorder on types.*

Lemma 6.8. (Constraint Resolution) *Let φ be a set of type, effect and call graph constraints. If φ is well-formed and if $\overline{\varphi}$ does not fail, then $\overline{\varphi} \models \varphi$.*

Now we return to our main goal, that is to prove the correctness of our inference algorithm with respect to the static semantics. The soundness theorem states that the inferred type, effect and call graph are provable in the static semantics, assuming the solution of the collected constraints.

Theorem 6.9 (Soundness) *If* $\mathrm{Infer}(\mathcal{E}, \varphi, e) = (\theta, \tau, \sigma, \gamma, \varphi')$ *then* $\overline{\varphi}'(\theta\overline{\mathcal{E}}) \vdash e : \overline{\varphi}'\tau, \overline{\varphi}'\sigma, \overline{\varphi}'\gamma$.

The completeness theorem states that the inferred type is principal with respect to the substitution on variables and that the reconstructed effect and call graph are minimal with respect to the subsumption on respectively effects and call graphs.

Theorem 6.10 (Completeness) *Let $\mathcal{E}$ and φ be well-formed and $\theta'' \models \varphi$. If $\theta''\overline{\mathcal{E}} \vdash e : \tau', \sigma', \gamma'$ then* $\mathrm{Infer}(\mathcal{E}, \varphi, e) = (\theta, \tau, \sigma, \gamma, \varphi')$ *and there exists $\theta' \models \varphi'$ such that $\theta''\overline{\mathcal{E}} = \theta'(\theta\overline{\mathcal{E}}), \tau' = \theta'\tau$, $\theta'\sigma \preccurlyeq \sigma'$ and $\theta'\gamma \preccurlyeq \gamma'$.*

7 Conclusion

In this paper, we reported a type-based framework, that resolves the problem of control-flow analysis for concurrent and functional languages. We presented an efficient algorithm that propagates automatically types, communication effects and call graphs. The algorithm comes with a logical characterization that consists of a type proof system. The latter operates on a Concurrent ML core-syntax: a strongly typed, polymorphic kernel that supports higher-order functions and concurrency primitives. Effects are represented as algebraic terms that record communication effects resulting from channel creation, sending and receiving. Call graphs record function calls and are captured by a term algebra that is close to usual process algebras. Types are annotated with effects and call graphs. For the sake of flexibility, a subtyping relation is considered on the type algebra. We present the language syntax together with its static semantics that consists of the typing rules and an inference algorithm. The latter is proved to be consistent and complete with respect to the typing rules. We have made an implementation of this algorithm using C, LeX and Yacc on a CML interpreter. Actually, the program is about 8000 lines of code and shows very good performance.

As a future research, we plan to extend this framework in order to capture a richer and a more precise information. In addition, we project to develop a framework that unifies our control-flow analysis technique with the one based on abstract interpretation.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* The Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Company, London, 1974.

2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986.
3. D. Berry, A.J.R.G. Milner, and D. Turner. A semantics for ML concurrency primitives. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, 1992.
4. D. Bolignano and M. Debbabi. A coherent type inference system for a concurrent, functional and imperative programming language. In *Proceedings of the AMAST'93 Conference*. Springer Verlag, June 1993.
5. D. Bolignano and M. Debbabi. A denotational model for the integration of concurrent functional and imperative programming. In *Proceedings of the ICCI'93 Conference*. IEEE, May 1993.
6. D. Bolignano and M. Debbabi. A semantic theory for ML higher order concurrency primitives. In *Proceedings of the NAPAW'93 Workshop*. Cornell University, August 1993.
7. D. Bolignano and M. Debbabi. Vers une semantique des langages parallèles, fonctionnels avec des extensions impératives. In *Actes des Journées Francophones des Langages Applicatifs, JFLA'93*. INRIA, February 1993.
8. D. Bolignano and M. Debbabi. A semantic theory for CML. In *Proceedings of the TACS'94 Conference*. Springer Verlag, April 1994.
9. Dominique Boucher and Marc Feeley. Abstract compilation: a new implementation paradigm for static analysis. In *Proceedings of the 1996 Conference on Compiler Construction*, Linkoping, Sude, april 1996.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, New York, NY, 1977. ACM.
11. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 269–282, New York, NY, 1979. ACM.
12. P. Cousot and R. Cousot. Inductive definitions, semantics, and abstract interpretation. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, NM, January 1992.
13. L. M. M. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages, Albuquerque*, pages 207–212, New York, NY, 1982. ACM.
14. M. Debbabi. *Intégration des paradigmes de programmation parallèle, fonctionnelle et impérative : fondements sémantiques.* Université Paris Sud, Centre d'Orsay, July 1994. Thèse de Doctorat.
15. D.K. Gifford, P. Jouvelot, J.M. Lucassen, and M.A. Sheldon. Fx-87 reference manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, September 1987.
16. Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the 1995 ACM Symposium on Principles of Programming Languages*, January 1995.
17. P. Jouvelot and D.K. Gifford. Communication effects for message-based concurrency. Technical Report MIT/LCS/TM-386, MIT Laboratory for Computer Science, February 1989.
18. X. Leroy. *Typage polymorphe d'un langage algorithmique.* PhD thesis, Université de Paris VII, June 1992.

19. J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988.
20. A.J.R.G. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
21. A.J.R.G. Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science 92*, pages 281–305. Springer-Verlag, 1991.
22. F. Nielson and H.R. Nielson. From CML to process algebras. Technical Report DAIMI PB - 433, Computer science department, Aarhus university, March 1993.
23. Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 84–97, Portland, Oregon, January 17–21, 1994. ACM Press. Extended abstract.
24. J.H. Reppy. First-class synchronous operations in standard ML. Technical Report TR89-1068, Dept. of Computer Science, Cornell University, 1989.
25. J.H Reppy. Concurrent programming with events - the Concurrent ML manual. Technical report, Department of Computer Science, Cornell University, November 1990.
26. J.H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 PLDI*, pages 294–305. SIGPLAN Notices 26(6), 1991.
27. J.H Reppy. An operational semantics of first-class synchronous operations. Technical Report TR 91-1232, Department of Computer Science, Cornell University, August 1991.
28. O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 164–174, Atlanta, GA, June 1988.
29. O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. also published as CMU Technical Report CMU-CS-91-145.
30. O. Shivers. Data-flow analysis and type recovery in scheme. Report CMU-CS-90-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1990.
31. O. Shivers. The semantics of scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, pages 190–198, New Haven, CN, June 1991.
32. Dan Stefanescu and Yuli Zhou. An equational framework for the flow analysis of higher-order functional programs. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 318–327, 1994.
33. J. Talpin and P. Jouvelot. The type and effect discipline. In *Proc. Logic in Computer Science*, 1992.
34. Y. M. Tang. *Systèmes d'Effet et Interprétation Abstraite pour l'Analyse de Flot de Contrôle*. PhD thesis, Université de Paris VI, March 1994.
35. Y. M. Tang and P. Jouvelot. Separate abstract interpretation for control-flow analysis. In *Proceedings of the TACS'94 Conference*. Springer Verlag, April 1994.
36. B. Thomsen. Polymorphic sorts and types for concurrent functional programs. Technical Report Draft, ECRC, Munich, March 1993.

This article was processed using the LaTeX macro package with LLNCS style