CLL: A Concurrent Language Built from Logical Principles

Deepak Garg

January, 2005
CMU-CS-05-104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

e-mail: dg+@cs.cmu.edu

This work has been partially supported by NSF Grant CCR-0306313 Efficient Logical Frameworks.

Keywords: Concurrent language, propositions as types, linear logic, logic programming, monad

Abstract

We present CLL, a concurrent programming language that symmetrically integrates functional and concurrent logic programming. First, a core functional language is obtained from a proof-term assignment to a variant of intuitionistic linear logic, called FOMLL, via the Curry-Howard isomorphism. Next, we introduce a Chemical Abstract Machine (CHAM) whose molecules are typed terms of this functional language. Rewrite rules for this CHAM are derived by augmenting proof-search rules for FOMLL with proof-terms. We show that this CHAM is a powerful concurrent language and that the linear connectives $\otimes$, $\exists$, $\oplus$, $\multimap$ and $\&$ correspond to process-calculi connectives for parallel composition, name restriction, internal choice, input prefixing and external choice respectively. We also demonstrate that communication and synchronization between CHAM terms can be performed through proof-search on the types of terms. Finally, we embed this CHAM as a construct in our functional language to allow interleaving functional and concurrent computation in CLL.

Contents

1 Introduction

There are several ways to design a typed concurrent programming language. We may start from a syntax and operational semantics for the terms of the language and add types in order to guarantee certain properties of typed terms. Such properties include but are not limited to type-safety, deadlock freedom and several security properties. Examples of such languages are typed variants of the π-calculus [20, 21, 22], join-calculus [16], CML [31] and Concurrent Haskell [28]. A completely different approach is to begin from a logic and lift it to a type system for a programming language using the Curry-Howard isomorphism. Proof-terms that are witnesses for proofs in the logic become the terms of the programming language and proof normalization corresponds to the operational semantics. This approach has been successfully applied to the design of functional programming languages. When we come to the concurrent paradigm where we allow creation of processes executing in parallel and communicating with each other through one of several mechanisms like shared memory, message queues or synchronization constructs like semaphores, monitors and events, attempts to design languages using the Curry-Howard isomorphism have mostly been theoretical. Most work [1, 2] in this direction is restricted to classical linear logic [18] and away from practice.[1]

A completely different meeting point for logic and concurrent programming is concurrent logic programming [34]. In this approach, one uses parallelism inherent in proof-search to design a logic programming language which simulates concurrent process behavior. As is usual with all logic programming, only predicates and logical propositions play a part in programming and proof-terms are not used. Examples of languages of this kind are Concurrent Prolog[33] and FCP[23].

In this report, we use both the Curry-Howard isomorphism and proof-search to design a concurrent programming language from logical principles. We call this language CLL (Concurrent Linear Language). Our underlying logic is a first-order intuitionistic linear logic where all right synchronous connectives $(\otimes, \oplus, 1, \exists)$ are restricted to a monad. We refer to this logic as FOMLL (First-Order Monadic Linear Logic). Using *linear* logic to build the type system for a concurrent language seems a natural choice since processes are linear entities. Ever since Girard's first work on linear logic [18], deep connections between linear logic and concurrency have been suggested. For example, Abramsky develops a concurrent computational interpretation of classical linear logic in [1]. FOMLL differs from the logic used by Abramsky in two essential ways. First, it is intuitionistic. Second, it is equipped with a monad. We use a monad in FOMLL because concurrent computations have *effects* like deadlocks and the monad separates pure functional terms from effectful concurrent computations, enabling us to prove a type-safety theorem. This use of monads goes back to Moggi's work [26] and similar uses of monads in concurrent languages like CML, Concurrent Haskell and Facile [31, 28, 17]. FOMLL has also been used in the Concurrent Logical Framework [35] which has been used to represent several concurrent languages [12].

We design CLL in three steps. First, we construct a purely functional language (called fCLL for *functional* CLL) by adding proof-terms to FOMLL. fCLL admits basic linear functional constructs like abstraction, linear pairing, disjunctions, replication and recursive types, recursion and first-order dependent types. It also allows parallelism - parts of programs may be evaluated in parallel. However, there is no primitive for communication between parallel processes. In the second step, we embed fCLL in a concurrent logic programming language called lCLL (*logic programming* CLL). The semantics of this logic programming

[1] Abramsky's work [1] mentions some computational interpretations of *intuitionistic* linear logic also. However, these are sequential, not concurrent, interpretations and are not of much interest in the context of this report.

language are presented as a Chemical Abstract Machine (CHAM) [4, 5, 7]. Molecules in *l*CLL CHAM configurations are terms of *f*CLL annotated with their types. Rewrite rules for these CHAM configurations are derived from proof-search rules for FOMLL. *l*CLL differs from other logic programming languages in two respects. First, we use the forward style of proof-search, not the traditional backward style. Second, proof-terms obtained during proof-search play a computational role in *l*CLL, which is not the case with other logic programming languages. *l*CLL is a powerful concurrent language that can encode all basic concurrency constructs like input and output processes, parallel composition for processes, choices, communication and even n-way synchronization. In the third step, we embed *l*CLL back in *f*CLL as a language construct. This makes functional and concurrent logic programming symmetric in the language. Since *l*CLL configurations produce side effects like deadlocks, we restrict all CHAM configurations to the monad in *f*CLL. The resultant language is called *full*-CLL. We sometimes drop the prefix 'full' if it is clear from context.

An implementation of full-CLL in the Concurrent Logical Framework is available from the author's homepage at `http://www.cs.cmu.edu/~dg`.

The contributions of this work are as follows. First, we show that proof-search in logic has an interesting computational interpretation - it can be viewed as a procedure to link together programs to form larger programs, which can then be executed. Working with FOMLL, we also show how proof-search can be exploited to add concurrency constructs to a programming language. Second, we demonstrate how functional and logic programming can be symmetrically integrated in a single framework that allows interleaving functional computation and proof-search. Third, we establish that functional and concurrent programming can be integrated symmetrically in a typed setting. In particular, we describe a method that allows concurrent computations inside functional programs to return non-trivial results, which can be used for further functional evaluation. Finally, we show that there is a correspondence between various concurrent constructs like parallel composition, name restriction, choices etc. and connectives of linear logic like $\otimes$, $\exists$ and $\&$.

Organization of the report. In section 2 we present the syntax, types and semantics of *f*CLL. We prove a type-safety result for this language and illustrate the expressiveness of our parallel construct with some examples. In section 3 we build the concurrent logic programming *l*CLL and prove a type-preservation result for it. *l*CLL is integrated with *f*CLL as a monadic construct to obtain full-CLL in section 4. We prove a type-safety theorem for the whole language. A number of examples to illustrate the constructs in full-CLL are presented in section 5. Section 6 discusses related work and concludes the report.

2 *f*CLL: Functional Programming in CLL

Syntax. As mentioned in the introduction, *f*CLL is the functional core of CLL. It is designed from an underlying logic (FOMLL), which under the Curry-Howard isomorphism corresponds to the type system. Hence the syntax of types is presented first. We assume a number of *sorts*, which are finite or infinite sets of index refinements (index terms are denoted by t). Index variables are denoted by i. See [36] for a detailed description of index refinements. Sort names are denoted by γ and its variants. Atomic type constructors denoted by P and its decorated variants have *kinds* which are given by the grammar:

$$\begin{array}{rcl} K & ::= & \texttt{Type} \\ & | & \gamma \to K \end{array}$$

We assume the existence of at least one infinite sort, namely the sort of channel names. This sort is called chan. Channels are denoted by the letter k and its decorated variants. We assume the existence of some implicit signature that gives the kinds of all atomic type constructors.

Types in CLL are derived from a variant of first-order intuitionistic linear logic [35, 13, 19] called FOMLL. We classify types into two categories based on the top level type constructor. If the top level constructor is atomic, &, $\to$, $\multimap$ or $\forall$, we call the type *asynchronous* following Andreoli[3]. In a sequent style presentation of linear logic, the right rules for asynchronous constructors are invertible, whereas their left rules are not. If the top constructor is !, $\otimes$, 1, $\oplus$, $\exists$ or μ, we call the type *synchronous*. In sharp contrast to asynchronous connectives, right rules for synchronous connectives are not invertible, whereas their left rules are. All synchronous types are restricted to a monad, whose constructor is denoted by $\{\ldots\}$. Types are generated by the following grammar:

A, B	$::=$		(Asynchronous types)
		$P\ t_1 \ldots t_n$	(Atomic types)
	$\mid$	$A \& B$	(With or additive conjunction)
	$\mid$	$A \to B$	(Unrestricted implication)
	$\mid$	$A \multimap B$	(Linear implication)
	$\mid$	$\{S\}$	(Monadic type)
	$\mid$	$\forall i : \gamma.A$	(Universal quantification)
S	$::=$		(Synchronous or monadic types)
		A	(Base synchronous types)
	$\mid$	$S_1 \otimes S_2$	(Tensor or multiplicative conjunction)
	$\mid$	1	(Unit of tensor)
	$\mid$	$S_1 \oplus S_2$	(Additive disjunction)
	$\mid$	$!A$	(Replication or exponential)
	$\mid$	$\mu\alpha.S$	(Iso-recursive type)
	$\mid$	$\exists i : \gamma.S$	(Existential quantification)
	$\mid$	α	(Recursive type variable)

For proof-terms, we distinguish three classes of terms. "Pure" terms, sometimes simply called terms, denoted by N, represent proofs of asynchronous types. Proofs of synchronous types are represented by two classes of syntax: monadic terms, denoted by M and expressions denoted by E. In general, monadic terms are constructive; they correspond to introduction rules of synchronous connectives. Expressions correspond to elimination terms and are the site of all parallelism in fCLL, as discussed later. The whole monad is presented in a judgmental style [29]. The syntax of terms and expressions in the language is given below. We assume the existence of three disjoint and infinite sets of variables - term variables denoted by $x, y, \ldots$, recursion variables denoted by $u, v, \ldots$ and "choice" variables denoted by $\zeta, \varsigma, \ldots$

Terms, N	$::=$	$x \mid \langle N_1, N_2 \rangle \mid \pi_1 N \mid \pi_2 N \mid \lambda x.N \mid \hat{\lambda} x.N \mid N_1\ N_2 \mid N_1 \mathbin{\hat{}} N_2 \mid \{E\}$
		$\mid \Lambda i : \gamma.N \mid N\ [t]$
Monadic terms, M	$::=$	$N \mid M_1 \otimes M_2 \mid 1 \mid \underline{\mathtt{inl}}\ M \mid \underline{\mathtt{inr}}\ M \mid !N$
		$\mid \underline{\mathtt{fold}}(M) \mid u \mid \mu u.M \mid [t, M] \mid M_1 \vert_\zeta M_2$
Expressions, E	$::=$	$M \mid \underline{\mathtt{let}}\ \{p\} = N\ \underline{\mathtt{in}}\ E \mid E_1 \vert_\zeta E_2$
patterns, p	$::=$	$x \mid 1 \mid p_1 \otimes p_2 \mid p_1 \vert_\zeta p_2 \mid !x \mid [i, p] \mid \underline{\mathtt{fold}}(p)$

For elimination of the synchronous connectives, $\otimes, \oplus, 1, \exists$ and μ, we use let constructions similar to [10]. As opposed to usual elimination rules, which correspond to natural deduction style eliminations, the use of lets gives rise to rules corresponding to left sequent rules of the sequent calculus. Choice variables $\{\zeta, \varsigma, \ldots\}$ are used to distinguish case branches for eliminating the connective $\oplus$. For a detailed description of this treatment see [10]. For clarity, we sometimes annotate bound variables and <u>`fold`</u> constructs with their types.

Type System. We use four contexts in our typing judgments: Σ (index variable context), Γ (unrestricted context), Δ (linear context) and Ψ (recursion context). The grammars generating these contexts are:

$$
\begin{array}{lcl}
\Sigma & ::= & \cdot \mid \Sigma,\ i : \gamma \\
\Gamma & ::= & \cdot \mid \Gamma,\ x : A \\
\Delta & ::= & \cdot \mid \Delta,\ p : S \quad \text{if}\ p \Rightarrow S \\
\Psi & ::= & \cdot \mid \Psi,\ u : S
\end{array}
$$

The judgment $p \Rightarrow S$, read as "p matches S" is described in figure 1. Subsequently, it is assumed that whenever $p : S$ occurs in a context, $p \Rightarrow S$. Given a context, the variables it defines are called its defined variables, `dv`. Related concepts are defined *linear* variables, `dlv` and defined *index* variables, `div`. These are precisely described in figure 2. Given a context $\Sigma; \Gamma; \Delta; \Psi$, we assume that the sets $\mathtt{dv}(\Gamma)$, $\mathtt{dv}(\Delta)$, $\mathtt{dv}(\Psi)$, $\mathtt{div}(\Sigma)$ and $\mathtt{div}(\Delta)$ are all pairwise disjoint. We use four typing judgments in our type system:

$$\Sigma; \Gamma; \Delta; \Psi \ \vdash\ N : A$$

$$\Sigma; \Gamma; \Delta; \Psi \ \vdash\ M \# S$$

$$\Sigma; \Gamma; \Delta; \Psi \ \vdash\ E \div S$$

$$\Sigma \ \vdash\ t : \gamma$$

The last judgment is external to the language and we do not specify how we check the well-sortedness of refinement terms. We simply assume the following properties of this judgment:

1. Substitution: If $\Sigma \vdash t : \gamma$ and $\Sigma, i : \gamma \vdash t' : \gamma'$, then $\Sigma \vdash t'\,[t/i] : \gamma'$.

2. Weakening: If $\Sigma \vdash t : \gamma$, then $\Sigma, i : \gamma \vdash t : \gamma$.

3. Strengthening: If $\Sigma, i : \gamma \vdash t : \gamma$ and $i \notin t$, then $\Sigma \vdash t : \gamma$.

The other three typing judgments assume that all types in Γ, Δ and Ψ are well-formed with respect to the refinement term context Σ. The type $P\ t_1 \ldots t_n$ is well formed in Σ if $\mathtt{Kind}(P) = \gamma_1 \ldots \gamma_n \to \mathtt{Type}$ and $\Sigma \vdash t_i : \gamma_i$ for $i = 1 \ldots n$. The well formedness of other types is obtained by lifting this relation in the standard way. The typing rules for *f*CLL are given in figures 3, 4, 5 and 6. It may be observed here that there is no $\oplus - L$ rule for terms similar to the rules $\oplus - L_M$ and $\oplus - L_E$ (see figure 6) because we do not allow choice branches in pure terms. This is done because we found that in practice choice branches in pure terms are never needed.

Operational Semantics. We use call-by-value semantics for *f*CLL. However, certain constructs have to be evaluated lazily due to linearity constraints and the presence of a monad. For example, pairs at the level of terms have to be lazy because the two components of a pair share the same linear resources and only the component that will be used in the remaining computation should be evaluated. Thus evaluation of the components of a pair is postponed till one of the components is projected. The monad is also a lazy construct because it encloses expressions, whose evaluation can have side effects. We do not evaluate the body of a functional abstraction ($\lambda x.N$, $\hat{\lambda} x.N$ and $\Lambda i.N$), since evaluation is restricted to *closed* terms,

$$\frac{}{x \Rightarrow A} \qquad \frac{}{!x \Rightarrow !A}$$

$$\frac{}{1 \Rightarrow 1} \qquad \frac{p \Rightarrow S}{[i_1, p] \Rightarrow \exists i_2 : \gamma.S}$$

$$\frac{p_1 \Rightarrow S_1 \qquad p_2 \Rightarrow S_2}{p_1 \otimes p_2 \Rightarrow S_1 \otimes S_2} \qquad \frac{p \Rightarrow S(\mu\alpha.S(\alpha))}{\underline{\texttt{fold}}(p) \Rightarrow \mu\alpha.S(\alpha)}$$

$$\frac{p_1 \Rightarrow S_1 \qquad p_2 \Rightarrow S_2}{p_1|_\zeta p_2 \Rightarrow S_1 \oplus S_2}$$

Figure 1: $p \Rightarrow S$

$\texttt{dv}(x) = \{x\}$	$\texttt{dv}(!x) = \{x\}$
$\texttt{dv}(p_1 \otimes p_2) = \texttt{dv}(p_1) \cup \texttt{dv}(p_2)$	$\texttt{dv}(1) = \phi$
$\texttt{dv}(p_1|_\zeta p_2) = \{\zeta\} \cup \texttt{dv}(p_1) \cup \texttt{dv}(p_2)$	$\texttt{dv}([i, p]) = \texttt{dv}(p)$
$\texttt{dv}(\underline{\texttt{fold}}(p)) = \texttt{dv}(p)$	
$\texttt{dv}(\cdot) = \phi$	$\texttt{dv}(\Gamma, x : A) = \texttt{dv}(\Gamma) \cup \{x\}$
$\texttt{dv}(\Delta, p : S) = \texttt{dv}(\Delta) \cup \texttt{dv}(p)$	$\texttt{dv}(\Psi, u : S) = \texttt{dv}(\Psi) \cup \{u\}$
$\texttt{dv}(\Sigma) = \phi$	
$\texttt{dlv}(x) = \{x\}$	$\texttt{dlv}(!x) = \phi$
$\texttt{dlv}(p_1 \otimes p_2) = \texttt{dlv}(p_1) \cup \texttt{dlv}(p_2)$	$\texttt{dlv}(1) = \phi$
$\texttt{dlv}(p_1|_\zeta p_2) = \texttt{dlv}(p_1) \cup \texttt{dlv}(p_2)$	$\texttt{dlv}([i, p]) = \texttt{dlv}(p)$
$\texttt{dlv}(\underline{\texttt{fold}}(p)) = \texttt{dlv}(p)$	
$\texttt{dlv}(\cdot) = \phi$	$\texttt{dlv}(\Delta, p : S) = \texttt{dlv}(\Delta) \cup \texttt{dlv}(p)$
$\texttt{dlv}(\Gamma) = \phi$	$\texttt{dlv}(\Psi) = \phi$
$\texttt{dlv}(\Sigma) = \phi$	
$\texttt{div}(x) = \phi$	$\texttt{div}(!x) = \phi$
$\texttt{div}(p_1 \otimes p_2) = \texttt{div}(p_1) \cup \texttt{div}(p_2)$	$\texttt{div}(1) = \phi$
$\texttt{div}(p_1|_\zeta p_2) = \texttt{div}(p_1) \cup \texttt{div}(p_2)$	$\texttt{div}([i, p]) = \{i\}$
$\texttt{div}(\underline{\texttt{fold}}(p)) = \texttt{div}(p)$	
$\texttt{div}(\cdot) = \phi$	$\texttt{div}(\Delta, p : S) = \texttt{div}(\Delta) \cup \texttt{div}(p)$
$\texttt{div}(\Gamma) = \phi$	$\texttt{div}(\Psi) = \phi$
$\texttt{div}(\Psi) = \phi$	$\texttt{div}(\Sigma) = dom(\Sigma)$

Figure 2: Defined variables of patterns and contexts

$$\dfrac{}{\Sigma;\Gamma;x:A;\Psi\ \vdash\ x:A}\ \text{Hyp1} \qquad \dfrac{}{\Sigma;\Gamma,x:A;\cdot;\Psi\ \vdash\ x:A}\ \text{Hyp2}$$

$$\dfrac{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ N_1:A_1 \qquad \Sigma;\Gamma;\Delta;\Psi\ \vdash\ N_2:A_2}{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ \langle N_1,N_2\rangle : A_1\&A_2}\ \&\text{-I}$$

$$\dfrac{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ N:A_1\&A_2}{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ \pi_1 N:A_1}\ \&\text{-}E_1 \qquad \dfrac{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ N:A_1\&A_2}{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ \pi_2 N:A_2}\ \&\text{-}E_2$$

$$\dfrac{\Sigma;\Gamma,x:A;\Delta;\Psi\ \vdash\ N:B}{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ \lambda x.N:A\rightarrow B}\ \rightarrow\text{-I} \qquad \dfrac{\Sigma;\Gamma;\Delta,x:A;\Psi\ \vdash\ N:B}{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ \hat{\lambda} x.N:A\multimap B}\ \multimap\text{-I}$$

$$\dfrac{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ N_1:A\rightarrow B \qquad \Sigma;\Gamma;\cdot;\Psi\ \vdash\ N_2:A}{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ N_1\ N_2:B}\ \rightarrow\text{-E}$$

$$\dfrac{\Sigma;\Gamma;\Delta_1;\Psi\ \vdash\ N_1:A\multimap B \qquad \Sigma;\Gamma;\Delta_2;\Psi\ \vdash\ N_2:A}{\Sigma;\Gamma;\Delta_1,\Delta_2;\Psi\ \vdash\ N_1\,\hat{}\,N_2:B}\ \multimap\text{-E}$$

$$\dfrac{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ E\ \div\ S}{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ \{E\}:\{S\}}\ \{\}\text{-I}$$

$$\dfrac{\Sigma,i:\gamma;\Gamma;\Delta;\Psi\ \vdash\ N:A}{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ \Lambda i:\gamma.N:\forall i:\gamma.A}\ \forall\text{-I}$$

$$\dfrac{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ N:\forall i:\gamma.A(i) \qquad \Sigma\ \vdash\ t:\gamma}{\Sigma;\Gamma;\Delta;\Psi\ \vdash\ N\ [t]:A(t)}\ \forall\text{-E}$$

Figure 3: Type system for Terms

$$\frac{\Sigma;\Gamma;\Delta;\Psi \vdash N : A}{\Sigma;\Gamma;\Delta;\Psi \vdash N \# A}\ {:}\#$$

$$\frac{}{\Sigma;\Gamma;\cdot;\Psi, u : S \vdash u \# S}\ \text{Hyp3} \qquad \frac{}{\Sigma;\Gamma;\cdot;\Psi \vdash 1 \# 1}\ \text{1-R}$$

$$\frac{\Sigma;\Gamma;\Delta_1;\Psi \vdash M_1 \# S_1 \qquad \Sigma;\Gamma;\Delta_2;\Psi \vdash M_2 \# S_2}{\Sigma;\Gamma;\Delta_1,\Delta_2;\Psi \vdash M_1 \otimes M_2 \# S_1 \otimes S_2}\ \otimes\text{-R}$$

$$\frac{\Sigma;\Gamma;\Delta;\Psi \vdash M \# S_1}{\Sigma;\Gamma;\Delta;\Psi \vdash \underline{\mathtt{inl}}\, M \# S_1 \oplus S_2}\ \oplus\text{-}R_1 \qquad \frac{\Sigma;\Gamma;\Delta;\Psi \vdash M \# S_2}{\Sigma;\Gamma;\Delta;\Psi \vdash \underline{\mathtt{inr}}\, M \# S_1 \oplus S_2}\ \oplus\text{-}R_2$$

$$\frac{\Sigma;\Gamma;\cdot;\Psi \vdash N : A}{\Sigma;\Gamma;\cdot;\Psi \vdash\ !N \#\ !A}\ \text{!-R}$$

$$\frac{S = \mu\alpha.S'(\alpha) \qquad \Sigma;\Gamma;\Delta;\Psi \vdash M \# S'(S)}{\Sigma;\Gamma;\Delta;\Psi \vdash \underline{\mathtt{fold}}(M) \# S}\ \underline{\mathtt{fold}}\text{-R}$$

$$\frac{\Sigma;\Gamma;\cdot;\Psi, u : S \vdash M \# S}{\Sigma;\Gamma;\cdot;\Psi \vdash \mu u.M \# S}\ \text{rec}$$

$$\frac{\Sigma;\Gamma;\Delta;\Psi \vdash M \# S(t) \qquad \Sigma \vdash t : \gamma}{\Sigma;\Gamma;\Delta;\Psi \vdash [t, M] \# \exists i : \gamma.S(i)}\ \exists\text{-R}$$

Figure 4: Type system for Monadic Terms

$$\frac{\Sigma;\Gamma;\Delta;\Psi \vdash M \# S}{\Sigma;\Gamma;\Delta;\Psi \vdash M \div S}\ \# \div$$

$$\frac{\Sigma;\Gamma;\Delta_1;\Psi \vdash N : \{S\} \qquad \Sigma;\Gamma;\Delta_2, p : S;\Psi \vdash E \div S'}{\Sigma;\Gamma;\Delta_1,\Delta_2;\Psi \vdash \underline{\mathtt{let}}\ \{p\} = N\ \underline{\mathtt{in}}\ E \div S'}\ \{\}\text{-E}$$

Figure 5: Type system for Expressions

$$T \,\%\, Z ::= N : A \mid M \# S \mid E \div S$$

$$\frac{\Sigma; \Gamma, x : A; \Delta; \Psi \vdash T \,\%\, Z}{\Sigma; \Gamma; \Delta, !x :!A; \Psi \vdash T \,\%\, Z}\ \text{!-L} \qquad \frac{\Sigma; \Gamma; \Delta, p_1 : S_1, p_2 : S_2; \Psi \vdash T \,\%\, Z}{\Sigma; \Gamma; \Delta, p_1 \otimes p_2 : S_1 \otimes S_2; \Psi \vdash T \,\%\, Z}\ \otimes\text{-L}$$

$$\frac{\Sigma; \Gamma; \Delta; \Psi \vdash T \,\%\, Z}{\Sigma; \Gamma; \Delta, 1 : 1; \Psi \vdash T \,\%\, Z}\ \text{1-L} \qquad \frac{\Sigma; \Gamma; \Delta, p : S(\mu\alpha.S(\alpha)); \Psi \vdash T \,\%\, Z}{\Sigma; \Gamma; \Delta, \underline{\texttt{fold}}(p) : \mu\alpha.S(\alpha); \Psi \vdash T \,\%\, Z}\ \underline{\texttt{fold}}\text{-L}$$

$$\frac{\Sigma, i : \gamma; \Gamma; \Delta, p : S; \Psi \vdash T \,\%\, Z}{\Sigma; \Gamma; \Delta, [i, p] : \exists i : \gamma.S; \Psi \vdash T \,\%\, Z}\ \exists\text{-L}\ (i \notin \Sigma, \Gamma, \Delta, p, \Psi, Z)$$

$$\frac{\Sigma; \Gamma; \Delta, p_1 : S_1; \Psi \vdash M_1 \# S \qquad \Sigma; \Gamma; \Delta, p_2 : S_2; \Psi \vdash M_2 \# S}{\Sigma; \Gamma; \Delta, p_1|_\zeta p_2 : S_1 \oplus S_2; \Psi \vdash M_1|_\zeta M_2 \# S}\ \oplus\text{-}L_M$$

$$\frac{\Sigma; \Gamma; \Delta, p_1 : S_1; \Psi \vdash E_1 \div S \qquad \Sigma; \Gamma; \Delta, p_2 : S_2; \Psi \vdash E_2 \div S}{\Sigma; \Gamma; \Delta, p_1|_\zeta p_2 : S_1 \oplus S_2; \Psi \vdash E_1|_\zeta E_2 \div S}\ \oplus\text{-}L_E$$

Figure 6: Type system : Left rules for patterns

monadic terms and expressions only. We call a term, monadic term or expression closed if it has no free variables. Apart from these restrictions, all other constructs in *f*CLL ($\otimes$, $\underline{\texttt{inl}}$, $\underline{\texttt{inr}}$, !, $\underline{\texttt{fold}}$ and existentials) are evaluated eagerly. Values for *f*CLL are described below:

Term values, V	$::=$	$\lambda x.N \mid \hat{\lambda} x.N \mid \{E\} \mid \langle N_1, N_2 \rangle \mid \Lambda i.N$
Monadic values, M_v	$::=$	$V \mid M_{v_1} \otimes M_{v_2} \mid 1 \mid \underline{\texttt{inl}}\, M_v \mid \underline{\texttt{inr}}\, M_v \mid !V \mid \underline{\texttt{fold}}\, M_v \mid [t, M_v]$

There are no values at the expression level, because expressions evaluate to monadic values. We define two operations on terms, monadic terms and expressions of *f*CLL: $\texttt{left}_\zeta$ and $\texttt{right}_\zeta$, which are the left and right case branches for the choice variable ζ, respectively. Figure 7 defines some of the interesting cases of these operations. The definitions for the remaining syntactic constructors are obtained by lifting these definitions homomorphically. Thus $\texttt{left}_\zeta(x) = x$, $\texttt{left}_\zeta(\lambda x.N) = \lambda x.\texttt{left}_\zeta(N)$, $\texttt{left}_\zeta(N_1\ N_2) = \texttt{left}_\zeta(N_1)\ \texttt{left}_\zeta(N_2)$, $\texttt{left}_\zeta(M_1 \otimes M_2) = \texttt{left}_\zeta(M_1) \otimes \texttt{left}_\zeta(M_2)$, $\texttt{left}_\zeta(\{E\}) = \{\texttt{left}_\zeta(E)\}$, etc. The result of substituting a monadic value M_v for a pattern p of the corresponding "shape" in a program T ($\texttt{subst}(M/p, T)$) is given in figure 8. This substitution is defined by induction on the structure of the pattern p. The base substitutions $[V/x]$ and $[t/i]$ are the usual capture avoiding substitutions for free variables and free index variables respectively.

The call-by-value evaluation rules for *f*CLL are given in figures 9, 10 and 11. We use three reduction judgments - $N \rightarrow N'$, $M \mapsto M'$ and $E \hookrightarrow E'$. The rules are standard. We allow reductions of M_1 and M_2 to interleave when we reduce $M_1 \otimes M_2$. Thus the two components of a tensor may be evaluated in parallel. For reasons mentioned earlier, pairs and the monad at the term level are evaluated lazily. The reduction rules for abstractions and applications are standard call-by-value.

$$\begin{array}{ll}\mathtt{left}_\zeta(M_1|_\zeta M_2) = M_1 & \mathtt{left}_\zeta(M_1|_\epsilon M_2) = \mathtt{left}_\zeta(M_1)|_\epsilon \mathtt{left}_\zeta(M_2) \quad \epsilon \neq \zeta \\ \mathtt{left}_\zeta(E_1|_\zeta E_2) = E_1 & \mathtt{left}_\zeta(E_1|_\epsilon E_2) = \mathtt{left}_\zeta(E_1)|_\epsilon \mathtt{left}_\zeta(E_2) \quad \epsilon \neq \zeta \end{array}$$

$$\mathtt{left}_\zeta(\underline{\mathtt{let}}\ \{p\} = N\ \underline{\mathtt{in}}\ E) = (\underline{\mathtt{let}}\ \{p\} = \mathtt{left}_\zeta(N)\ \underline{\mathtt{in}}\ \mathtt{left}_\zeta(E)) \quad (\zeta \notin \mathtt{dv}(p))$$

$$\begin{array}{ll}\mathtt{right}_\zeta(M_1|_\zeta M_2) = M_2 & \mathtt{right}_\zeta(M_1|_\epsilon M_2) = \mathtt{right}_\zeta(M_1)|_\epsilon \mathtt{right}_\zeta(M_2) \quad \epsilon \neq \zeta \\ \mathtt{right}_\zeta(E_1|_\zeta E_2) = E_2 & \mathtt{right}_\zeta(E_1|_\epsilon E_2) = \mathtt{right}_\zeta(E_1)|_\epsilon \mathtt{right}_\zeta(E_2) \quad \epsilon \neq \zeta \end{array}$$

$$\mathtt{right}_\zeta(\underline{\mathtt{let}}\ \{p\} = N\ \underline{\mathtt{in}}\ E) = (\underline{\mathtt{let}}\ \{p\} = \mathtt{right}_\zeta(N)\ \underline{\mathtt{in}}\ \mathtt{right}_\zeta(E)) \quad (\zeta \notin \mathtt{dv}(p))$$

Figure 7: Choice projections for fCLL

$$T ::= N \mid M \mid E$$

$$\begin{array}{ll}\mathtt{subst}(1/1, T) = T & \mathtt{subst}(V/x, T) = T[V/x] \\ \mathtt{subst}(!V/!x, T) = \mathtt{subst}(V/x, T) & \mathtt{subst}([t, M_v]/[i, p], T) = \mathtt{subst}(M_v/p, T[t/i]) \end{array}$$

$$\mathtt{subst}(\underline{\mathtt{inl}}\ M_v/(p_1|_\zeta p_2), T) = \mathtt{subst}(M_v/p_1, \mathtt{left}_\zeta(T))$$

$$\mathtt{subst}(\underline{\mathtt{inr}}\ M_v/(p_1|_\zeta p_2), T) = \mathtt{subst}(M_v/p_2, \mathtt{right}_\zeta(T))$$

$$\mathtt{subst}(M_{v_1} \otimes M_{v_2}/p_1 \otimes p_2, T) = \mathtt{subst}(M_{v_2}/p_2, \mathtt{subst}(M_{v_1}/p_1, T))$$

$$\mathtt{subst}(\underline{\mathtt{fold}}(M_v)/\underline{\mathtt{fold}}(p), T) = \mathtt{subst}(M_v/p, T)$$

Figure 8: Substitution of monadic values for patterns

$$\frac{N \;\rightarrow\; N'}{\pi_1\, N \;\rightarrow\; \pi_1 N'} \rightarrow \pi_1 \qquad \frac{N \;\rightarrow\; N'}{\pi_2\, N \;\rightarrow\; \pi_2 N'} \rightarrow \pi_2$$

$$\frac{}{\pi_1\, \langle N_1, N_2\rangle \;\rightarrow\; N_1} \rightarrow \langle\rangle\pi_1 \qquad \frac{}{\pi_2\, \langle N_1, N_2\rangle \;\rightarrow\; N_2} \rightarrow \langle\rangle\pi_2$$

$$\frac{N_1 \;\rightarrow\; N_1'}{N_1\, N_2 \;\rightarrow\; N_1'\, N_2} \rightarrow APP_1 \qquad \frac{N_2 \;\rightarrow\; N_2'}{V\, N_2 \;\rightarrow\; V\, N_2'} \rightarrow APP_2 \qquad \frac{}{(\lambda x.N)\; V \;\rightarrow\; N[V/x]} \rightarrow \lambda APP$$

$$\frac{N_1 \;\rightarrow\; N_1'}{N_1 \,\hat{}\, N_2 \;\rightarrow\; N_1' \,\hat{}\, N_2} \rightarrow LAPP_1 \qquad \frac{N_2 \;\rightarrow\; N_2'}{V \,\hat{}\, N_2 \;\rightarrow\; V \,\hat{}\, N_2'} \rightarrow LAPP_2 \qquad \frac{}{(\hat{\lambda} x.N) \,\hat{}\, V \;\rightarrow\; N[V/x]} \rightarrow \hat{\lambda} LAPP$$

$$\frac{N \;\rightarrow\; N'}{N\; [t] \;\rightarrow\; N'\; [t]} \rightarrow \forall \qquad \frac{}{(\Lambda i : \gamma.N)\; [t] \;\rightarrow\; N[t/i]} \rightarrow \Lambda APP$$

Figure 9: Reduction for terms, $N \;\rightarrow\; N'$

$$\frac{N \;\rightarrow\; N'}{N \;\mapsto\; N'} \rightarrow\mapsto \qquad \frac{N \;\rightarrow\; N'}{!N \;\mapsto\; !N'} \mapsto !$$

$$\frac{M_1 \;\mapsto\; M_1'}{M_1 \otimes M_2 \;\mapsto\; M_1' \otimes M_2} \mapsto \otimes_1 \qquad \frac{M_2 \;\mapsto\; M_2'}{M_1 \otimes M_2 \;\mapsto\; M_1 \otimes M_2'} \mapsto \otimes_2$$

$$\frac{M \;\mapsto\; M'}{\underline{\texttt{inl}}\, M \;\mapsto\; \underline{\texttt{inl}}\, M'} \mapsto \oplus_1 \qquad \frac{M \;\mapsto\; M'}{\underline{\texttt{inr}}\, M \;\mapsto\; \underline{\texttt{inr}}\, M'} \mapsto \oplus_2$$

$$\frac{M \;\mapsto\; M'}{\underline{\texttt{fold}}\, M \;\mapsto\; \underline{\texttt{fold}}\, M'} \mapsto FOLD$$

$$\frac{M \;\mapsto\; M'}{[t, M] \;\mapsto\; [t, M']} \mapsto \exists \qquad \frac{}{\mu u.M \;\mapsto\; M[\mu u.M/u]} \mapsto \mu$$

Figure 10: Reduction for monadic terms, $M \;\mapsto\; M'$

$$\frac{M \;\mapsto\; M'}{M \;\hookrightarrow\; M'} \mapsto\hookrightarrow \qquad \frac{}{\underline{\texttt{let}}\; \{p\} = \{M_v\}\; \underline{\texttt{in}}\; E \;\hookrightarrow\; \texttt{subst}(M_v/p, E)} \hookrightarrow LETRED$$

$$\frac{N \;\rightarrow\; N'}{\underline{\texttt{let}}\; \{p\} = N\; \underline{\texttt{in}}\; E \;\hookrightarrow\; \underline{\texttt{let}}\; \{p\} = N'\; \underline{\texttt{in}}\; E} \hookrightarrow LET_1$$

$$\frac{E \;\hookrightarrow\; E'}{\underline{\texttt{let}}\; \{p\} = \{E\}\; \underline{\texttt{in}}\; E_1 \;\hookrightarrow\; \underline{\texttt{let}}\; \{p\} = \{E'\}\; \underline{\texttt{in}}\; E_1} \hookrightarrow LET_2$$

Figure 11: Reduction for expressions, $E \;\hookrightarrow\; E'$

Contexts for expression evaluation:

$$\begin{array}{rcl} \mathcal{C}[\,] & ::= & [\,] \\ & & \mid \underline{\texttt{let}}\,\{p\} = N\ \underline{\texttt{in}}\,\mathcal{C}[\,] \\ & & \mid \underline{\texttt{let}}\,\{p\} = \{\mathcal{C}[\,]\}\ \underline{\texttt{in}}\,E \end{array}$$

Reduction rules:

$$\frac{M \ \mapsto\ M' \qquad M \text{ closed}}{\mathcal{C}[M] \ \hookrightarrow\ \mathcal{C}[M']} \mapsto\hookrightarrow$$

$$\frac{(\underline{\texttt{let}}\,\{p\} = \{M_v\}\ \underline{\texttt{in}}\,E) \text{ closed}}{\mathcal{C}[\underline{\texttt{let}}\,\{p\} = \{M_v\}\ \underline{\texttt{in}}\,E] \ \hookrightarrow\ \mathcal{C}[\texttt{subst}(M_v/p, E)]} \hookrightarrow_{LETRED}$$

$$\frac{N \ \rightarrow\ N' \qquad N \text{ closed}}{\mathcal{C}[\underline{\texttt{let}}\,\{p\} = N\ \underline{\texttt{in}}\,E] \ \hookrightarrow\ \mathcal{C}[\underline{\texttt{let}}\,\{p\} = N'\ \underline{\texttt{in}}\,E]} \hookrightarrow_{LET_1}$$

Figure 12: Generalized evaluation rules for expressions

2.1 Parallel Evaluation in Expressions

Consider the following expression when $\texttt{dv}(p_1) \cap \texttt{fv}(N_2) = \phi$.

$$\begin{array}{rcl} E & = & \underline{\texttt{let}}\,\{p_1\} = N_1\ \underline{\texttt{in}} \\ & & \quad \underline{\texttt{let}}\,\{p_2\} = N_2\ \underline{\texttt{in}}\,E' \end{array}$$

If E is a closed expression, then according to the rules in figure 11, it is evaluated as follows. First N_1 is evaluated to a term of the form $\{E_1\}$. Then E_1 is evaluated to a monadic value M_{v_1} of shape p_1. This monadic value is then substituted for p_1 in the expression $\underline{\texttt{let}}\,\{p_2\} = N_2\ \underline{\texttt{in}}\,E'$. Subsequently, N_2 is evaluated. But by the given condition N_2 is a closed term. Hence $subst(M_{v_1}/p_1, N_2) = N_2$ and therefore there is no need to postpone the evaluation of N_2 until N_1 is completely evaluated. We may interleave or parallelize the evaluation of N_1 and N_2, without affecting the result of the computation. This idea allows us to generalize the evaluation rules for expressions to those shown in figure 12. The generalized rules for evaluating expressions are presented using evaluation contexts on expressions. We obtain parallel evaluation from these rules using the following heuristic - if E is a closed expression and $E = \mathcal{C}_1[E_1] = \mathcal{C}_2[E_2]$ where E_1 and E_2 are closed and non-overlapping sub-expressions of E, then E_1 and E_2 may be evaluated in parallel.

Even though these generalized rules allow parts of expressions to evaluate in parallel, they provide no primitive for communication between simultaneously evaluating sub-terms. In section 3, we introduce *l*CLL, which is a concurrent logic programming language that allows asynchronous message passing between parallel processes.

2.2 Type-Safety

We establish the type-safety theorem for *f*CLL by proving progress and preservation theorems. The progress theorem states that a typed term is either a value or it can step further. The preservation theorem says that

$$\frac{}{\Sigma; \Gamma; \Delta \Longleftarrow \Sigma; \Gamma; \Delta, 1 : 1} \Leftarrow -1 \qquad \frac{}{\Sigma; \Gamma, x : A; \Delta \Longleftarrow \Sigma; \Gamma; \Delta, !x :!A} \Leftarrow -!$$

$$\frac{}{\Sigma; \Gamma; \Delta, p_1 : S_1, p_2 : S_2 \Longleftarrow \Sigma; \Gamma; \Delta, p_1 \otimes p_2 : S_1 \otimes S_2} \Leftarrow -\otimes$$

$$\frac{}{\Sigma, i : \gamma; \Gamma; \Delta, p : S(i) \Longleftarrow \Sigma; \Gamma; \Delta, [i, p] : \exists i : \gamma.S(i)} \Leftarrow -\exists$$

$$\frac{}{\Sigma; \Gamma; \Delta, p : S(\mu\alpha.S(\alpha)) \Longleftarrow \Sigma; \Gamma; \Delta, \underline{\texttt{fold}}(p) : \mu\alpha.S(\alpha)} \Leftarrow -\underline{\texttt{fold}}$$

$$\frac{}{\Sigma; \Gamma; \Delta \Longleftarrow \Sigma; \Gamma; \Delta} \Leftarrow -REF \qquad \frac{\Sigma; \Gamma; \Delta \Longleftarrow \Sigma'; \Gamma'; \Delta' \qquad \Sigma'; \Gamma'; \Delta' \Longleftarrow \Sigma''; \Gamma''; \Delta''}{\Sigma; \Gamma; \Delta \Longleftarrow \Sigma''; \Gamma''; \Delta''} \Leftarrow -TRANS$$

Figure 13: Context entailment, $\Sigma; \Gamma; \Delta \Longleftarrow \Sigma'; \Gamma'; \Delta'$

reduction of a typed term under the evaluation rules preserves its type. Together these two imply *type-safety* i.e. any typed term either evaluates to a value or diverges indefinitely. In order to establish these theorems, we need a few results.

Notation 1. We use $T \mathbin{\%} Z$ to denote any of $N : A$, $M \# S$ or $E \div S$.

Definition 1 (Context Entailment). The relation $\Sigma; \Gamma; \Delta \Longleftarrow \Sigma'; \Gamma'; \Delta'$, read as $\Sigma; \Gamma; \Delta$ entails $\Sigma'; \Gamma'; \Delta'$, is shown in figure 13.

Lemma 1 ($\Longleftarrow$ properties).

1. If $\Sigma; \Gamma; \Delta \Longleftarrow \Sigma'; \Gamma'; \Delta'$, then $\Sigma \supseteq \Sigma'$ and $\Gamma \supseteq \Gamma'$.
2. If $\Sigma; \Gamma; \Delta \Longleftarrow \Sigma'; \Gamma'; \Delta'$, then
   (a) $\texttt{dv}(\Gamma) \cup \texttt{dv}(\Delta) = \texttt{dv}(\Gamma') \cup \texttt{dv}(\Delta')$
   (b) $\texttt{dlv}(\Delta) = \texttt{dlv}(\Delta')$
   (c) $\texttt{div}(\Sigma) \cup \texttt{div}(\Delta) = \texttt{div}(\Sigma') \cup \texttt{div}(\Delta')$
3. If $\mathcal{D} :: \Sigma; \Gamma; \Delta; \Psi \vdash T \mathbin{\%} Z$ and $\Sigma; \Gamma; \Delta \Longleftarrow \Sigma'; \Gamma'; \Delta'$, then $\mathcal{D}$ can be *extended* to a derivation $\mathcal{D}' :: \Sigma'; \Gamma'; \Delta'; \Psi \vdash T \mathbin{\%} Z$ using the rules $\otimes - L$, $1 - L$, $\exists - L$, $! - L$ and $\underline{\texttt{fold}} - L$ only.
4. (Weakening) If $\Sigma; \Gamma; \Delta \Longleftarrow \Sigma'; \Gamma'; \Delta'$, then
   (a) $\Sigma, \Sigma''; \Gamma; \Delta \Longleftarrow \Sigma', \Sigma''; \Gamma'; \Delta'$
   (b) $\Sigma; \Gamma, \Gamma''; \Delta \Longleftarrow \Sigma'; \Gamma', \Gamma''; \Delta'$
   (c) $\Sigma; \Gamma; \Delta, \Delta'' \Longleftarrow \Sigma'; \Gamma'; \Delta', \Delta''$

Proof. In each case by induction on the given derivation $\Sigma; \Gamma; \Delta \Longleftarrow \Sigma'; \Gamma'; \Delta'$.

Definition 2 (Height of a derivation). The height of a derivation $\mathcal{D}$, $\texttt{height}(\mathcal{D})$ is defined to be the length of longest path from the conclusion to any leaf.

Lemma 2 (Weakening). If $\Sigma; \Gamma; \Delta; \Psi \vdash \psi$, then

1. $\Sigma, i : \gamma; \Gamma; \Delta; \Psi \vdash \psi$.
2. $\Sigma; \Gamma, x : A; \Delta; \Psi \vdash \psi$.
3. $\Sigma; \Gamma; \Delta; \Psi, u : S \vdash \psi$.

Proof. By induction on the given derivation.

Lemma 3 (Left inversion).

1. If $\Sigma; \Gamma; \Delta, p : 1; \Psi \vdash \psi$, then $p = 1$ and $\Sigma; \Gamma; \Delta; \Psi \vdash \psi$.
2. If $\Sigma; \Gamma; \Delta; p :!A; \Psi \vdash \psi$, then $p =!x$ and $\Sigma; \Gamma, x : A; \Delta; \Psi \vdash \psi$.
3. If $\Sigma; \Gamma; \Delta, p : A; \Psi \vdash \psi$, then $p = x$.
4. If $\Sigma; \Gamma; \Delta, p : S_1 \otimes S_2; \Psi \vdash \psi$, then $p = p_1 \otimes p_2$ and $\Sigma; \Gamma; \Delta, p_1 : S_1, p_2 : S_2; \Psi \vdash \psi$.
5. If $\Sigma; \Gamma; \Delta, p : \exists i : \gamma.S; \Psi \vdash \psi$, then $p = [i, p_1]$ and $\Sigma, i : \gamma; \Gamma; \Delta, p_1 : S; \Psi \vdash \psi$.
6. If $\Sigma; \Gamma; \Delta, p : S_1 \oplus S_2; \Psi \vdash \psi$, then $p = p_1|_\zeta p_2$, $\Sigma; \Gamma; \Delta, p_1 : S_1; \Psi \vdash \texttt{left}_\zeta(\psi)$ and $\Sigma; \Gamma; \Delta, p_2 : S_2; \Psi \vdash \texttt{right}_\zeta(\psi)$. ($\texttt{left}_\zeta(N : A) = \texttt{left}_\zeta(N) : A$, etc.)
7. If $\Sigma; \Gamma; \Delta, p : \mu\alpha.S(\alpha); \Psi \vdash \psi$, then $p = \underline{\texttt{fold}}(p')$ and $\Sigma; \Gamma; \Delta, p' : S(\mu\alpha.S(\alpha)); \Psi \vdash \psi$.

Proof. Each statement may be separately proved by induction on the given typing derivation.

Lemma 4 (Strong right value inversion).

1. If $\mathcal{D} :: \Sigma; \Gamma; \Delta; \Psi \vdash V : A \to B$ then $V = \lambda x.N$ and there is a derivation $\mathcal{D}' :: \Sigma; \Gamma, x : A; \Delta; \Psi \vdash N : B$ with $\texttt{height}(\mathcal{D}') \leq \texttt{height}(\mathcal{D})$.
2. If $\mathcal{D} :: \Sigma; \Gamma; \Delta; \Psi \vdash V : A \multimap B$ then $V = \hat{\lambda} x.N$ and there is a derivation $\mathcal{D}' :: \Sigma; \Gamma; \Delta, x : A; \Psi \vdash N : B$ with $\texttt{height}(\mathcal{D}') \leq \texttt{height}(\mathcal{D})$.
3. If $\mathcal{D} :: \Sigma; \Gamma; \Delta; \Psi \vdash V : A \& B$ then $V = \langle N_1, N_2 \rangle$ and there are derivations $\mathcal{D}_1 :: \Sigma; \Gamma; \Delta; \Psi \vdash N_1 : A$ and $\mathcal{D}_2 :: \Sigma; \Gamma; \Delta; \Psi \vdash N_2 : B$ with $\texttt{height}(\mathcal{D}_1) \leq \texttt{height}(\mathcal{D})$ and $\texttt{height}(\mathcal{D}_2) \leq \texttt{height}(\mathcal{D})$.
4. If $\mathcal{D} :: \Sigma; \Gamma; \Delta; \Psi \vdash V : \{S\}$ then $V = \{E\}$ and there is a derivation $\mathcal{D}' :: \Sigma; \Gamma; \Delta; \Psi \vdash E \div S$ with $\texttt{height}(\mathcal{D}') \leq \texttt{height}(\mathcal{D})$.
5. If $\mathcal{D} :: \Sigma; \Gamma; \Delta; \Psi \vdash V : \forall i : \gamma.A(i)$, $i \notin \Sigma$ then $V = \Lambda i : \gamma.N(i)$ and there is a derivation $\mathcal{D}' :: \Sigma, i : \gamma; \Gamma; \Delta; \Psi \vdash N(i) : A(i)$ with $\texttt{height}(\mathcal{D}') \leq \texttt{height}(\mathcal{D})$.

Proof. Each statement can be proved separately by induction on the given typing derivation.

Lemma 5 (Right monadic value inversion).

1. If $\Sigma; \Gamma; \Delta; \Psi \vdash M_v \# A$, then $M_v = V$ and $\Sigma; \Gamma; \Delta; \Psi \vdash V : A$.

2. If $\Sigma; \Gamma; \Delta; \Psi \vdash M_v \# !A$, then $M_v = !V$ and for some Σ' and Γ', $\Sigma'; \Gamma'; \cdot; \Psi \vdash V : A$ and $\Sigma'; \Gamma'; \cdot \Longleftarrow \Sigma; \Gamma; \Delta$.

3. If $\Sigma; \Gamma; \Delta; \Psi \vdash M_v \# S_1 \otimes S_2$, then $M_v = M_{v_1} \otimes M_{v_2}$ and for some $\Sigma', \Gamma', \Delta'_1, \Delta'_2$, the following three hold:

   (a) $\Sigma'; \Gamma'; \Delta'_1, \Delta'_2 \Longleftarrow \Sigma; \Gamma; \Delta$

   (b) $\Sigma'; \Gamma'; \Delta'_1; \Psi \vdash M_{v_1} \# S_1$

   (c) $\Sigma'; \Gamma'; \Delta'_2; \Psi \vdash M_{v_2} \# S_2$

4. If $\Sigma; \Gamma; \Delta; \Psi \vdash M_v \# 1$, then $M_v = 1$ and for some Σ' and Γ', $\Sigma'; \Gamma'; \cdot \Longleftarrow \Sigma; \Gamma; \Delta$.

5. If $\Sigma; \Gamma; \Delta; \Psi \vdash M_v \# S_1 \oplus S_2$, then one of the following holds

   (a) $M_v = \underline{\texttt{inl}}\, M'_v$ and $\Sigma; \Gamma; \Delta; \Psi \vdash M'_v \# S_1$

   (b) $M_v = \underline{\texttt{inr}}\, M'_v$ and $\Sigma; \Gamma; \Delta; \Psi \vdash M'_v \# S_2$

6. If $\Sigma; \Gamma; \Delta; \Psi \vdash M_v \# \exists i : \gamma.S(i)$, then $M_v = [t, M'_v]$ and for some Σ', Γ' and Δ', the following three hold:

   (a) $\Sigma'; \Gamma'; \Delta' \Longleftarrow \Sigma; \Gamma; \Delta$

   (b) $\Sigma' \vdash t : \gamma$

   (c) $\Sigma'; \Gamma'; \Delta'; \Psi \vdash M'_v \# S(t)$

7. If $\Sigma; \Gamma; \Delta; \Psi \vdash M_v \# \mu\alpha.S(\alpha)$, then $M_v = \underline{\texttt{fold}}(M'_v)$ and $\Sigma; \Gamma; \Delta; \Psi \vdash M'_v \# S(\mu\alpha.S(\alpha))$.

Proof. In each case by induction on the given typing derivation.

Lemma 6 (Strong right inversion for expressions).

1. If $\mathcal{D} :: \Sigma; \Gamma; \Delta; \Psi \vdash M \div S$, then there exists $\mathcal{D}' :: \Sigma; \Gamma; \Delta; \Psi \vdash M \# S$ with $\texttt{height}(\mathcal{D}') \leq \texttt{height}(\mathcal{D})$.

2. If $\mathcal{D} :: \Sigma; \Gamma; \Delta; \Psi \vdash \underline{\texttt{let}}\ \{p\} = N\ \underline{\texttt{in}}\ E \div S'$, then there exist $\Sigma', \Gamma', \Delta'_1, \Delta'_2, S, \mathcal{D}_1, \mathcal{D}_2$ such that

   (a) $\Sigma'; \Gamma'; \Delta'_1, \Delta'_2 \Longleftarrow \Sigma; \Gamma; \Delta$

   (b) $\mathcal{D}_1 :: \Sigma'; \Gamma'; \Delta'_1; \Psi \vdash N : \{S\}$ and $\texttt{height}(\mathcal{D}_1) \leq \texttt{height}(\mathcal{D})$

   (c) $\mathcal{D}_2 :: \Sigma'; \Gamma'; \Delta'_2, p : S; \Psi \vdash E \div S'$ and $\texttt{height}(\mathcal{D}_2) \leq \texttt{height}(\mathcal{D})$

Proof. By induction on the given typing derivation.

Lemma 7 (Substitution).

1. If $\mathcal{D} :: \Sigma; \Gamma; \Delta; \Psi \vdash V : A$ and $\mathcal{D}' :: \Sigma; \Gamma; \Delta', x : A; \Psi \vdash T \,\%\, Z$, then $\Sigma; \Gamma; \Delta, \Delta'; \Psi \vdash T[V/x] \,\%\, Z$.

2. If $\Sigma \vdash t : \gamma$ and $\mathcal{D}' :: \Sigma, i : \gamma; \Gamma; \Delta; \Psi \vdash T(i) \,\%\, Z(i)$, then $\Sigma; \Gamma[t/i]; \Delta[t/i]; \Psi[t/i] \vdash T(t) \,\%\, Z(t)$.

3. If $\mathcal{D} :: \Sigma; \Gamma; \Delta; \Psi \vdash M_v \# S$ and $\mathcal{D}' :: \Sigma; \Gamma; \Delta', p : S; \Psi \vdash T \,\%\, Z$, then
   $\Sigma; \Gamma; \Delta, \Delta'; \Psi \vdash \texttt{subst}(M_v/p, T) \,\%\, Z$.

4. If $\mathcal{D} :: \Sigma; \Gamma; .; \Psi \vdash V : A$ and $\mathcal{D}' :: \Sigma; \Gamma, x : A; \Delta'; \Psi \vdash T \,\%\, Z$, then $\Sigma; \Gamma; \Delta'; \Psi \vdash T[V/x] \,\%\, Z$.

5. If $\mathcal{D} :: \Sigma; \Gamma; .; \Psi \vdash M \# S$ and $\mathcal{D}' :: \Sigma; \Gamma; \Delta'; \Psi, u : S \vdash T \,\%\, Z$, then $\Sigma; \Gamma; \Delta'; \Psi \vdash T[M/u] \,\%\, Z$.

Proof. Statements (1), (2), (4) and (5) can be proved by induction on derivation $\mathcal{D}'$. To prove (3), we use induction on the derivation $\mathcal{D}$, lemmas 2 and 3 and statements (1) and (2).

Lemma 8 (Preservation).

1. If $\Sigma; \Gamma; \Delta; \Psi \vdash N : A$ and $N \rightarrow N'$, then $\Sigma; \Gamma; \Delta; \Psi \vdash N' : A$.
2. If $\Sigma; \Gamma; \Delta; \Psi \vdash M \# S$ and $M \mapsto M'$, then $\Sigma; \Gamma; \Delta; \Psi \vdash M' \# S$.
3. If $\Sigma; \Gamma; \Delta; \Psi \vdash E \div S$ and $E \hookrightarrow E'$, then $\Sigma; \Gamma; \Delta; \Psi \vdash E' \div S$.

Proof. By simultaneous induction on the height of the given typing derivation, using lemmas 7, 6 and 4. For the case of expressions, we perform a sub-induction on the evaluation context $\mathcal{C}[\,]$ and use lemma 6.

Lemma 9 (Progress).

1. If $\Sigma; \cdot; \cdot; \cdot \vdash N : A$, then either $N = V$ or for some N', $N \rightarrow N'$.
2. If $\Sigma; \cdot; \cdot; \cdot \vdash M \# S$, then either $M = M_v$ or for some M', $M \mapsto M'$.
3. If $\Sigma; \cdot; \cdot; \cdot \vdash E \div S$, then either $E = M_v$ or for some E', $E \hookrightarrow E'$.

Proof. By induction on the given typing derivation.

Theorem 1 (Type-Safety).

1. If $\Sigma; \cdot; \cdot; \cdot \vdash N : A$ and $N \rightarrow^* N'$, then either $N' = V$ or there exists N'' such that $N' \rightarrow N''$.
2. If $\Sigma; \cdot; \cdot; \cdot \vdash M \# S$ and $M \mapsto^* M'$, then either $M' = M_v$ or there exists M'' such that $M' \mapsto M''$.
3. If $\Sigma; \cdot; \cdot; \cdot \vdash E \div S$ and $E \hookrightarrow^* E'$, then either $E' = M_v$ or there exists E'' such that $E' \hookrightarrow E''$.

Proof. By induction on the number of steps in the reduction. The statement at the base case (no reduction) is the same as the progress lemma (lemma 9). For the induction step, we use preservation (lemma 8).

2.3 Examples

In this section we explain program construction in *f*CLL through a number of examples. We present these examples in ML-like syntax. We assume that we have named and recursive functions, conditionals and datatype constructions at the term level, which may be added to *f*CLL presented above in a straightforward manner.

Divide and Conquer. Our first example is a general divide and conquer program. Let us suppose we have a type P of problems and a type A of solutions. A general divide and conquer method assumes the following input functions:

1. `divide` : $P \rightarrow P \times P$ that divides a given problem into two subproblems, each of which is strictly smaller than the original.

2. `istrivial` : $P \rightarrow$ `bool` that decides if the input problem is at its base case.

3. `solve` : $P \rightarrow A$ that returns the solution to a base case problem.

4. `merge` : $A \rightarrow A \rightarrow A$ that combines solutions of two subproblems obtained using `divide` into a solution to the original problem.

In *f*CLL, we have no product type (which would be present in a non-linear language). So we encode the product type as $A \times B = \{!A \otimes !B\}$. Then we have the following divide and conquer function, `divAndConquer`:

$$
\begin{array}{l}
\texttt{divAndConquer} = \\
\quad \lambda(divide) : P \rightarrow \{!P \otimes !P\}.\ \lambda(istrivial) : P \rightarrow \texttt{bool}. \\
\quad \lambda(solve) : P \rightarrow \{!A\}.\ \lambda(merge) : A \rightarrow A \rightarrow \{!A\}.\ \lambda p : P. \\
\quad\quad \underline{\texttt{if}}\ (istrivial\ p)\ \underline{\texttt{then}}\ solve\ p \\
\quad\quad \underline{\texttt{else}} \\
\quad\quad \{ \\
\quad\quad\quad \underline{\texttt{let}}\ \{!p_1 \otimes !p_2\} = divide\ p\ \underline{\texttt{in}} \\
\quad\quad\quad \underline{\texttt{let}}\ \{!s_1\} = \texttt{divAndConquer}\ divide\ istrivial\ solve\ merge\ p_1\ \underline{\texttt{in}} \\
\quad\quad\quad \underline{\texttt{let}}\ \{!s_2\} = \texttt{divAndConquer}\ divide\ istrivial\ solve\ merge\ p_2\ \underline{\texttt{in}} \\
\quad\quad\quad \underline{\texttt{let}}\ \{!s\} = merge\ s_1\ s_2\ \underline{\texttt{in}} \\
\quad\quad\quad\quad !s \\
\quad\quad \}
\end{array}
$$

The return type of `divAndConquer` is $\{!A\}$. Observe that in this program, the two $\underline{\texttt{let}}$ eliminations corresponding to the two recursive calls may occur in parallel i.e. the terms `divAndConquer` *divide istrivial solve merge* p_1 and `divAndConquer` *divide istrivial solve merge* p_2 can be evaluated simultaneously. This is because the second term does not use the variable s_1. Therefore the program above attains the parallelism that is expected in a divide and conquer approach.

Bellman-Ford algorithm. We now present a parallel implementation of Bellman-Ford algorithm for single source shortest paths in directed, non-negative edge-weighted graphs. Assume that a directed graph $\mathcal{G}$ has n vertices, numbered $1, \ldots, n$. For each vertex we have a list of incoming edges called the adjacency list of the vertex. Each element of the adjacency list is a pair, the first member of which is an integer, which is the source vertex of the edge and the second member is a non-negative real number, which is the weight of the edge. The whole graph is represented as a list of adjacency lists, one adjacency list for each vertex. During the course of the algorithm, we have approximations of shortest distance from the source vertex to each vertex. These are stored as a list of reals. The type of this list is called `distlist`.

```
type distlist = {!real} list
type adjlist = {!int⊗ !real} list
type edgelist = {!adjlist} list
```

The following function finds the ith element of a list l.

```
val find:  'a list → int → 'a
find =
   λl:'a list.  λi:int.
      if (i = 1) then head(l) else find(tail(l))(i − 1)
```

The main routine of the Bellman-Ford algorithm is a relaxation procedure that takes as input a vertex, v (which in our case is completely represented by the adjacency list, al. The exact vertex number of the vertex is immaterial), a presently known approximation of shortest distances to all vertices from the source (called dl), the present known approximation of the shortest distance from the source to v and returns a new approximation to the shortest distance from source to v.

```
val relax:  adjlist → distlist → real → {!real}
relax =
   λ(al):adjlist.  λ(dl):distlist.  λd:real.
      case(al) of
         [] => {!d}
         | (a :: al) =>
            {
               let{!v⊗!w} = a in
               let{!d'} = find dl v in
               let{!d''} = relax al dl d in
                  !min(d'', d' + w)
            }
```

The calls `find` dl v and `relax` al dl d can be reduced in parallel in the above function. The main loop of the Bellman-Ford algorithm is a function `relaxall`, that applies the function `relax` to all the vertices. To make the code simpler, we assume that this function actually takes as argument *two copies* of the distance list. To make the code more presentable, we drop the λ-calculus notation and use standard ML's `fun` notation.

```
val relaxall:  edgelist → distlist → distlist → {!distlist}
fun relaxall [] [] dl = {![]}
|  relaxall (al :: el) (d :: dl') (dl) =
   {
      let{!al'} = al in
      let{!d'} = d in
      let{!d1} = relax al' dl d' in
      let{!l} = relaxall el dl' dl in
         !({!d1} :: l)
   }
```

In the above function, the calls relax al' dl d' and relaxall el dl' dl can be reduced in parallel. This results in simultaneous relaxation for the whole graph.

Suppose now that our source vertex is the vertex 1. Then we can initialize the distlist to the value $[0, \infty, \ldots, \infty]$. Using this initial value of the distlist, we iterate the function relaxall n times. The resultant value of distlist is the list of minimum distances from the source (vertex 1) to all the other vertices. The function BF below takes as input a graph (in the form of an edgelist) and returns the minimum distances to all vertices from the vertex 1.

```
fun BF (el: edgelist) =
    (* makedistlist: int → {!distlist} *)
    let fun makedistlist 0 = {![]}
        | makedistlist k =
          {
             let{!l} = makelist (k − 1)
             in
                !({!∞} :: l)
          }

        (* loop: int → distlist → {!distlist} *)
        fun loop 0 dl = {!dl}
        | loop k dl =
          {
             let{!dl′} = relaxall el dl dl
             in
                loop (k − 1) dl′
          }

        (* length: 'a list → {!int} *)
        fun length [] = {!0}
        | length (x :: l) =
          {
             let{!len} = length(l)
             in
                !(1 + len)
          }

        in
        {
           let{!n} = length el in
           let{!l} = makedistlist (n − 1) in
           let{!dl} = loop n ({!0} :: l) in
              !dl
        }
        end
```

3 *l*CLL: Concurrent Logic Programming in CLL

As mentioned in the introduction, an alternate paradigm used for concurrent languages is concurrent logic programming [34]. In this paradigm, proof-search in logic simulates computation and assignment to variables as well as communication are implemented through unification. Concurrency is inherent in such a setting because many parts of a proof can be computed or searched in parallel. We use similar ideas to create a concurrent logic programming language that allows concurrent computation of terms. We call this language *l*CLL.

*l*CLL differs significantly from other logic programming languages. Traditionally, logic programming uses only logical propositions and predicates but no proof-terms. Even if proof-terms are synthesized by the proof-search mechanism, they are merely witnesses to the proof found by the search. They play no computational role. In sharp contrast, we interpret proof-terms as programs and use the proof-search mechanism to link programs together. This linking mechanism is directed by the types of the terms that can be viewed as logical propositions through the Curry-Howard isomorphism. The whole idea may be viewed as an extension of the Curry-Howard isomorphism to include proof-search - in computational terms, proof-search corresponds to linking together programs using their types. For example, if $N_1 : A \multimap B$ and $N_2 : A$, then the proof-search mechanism can link N_1 and N_2 together to produce the term $N_1 \mathbin{\hat{}} N_2 : B$. *l*CLL extends this idea to all the connectives in FOMLL, and is rich enough to express most concurrency constructs.

We present *l*CLL as a Chemical Abstract Machine (CHAM)[4, 5]. The molecules in *l*CLL CHAM configurations are *f*CLL programs annotated by their types. The rewrite rules for these CHAM configurations are derived by modifying the inference rules for a proof-search method for FOMLL. One question that remains at this stage is *which* proof-search method we use for FOMLL and we answer this question next.

Proof-search in logic can be implemented in two different but related styles. In the *backward* style, search is started from the proposition to be proved as the goal. Each possible rule (assumption of the form $A \to B$) that can be used to conclude this goal is then considered and the premises of the rule applied become the subgoals for the proof-search. This process of matching a goal against the conclusion of a rule and making the rule's premises the subgoals for the remaining search is called *backchaining*. It is continued till the set of goals contains only axioms or no more rules apply. In the former case, the (sub) goal is provable. In the latter case, the (sub) goal cannot be proved and the proof-search must backtrack and find other possible rules to apply to some earlier goals in the search. There is an inherent non-determinism in this proof-search mechanism - at any step, one may apply any of the possible rules whose conclusion matches the goal at hand. This kind of non-determinism is called *don't-know* non-determinism. Since there is a possibility of backtracking if a bad rule is applied, search strategies are complete in the sense that proof-search will always find a proof of a proposition that is true. Most logic programming languages like Prolog use this style of theorem-proving.

A very different approach to proof-search is to start by assuming that the only known facts are axioms and then apply rules to known facts to obtain more facts which are true. This can be continued till the goal to be proved is among the facts known to be true, or no new facts can be concluded. In the former case, proof-search succeeds whereas in the latter case it fails. The process of applying a rule to known facts to obtain more facts in called *forward chaining*. In this style, search is exhaustive and does not require backtracking. This is known as the *forward* or *inverse* style of theorem-proving. One important aspect of the forward style in linear logic is that the facts obtained during this method are also linear. As a result, each fact may be used to conclude exactly one more fact and subsequently be removed from the set of known facts. This re-introduces non-determinism in the facts we choose to conclude. It also introduces the need to backtrack if we want completeness. However, in some applications, incompleteness is acceptable and the forward method is implemented without backtracking. Such implementations work as follows. At any stage, the proof-search procedure non-deterministically picks up any of the facts that it can conclude and continues. Such a proof-search procedure is non-deterministic in a sense different from don't-know non-determinism. The procedure simply concludes an arbitrary selection of facts and terminates, without caring about the goal. Hence this non-determinism is called *don't-care* non-determinism. A large number of concurrent logic programming languages use this non-determinism because it closely resembles non-deterministic synchronization between parallel processes in process calculi. We also choose to use this method of proof-search for *l*CLL. In our case, using this method is even more advantageous because we use proof-search to link programs together and execute them. Backtracking in such a setting is counter-intuitive and computationally expensive.

Our computation strategy for *l*CLL CHAM configurations is as follows. Each CHAM configuration is started with a certain number of type-annotated terms and a goal type. Once started, the configuration is allowed to rewrite according to a specific set of non-deterministic rules, which are based on forward chaining rules for proof-search in FOMLL. We do not backtrack. *If ever* the CHAM configuration reaches a state where it has exactly one term of the goal type, computation of the *l*CLL CHAM configuration succeeds, else it fails. Thus we use *don't-care* non-determinism and CHAM configurations can get stuck without reaching the goal. As a result, we do not have a progress lemma for *l*CLL as a whole. However, we develop a notion of types for CHAM configurations and prove a type-preservation lemma for CHAM rewrite moves. This preservation lemma implies a weak type-safety theorem for *l*CLL CHAM configurations. This theorem states that *individual f*CLL terms in *l*CLL CHAM configurations obtained by rewriting well-typed CHAM configurations are either values or they can reduce further. As before, we are interested in the execution of closed terms only and we assume that terms in *l*CLL CHAMs do not have free variables in them. In order to demonstrate the expressiveness of *l*CLL, we present a translation of an asynchronous π-calculus [6] to it. Examples of more sophisticated programs in *l*CLL are described in section 5.

3.1 Introducing *l*CLL

We introduce constructs and rewrite rules in *l*CLL step by step. Informally described, our CHAM solutions consist of *f*CLL terms labeled by their types. We use the notation $\hat{\Delta}$ for such solutions.

$$\hat{\Delta} \ ::= \ \cdot \mid \hat{\Delta}, N : A \mid \hat{\Delta}, M \# S \mid \hat{\Delta}, E \ \div \ S$$

The rewrite rules on these solutions fall into three categories. The first one, called *structural rules* allow rewrite of monadic values. These rules, in general, are derived from the left rules for synchronous connectives of a sequent style presentation of FOMLL. Like their logical counterparts, they are invertible. They correspond to heat-cool rules in the CHAM terminology. However, like other well-designed CHAMs, *l*CLL uses these rules in the forward (heating) direction only. We use the symbol $\rightharpoonup$ for structural rules oriented in the forward direction.[2] The second set of rules is *functional rules* that allow in-place computation of terms using the evaluation rules for *f*CLL. These rules do not affect the types of terms as shown in lemma 8. They are not invertible and correspond to administrative moves in CHAMs. We denote functional rules using the

[2]Traditionally, the symbol $\rightleftharpoons$ is used to denote heat-cool rules in CHAMs, in order to emphasize their reversibility. We use $\rightharpoonup$ instead of $\rightleftharpoons$ to emphasize that *l*CLL uses these rules in the forward direction only.

symbol $\twoheadrightarrow$. The final set of rules is derived from left rules for asynchronous connectives of FOMLL. These rules are called *reaction rules* because of their close connection to reaction rules in CHAMs. Reaction rules are also non-invertible. They are denoted by $\longrightarrow$. We do not have any rules corresponding to right sequent rules of asynchronous connectives, because from the point of view of a programming language they correspond to synthesis of abstractions (functions) and additive conjunctions (choices). For example, consider the following typing rule.

$$\frac{\Sigma; \Gamma; \Delta, x : A; \Psi \vdash N : B}{\Sigma; \Gamma; \Delta; \Psi \vdash \hat{\lambda} x.N : A \multimap B} \multimap\text{-I}$$

If used as a rule in a proof-search, the proof-term $\hat{\lambda} x.N$ is *synthesized* by the proof-search mechanism. However, from the point of view of a programming language, this proof-term is a function whose exact behavior is not known and hence we do not use the above and similar rules in our proof-search.

3.1.1 Structural Rules for Monadic Values and Synchronous Connectives

As mentioned earlier, structural rules are derived from left rules for synchronous connectives and like their logical counterparts, they are invertible. For practical reasons, we use them in the forward direction only. The principal terms on which they apply are always monadic values. We systematically derive structural rules for all synchronous connectives by looking at the corresponding typing rules.

Multiplicative Conjunction, $(\otimes)$. Consider the left rule for tensor in FOMLL:

$$\frac{\Sigma; \Gamma; \Delta, p_1 : S_1, p_2 : S_2; \Psi \vdash \psi}{\Sigma; \Gamma; \Delta, p_1 \otimes p_2 : S_1 \otimes S_2; \Psi \vdash \psi} \otimes\text{-L}$$

This rule is invertible, as proved in lemma 3. In order to derive a CHAM rewrite rule from this rule, we substitute a monadic value M_v for $p_1 \otimes p_2$. Since the type is $S_1 \otimes S_2$, $M_v = M_{v_1} \otimes M_{v_2}$. This gives us the following structural rule:

$$\hat{\Delta}, (M_{v_1} \otimes M_{v_2}) \# (S_1 \otimes S_2) \rightharpoonup \hat{\Delta}, M_{v_1} \# S_1, M_{v_2} \# S_2$$

Linear Unit (1). Reasoning as above, we arrive at the following rule for the unit:

$$\hat{\Delta}, 1 \# 1 \rightharpoonup \hat{\Delta}$$

Additive disjunction $(\oplus)$. Consider the left rule for additive disjunction:

$$\frac{\Sigma; \Gamma; \Delta, p_1 : S_1; \Psi \vdash E_1 \div S \qquad \Sigma; \Gamma; \Delta, p_2 : S_2; \Psi \vdash E_2 \div S}{\Sigma; \Gamma; \Delta, p_1|_\zeta p_2 : S_1 \oplus S_2; \Psi \vdash E_1|_\zeta E_2 \div S} \oplus\text{-}L_E$$

From the invertibility of this rule it follows that whenever we can use $S_1 \oplus S_2$ to prove some conclusion S, we can also use either S_1 or S_2 to prove S. Operationally, this decision can be made using the actual term that has type $S_1 \oplus S_2$. If it has the form $\underline{\texttt{inl}}\ M_1$, then we use S_1 and if it has the form $\underline{\texttt{inr}}\ M_2$, we use S_2. This intuition gives us the following two rewrite rules:

$$\hat{\Delta}, \underline{\texttt{inl}}\ M_v \# (S_1 \oplus S_2) \rightharpoonup \hat{\Delta}, M_v \# S_1$$

$$\hat{\Delta}, \underline{\texttt{inr}}\ M_v \# (S_1 \oplus S_2) \rightharpoonup \hat{\Delta}, M_v \# S_2$$

Thus $\oplus$ acts as an *internal choice* operator in our language.[3]

Iso-recursive type $(\mu\alpha.S(\alpha))$**.** We use the following rule for iso-recursive types.

$$\hat{\Delta}, \underline{\texttt{fold}}(M_v) \mathrel{\#} \mu\alpha.S(\alpha) \rightharpoonup \hat{\Delta}, M_v \mathrel{\#} S(\mu\alpha.S(\alpha))$$

Existential quantification $(\exists)$**.** The left rule for existentials is:

$$\frac{\Sigma, i : \gamma; \Gamma; \Delta, p : S; \Psi \vdash \psi}{\Sigma; \Gamma; \Delta, [i, p] : \exists i : \gamma.S; \Psi \vdash \psi} \ \exists\text{-L}$$

This rule suggests that in our rewrite system we add a context Σ of index variables and the rule below. We use the symbol $\|$ to separate different kinds of contexts in CHAMs.

$$\Sigma \,\|\, \hat{\Delta}, [t, M_v] \mathrel{\#} \exists i : \gamma.S(i) \rightharpoonup \Sigma, i : \gamma \,\|\, \hat{\Delta}, M_v \mathrel{\#} S(i)$$

While attractive from a logical point of view, such a rule is not sound for a programming language. First, M_v does not have type $S(i)$. Instead it has the type $S(t)$. Thus the right hand side of this rewrite rule is "ill-formed". Second, we have completely lost the abstracted term t, which is not good from a programming perspective. The other alternative shown below is "type correct" but eliminates the abstraction over t, which is contradictory to the idea of using the $\exists$ quantifier.

$$\hat{\Delta}, [t, M_v] \mathrel{\#} \exists i : \gamma.S(i) \rightharpoonup \hat{\Delta}, M_v \mathrel{\#} S(t)$$

To correctly implement this rule, we keep the abstraction t/i in a separate context of substitutions. We denote this context by $\hat{\sigma}$.

$$\hat{\sigma} ::= \cdot \mid \hat{\sigma}, t/i : \gamma$$

Our correct rewrite rule is:

$$\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Delta}, [t, M_v] \mathrel{\#} \exists i : \gamma.S(i) \rightharpoonup \Sigma, i : \gamma \,\|\, \hat{\sigma}, t/i : \gamma \,\|\, \hat{\Delta}, M_v \mathrel{\#} S(i) \quad (i \text{ fresh})$$

If we have a configuration $\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Delta}, M_v \mathrel{\#} S$, then M_v has the type $S[\hat{\sigma}]$, where $S[\hat{\sigma}]$ is the result of applying the substitution $\hat{\sigma}$ to S. Since nested existentials may be present, the i chosen at each step is fresh. An important invariant we maintain while evaluating *l*CLL CHAM configurations is that terms, monadic terms and expressions in CHAM solutions are always closed under $\hat{\sigma}$, i.e. if $T \mathrel{\%} Z \in \hat{\Delta}$ in a CHAM configuration $\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Delta}$, then $\texttt{fv}(T) \cap dom(\hat{\sigma}) = \phi$. Thus the substitution $\hat{\sigma}$ is meant to be applied only to types, not to proof-terms.

Exponential $(!)$**.** Consider the left rule for exponentials:

$$\frac{\Sigma; \Gamma, x : A; \Delta; \Psi \vdash \psi}{\Sigma; \Gamma; \Delta, !x :!A; \Psi \vdash \psi} \ !\text{-L}$$

This rule suggests that we introduce a new solution $\hat{\Gamma}$ of unrestricted values i.e. values that may be used any number of times. Due to typing restrictions in *f*CLL, all such values must have asynchronous types.

$$\hat{\Gamma} ::= \cdot \mid \hat{\Gamma}, V : A$$

[3] The proof-terms $\underline{\texttt{inl}}\ M_v$ and $\underline{\texttt{inr}}\ M_v$ play an important role in the use of these rules. In linear logic without proof-terms, it is not possible to conclude S_1 and S_2 from an assumption $S_1 \oplus S_2$.

The corresponding rewrite rule is:

$$\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, !V \# !A \rightharpoonup \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma}, V : A \parallel \hat{\Delta}$$

Type Ascription. Once an expression evaluates to a monadic value or a monadic term of type A evaluates to a value, we need to be able to change its type ascription in order to evaluate further. This is achieved by the following rules.

$$\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, V \# A \rightharpoonup \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, V : A$$

$$\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, M_v \div S \rightharpoonup \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, M_v \# S$$

3.1.2 Functional Rules for In-place Computation

We also need some rules to allow evaluation of terms and expressions to reduce them to values. One such rule is the following:

$$\frac{N \rightarrow N'}{\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, N : A \twoheadrightarrow \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, N' : A}$$

The type preservation lemma (lemma 8) guarantees that if N has type A, then so does N'. The remaining rules in this category are:

$$\frac{M \mapsto M'}{\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, M \# S \twoheadrightarrow \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, M' \# S}$$

$$\frac{E \hookrightarrow E'}{\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, E \div S \twoheadrightarrow \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, E' \div S}$$

3.1.3 Summary of Structural and Functional Rules

We summarize all the structural and functional rules in figure 14.

3.1.4 Reaction Rules for Term Values and Asynchronous Connectives

Consider a sequent style presentation of the asynchronous connectives of FOMLL ($\&$, $\multimap$, $\rightarrow$, $\forall$). The left or elimination rules for such a presentation are given in figure 15. Using these rules, we can derive one possible set of CHAM reaction rules for *l*CLL, as given below. These logic rules are not invertible and the corresponding CHAM rules are irreversible. As we shall see, these rules are too general to be useful for concurrent programming, and we will replace them with a different set of rules later.

$$\begin{array}{lcl}
\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma}, V : A \parallel \hat{\Delta} & \longrightarrow & \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma}, V : A \parallel \hat{\Delta}, V : A \\
\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, \langle N_1, N_2 \rangle : A_1 \& A_2 & \longrightarrow & \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, N_1 : A_1 \\
\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, \langle N_1, N_2 \rangle : A_1 \& A_2 & \longrightarrow & \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, N_2 : A_2 \\
\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, \{E\} : \{S\} & \longrightarrow & \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, E \div S \\
\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, V_1 : A \multimap B, V_2 : A & \longrightarrow & \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, V_1 \,\hat{}\, V_2 : B \\
\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma}, V_2 : A \parallel \hat{\Delta}, V_1 : A \rightarrow B & \longrightarrow & \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma}, V_2 : A \parallel \hat{\Delta}, V_1\ V_2 : B
\end{array}$$

$$\frac{\Sigma \vdash t : \gamma}{\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, N : \forall i : \gamma . A(i) \longrightarrow \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, N\ [t] : A(t)}$$

CHAM solutions

$$
\begin{array}{lcl}
\hat{\Delta} & ::= & \cdot \mid \hat{\Delta}, N : A \mid \hat{\Delta}, M \mathrel{\#} S \mid \hat{\Delta}, E \div S \\
\hat{\Gamma} & ::= & \cdot \mid \hat{\Gamma}, V : A \\
\hat{\sigma} & ::= & \cdot \mid \hat{\sigma}, t/i : \gamma
\end{array}
$$

CHAM configurations

$$\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}$$

Structural rules, $\boldsymbol{\Sigma} \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\boldsymbol{\Gamma}} \mathbin{\boldsymbol{|}} \hat{\boldsymbol{\Delta}} \rightharpoonup \boldsymbol{\Sigma} \mathbin{\boldsymbol{|}} \hat{\sigma}' \mathbin{\boldsymbol{|}} \hat{\boldsymbol{\Gamma}}' \mathbin{\boldsymbol{|}} \hat{\boldsymbol{\Delta}}'$

$$
\begin{array}{rcll}
\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, (M_{v_1} \otimes M_{v_2}) \mathrel{\#} (S_1 \otimes S_2) & \rightharpoonup & \Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, M_{v_1} \mathrel{\#} S_1, M_{v_2} \mathrel{\#} S_2 & (\rightharpoonup -\otimes) \\
\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, 1 \mathrel{\#} 1 & \rightharpoonup & \Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta} & (\rightharpoonup -1) \\
\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, \underline{\mathtt{inl}}\ M_v \mathrel{\#} (S_1 \oplus S_2) & \rightharpoonup & \Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, M_v \mathrel{\#} S_1 & (\rightharpoonup -\oplus_1) \\
\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, \underline{\mathtt{inr}}\ M_v \mathrel{\#} (S_1 \oplus S_2) & \rightharpoonup & \Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, M_v \mathrel{\#} S_2 & (\rightharpoonup -\oplus_2) \\
\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, \underline{\mathtt{fold}}(M_v) \mathrel{\#} \mu\alpha.S(\alpha) & \rightharpoonup & \Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, M_v \mathrel{\#} S(\mu\alpha.S(\alpha)) & (\rightharpoonup -\mu) \\
\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, [t, M_v] \mathrel{\#} \exists i : \gamma.S(i) & \rightharpoonup & \Sigma, i : \gamma \mathbin{\boldsymbol{|}} \hat{\sigma}, t/i : \gamma \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, M_v \mathrel{\#} S(i) & (\rightharpoonup -\exists) \\
\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, !V \mathrel{\#} !A & \rightharpoonup & \Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma}, V : A \mathbin{\boldsymbol{|}} \hat{\Delta} & (\rightharpoonup -!) \\
\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, V \mathrel{\#} A & \rightharpoonup & \Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, V : A & (\rightharpoonup - \# :) \\
\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, M_v \div S & \rightharpoonup & \Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, M_v \mathrel{\#} S & (\rightharpoonup - \div \#)
\end{array}
$$

Functional rules, $\boldsymbol{\Sigma} \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\boldsymbol{\Gamma}} \mathbin{\boldsymbol{|}} \hat{\boldsymbol{\Delta}} \twoheadrightarrow \boldsymbol{\Sigma}' \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\boldsymbol{\Gamma}} \mathbin{\boldsymbol{|}} \hat{\boldsymbol{\Delta}}'$

$$
\frac{N \rightarrow N'}{\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, N : A \twoheadrightarrow \Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, N' : A} \twoheadrightarrow - \rightarrow
$$

$$
\frac{M \mapsto M'}{\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, M \mathrel{\#} S \twoheadrightarrow \Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, M' \mathrel{\#} S} \twoheadrightarrow - \mapsto
$$

$$
\frac{E \hookrightarrow E'}{\Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, E \div S \twoheadrightarrow \Sigma \mathbin{\boldsymbol{|}} \hat{\sigma} \mathbin{\boldsymbol{|}} \hat{\Gamma} \mathbin{\boldsymbol{|}} \hat{\Delta}, E' \div S} \twoheadrightarrow - \hookrightarrow
$$

Figure 14: Structural and Functional rules for the CHAM

Contexts

$$\begin{array}{lcl} \Gamma & ::= & \cdot \mid \Gamma, A \\ \Delta & ::= & \cdot \mid \Delta, A \mid \Delta, S \end{array}$$

$$\psi \;::=\; S \mid A$$

$\boldsymbol{\Sigma}; \boldsymbol{\Gamma}; \boldsymbol{\Delta} \;\rightarrow\; \psi$

$$\frac{}{\Sigma; \Gamma; A \;\rightarrow\; A} \qquad \frac{\Sigma; \Gamma, A; \Delta, A \;\rightarrow\; \psi}{\Sigma; \Gamma, A; \Delta \;\rightarrow\; \psi}$$

$$\frac{\Sigma; \Gamma; \Delta, A \;\rightarrow\; \psi}{\Sigma; \Gamma; \Delta, A \& B \;\rightarrow\; \psi} \qquad \frac{\Sigma; \Gamma; \Delta, B \;\rightarrow\; \psi}{\Sigma; \Gamma; \Delta, A \& B \;\rightarrow\; \psi}$$

$$\frac{\Sigma; \Gamma; \Delta_1 \;\rightarrow\; A \qquad \Sigma; \Gamma; \Delta_2, B \;\rightarrow\; \psi}{\Sigma; \Gamma; \Delta_1, \Delta_2, A \multimap B \;\rightarrow\; \psi}$$

$$\frac{\Sigma; \Gamma; \cdot \;\rightarrow\; A \qquad \Sigma; \Gamma; \Delta, B \;\rightarrow\; \psi}{\Sigma; \Gamma; \Delta, A \rightarrow B \;\rightarrow\; \psi}$$

$$\frac{\Sigma \;\vdash\; t : \gamma \qquad \Sigma; \Gamma; \Delta, A(t) \;\rightarrow\; \psi}{\Sigma; \Gamma; \Delta, \forall i : \gamma. A(i) \;\rightarrow\; \psi} \qquad \frac{\Sigma; \Gamma; \Delta, S \;\rightarrow\; S'}{\Sigma; \Gamma; \Delta, \{S\} \;\rightarrow\; S'}$$

Figure 15: Sequent calculus left rules for asynchronous connectives

These simple rewrite rules, however, allow too much non-determinism to be useful in a programming language. For example, consider the following configuration which uses an index refinement by sort γ on the types A and B:

$$k : \gamma, m : \gamma \parallel \cdot \parallel \cdot \parallel V_1 : A(k), V_2 : (\forall i : \gamma. A(i) \multimap \{B(i)\})$$

We expect the reaction to go as follows:

$$\begin{array}{ll} & k : \gamma, m : \gamma \parallel \cdot \parallel \cdot \parallel V_1 : A(k), V_2 : (\forall i : \gamma. A(i) \multimap \{B(i)\}) \\ \longrightarrow & k : \gamma, m : \gamma \parallel \cdot \parallel \cdot \parallel V_1 : A(k), V_2\,[k] : A(k) \multimap \{B(k)\} \\ \twoheadrightarrow^* & k : \gamma, m : \gamma \parallel \cdot \parallel \cdot \parallel V_1 : A(k), V_2' : A(k) \multimap \{B(k)\} \\ \longrightarrow & k : \gamma, m : \gamma \parallel \cdot \parallel \cdot \parallel V_2' \ \hat{}\ V_1 : \{B(k)\} \end{array}$$

However, there is another possible sequence of reactions that gets stuck:

$$\begin{array}{ll} & k : \gamma, m : \gamma \parallel \cdot \parallel \cdot \parallel V_1 : A(k), V_2 : (\forall i : \gamma. A(i) \multimap \{B(i)\}) \\ \longrightarrow & k : \gamma, m : \gamma \parallel \cdot \parallel \cdot \parallel V_1 : A(k), V_2\,[m] : A(m) \multimap \{B(m)\} \\ \twoheadrightarrow^* & k : \gamma, m : \gamma \parallel \cdot \parallel \cdot \parallel V_1 : A(k), V_2'' : A(m) \multimap \{B(m)\} \end{array}$$

At this point the CHAM configuration cannot react further. This problem has occurred because we chose a wrong term (m) to instantiate the universal quantifier at the first step. In order to rectify this situation, we use chaining of reactions i.e. a "look-ahead" to see what reactions to perform. One important problem at this stage is to decide how much look-ahead to perform. At one extreme, we have seen that performing no look-ahead at all is not a good forward chaining strategy. The other possible extreme is to perform complete look-ahead, i.e. to predict all reaction steps needed to reach the desired goal. Unfortunately, this problem is undecidable. Therefore, we choose a solution which lies between the two extremes. We let each reaction chain correspond to exactly one *focusing* step in a focused theorem prover for our underlying logic.

Focusing in proof-search [3] was introduced as a method of reducing the search space of proofs. Traditionally, focusing is used in backwards proof-search only. However it is possible to use focusing in a forward chaining procedure (see for example [14]). Here we will try to combine focusing and forward reasoning to obtain reaction rules for *l*CLL CHAM configurations that do not have the problem mentioned above. We will start by combining forward chaining and focusing for FOMLL without proof-terms. Later we will add proof-terms to our rules to obtain the reaction rules for *l*CLL.

Focusing with forward chaining in FOMLL. In a forward chaining procedure new facts are concluded from known facts based on some rules. At any point of time in a forward chaining procedure, a certain set of linear facts and a certain set of unrestricted facts are known to be true. We denote these using the conventional notation Δ and Γ respectively.

$$\begin{array}{lcl} \Delta & ::= & \cdot \mid \Delta, A \mid \Delta, S \\ \Gamma & ::= & \cdot \mid \Gamma, A \end{array}$$

Since we are working at first order, the known facts can be parametric in some indexes. We record these indexes explicitly in a context of parameters, Σ.

$$\Sigma ::= \cdot \mid \Sigma, i : \gamma$$

We represent the facts known at any time using the notation $\Sigma; \Gamma; \Delta$. Now the principal judgment in a forward chaining procedure is a rewrite judgment $\Sigma; \Gamma; \Delta \;\rightarrow\; \Sigma'; \Gamma'; \Delta'$, which means that given the facts $\Sigma; \Gamma; \Delta$, we can conclude the facts $\Sigma'; \Gamma'; \Delta'$. In this judgment, Σ', Γ' and Δ' are *outputs*. As it turns out, since we are dealing only with asynchronous connectives here, Σ and Γ do not change in this judgment. Thus we can write this judgment more explicitly as $\Sigma; \Gamma; \Delta \;\rightarrow\; \Sigma; \Gamma; \Delta'$. We have already seen some examples of rules of this judgment(with proof-terms) earlier. We abandoned these rules because they are ineffective for linking programs. The rules we saw earlier are reproduced below without proof-terms.

$$\begin{array}{lcl}
\Sigma; \Gamma; \Delta, A_1 \& A_2 & \rightarrow & \Sigma; \Gamma; \Delta, A_1 \\
\Sigma; \Gamma; \Delta, A_1 \& A_2 & \rightarrow & \Sigma; \Gamma; \Delta, A_2 \\
\Sigma; \Gamma; \Delta, A \multimap B, A & \rightarrow & \Sigma; \Gamma; \Delta, B \\
\Sigma; \Gamma, A; \Delta, A \rightarrow B & \rightarrow & \Sigma; \Gamma, A; \Delta, B
\end{array}$$

$$\frac{\Sigma \;\vdash\; t : \gamma}{\Sigma; \Gamma; \Delta, \forall i : \gamma.A(i) \;\rightarrow\; \Sigma; \Gamma; \Delta, A(t)}$$

As we saw, these rules are too general in the sense that they allow too many possible computations, all of which are not desirable. Going back to our example of the computation that got stuck, we observe that we wanted to forward chain the two types $A(k)$ and $\forall i : \gamma.A(i) \multimap \{B(i)\}$ to conclude $\{B(k)\}$. This required instantiation of the second type with k to obtain $A(k) \multimap \{B(k)\}$ and then an elimination of $\multimap$ to obtain $\{B(k)\}$. However, using the rules presented above, we could also instantiate $\forall i : \gamma.A(i) \multimap \{B(i)\}$ with m instead of k and reach a deadlock. As we noticed, we need to perform a look-ahead, or a certain amount of reasoning to decide that we have to instantiate the second type with k, not m. This kind of look-ahead can be done using focusing. Rather than arbitrarily selecting m or k to instantiate $\forall i : \gamma.A(i) \multimap \{B(i)\}$, we begin a *focus* on $\forall i : \gamma.A(i) \multimap \{B(i)\}$. Once this type is under a focus, we perform *backwards reasoning with backtracking* on this type to decide what to do till we have either successfully concluded a fact, or we have exhausted all possibilities.

We present a focused forward chaining procedure for FOMLL (without proof-terms) in two steps. In the first step, we present some focusing rules that allow us to conclude a *single* fact of the form $\{S\}$ from a set of facts $\Sigma; \Gamma; \Delta$. This judgment is written $\Sigma; \Gamma; \Delta \;\rightarrow\; \{S\}$. In the second step, we modify some of these rules to obtain a focusing forward chaining procedure for FOMLL. The principal judgment of this procedure is as mentioned before - $\Sigma; \Gamma; \Delta \;\rightarrow\; \Sigma; \Gamma; \Delta'$.

Concluding single facts with forward chaining in FOMLL. We begin with the first step i.e. we present focusing rules that allow us to conclude a single fact of the form $\{S\}$ from a set of facts $\Sigma; \Gamma; \Delta$. This process consumes the facts in Δ. Figures 16 and 17 show four related judgments. All rules in these figures are used backwards. The rules in figure 16 will later be replaced by a new set of rules to obtain a focused forward chaining procedure for FOMLL.

The principal judgment in figure 16 is $\Sigma; \Gamma; \Delta \;\rightarrow\; \{S\}$. We read this judgment as follows - "if we can deduce $\Sigma; \Gamma; \Delta \;\rightarrow\; \{S\}$ from the rules in figures 16 and 17 using backward reasoning, then in a forward chaining procedure we can conclude the linear fact $\{S\}$ from the unrestricted facts in Γ and the linear facts in Δ". Thus this judgment allows us to combine forward and backward reasoning. The proposition $\{S\}$ is an output of this judgment. In order to deduce $\Sigma; \Gamma; \Delta \;\rightarrow\; \{S\}$, we must first focus on a fact in either Γ or Δ. This is done using one of the two rules that conclude $\Sigma; \Gamma; \Delta \;\rightarrow\; \{S\}$. Once we have focused on a formula, we move to the second judgment in figure 16 - $\Sigma; \Gamma; \Delta; A \;\Rightarrow\; \{S\}$. The proposition A at the

Contexts

$$\begin{array}{lcl} \Gamma & ::= & \cdot \mid \Gamma, A \\ \Delta & ::= & \cdot \mid \Delta, A \mid \Delta, S \end{array}$$

$\mathbf{\Sigma; \Gamma; \Delta \ \rightarrow \ \{S\}}$ **(Inputs:** Σ, Γ, Δ**; Output:** $\{S\}$**)**

$$\frac{\Sigma; \Gamma; \Delta; A \ \Rightarrow \ \{S\}}{\Sigma; \Gamma; \Delta, A \ \rightarrow \ \{S\}} \rightarrow -1 \qquad \frac{\Sigma; \Gamma, A; \Delta; A \ \Rightarrow \ \{S\}}{\Sigma; \Gamma, A; \Delta \ \rightarrow \ \{S\}} \rightarrow -2$$

$\mathbf{\Sigma; \Gamma; \Delta; A \ \Rightarrow \ \{S\}}$ **(Inputs:** $\Sigma, \Gamma, \Delta, A$**; Output:** $\{S\}$**)**

$$\frac{}{\Sigma; \Gamma; \cdot; \{S\} \ \Rightarrow \ \{S\}} \Rightarrow -HYP \qquad \frac{\Sigma \ \vdash \ t : \gamma \qquad \Sigma; \Gamma; \Delta; A(t) \ \Rightarrow \ \{S\}}{\Sigma; \Gamma; \Delta; \forall i : \gamma . A(i) \ \Rightarrow \ \{S\}} \Rightarrow -\forall$$

$$\frac{\Sigma; \Gamma; \Delta; A_1 \ \Rightarrow \ \{S\}}{\Sigma; \Gamma; \Delta; A_1 \& A_2 \ \Rightarrow \ \{S\}} \Rightarrow -\&_1 \qquad \frac{\Sigma; \Gamma; \Delta; A_2 \ \Rightarrow \ \{S\}}{\Sigma; \Gamma; \Delta; A_1 \& A_2 \ \Rightarrow \ \{S\}} \Rightarrow -\&_2$$

$$\frac{\Sigma; \Gamma; \Delta_1 \ \rightarrow_{\mathcal{A}} \ P \qquad \Sigma; \Gamma; \Delta_2; B \ \Rightarrow \ \{S\}}{\Sigma; \Gamma; \Delta_1, \Delta_2; P \multimap B \ \Rightarrow \ \{S\}} \Rightarrow - \multimap$$

$$\frac{\Sigma; \Gamma; \cdot \ \rightarrow_{\mathcal{A}} \ P \qquad \Sigma; \Gamma; \Delta; B \ \Rightarrow \ \{S\}}{\Sigma; \Gamma; \Delta; P \rightarrow B \ \Rightarrow \ \{S\}} \Rightarrow - \rightarrow$$

Figure 16: Focused left rules for asynchronous connectives (Part I)

$\mathbf{\Sigma; \Gamma; \Delta} \ \rightarrow_{\mathcal{A}} \ \mathbf{P}$ **(Inputs:** $\Sigma, \Gamma, \Delta, P$**; Outputs: None)**

$$\frac{\Sigma; \Gamma; \Delta; A \ \Rightarrow_{\mathcal{A}} \ P}{\Sigma; \Gamma; \Delta, A \ \rightarrow_{\mathcal{A}} \ P} \rightarrow_{\mathcal{A}} -1 \qquad \frac{\Sigma; \Gamma, A; \Delta; A \ \Rightarrow_{\mathcal{A}} \ P}{\Sigma; \Gamma, A; \Delta \ \rightarrow_{\mathcal{A}} \ P} \rightarrow_{\mathcal{A}} -2$$

$\mathbf{\Sigma; \Gamma; \Delta; A} \ \Rightarrow_{\mathcal{A}} \ \mathbf{P}$ **(Inputs:** $\Sigma, \Gamma, \Delta, A, P$**; Outputs: None)**

$$\frac{}{\Sigma; \Gamma; \cdot; P \ \Rightarrow_{\mathcal{A}} \ P} \Rightarrow_{\mathcal{A}} -HYP \qquad \frac{\Sigma \ \vdash \ t : \gamma \qquad \Sigma; \Gamma; \Delta; A(t) \ \Rightarrow_{\mathcal{A}} \ P}{\Sigma; \Gamma; \Delta; \forall i : \gamma . A(i) \ \Rightarrow_{\mathcal{A}} \ P} \Rightarrow_{\mathcal{A}} -\forall$$

$$\frac{\Sigma; \Gamma; \Delta; A_1 \ \Rightarrow_{\mathcal{A}} \ P}{\Sigma; \Gamma; \Delta; A_1 \& A_2 \ \Rightarrow_{\mathcal{A}} \ P} \Rightarrow_{\mathcal{A}} -\&_1 \qquad \frac{\Sigma; \Gamma; \Delta; A_2 \ \Rightarrow_{\mathcal{A}} \ P}{\Sigma; \Gamma; \Delta; A_1 \& A_2 \ \Rightarrow_{\mathcal{A}} \ P} \Rightarrow_{\mathcal{A}} -\&_2$$

$$\frac{\Sigma; \Gamma; \Delta_1 \ \rightarrow_{\mathcal{A}} \ P' \qquad \Sigma; \Gamma; \Delta_2; B \ \Rightarrow_{\mathcal{A}} \ P}{\Sigma; \Gamma; \Delta_1, \Delta_2; P' \multimap B \ \Rightarrow_{\mathcal{A}} \ P} \Rightarrow_{\mathcal{A}} - \multimap$$

$$\frac{\Sigma; \Gamma; \cdot \ \rightarrow_{\mathcal{A}} \ P' \qquad \Sigma; \Gamma; \Delta; B \ \Rightarrow_{\mathcal{A}} \ P}{\Sigma; \Gamma; \Delta; P' \rightarrow B \ \Rightarrow_{\mathcal{A}} \ P} \Rightarrow_{\mathcal{A}} - \rightarrow$$

Figure 17: Focused left rules for asynchronous connectives (Part II)

end of the three contexts is the formula in focus. $\{S\}$ is an output in this judgment also. In this judgment, we keep eliminating the top level connective of the formula in focus till we are left with a single formula of the form $\{S\}$. This formula $\{S\}$ becomes the output of the judgment. If this does not happen, we must backtrack and find some other sequence of eliminations to apply.

In order to eliminate an implication ($\multimap$ or $\rightarrow$), we have to show that the argument of the implication is provable. This requires the introduction of the auxiliary judgments shown in figure 17. They are $\Sigma; \Gamma; \Delta \;\rightarrow_{\mathcal{A}}\; P$ and $\Sigma; \Gamma; \Delta; A \;\Rightarrow_{\mathcal{A}}\; P$. The symbol $\mathcal{A}$ in the subscript of $\Rightarrow_{\mathcal{A}}$ and $\rightarrow_{\mathcal{A}}$ stands for *auxiliary*. These judgments are exactly like those in figure 16 with three differences. First, the conclusion is an *atomic proposition* P instead of $\{S\}$. The proposition must be atomic because as mentioned earlier, our backwards reasoning does not use any right rules. Second, these judgments are not principal; even if we can conclude $\Sigma; \Gamma; \Delta \;\rightarrow_{\mathcal{A}}\; P$, the forward chaining procedure cannot conclude P from the unrestricted facts Γ and the linear facts Δ. The judgments of figure 17 can be used only to prove that the formula needed in the argument of an implication actually holds. Third, the conclusion of the sequent, P, is an input in the auxiliary judgments. On the other hand, the conclusion $\{S\}$ is an output in the judgments of figure 16.

There are some remarks to be made here. The principal judgment $\Sigma; \Gamma; \Delta \;\rightarrow\; \{S\}$ always has a fact of the type $\{S\}$ in the conclusion. Thus the forward chaining procedure always concludes facts of the form $\{S\}$ when it uses focusing. Further, backward search never eliminates a monad in focus. Thus the monad is also the constructor where backwards reasoning stops. Having such a clear demarcation of where backward reasoning stops is essential in writing correct programs.

Forward chaining rules for FOMLL. So far we have seen how we can combine focusing with forward chaining to successfully conclude a *single* fact from a number of given facts. We now come to the second step. We use the rules in figures 16 and 17 to obtain a forward chaining procedure for FOMLL. As mentioned earlier, the principal judgment we want to obtain is $\Sigma; \Gamma; \Delta \;\rightarrow\; \Sigma; \Gamma; \Delta'$. This judgment is to be read as follows - "given the parametric index assumptions in Σ and the unrestricted facts Γ, we can conclude the linear facts Δ' from the linear facts Δ". We already know how to conclude a single fact from a given set of facts. Now we allow this deduction to occur in any arbitrary context. We want to say that if $\Sigma; \Gamma; \Delta \;\rightarrow\; \{S\}$, then we can conclude $\Sigma, \Sigma''; \Gamma, \Gamma''; \{S\}, \Delta''$ from the facts $\Sigma, \Sigma''; \Gamma, \Gamma''; \Delta, \Delta''$ for arbitrary Σ'', Γ'' and Δ''. We can integrate this closure under contexts directly into the backwards search rules. To do that, we reformulate the rules of figure 16. These modified rules are shown in figure 18. A step-by-step explanation of the transformation of rules is given below.

We begin by changing the judgment $\Sigma; \Gamma; \Delta \;\rightarrow\; \{S\}$ to $\Sigma; \Gamma; \Delta \;\rightarrow\; \Sigma; \Gamma; \Delta'$. There are two rules to derive this new judgment, both of which are shown in figure 18. This judgment is the principal forward chaining judgment for FOMLL and the context Δ' is an output. Next we revise the judgment $\Sigma; \Gamma; \Delta; A \;\Rightarrow\; \{S\}$. We change this to $\Sigma; \Gamma; \Delta; A \;\Rightarrow\; \Sigma; \Gamma; \Delta'$. We do this one by one for all the rules. In the rule $\Rightarrow -HYP$ (see figure 16), we conclude $\{S\}$ if we have focused on $\{S\}$. Since we are now implementing a forward chaining rewrite judgment, we can change this to the unconditional rewrite rule $\Sigma; \Gamma; \Delta; \{S\} \;\Rightarrow\; \Sigma; \Gamma; \Delta, \{S\}$. In the rule $\Rightarrow -\forall$ (figure 16), we instantiate a universally quantified term in focus as we move from the conclusion of the rule to the premises and the right hand side of the sequents in the second premise and conclusion is the same. This gives us the following revised rule:

$$\frac{\Sigma \vdash t : \gamma \qquad \Sigma; \Gamma; \Delta; A(t) \;\Rightarrow\; \Sigma; \Gamma; \Delta'}{\Sigma; \Gamma; \Delta; \forall i : \gamma . A(i) \;\Rightarrow\; \Sigma; \Gamma; \Delta'} \Rightarrow -\forall$$

Contexts

$$\begin{array}{lcl} \Gamma & ::= & \cdot \mid \Gamma, A \\ \Delta & ::= & \cdot \mid \Delta, A \mid \Delta, S \end{array}$$

$\mathbf{\Sigma; \Gamma; \Delta \ \rightarrow \ \Sigma; \Gamma; \Delta'}$ **(Inputs:** Σ, Γ, Δ**; Output:** Δ'**)**

$$\frac{\Sigma; \Gamma; \Delta; A \ \Rightarrow \ \Sigma; \Gamma; \Delta'}{\Sigma; \Gamma; \Delta, A \ \rightarrow \ \Sigma; \Gamma; \Delta'} \rightarrow -1 \qquad \frac{\Sigma; \Gamma, A; \Delta; A \ \Rightarrow \ \Sigma; \Gamma; \Delta'}{\Sigma; \Gamma, A; \Delta \ \rightarrow \ \Sigma; \Gamma; \Delta'} \rightarrow -2$$

$\mathbf{\Sigma; \Gamma; \Delta; A \ \Rightarrow \ \Sigma; \Gamma; \Delta'}$ **(Inputs:** $\Sigma, \Gamma, \Delta, A$**; Output:** Δ'**)**

$$\frac{}{\Sigma; \Gamma; \Delta; \{S\} \ \Rightarrow \ \Sigma; \Gamma; \Delta, \{S\}} \Rightarrow -HYP \qquad \frac{\Sigma \ \vdash \ t : \gamma \qquad \Sigma; \Gamma; \Delta; A(t) \ \Rightarrow \ \Sigma; \Gamma; \Delta'}{\Sigma; \Gamma; \Delta; \forall i : \gamma. A(i) \ \Rightarrow \ \Sigma; \Gamma; \Delta'} \Rightarrow -\forall$$

$$\frac{\Sigma; \Gamma; \Delta; A_1 \ \Rightarrow \ \Sigma; \Gamma; \Delta'}{\Sigma; \Gamma; \Delta; A_1 \& A_2 \ \Rightarrow \ \Sigma; \Gamma; \Delta'} \Rightarrow -\&_1 \qquad \frac{\Sigma; \Gamma; \Delta; A_2 \ \Rightarrow \ \Sigma; \Gamma; \Delta'}{\Sigma; \Gamma; \Delta; A_1 \& A_2 \ \Rightarrow \ \Sigma; \Gamma; \Delta'} \Rightarrow -\&_2$$

$$\frac{\Sigma; \Gamma; \Delta_1 \ \rightarrow_{\mathcal{A}} \ P \qquad \Sigma; \Gamma; \Delta_2; B \ \Rightarrow \ \Sigma; \Gamma; \Delta'}{\Sigma; \Gamma; \Delta_1, \Delta_2; P \multimap B \ \Rightarrow \ \Sigma; \Gamma; \Delta'} \Rightarrow - \multimap$$

$$\frac{\Sigma; \Gamma; \cdot \ \rightarrow_{\mathcal{A}} \ P \qquad \Sigma; \Gamma; \Delta; B \ \Rightarrow \ \Sigma; \Gamma; \Delta'}{\Sigma; \Gamma; \Delta; P \rightarrow B \ \Rightarrow \ \Sigma; \Gamma; \Delta'} \Rightarrow - \rightarrow$$

Figure 18: Judgments $\rightarrow$ and $\Rightarrow$ for forward chaining in FOMLL

The rules $\Rightarrow \ -\&_1$ and $\Rightarrow \ -\&_2$ can be modified similarly (see figure 18). For the rules $\Rightarrow \ - \ \multimap$ and $\Rightarrow - \rightarrow$, we need to replace the $\{S\}$ in the right hand sides of the sequents by $\Sigma; \Gamma; \Delta'$. In this manner we can revise the entire system in figure 16 and obtain the system in figure 18. The judgments $\rightarrow_{\mathcal{A}}$ and $\Rightarrow_{\mathcal{A}}$ in figure 17 are auxiliary and do not change.

In summary, the rules of figures 17 and 18 are focused rewrite rules for a forward chaining procedure for FOMLL. The principal judgment is $\Sigma; \Gamma; \Delta \ \rightarrow \ \Sigma; \Gamma; \Delta'$ (figure 18). It is used as follows. If using backward reasoning we can conclude the judgment $\Sigma; \Gamma; \Delta \ \rightarrow \ \Sigma; \Gamma; \Delta'$, then in a forward chaining procedure for FOMLL, we can conclude the linear facts Δ' from the linear facts Δ, if the unrestricted context has the the facts Γ. Further, this conclusion is parametric in the assumptions in Σ. As remarked earlier, the backward search procedure constructs the set of facts Δ'. We now augment these rules with proof-terms to obtain reaction rules for CHAM configurations.

Reaction rules for CHAMs. We augment the rules in figures 18 and 17 with proof-terms to obtain focused reaction rules for *l*CLL-CHAMs. These new rules are shown in figures 19 and 20. The judgments $\longrightarrow$, $\Longrightarrow$, $\longrightarrow_{\mathcal{A}}$ and $\Longrightarrow_{\mathcal{A}}$ are obtained by adding proof-terms to the judgments $\rightarrow$, $\Rightarrow$, $\rightarrow_{\mathcal{A}}$ and $\Rightarrow_{\mathcal{A}}$ respectively. We also add the context of substitutions $\hat{\sigma}$ to all our judgments. This process is straightforward. As an illustration, we explain some of the rules. In the rule $\Longrightarrow -\&_1$, we have a focus on a formula $A_1 \& A_2$, whose proof-term is N. As we reason backwards, we replace $A_1 \& A_2$ by A_1 and its witness N by $\pi_1\ N$, which is a proof of A_1. In the rule $\Longrightarrow -\forall$, we instantiate the proof N of $\forall i : \gamma.A(i)$ by a concrete index term. Since we assumed earlier that index variables in the domain of $\hat{\sigma}$ must not occur in terms inside configurations, we instantiate N with $t[\hat{\sigma}]$ instead of t.[4] Observe that $N\ [t[\hat{\sigma}]]$ has the type $A(t)[\hat{\sigma}]$ if N has the type $\forall i : \gamma.A(i)$ and $t : \gamma$. The rule $\longrightarrow -\{\}$ is a new rule to eliminate the monadic constructor, as mentioned earlier. It is instructive to compare figures 18 and 17 with figures 19 and 20 respectively.

As for the case of FOMLL, these rules are conditional rewrite rules for CHAM configurations that use backward reasoning and focusing. The principal judgment here is $\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \longrightarrow \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}'$. The interpretation is the same as before - if we can conclude the judgment $\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \longrightarrow \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}'$ using backward reasoning, then the CHAM configuration $\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$ rewrites to the configuration $\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}'$ using a single reaction step.

The type P in the judgment $\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \ \Longrightarrow_{\mathcal{A}}\ N : P$ must be atomic because we do not use right rules for asynchronous connectives in our proof-search. One consequence of this is that all arguments passed to functions during the focusing steps have atomic types i.e. in the rules $\Longrightarrow - \multimap$ and $\Longrightarrow - \rightarrow$, the term N_1 is forced to have an atomic type P (see figure 19). In order to pass values of other types to functions during the linking steps, the values must be abstracted to atomic types. This requires an extension of the language with suitable primitives like datatypes as in ML.[5]

This completes our discussion of rewrite rules for *l*CLL CHAMs. In summary, there are three types of rewrite rules in *l*CLL: structural ($\rightharpoonup$), functional ($\twoheadrightarrow$) and reaction ($\longrightarrow$). Structural and functional rules are shown in figure 14. Reaction rules require some backward reasoning. They are shown in figures 19 and 20.

3.2 Programming Technique: Creating Private Names

We illustrate here how to use the existential quantifier to create fresh names. Suppose we have a *constant* $c_\gamma : \gamma$ in the sort γ. Now consider a typed monadic term of the form $[c_\gamma, M(c_\gamma)] \# \exists i : \gamma.S(i)$. If this term is put in a solution, the only way it rewrites is using the rule $\rightharpoonup -\exists$:

$$\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, [c_\gamma, M(c_\gamma)] \# \exists i : \gamma.S(i) \ \rightharpoonup\ \Sigma, k : \gamma \parallel \hat{\sigma}, c_\gamma/k : \gamma \parallel \hat{\Gamma} \parallel \hat{\Delta}, M(c_\gamma) \# S(k)$$

In the type $S(i)$ on the right hand side, c_γ has been abstracted by k which is a fresh name by the side condition on this rewrite rule. In effect, we have created a fresh name k of sort γ. We can use this mechanism to create more private names using the same name c_γ for the index term again and again. Based on this idea, we define a new language construct as follows.

```
priv k:γ in M # S(k) =
    [c_γ, M[c_γ/k] ] # ∃i:γ.S(i)
```

[4]Since this rule is used backward, t is determined through unification in a practical implementation. Thus in practice this rule is implemented as follows. We replace i with a unification variable X to obtain the type $A(X)$. Unification on types determines the exact index term t that substitutes X. Then we instantiate N with $t[\hat{\sigma}]$.

[5]Datatypes can also be implemented by introducing existential, recursive and sum types at the level of pure terms.

$\mathbf{\Sigma} \,|\, \hat{\sigma} \,|\, \hat{\mathbf{\Gamma}} \,|\, \hat{\mathbf{\Delta}} \;\longrightarrow\; \mathbf{\Sigma} \,|\, \hat{\sigma} \,|\, \hat{\mathbf{\Gamma}} \,|\, \hat{\mathbf{\Delta}}'$ **(Inputs:** $\Sigma, \hat{\sigma}, \hat{\Gamma}, \hat{\Delta}$**; Output:** $\hat{\Delta}'$**)**

$$\frac{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta} \,|\, V : A \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'}{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}, V : A \longrightarrow \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'} \longrightarrow - \implies -1$$

$$\frac{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma}, V : A \,|\, \hat{\Delta} \,|\, V : A \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma}, V : A \,|\, \hat{\Delta}'}{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma}, V : A \,|\, \hat{\Delta} \longrightarrow \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma}, V : A \,|\, \hat{\Delta}'} \longrightarrow - \implies -2$$

$$\frac{}{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}, \{E\} : \{S\} \longrightarrow \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}, E \div S} \longrightarrow -\{\}$$

$\mathbf{\Sigma} \,|\, \hat{\sigma} \,|\, \hat{\mathbf{\Gamma}} \,|\, \hat{\mathbf{\Delta}} \,|\, \mathbf{N} : \mathbf{A} \implies \mathbf{\Sigma} \,|\, \hat{\sigma} \,|\, \hat{\mathbf{\Gamma}} \,|\, \hat{\mathbf{\Delta}}'$ **(Inputs:** $\Sigma, \hat{\sigma}, \hat{\Gamma}, \hat{\Delta}, N, A$**; Output:** $\hat{\Delta}'$**)**

$$\frac{}{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta} \,|\, N : \{S\} \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}, N : \{S\}} \implies -HYP$$

$$\frac{\Sigma \vdash t : \gamma \qquad \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta} \,|\, N\, [t[\hat{\sigma}]] : A(t) \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'}{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta} \,|\, N : \forall i : \gamma. A(i) \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'} \implies -\forall$$

$$\frac{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta} \,|\, \pi_1\, N : A_1 \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'}{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta} \,|\, N : A_1 \& A_2 \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'} \implies -\&_1$$

$$\frac{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta} \,|\, \pi_2\, N : A_2 \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'}{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta} \,|\, N : A_1 \& A_2 \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'} \implies -\&_2$$

$$\frac{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}_1 \longrightarrow_{\mathcal{A}} N_1 : P \qquad \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}_2 \,|\, N_2 \,\hat{}\, N_1 : A \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'}{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}_1, \hat{\Delta}_2 \,|\, N_2 : P \multimap A \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'} \implies - \multimap$$

$$\frac{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \cdot \longrightarrow_{\mathcal{A}} N_1 : P \qquad \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta} \,|\, N_2\, N_1 : A \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'}{\Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta} \,|\, N_2 : P \to A \implies \Sigma \,|\, \hat{\sigma} \,|\, \hat{\Gamma} \,|\, \hat{\Delta}'} \implies - \to$$

Figure 19: Reaction rules for the CHAM (Part I)

$\mathbf{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \;\longrightarrow_{\mathcal{A}}\; N : P}$ **(Inputs:** $\Sigma, \hat{\sigma}, \hat{\Gamma}, \hat{\Delta}, P$**; Output:** N**)**

$$\frac{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \,\|\, V : A \;\Longrightarrow_{\mathcal{A}}\; \Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta}'}{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta}, V : A \;\longrightarrow_{\mathcal{A}}\; N : P} \longrightarrow_{\mathcal{A}} - \Longrightarrow_{\mathcal{A}} -1$$

$$\frac{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma}, V : A \,\|\, \hat{\Delta} \,\|\, V : A \;\Longrightarrow_{\mathcal{A}}\; \Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma}, V : A \,\|\, \hat{\Delta}'}{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma}, V : A \,\|\, \hat{\Delta} \;\longrightarrow_{\mathcal{A}}\; N : P} \longrightarrow_{\mathcal{A}} - \Longrightarrow_{\mathcal{A}} -2$$

$\mathbf{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \,\|\, N' : A \;\Longrightarrow_{\mathcal{A}}\; N : P}$ **(Inputs:** $\Sigma, \hat{\sigma}, \hat{\Gamma}, \hat{\Delta}, N', A, P$**; Output:** N**)**

$$\frac{}{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \cdot \,\|\, N : P \;\Longrightarrow_{\mathcal{A}}\; N : P} \Longrightarrow_{\mathcal{A}} -HYP$$

$$\frac{\Sigma \vdash t : \gamma \qquad \Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \,\|\, N' \; [t[\hat{\sigma}]] : A(t) \;\Longrightarrow_{\mathcal{A}}\; N : P}{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \,\|\, N' : \forall i : \gamma. A(i) \;\Longrightarrow_{\mathcal{A}}\; N : P} \Longrightarrow_{\mathcal{A}} -\forall$$

$$\frac{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \,\|\, \pi_1 \; N' : A_1 \;\Longrightarrow_{\mathcal{A}}\; N : P}{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \,\|\, N' : A_1 \& A_2 \;\Longrightarrow_{\mathcal{A}}\; N : P} \Longrightarrow_{\mathcal{A}} -\&_1$$

$$\frac{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \,\|\, \pi_2 \; N' : A_2 \;\Longrightarrow_{\mathcal{A}}\; N : P}{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \,\|\, N' : A_1 \& A_2 \;\Longrightarrow_{\mathcal{A}}\; N : P} \Longrightarrow_{\mathcal{A}} -\&_2$$

$$\frac{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta}_1 \;\longrightarrow_{\mathcal{A}}\; N_1 : P \qquad \Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta}_2 \,\|\, N_2 \,\hat{}\, N_1 : A \;\Longrightarrow_{\mathcal{A}}\; N : P'}{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta}_1, \hat{\Delta}_2 \,\|\, N_2 : P \multimap A \;\Longrightarrow_{\mathcal{A}}\; N : P'} \Longrightarrow_{\mathcal{A}} - \multimap$$

$$\frac{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \cdot \;\longrightarrow_{\mathcal{A}}\; N_1 : P \qquad \Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \,\|\, N_2 \; N_1 : A \;\Longrightarrow_{\mathcal{A}}\; N : P'}{\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \,\|\, N_2 : P \rightarrow A \;\Longrightarrow_{\mathcal{A}}\; N : P'} \Longrightarrow_{\mathcal{A}} - \rightarrow$$

Figure 20: Reaction rules for the CHAM (Part II)

Syntax

$$\begin{array}{lcll} A & ::= & \bar{x}\langle y_1 \ldots y_n\rangle \mid x(y_1 \ldots y_n).P & \text{(Actions)} \\ C & ::= & A \mid C + C & \text{(External Choice)} \\ P & ::= & C \mid P|P \mid \nu x.P \mid 0 & \text{(Processes)} \end{array}$$

CHAM solutions

$$\begin{array}{lcll} m & ::= & P \mid \nu x.S & \text{(Molecules)} \\ S & ::= & \phi \mid S \uplus \{m\} & \text{(Solutions)} \end{array}$$

Equations on terms and solutions

$$\begin{array}{rcll} C_1 + (C_2 + C_3) & = & (C_1 + C_2) + C_3 & \\ C_1 + C_2 & = & C_2 + C_1 & \\ \nu x.P & = & \nu y.P[y/x] & y \notin P \\ \nu x.S & = & \nu y.S[y/x] & y \notin S \end{array}$$

CHAM semantics

$$\begin{array}{rcl} x(y_1 \ldots y_n).P + C_1\ ,\ \bar{x}z_1 \ldots z_n + C_2 & \rightarrow & P[z_1/y_1] \ldots [z_n/y_n] \\ \nu x.P & \rightharpoonup & \nu x.\{P\} \\ \nu x.S, P & \rightharpoonup & \nu x.(S \uplus \{P\}) \\ P_1|P_2 & \rightharpoonup & P_1, P_2 \\ 0 & \rightharpoonup & \end{array}$$

Figure 21: The π-calculus: syntax and semantics

The typing rule for this construct is

$$\frac{\Sigma, k : \gamma; \Gamma; \Delta; \Psi \ \vdash\ M \# S(k)}{\Sigma; \Gamma; \Delta; \Psi \ \vdash\ (\texttt{priv}\, k : \gamma \ \texttt{in}\ M \# S(k)) \# \exists i : \gamma.S(i)}\ \texttt{priv}$$

3.3 Example: Encoding the π-calculus

In this section, we show an encoding of variant of the π-calculus [6] in *l*CLL. The syntax and semantics of the π-calculus we use are shown in figure 21. The encoding we choose for this calculus is based on an encoding of a similar calculus in MSR [11]. We assume that the signature for our language contains a family of type constructors $\texttt{out}_\texttt{n}$ for $n = 0, 1, \ldots$. These have the kinds:

$$\begin{array}{lcl} \texttt{out}_0 & : & \texttt{chan} \rightarrow \texttt{Type} \\ \texttt{out}_1 & : & \texttt{chan} \rightarrow \texttt{chan} \rightarrow \texttt{Type} \\ \texttt{out}_2 & : & \texttt{chan} \rightarrow \texttt{chan} \rightarrow \texttt{chan} \rightarrow \texttt{Type} \\ \vdots & & \end{array}$$

Assume also that we have a family of constants $\underline{\texttt{out}}_0, \underline{\texttt{out}}_1, \ldots$ having the types:

$$\begin{array}{lcl} \underline{\texttt{out}}_0 & : & \forall x : \texttt{chan}.\ \texttt{out}_0\ x \\ \underline{\texttt{out}}_1 & : & \forall x : \texttt{chan}.\ \forall y_1 : \texttt{chan}.\ \texttt{out}_1\ x\ y_1 \\ \underline{\texttt{out}}_2 & : & \forall x : \texttt{chan}.\ \forall y_1 : \texttt{chan}.\ \forall y_2 : \texttt{chan}.\ \texttt{out}_2\ x\ y_1\ y_2 \\ \vdots & & \end{array}$$

In effect, for any n, and any $k_1, \ldots, k_{n+1}$: chan, $\texttt{out}_\texttt{n}\ k_1 \ldots k_{n+1}$ is actually a *singleton type* i.e. the only closed value of this type is $\underline{\texttt{out}}_\texttt{n}\ [k_1] \ldots [k_{n+1}]$. Let us also assume a family of destructor functions $\underline{\texttt{destroyout}}_\texttt{n}$ which have the types:

$$\begin{array}{lcl} \underline{\texttt{destroyout}}_0 & : & \forall x : \texttt{chan}.\ \texttt{out}_0\ x \multimap \{1\} \\ \underline{\texttt{destroyout}}_1 & : & \forall x : \texttt{chan}.\ \forall y_1 : \texttt{chan}.\ \texttt{out}_1\ x\ y_1 \multimap \{1\} \\ \underline{\texttt{destroyout}}_2 & : & \forall x : \texttt{chan}.\ \forall y_1 : \texttt{chan}.\ \forall y_2 : \texttt{chan}.\ \texttt{out}_2\ x\ y_1\ y_2 \multimap \{1\} \\ \vdots & & \end{array}$$

The corresponding reduction rule is:

$$\frac{}{\underline{\texttt{destroyout}}_\texttt{n}\ [k_1] \ldots [k_{n+1}]\ \hat{}\ (\underline{\texttt{out}}_\texttt{n}\ [k_1] \ldots [k_{n+1}]) \ \to\ \{1\}}$$

We now translate the π-calculus into our language. Every π-calculus term is translated into a type and a term. These translations are shown in figure 22. Let $\texttt{fn}(A)$, $\texttt{fn}(C)$ and $\texttt{fn}(P)$ stand for the free names contained in an action, choice and process respectively. Then the following typing lemma holds.

Lemma 10 (Typing of translated terms).

1. $\texttt{fn}(A) : \texttt{chan}; \cdot; \cdot; \cdot\ \vdash\ \ulcorner A \urcorner : \ulcorner\!\ulcorner A \urcorner\!\urcorner$
2. $\texttt{fn}(C) : \texttt{chan}; \cdot; \cdot; \cdot\ \vdash\ \ulcorner C \urcorner : \ulcorner\!\ulcorner C \urcorner\!\urcorner$
3. $\texttt{fn}(P) : \texttt{chan}; \cdot; \cdot; \cdot\ \vdash\ \ulcorner P \urcorner \,\#\, \ulcorner\!\ulcorner P \urcorner\!\urcorner$

Proof. By induction on the structure of the π-calculus term A, C or P.

Definition 3 (Translation of π-terms). We define the translation, $\langle P \rangle$ of a π-term P as the CHAM configuration $\texttt{fn}(P) : \texttt{chan} \parallel \cdot \parallel \cdot \parallel \ulcorner P \urcorner \,\#\, \ulcorner\!\ulcorner P \urcorner\!\urcorner$.

To illustrate how reductions occur in this framework, consider the π-process, $P = (\bar{x}\langle y \rangle + C_1) \mid (x(z).0 + C_2)$. The translation of this process at types and terms is:

$$\begin{array}{lcl} \ulcorner\!\ulcorner P \urcorner\!\urcorner & = & (\texttt{out}_1\ x\ y \,\&\, \ulcorner\!\ulcorner C_1 \urcorner\!\urcorner) \otimes ((\forall z : \texttt{chan}.\ \texttt{out}_1\ x\ z \multimap \{1\}) \,\&\, \ulcorner\!\ulcorner C_2 \urcorner\!\urcorner) \\ \ulcorner P \urcorner & = & \langle \underline{\texttt{out}}_1\ [x]\ [y], \ulcorner C_1 \urcorner \rangle \otimes \langle N_1(x), \ulcorner C_2 \urcorner \rangle \end{array}$$

where $N_1(x)$ is an abbreviation defined as follows:

$$\begin{array}{lcl} N_1(x) & = & \Lambda z : \texttt{chan}.\ \hat{\lambda} m : \texttt{out}_1\ x\ z. \\ & & \{ \\ & & \quad \underline{\texttt{let}}\ \{1\} = \underline{\texttt{destroyout}}_1\ [x]\ [z]\ \hat{}\ m \\ & & \quad \underline{\texttt{in}}\ 1 \\ & & \} \end{array}$$

Translation into types

$$
\begin{array}{lcl}
\ulcorner\!\ulcorner \bar{x}\langle y_1,\ldots y_n\rangle \urcorner\!\urcorner & = & \mathtt{out_n}\; x\; y_1 \ldots y_n \\
\ulcorner\!\ulcorner x(y_1,\ldots,y_n).P \urcorner\!\urcorner & = & \forall y_1 \ldots y_n : \mathtt{chan}.\; \mathtt{out_n}\; x\; y_1 \ldots y_n \multimap \{\ulcorner\!\ulcorner P \urcorner\!\urcorner\} \\
\ulcorner\!\ulcorner C_1 + C_2 \urcorner\!\urcorner & = & \ulcorner\!\ulcorner C_1 \urcorner\!\urcorner \,\&\, \ulcorner\!\ulcorner C_2 \urcorner\!\urcorner \\
\ulcorner\!\ulcorner 0 \urcorner\!\urcorner & = & 1 \\
\ulcorner\!\ulcorner P_1 | P_2 \urcorner\!\urcorner & = & \ulcorner\!\ulcorner P_1 \urcorner\!\urcorner \otimes \ulcorner\!\ulcorner P_2 \urcorner\!\urcorner \\
\ulcorner\!\ulcorner \nu x.P \urcorner\!\urcorner & = & \exists x : \mathtt{chan}. \ulcorner\!\ulcorner P \urcorner\!\urcorner
\end{array}
$$

Translation into terms

$$
\begin{array}{lcl}
\ulcorner \bar{x}\langle y_1,\ldots y_n\rangle \urcorner & = & \underline{\mathtt{out}}_{\mathtt{n}}\; [x]\; [y_1] \ldots [y_n] \\
\ulcorner x(y_1,\ldots,y_n).P \urcorner & = & \Lambda y_1 \ldots y_n : \mathtt{chan}.\; \hat{\lambda} m : \mathtt{out_n}\; x\; y_1 \ldots y_n. \\
 & & \{ \\
 & & \quad \underline{\mathtt{let}}\; \{1\} = \underline{\mathtt{destroyout}}_{\mathtt{n}}\; [x]\; [y_1] \ldots\; [y_n] \,\hat{}\, m \\
 & & \quad \underline{\mathtt{in}} \\
 & & \qquad \ulcorner P \urcorner \\
 & & \} \\
\ulcorner C_1 + C_2 \urcorner & = & \langle \ulcorner C_1 \urcorner, \ulcorner C_2 \urcorner \rangle \\
\ulcorner 0 \urcorner & = & 1 \\
\ulcorner P_1 | P_2 \urcorner & = & \ulcorner P_1 \urcorner \otimes \ulcorner P_2 \urcorner \\
\ulcorner \nu x.P \urcorner & = & \mathtt{priv}\;\; x : \mathtt{chan}\; \mathtt{in}\, \ulcorner P \urcorner
\end{array}
$$

Figure 22: Translation of the π-calculus

The process P reduces to 0 in the π-calculus. Correspondingly we have the following reduction sequence on the translated term:

$$
\begin{array}{lll}
\langle P \rangle & = & x : \mathtt{chan}, y : \mathtt{chan} \parallel \cdot \parallel \cdot \parallel \ulcorner P \urcorner \mathbin{\#} \ulcorner\!\ulcorner P \urcorner\!\urcorner \\
 & \rightharpoonup & x : \mathtt{chan}, y : \mathtt{chan} \parallel \cdot \parallel \cdot \parallel \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner \rangle \mathbin{\#} (\mathtt{out}_1\ x\ y \mathbin{\&} \ulcorner\!\ulcorner C_1 \urcorner\!\urcorner), \\
 & & \langle N_1(x), \ulcorner C_2 \urcorner \rangle \mathbin{\#} ((\forall z : \mathtt{chan}.\ \mathtt{out}_1\ x\ z \multimap \{1\}) \mathbin{\&} \ulcorner\!\ulcorner C_2 \urcorner\!\urcorner) \\
 & \rightharpoonup^2 & x : \mathtt{chan}, y : \mathtt{chan} \parallel \cdot \parallel \cdot \parallel \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner \rangle : (\mathtt{out}_1\ x\ y \mathbin{\&} \ulcorner\!\ulcorner C_1 \urcorner\!\urcorner), \\
 & & \langle N_1(x), \ulcorner C_2 \urcorner \rangle : ((\forall z : \mathtt{chan}.\ \mathtt{out}_1\ x\ z \multimap \{1\}) \mathbin{\&} \ulcorner\!\ulcorner C_2 \urcorner\!\urcorner) \\
 & \longrightarrow & x : \mathtt{chan}, y : \mathtt{chan} \parallel \cdot \parallel \cdot \parallel (\pi_1\ \langle N_1(x), \ulcorner C_2 \urcorner \rangle)\ [y]\ \hat{}\ (\pi_1\ \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner \rangle) : \{1\} \quad (1) \\
 & \twoheadrightarrow^* & x : \mathtt{chan}, y : \mathtt{chan} \parallel \cdot \parallel \cdot \parallel \{\underline{\mathtt{let}}\ \{1\} = \underline{\mathtt{destroyout}_1}\ [x]\ [y]\ \hat{}\ (\underline{\mathtt{out}_1}\ [x]\ [y])\ \underline{\mathtt{in}}\ 1\} : \{1\} \\
 & \longrightarrow & x : \mathtt{chan}, y : \mathtt{chan} \parallel \cdot \parallel \cdot \parallel (\underline{\mathtt{let}}\ \{1\} = \underline{\mathtt{destroyout}_1}\ [x]\ [y]\ \hat{}\ (\underline{\mathtt{out}_1}\ [x]\ [y])\ \underline{\mathtt{in}}\ 1) \div 1 \quad (2) \\
 & \twoheadrightarrow & x : \mathtt{chan}, y : \mathtt{chan} \parallel \cdot \parallel \cdot \parallel (\underline{\mathtt{let}}\ \{1\} = \{1\}\ \underline{\mathtt{in}}\ 1) \div 1 \\
 & \twoheadrightarrow & x : \mathtt{chan}, y : \mathtt{chan} \parallel \cdot \parallel \cdot \parallel 1 \div 1 \\
 & \rightharpoonup & x : \mathtt{chan}, y : \mathtt{chan} \parallel \cdot \parallel \cdot \parallel 1 \mathbin{\#} 1 \\
 & \rightharpoonup & x : \mathtt{chan}, y : \mathtt{chan} \parallel \cdot \parallel \cdot \parallel \cdot
\end{array}
$$

The *reaction* steps here have been marked (1) and (2). Step (2) is the elimination of $\{\dots\}$ in the monad. Step (1) requires some backwards reasoning and the exact proof that allows this step is shown in figure 23. All rules used in this proof are from figure 19. It is instructive to observe that chaining of reactions allows us to simulate the correct behavior of external choice in the π-calculus. The remaining steps in the above reduction are either structural rearrangement ($\rightharpoonup$) or functional evaluation ($\twoheadrightarrow$).

The rewrite steps shown above show how π-calculus reductions are simulated in the translation. The exact formulation of a correctness result for the translation requires a notion of observation on *l*CLL-CHAM configurations, which is a subject of future research.

3.4 Types for *l*CLL CHAM Configurations

Since terms in *l*CLL CHAM configurations already have types with them, we need to create a judgment like $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$ in order to type *l*CLL CHAM configurations. The obvious intuition of splitting resources in Δ for all terms in $\hat{\Delta}$ does not not work. For example, consider proving $\cdot; \cdot; \cdot; \Delta \vdash \cdot \parallel \cdot \parallel \cdot \parallel x_1 : A_1, x_2 : A_2$ when $\Delta = x_1 \otimes x_2 : A_1 \otimes A_2$. Intuitively, we want this to be provable but Δ is a singleton and cannot be split. From such observations, we arrive at the following definition.

Definition 4 (*l*CLL typing relation). Let $\Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$ be a CHAM configuration. Let $\hat{\Delta} = T_1 \mathbin{\%} Z_1 \dots T_n \mathbin{\%} Z_n$ and $\hat{\Gamma} = V_1 : A_1 \dots V_m : A_m$. We say that $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$ iff there exist $\Delta_1 \dots \Delta_n$, Γ' and Σ'' such that the following conditions hold:

1. $\Sigma'' \supseteq \Sigma'$
2. $\Sigma''; \Gamma'; \Delta_1 \dots \Delta_n \impliedby \Sigma; \Gamma; \Delta$
3. For each $t/i : \gamma \in \hat{\sigma}$, $i : \gamma \in \Sigma'$ and $\Sigma'' \vdash t : \gamma$
4. For $1 \le i \le n$, $\Sigma''; \Gamma'; \Delta_i; \Psi \vdash T_i \mathbin{\%} Z_i[\hat{\sigma}]$
5. For $1 \le j \le m$, $\Sigma''; \Gamma'; \cdot; \Psi \vdash V_j : A_j[\hat{\sigma}]$

$$
\begin{aligned}
\Sigma &= x : \mathtt{chan}, y : \mathtt{chan} \\
\gamma &= (\pi_1\ \langle N_1(x), \ulcorner C_2 \urcorner\rangle)\ [y] \,\hat{}\, (\pi_1\ \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle) : \{1\}
\end{aligned}
$$

$$
\mathcal{D}_1 \;=\; \dfrac{\dfrac{\dfrac{}{\begin{array}{c}\Sigma \,\|\, \cdot \,\|\, \cdot \,\|\, \cdot \,\|\, \pi_1\langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle : \mathtt{out}_1\ x\ y \\ \Longrightarrow_{\mathcal{A}} \\ \pi_1\langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle : \mathtt{out}_1\ x\ y\end{array}} \Longrightarrow_{\mathcal{A}} -HYP}{\begin{array}{c}\Sigma \,\|\, \cdot \,\|\, \cdot \,\|\, \cdot \,\|\, \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle : \mathtt{out}_1\ x\ y \,\&\, \ulcorner\ulcorner C_1 \urcorner\urcorner \\ \Longrightarrow_{\mathcal{A}} \\ \pi_1\langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle : \mathtt{out}_1\ x\ y\end{array}} \Longrightarrow_{\mathcal{A}} -\&_1}{\begin{array}{c}\Sigma \,\|\, \cdot \,\|\, \cdot \,\|\, \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle : \mathtt{out}_1\ x\ y \,\&\, \ulcorner\ulcorner C_1 \urcorner\urcorner \\ \longrightarrow_{\mathcal{A}} \\ \pi_1\langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle : \mathtt{out}_1\ x\ y\end{array}} \longrightarrow_{\mathcal{A}} - \Longrightarrow_{\mathcal{A}} -1
$$

$$
\dfrac{\dfrac{\dfrac{\mathcal{D}_1 \qquad \dfrac{}{\Sigma \,\|\, \cdot \,\|\, \cdot \,\|\, \cdot \,\|\, (\pi_1\ \langle N_1(x), \ulcorner C_2 \urcorner\rangle)\ [y] \,\hat{}\, (\pi_1\ \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle) : \{1\} \Longrightarrow \gamma} \Longrightarrow -HYP}{\begin{array}{c}\Sigma \,\|\, \cdot \,\|\, \cdot \,\|\, \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle : \mathtt{out}_1\ x\ y \,\&\, \ulcorner\ulcorner C_1 \urcorner\urcorner \,\| \\ (\pi_1\langle N_1(x), \ulcorner C_2 \urcorner\rangle)\ [y] : \mathtt{out}_1\ x\ y \multimap \{1\}\end{array} \Longrightarrow \gamma} \Longrightarrow - \multimap}{\begin{array}{c}\Sigma \,\|\, \cdot \,\|\, \cdot \,\|\, \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle : \mathtt{out}_1\ x\ y \,\&\, \ulcorner\ulcorner C_1 \urcorner\urcorner \,\| \\ \pi_1\langle N_1(x), \ulcorner C_2 \urcorner\rangle : (\forall z : \mathtt{chan}.\ \mathtt{out}_1\ x\ z \multimap \{1\})\end{array} \Longrightarrow \gamma} \Longrightarrow -\forall
$$

$$
\dfrac{\dfrac{\begin{array}{c}\Sigma \,\|\, \cdot \,\|\, \cdot \,\|\, \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle : \mathtt{out}_1\ x\ y \,\&\, \ulcorner\ulcorner C_1 \urcorner\urcorner \,\| \\ \pi_1\langle N_1(x), \ulcorner C_2 \urcorner\rangle : (\forall z : \mathtt{chan}.\ \mathtt{out}_1\ x\ z \multimap \{1\})\end{array} \Longrightarrow \gamma}{\begin{array}{c}\Sigma \,\|\, \cdot \,\|\, \cdot \,\|\, \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle : \mathtt{out}_1\ x\ y \,\&\, \ulcorner\ulcorner C_1 \urcorner\urcorner \,\| \\ \langle N_1(x), \ulcorner C_2 \urcorner\rangle : (\forall z : \mathtt{chan}.\ \mathtt{out}_1\ x\ z \multimap \{1\} \,\&\, \ulcorner\ulcorner C_2 \urcorner\urcorner)\end{array} \Longrightarrow \gamma} \Longrightarrow -\&_1}{\begin{array}{c}\Sigma \,\|\, \cdot \,\|\, \cdot \,\|\, \langle \underline{\mathtt{out}_1}\ [x]\ [y], \ulcorner C_1 \urcorner\rangle : \mathtt{out}_1\ x\ y \,\&\, \ulcorner\ulcorner C_1 \urcorner\urcorner, \\ \langle N_1(x), \ulcorner C_2 \urcorner\rangle : (\forall z : \mathtt{chan}.\ \mathtt{out}_1\ x\ z \multimap \{1\} \,\&\, \ulcorner\ulcorner C_2 \urcorner\urcorner)\end{array} \longrightarrow \gamma} \longrightarrow - \Longrightarrow -1
$$

Figure 23: Main reduction step for π-calculus example

We also say that $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \parallel N : A$ iff $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}, N : A$.

This definition essentially allows the context Δ to be split into several contexts, one for each of the terms in $\hat{\Delta}$.

Lemma 11. Let $\Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$ be a CHAM configuration. Let $\hat{\Delta} = T_1 \mathbin{\%} Z_1 \ldots T_n \mathbin{\%} Z_n$, $\hat{\Gamma} = V_1 : A_1 \ldots V_m : A_m$ and suppose there exist $\Delta, \Delta_1 \ldots \Delta_n, \Gamma, \Sigma$ and Ψ such that the following hold:

1. $\Sigma \supseteq \Sigma'$.
2. For each $t/i : \gamma \in \hat{\sigma}$, $i : \gamma \in \Sigma'$ and $\Sigma \vdash t : \gamma$
3. For each $1 \leq i \leq n$, $\Sigma; \Gamma; \Delta_i; \Psi \vdash T_i \mathbin{\%} Z_i[\hat{\sigma}]$
4. For each $1 \leq j \leq m$, $\Sigma; \Gamma; \cdot; \Psi \vdash V_j : A_j[\hat{\sigma}]$
5. $\Sigma; \Gamma; \Delta; \Psi \vdash N : A[\hat{\sigma}]$

Then,

1. $\Sigma; \Gamma; \Delta_1 \ldots \Delta_n; \Psi \vdash \Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$.
2. $\Sigma; \Gamma; \Delta, \Delta_1 \ldots \Delta_n; \Psi \vdash \Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \parallel N : A$.
3. If $\Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \parallel N : A \Longrightarrow_{\mathcal{A}} N' : P$, then $\Sigma; \Gamma; \Delta, \Delta_1 \ldots \Delta_n; \Psi \vdash N' : P[\hat{\sigma}]$.
4. If $\Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \longrightarrow_{\mathcal{A}} N' : P$, then $\Sigma; \Gamma; \Delta_1 \ldots \Delta_n; \Psi \vdash N' : P[\hat{\sigma}]$.

Proof. Proof of (1) and (2) is immediate from definition 4. Proof of (3) and (4) follows by a mutual induction on the given rewrite derivation.

Lemma 12.

1. If $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$ and $\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \rightharpoonup \Sigma' \parallel \hat{\sigma}' \parallel \hat{\Gamma}' \parallel \hat{\Delta}'$, then $\Sigma \subseteq \Sigma'$ and $\Sigma'; \Gamma; \Delta; \Psi \vdash \Sigma' \parallel \hat{\sigma}' \parallel \hat{\Gamma}' \parallel \hat{\Delta}'$.
2. If $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$ and $\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \twoheadrightarrow \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}'$, then $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}'$.
3. If $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \parallel N' : A$ and $\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \parallel N' : A \Longrightarrow_{\mathcal{A}} N : P$, then $\Sigma; \Gamma; \Delta; \Psi \vdash N : P[\hat{\sigma}]$.
4. If $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$ and $\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \longrightarrow_{\mathcal{A}} N : P$, then $\Sigma; \Gamma; \Delta; \Psi \vdash N : P[\hat{\sigma}]$.
5. If $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \parallel N : A$ and $\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \parallel N : A \Longrightarrow \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}'$, then $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}'$.
6. If $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$ and $\Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta} \longrightarrow \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}'$, then $\Sigma; \Gamma; \Delta; \Psi \vdash \Sigma \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}'$.

Proof. Proof of (1) follows from lemma 5. (2) is immediate from lemma 8. (3) and (4) follow from lemma 11 and lemma 1(3). For (5) we use induction on the derivation of the given rewrite relation and lemma 11. (6) follows immediately from (5).

Definition 5 (*l*CLL CHAM moves). We define a *l*CLL CHAM rewrite move $\rightrightarrows$ as $\rightrightarrows \ = \ \rightharpoonup \cup \twoheadrightarrow \cup \longrightarrow$. $\rightrightarrows^*$ denotes the reflexive-transitive closure of $\rightrightarrows$.

Lemma 13 (*l*CLL preservation). If $\Sigma; \Gamma; \Delta; \Psi \ \vdash \ \Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta}$ and $\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \ \rightrightarrows^* \Sigma' \,\|\, \hat{\sigma}' \,\|\, \hat{\Gamma}' \,\|\, \hat{\Delta}'$, then $\Sigma \subseteq \Sigma'$ and $\Sigma'; \Gamma; \Delta; \Psi \ \vdash \ \Sigma' \,\|\, \hat{\sigma}' \,\|\, \hat{\Gamma}' \,\|\, \hat{\Delta}'$.

Theorem 2 (Type-safety for terms in *l*CLL CHAMs). If $\Sigma; \cdot; \cdot; \cdot \ \vdash \ \Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta}$ and $\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \ \rightrightarrows^* \Sigma' \,\|\, \hat{\sigma}' \,\|\, \hat{\Gamma}' \,\|\, \hat{\Delta}'$, then for any typed *f*CLL term $T \ \% \ Z$ in $\hat{\Delta}'$, it is the case that T is a value or T reduces to some T'.

Proof. Using lemma 13, $\Sigma'; \cdot; \cdot; \cdot \ \vdash \ \Sigma' \,\|\, \hat{\sigma}' \,\|\, \hat{\Gamma}' \,\|\, \hat{\Delta}'$. Let $\hat{\Delta}' = T_1 \ \% \ Z_1, \ldots, T_n \ \% \ Z_n$. By definition 4, there exist $\Sigma'', \Gamma', \Delta_1, \ldots, \Delta_n$ such that $\Sigma''; \Gamma'; \Delta_1, \ldots, \Delta_n \Longleftarrow \Sigma'; \cdot; \cdot$ and for each $1 \leq i \leq n$, $\Sigma''; \Gamma'; \Delta_i; \cdot \ \vdash \ T_i \ \% \ Z_i[\hat{\sigma}']$. Now from lemma 1 it is clear that $\Gamma' = \cdot$ and $\Delta_1 = \ldots = \Delta_n = \cdot$. Thus, for each $1 \leq i \leq n$, $\Sigma''; \cdot; \cdot; \cdot \ \vdash \ T_i \ \% \ Z_i[\hat{\sigma}']$. Using the progress lemma for *f*CLL (lemma 9), each T_i must either be a value, or it can reduce further.

3.5 Comparing process-calculi and *l*CLL

As seen from the encoding of the π-calculus in section 3.3, *l*CLL can encode several basic concurrency primitives. In fact, there is a correspondence between the constructs of process-calculi like the π-calculus and constructors of *l*CLL. Various common constructs of process-calculi, together with their equivalents in *l*CLL are listed below.

1. *Processes*. In general, we view monadic terms and expressions as processes in CHAM solutions.

2. *Parallelism*. Apart from parallelism introduced for expressions in section 2.1, monadic terms of the form $M_1 \otimes M_2$ can be viewed as processes reducing in parallel. Similarly, terms in CHAM solutions can be viewed as processes executing in parallel.

3. *Communication channels*. Communication channels can be simulated in *l*CLL using index refinements of a fixed sort (`chan`) as we do in section 3.3.

4. *Input prefixing*. The language constructs $\Lambda i.N$, $\lambda x : P.N$ and $\hat{\lambda} x : P.N$ together with the associated types $\forall i : \gamma.A$, $P \rightarrow B$ and $P \multimap B$ provide encodings for input processes.

5. *Asynchronous output*. Any value V of (possibly refined) atomic type P can be viewed as an output term without continuation because it can be linked to a term of type $P \multimap B$ as an input. If the value $V : P$ is linear, then this corresponds to an output that has to be used as an input to exactly one program. If it is unrestricted, then it can be used as input to any number of programs. Such an output term corresponds to an asynchronous broadcast.

6. *Name restriction*. We showed in section 3.2 that channels can be made private using abstraction semantics of the $\exists$ quantifier. The `priv` construct defined in that section can be used to create private channel names in CHAM executions.

7. *Choices*. The type constructor $\&$ and the associated term constructor $\langle N_1, N_2 \rangle$ act as an external choice operator in our logic programming language. The proof-search procedure can project out one component of a choice if it can be used to complete a link step. The type constructor $\oplus$ and the monadic term constructors $\underline{\texttt{inl}}$ and $\underline{\texttt{inr}}$ can be used to simulate internal choice in *l*CLL.

Process-calculus construct	Equivalent *l*CLL construct
Process, P	Monadic term(M), expression(E)
Parallel composition, $P_1 \vert P_2$	$M_1 \otimes M_2 \# S_1 \otimes S_2$
Communication channel	Index refinement of sort `chan`
Input prefixing, $x(y).P$	$\Lambda i.N : \forall i : \gamma.A$, $\lambda x.N : P \to B$ and $\hat{\lambda} x.N : P \multimap B$
Asynchronous output, $\bar{x}y$	Linear assumption $N : P$ where P is atomic
Name restriction, $\nu x.P$	$\texttt{priv}\ x : \texttt{chan}\ \texttt{in}\ M \# \exists x : \texttt{chan}.S$
Internal choice	$\underline{\texttt{inl}}\ M \# S_1 \oplus S_2$, $\underline{\texttt{inr}}\ M \# S_1 \oplus S_2$
External choice, $C_1 + C_2$	$\langle N_1, N_2 \rangle : A_1 \& A_2$
n-way input	$N : P_1 \multimap \ldots \multimap P_n \multimap B$
Communication and synchronization	Proof-search

Figure 24: Correspondence between process-calculi and *l*CLL

8. *n-way input.* Due to chaining of linking steps in the *l*CLL, we have a mechanism for n-way input in *l*CLL. For example, a receiver of type $P_1 \multimap P_2 \multimap B$ *always* synchronizes simultaneously with two senders of types P_1 and P_2.

9. *Communication and synchronization.* Communication and synchronization in *l*CLL occurs using reaction steps ($\longrightarrow$).

For an illustration of these constructs in *l*CLL, the reader is referred to the encoding of the π-calculus in section 3.3. Figure 24 shows a summary of the above correspondence between process-calculi constructs and *l*CLL connectives.

4 Full-CLL: Integrating *f*CLL and *l*CLL

*f*CLL described in section 2 is purely functional. Even though it admits some parallelism in the tensor and evaluation of expressions, it is essentially free from effects. The concurrent (logic) programming language *l*CLL described in section 3 allows an additional layer of concurrency over *f*CLL. In this section we integrate in the other direction - we allow concurrent logic programming to occur inside *f*CLL programs. Since concurrent computations can deadlock and get stuck, they are not free from effects. As a result, we confine such concurrent evaluation to the monad only. We extend the grammar for expressions with an additional construct as follows.

$$E ::= \ \ldots \mid \underline{\texttt{link}}\,(E \div S)\ \underline{\texttt{to}}\ G$$

where $G ::= A \mid !A \mid 1$ and A is any asynchronous type. G is called a goal type. Observe that G is a subset of the family of types. We do not allow arbitrary types as goals for reasons described later. The typing rule for the $\underline{\texttt{link}}$ construct is the following.

$$\frac{\Sigma; \Gamma; \Delta; \Psi \vdash E \div S}{\Sigma; \Gamma; \Delta; \Psi \vdash \underline{\texttt{link}}\,(E \div S)\ \underline{\texttt{to}}\ G \div G}\ \textit{LINK}$$

It is assumed in the above rule that G is well-formed in the context Σ. The $\underline{\texttt{link}}$ construct is always evaluated in the context of index variables Σ in which it is well-typed. To evaluate $\Sigma; \underline{\texttt{link}}\,(E \div S)\ \underline{\texttt{to}}\ G$

we start a new $lCLL$ CHAM configuration with only the term $E \div S$ in it i.e. we start with $\Sigma \parallel \cdot \parallel \cdot \parallel E \div S$. Then we let this configuration rewrite according to all the rules in figures 14, 19 and 20 *until it saturates* i.e. no more rewrite rules apply. If the configuration never saturates, computation runs forever and the $\underline{\texttt{link}}$ construct does not terminate. If the configuration saturates in $\Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$, then the computation of the $\underline{\texttt{link}}$ construct succeeds iff one of the following conditions holds.

1. $G = A$ and $\hat{\Delta} = V : A$. Then the whole construct evaluates to $\Sigma'; V$.
2. $G = A$, $\hat{\Delta} = \cdot$ and there exists $V : A \in \hat{\Gamma}$. In this case the whole construct evaluates to $\Sigma'; V$.
3. $G = !A$, $\hat{\Delta} = \cdot$ and there exists $V : A \in \hat{\Gamma}$. Then the whole construct evaluates to $\Sigma'; !V$.
4. $G = 1$ and $\hat{\Delta} = \cdot$. In this case the whole construct evaluates to $\Sigma'; 1$.

If none of these conditions hold, then the computation fails and evaluation deadlocks. The above conditions are summarized in the following evaluation rules.

$$\frac{\Sigma \parallel \cdot \parallel \cdot \parallel E \div S \ \rightrightarrows^* \ \Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel V : A}{\Sigma; \underline{\texttt{link}}\,(E \div S)\,\underline{\texttt{to}}\,A \ \hookrightarrow \ \Sigma'; V} \hookrightarrow LINK-1$$

$$\frac{\Sigma \parallel \cdot \parallel \cdot \parallel E \div S \ \rightrightarrows^* \ \Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma}, V : A \parallel \cdot}{\Sigma; \underline{\texttt{link}}\,(E \div S)\,\underline{\texttt{to}}\,A \ \hookrightarrow \ \Sigma'; V} \hookrightarrow LINK-2$$

$$\frac{\Sigma \parallel \cdot \parallel \cdot \parallel E \div S \ \rightrightarrows^* \ \Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma}, V : A \parallel \cdot}{\Sigma; \underline{\texttt{link}}\,(E \div S)\,\underline{\texttt{to}}\,!A \ \hookrightarrow \ \Sigma'; !V} \hookrightarrow LINK-3$$

$$\frac{\Sigma \parallel \cdot \parallel \cdot \parallel E \div S \ \rightrightarrows^* \ \Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \cdot}{\Sigma; \underline{\texttt{link}}\,(E \div S)\,\underline{\texttt{to}}\,1 \ \hookrightarrow \ \Sigma'; 1} \hookrightarrow LINK-4$$

where $\rightrightarrows \ = \ \rightharpoonup \cup \twoheadrightarrow \cup \longrightarrow$, as in definition 5. It is implicitly assumed in these rules that any CHAM configuration on the right of $\rightrightarrows^*$ is saturated (it cannot be rewritten using the relation $\rightrightarrows$). We also lift the evaluation relation $E \hookrightarrow E'$ from figure 12 to include the context Σ.

$$\frac{E \hookrightarrow E'}{\Sigma; E \hookrightarrow \Sigma; E'}$$

At this point we can explain why we restrict the goal G in the construct $\underline{\texttt{link}}\ (E \div S)\ \underline{\texttt{to}}\ G$ to the set $\{A, !A, 1\}$. The reason for disallowing arbitrary goals is that for goal types other than $\{A, !A, 1\}$, computation of the $\underline{\texttt{link}}$ construct will always fail because saturated CHAM configurations cannot contain terms having those types. Suppose, for example, we allow the type $G = S_1 \otimes S_2$ as a goal. In order for evaluation to succeed with $S_1 \otimes S_2$ as a goal, the CHAM rewriting would have to end in a configuration $\Sigma' \parallel \hat{\sigma} \parallel \hat{\Gamma} \parallel \hat{\Delta}$ where $\hat{\Delta} = M \# S_1 \otimes S_2$, for some M. However, this is impossible because at this point the CHAM configuration cannot be saturated. We can prove this as follows. By the progress theorem, either M is a value, or it can reduce further. If M is a value, it has to be of the form $M_{v_1} \otimes M_{v_2}$ and in that case we can apply the rule $\rightharpoonup -\otimes$ on the CHAM configuration. If M can reduce further, then the whole CHAM configuration can reduce using the rule $\twoheadrightarrow - \mapsto$. Thus an *l*CLL CHAM configuration cannot end with a monadic term of type $S_1 \otimes S_2$ in $\hat{\Delta}$. Similar arguments show that a CHAM configuration cannot saturate if it has a program of *any* synchronous type in it. For the particular set of goals, $\{A, !A, 1\}$, it is possible for CHAM computations to succeed without any synchronous types in them. Thus we limit goals to this set. Limiting goal types to the set $\{A, !A, 1\}$ may seem like a big restriction but in practice we found that other goal types are never needed.

We call the resultant language with the `link` construct full-CLL or CLL for brevity. Full-CLL symmetrically integrates functional and concurrent logic programming. Concurrent logic programming can be nested inside *f*CLL programs using the `link` construct. On the other hand, the functional rewrite rules in CHAMs allow functional evaluation inside concurrent logic programming. Execution of full-CLL programs occurs in interleaving phases of functional evaluation and concurrent logic programming.

An important remark related to programming in full-CLL is that it is essential that the top-level construct of any program that performs concurrent computation be an expression. This is because all concurrency in full-CLL is restricted to the `link` construct which is an expression, and evaluation of expressions coerced into terms is lazy (recall that $\{E\}$ is a value in CLL). If the top-level construct of a program is a term or a monadic term, then nested expressions in the program will never be evaluated, and hence the program will not perform any concurrent computation.

4.1 Type-Safety

Since the `link` construct may get stuck, full-CLL does not have a progress lemma at the level of expressions. However the monad in CLL is lazy and this lemma still holds at the level of terms and monadic terms. We also have a type preservation lemma at the level of terms, monadic terms and expressions. Type-safety lemmas and theorems for full-CLL are given below.

Lemma 14 (Preservation).

1. If $\Sigma; \Gamma; \Delta; \Psi \vdash N : A$ and $N \rightarrow N'$, then $\Sigma; \Gamma; \Delta; \Psi \vdash N' : A$.
2. If $\Sigma; \Gamma; \Delta; \Psi \vdash M \# S$ and $M \mapsto M'$, then $\Sigma; \Gamma; \Delta; \Psi \vdash M' \# S$.
3. If $\Sigma; \Gamma; \Delta; \Psi \vdash E \div S$ and $\Sigma; E \hookrightarrow \Sigma'; E'$, then $\Sigma \subseteq \Sigma'$ and $\Sigma'; \Gamma; \Delta; \Psi \vdash E' \div S$.

Proof. In this case we use induction on the given derivation to simultaneously prove this and lemmas 12 and 13.

Lemma 15 (Progress).

1. If $\Sigma; \cdot; \cdot; \cdot \vdash N : A$, then either $N = V$ or for some N', $N \rightarrow N'$.
2. If $\Sigma; \cdot; \cdot; \cdot \vdash M \# S$, then either $M = M_v$ or for some M', $M \mapsto M'$.

Proof. By induction on the given typing derivation. As expected, there is no progress lemma at the level of expressions.

Theorem 3 (Type-Safety).

1. If $\Sigma; \cdot; \cdot; \cdot \vdash N : A$ and $N \rightarrow^* N'$, then either $N' = V$ or there exists N'' such that $N' \rightarrow N''$.
2. If $\Sigma; \cdot; \cdot; \cdot \vdash M \# S$ and $M \mapsto^* M'$, then either $M' = M_v$ or there exists M'' and such that $M' \mapsto M''$.

Proof. By induction on the number of steps in the reduction, as for *f*CLL.

In full-CLL, nested *l*CLL-CHAM configurations contain full-CLL programs in place of *f*CLL programs. This has a significant effect on theorem 2 of section 3.4, which must be modified for *l*CLL-CHAM configurations that contain full-CLL programs. Since the proof of theorem 2 uses progress (lemma 9), which no longer holds for expressions in full-CLL, we expect to obtain only a weaker type-safety property for CHAMs embedded in full-CLL. Indeed, we can prove only the following theorem.

Theorem 4 (Type-safety for terms in CHAMs in full-CLL). If $\Sigma; \cdot; \cdot; \cdot \vdash \Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta}$ and $\Sigma \,\|\, \hat{\sigma} \,\|\, \hat{\Gamma} \,\|\, \hat{\Delta} \rightrightarrows^* \Sigma' \,\|\, \hat{\sigma}' \,\|\, \hat{\Gamma}' \,\|\, \hat{\Delta}'$, then for any typed full-CLL term $T \;\%\; Z$ in $\hat{\Delta}'$, it is the case that T is an expression or T is a value or T reduces to some T'.

The only reason full-CLL programs get stuck is that the forward chaining procedure in some nested link construct fails to reach its stated goal. In all practical problems that we encountered, we found that it was possible to write full-CLL programs in a way that embedded link constructs always succeed in producing the desired goal. An exploration of methods and techniques to prove the correctness of full-CLL programs formally is left to future work.

5 Programming Techniques and Examples

In order to illustrate the relatively new style of programming that CLL requires, we devote this section to developing programming techniques and examples of programs in full-CLL. The concurrency primitives already present in full-CLL are very simple (but expressive) and in order to write useful programs we need to build library code that implements more conventional concurrency primitives like buffered-asynchronous message passing, synchronous message passing, non-deterministic synchronous choices etc. We present this library code as a set of macros. The reasons for using macros in place of functional abstractions are clarity and brevity. The functional abstraction mechanisms in CLL (Λ, λ and $\hat{\lambda}$) are expressive enough to allow us to rewrite all the library code in this section as functions instead of macros. However, doing so results in more complicated implementations and types for the abstractions. Thus we use macros for library code in place of functions. Just as an illustration, we describe the implementation of the primitives for buffered-asynchronous message passing using functions instead of macros in section 5.4.

Many of the examples in this section are based on similar programs in John Reppy's book Concurrent Programming in ML [32]. As a convention, we write all macro names in **boldface**.

5.1 Example: A Concurrent Fibonacci Program

In this section we build a concurrent program to compute Fibonacci numbers. For this and subsequent examples, we assume that our language has fundamental functional constructs like basic types (integers, `int` and booleans, `bool`), datatypes (á la ML), recursion at the level of terms and conditional if-then-else constructs. All these may be added to the language in a straightforward manner. Fibonacci numbers are defined by the following equations.

$$\begin{array}{rcll} fib(0) & = & 1 & \\ fib(1) & = & 1 & \\ fib(n) & = & fib(n-1) + fib(n-2) & \quad n \geq 2 \end{array}$$

```
fun fib (n) =
    if (n = 0) then {!1}
    else if (n = 1) then {!1}
    else
    {
        let {!n1} = fib (n - 1)
        let {!n2} = fib (n - 2)
        in
            !(n1 + n2)
    }
```

Figure 25: The function `fib`

```
fun fibc (n) =
    if (n = 0) then {!1}
    else if (n = 1) then {!1}
    else
    {
        link
        (
            (fibc (n - 1)  ⊗  fibc (n - 2)  ⊗  λn1 : int. λn2 : int. {!(n1 + n2)})
              ÷  ({!int} ⊗ {!int} ⊗ (int → int → {!int}))
        ) to !int
    }
```

Figure 26: An *incorrect* function `fibc`

We can write a *parallel* version of the function `fib` as shown in figure 25. This function does not use any communication between processes executing in parallel and may be derived from the more general `divAndConquer` function described in section 2.3. It has the type $\texttt{int} \rightarrow \{!\texttt{int}\}$.

Figure 26 shows a *concurrent*, but incorrect implementation of `fib`. The function `fibc` has the type $\texttt{int} \rightarrow \{!\texttt{int}\}$. Given $n \geq 2$, we spawn a CHAM with three threads. The first two threads recursively compute $fib(n-1)$ and $fib(n-2)$. These two computations may spawn nested CHAMs during evaluation. Such nested CHAMs are distinct from each other and terms in different CHAMs cannot interact. The third thread is a synchronization thread that waits for the results of these two computations and adds them together to produce the result. This synchronization is performed automatically by the CHAM. As mentioned earlier, this implementation is incorrect and the reason for incorrectness is described below.

In the function `fibc`, there are four ways for the CHAM to proceed after $\texttt{fibc}(n-1)$ has evaluated to a value $\{!N_1\}$ and $\texttt{fibc}(n-2)$ evaluates to a value $\{!N_2\}$. In one case, n_1 gets instantiated to the result of evaluating N_1 and n_2 to the result of evaluating N_2:

$$
\begin{array}{ll}
 & \cdot \,\Vert\, \cdot \,\Vert\, \cdot \,\Vert\, \{!N_1\} : \{!\texttt{int}\}, \{!N_2\} : \{!\texttt{int}\}, \\
 & \lambda n_1 : \texttt{int}.\ \lambda n_2 : \texttt{int}.\ \{!(n_1 + n_2)\} : \texttt{int} \rightarrow \texttt{int} \rightarrow \{!\texttt{int}\} \\
\longrightarrow^2 & \cdot \,\Vert\, \cdot \,\Vert\, \cdot \,\Vert\, !N_1 \div !\texttt{int}, !N_2 \div !\texttt{int}, \\
 & \lambda n_1 : \texttt{int}.\ \lambda n_2 : \texttt{int}.\ \{!(n_1 + n_2)\} : \texttt{int} \rightarrow \texttt{int} \rightarrow \{!\texttt{int}\} \\
\rightharpoonup^2 & \cdot \,\Vert\, \cdot \,\Vert\, \cdot \,\Vert\, !N_1 \# !\texttt{int}, !N_2 \# !\texttt{int}, \\
 & \lambda n_1 : \texttt{int}.\ \lambda n_2 : \texttt{int}.\ \{!(n_1 + n_2)\} : \texttt{int} \rightarrow \texttt{int} \rightarrow \{!\texttt{int}\} \\
\twoheadrightarrow^* & \cdot \,\Vert\, \cdot \,\Vert\, \cdot \,\Vert\, !V_1 \# !\texttt{int}, !V_2 \# !\texttt{int}, \\
 & \lambda n_1 : \texttt{int}.\ \lambda n_2 : \texttt{int}.\ \{!(n_1 + n_2)\} : \texttt{int} \rightarrow \texttt{int} \rightarrow \{!\texttt{int}\} \\
\rightharpoonup^2 & \cdot \,\Vert\, \cdot \,\Vert\, V_1 : \texttt{int}, V_2 : \texttt{int} \,\Vert\, \lambda n_1 : \texttt{int}.\ \lambda n_2 : \texttt{int}.\ \{!(n_1 + n_2)\} : \texttt{int} \rightarrow \texttt{int} \rightarrow \{!\texttt{int}\} \\
\longrightarrow & \cdot \,\Vert\, \cdot \,\Vert\, V_1 : \texttt{int}, V_2 : \texttt{int} \,\Vert\, ((\lambda n_1 : \texttt{int}.\ \lambda n_2 : \texttt{int}.\ \{!(n_1 + n_2)\})\ V_1\ V_2) : \{!\texttt{int}\} \\
\twoheadrightarrow^2 & \cdot \,\Vert\, \cdot \,\Vert\, V_1 : \texttt{int}, V_2 : \texttt{int} \,\Vert\, \{!(V_1 + V_2)\} : \{!\texttt{int}\}
\end{array}
$$

In the second case the instantiations are swapped - n_1 is instantiated to V_2 and n_2 is instantiated to V_1. Assuming that $+$ is commutative, the result of both possible programs is the same and correct. However, observe that since V_1 and V_2 are unrestricted values in the configuration, it is possible to instantiate both n_1 and n_2 with either one of V_1 and V_2. This gives us two more possible incorrect computations. One of these is shown below.

$$
\begin{array}{ll}
 & \ldots \\
\rightharpoonup^2 & \cdot \,\Vert\, \cdot \,\Vert\, V_1 : \texttt{int}, V_2 : \texttt{int} \,\Vert\, \lambda n_1 : \texttt{int}.\ \lambda n_2 : \texttt{int}.\ \{!(n_1 + n_2)\} : \texttt{int} \rightarrow \texttt{int} \rightarrow \{!\texttt{int}\} \\
\longrightarrow & \cdot \,\Vert\, \cdot \,\Vert\, V_1 : \texttt{int}, V_2 : \texttt{int} \,\Vert\, ((\lambda n_1 : \texttt{int}.\ \lambda n_2 : \texttt{int}.\ \{!(n_1 + n_2)\})\ V_1\ V_1) : \{!\texttt{int}\} \\
\twoheadrightarrow^2 & \cdot \,\Vert\, \cdot \,\Vert\, V_1 : \texttt{int}, V_2 : \texttt{int} \,\Vert\, \{!(V_1 + V_1)\} : \{!\texttt{int}\}
\end{array}
$$

We can use index refinements to correct this function. Assume that we have a type constructor $\overline{\texttt{int}} : \texttt{chan} \rightarrow \texttt{Type}$ and the constructor-destructor pair `refineint` and `fetchint` with the typing rules

$$
\frac{\Sigma \ \vdash\ k : \texttt{chan} \qquad \Sigma; \Gamma; \Delta; \Psi \ \vdash\ N : \texttt{int}}{\Sigma; \Gamma; \Delta; \Psi \ \vdash\ \texttt{refineint}\ [k]\,\hat{}\,(N) : \overline{\texttt{int}}\ k}\ \overline{\texttt{int}} - I
$$

$$
\frac{\Sigma \ \vdash\ k : \texttt{chan} \qquad \Sigma; \Gamma; \Delta; \Psi \ \vdash\ N : \overline{\texttt{int}}\ k}{\Sigma; \Gamma; \Delta; \Psi \ \vdash\ \texttt{fetchint}\ [k]\,\hat{}\,(N) : \texttt{int}}\ \overline{\texttt{int}} - E
$$

and the reduction rules

$$
\frac{N \ \rightarrow\ N'}{\texttt{fetchint}\ [k]\,\hat{}\,N \ \rightarrow\ \texttt{fetchint}\ [k]\,\hat{}\,N'}
$$

$$
\frac{N \ \rightarrow\ N'}{\texttt{refineint}\ [k]\,\hat{}\,N \ \rightarrow\ \texttt{refineint}\ [k]\,\hat{}\,N'}
$$

$$
\frac{}{\texttt{fetchint}\ [k]\,\hat{}\,(\texttt{refineint}\ [k]\,\hat{}\,n) \ \rightarrow\ n}
$$

The function $\texttt{fibc}'$ shown in figure 27 is a correctly implemented concurrent version of `fib` that takes as input a channel name k and an integer n and returns $fib(n)$ refined by channel name k i.e. the value $(\texttt{refineint}\ [k]\,\hat{}\,(fib(n)))$. It has the type $\forall k : \texttt{chan}.\ \texttt{int} \rightarrow \{!(\overline{\texttt{int}}\ k)\}$. In this case there is exactly one possible program execution.

```
fun fibc′ [k] (n) =
  if (n = 0) then {!(refineint [k] ˆ 1)}
  else if (n = 1) then {!(refineint [k] ˆ 1)}
  else
  {
    link
    (
      (
        priv  k₁ : chan in
        priv  k₂ : chan in
              fibc′ [k₁] (n − 1)
            ⊗ fibc′ [k₂] (n − 2)
            ⊗ λn₁ : int̄ k₁. λn₂ : int̄ k₂.
               {!(refineint [k] ˆ ((fetchint [k₁] ˆ n₁) + (fetchint [k₂] ˆ n₂)))}
      )
     ÷ ∃k₁ : chan. ∃k₂ : chan. ({!(int̄ k₁)} ⊗ {!(int̄ k₂)}⊗
          (int̄ k₁ → int̄ k₂ → {!(int̄ k)}))
    ) to !(int̄ k)
  }
```

Figure 27: The function fibc′

5.2 Programming Technique: Buffered Asynchronous Message Passing

We assume that we have a conditional if-then-else construct for terms, monadic terms and expressions. This construct has the form $\underline{\texttt{if}}\ N\ \underline{\texttt{then}}\ T_1\ \underline{\texttt{else}}\ T_2$ (T stands for any of N, M or E). The associated typing and reduction rules are shown below.

$$T \ ::= \ \ldots \mid \underline{\texttt{if}}\ N\ \underline{\texttt{then}}\ T_1\ \underline{\texttt{else}}\ T_2$$

$$\frac{\Sigma;\Gamma;\Delta;\Psi \vdash N : \texttt{bool} \qquad \Sigma;\Gamma;\Delta';\Psi \vdash T_1 \,\%\, Z \qquad \Sigma;\Gamma;\Delta';\Psi \vdash T_2 \,\%\, Z}{\Sigma;\Gamma;\Delta,\Delta';\Psi \vdash \underline{\texttt{if}}\ N\ \underline{\texttt{then}}\ T_1\ \underline{\texttt{else}}\ T_2 \,\%\, Z}\ \underline{\texttt{if}}-\underline{\texttt{then}}-\underline{\texttt{else}}$$

$$\frac{N \ \rightarrow \ N'}{\underline{\texttt{if}}\ N\ \underline{\texttt{then}}\ T_1\ \underline{\texttt{else}}\ T_2 \ \hookrightarrow \ \underline{\texttt{if}}\ N'\ \underline{\texttt{then}}\ T_1\ \underline{\texttt{else}}\ T_2}$$

$$\underline{\texttt{if}}\ true\ \underline{\texttt{then}}\ T_1\ \underline{\texttt{else}}\ T_2 \ \hookrightarrow \ T_1$$

$$\underline{\texttt{if}}\ false\ \underline{\texttt{then}}\ T_1\ \underline{\texttt{else}}\ T_2 \ \hookrightarrow \ T_2$$

We now build a library of programs in CLL to allow us to write programs that use asynchronous, queue based message passing. For every channel name k that is to be used for communication of values of the asynchronous type B, we introduce a first-in, first-out queue of elements of type B into our CHAM solution. In order to distinguish various queues in a CHAM solution just by their types, we refine the queue type with channel names. Queues have the following abstract specification.

```
abstype queue_B:  chan → Type with
  empty:  ∀i : chan. queue_B i
```

$$
\begin{array}{l}
\texttt{push:}\quad \forall i:\mathtt{chan}.\ \mathtt{queue}_B\ \ i \multimap B \rightarrow \mathtt{queue}_B\ \ i \\
\texttt{isempty:}\quad \forall i:\mathtt{chan}.\ \mathtt{queue}_B\ \ i \multimap \{!\mathtt{bool} \otimes \mathtt{queue}_B\ \ i\} \\
\texttt{pop:}\quad \forall i:\mathtt{chan}.\ \mathtt{queue}_B\ \ i \multimap \{!B \otimes \mathtt{queue}_B\ \ i\} \\
\texttt{top:}\quad \forall i:\mathtt{chan}.\ \mathtt{queue}_B\ \ i \multimap \{!B \otimes \mathtt{queue}_B\ \ i\} \\
\texttt{destroy:}\quad \forall i:\mathtt{chan}.\ \mathtt{queue}_B\ \ i \multimap \{1\}
\end{array}
$$

The above first-in, first-out queue may be implemented using data structures like ML-style lists, which we assume are present in our language. The exact details of the implementation are not relevant to our discussion. A more important fact is that queues are linear objects in CHAM solutions, and hence can be used to capture the notion of state of communication on a particular channel. On the other hand, the data within the queue is non-linear and can be used multiple times. One can also design a different model of communication in which the data in the queue is linear. For any queue of type queue k, we view the elements in the queue as messages that are *pending to be read* on channel k. Message sending in this model is asynchronous in the sense that a sender simply appends its message to the end of a message queue and continues execution. It does not wait for a receiver to receive the message. Thus we can define a 'send' macro:

$$
\begin{array}{l}
\textbf{asyncsend}(k, N : B); M\ \ \#\ \ S\ \ = \\
\quad \hat{\lambda} q : \mathtt{queue}_B\ \ k.\ \ \{(\mathtt{push}\ \ [k]\ \ \hat{}\ \ q\ \ N)\ \ \otimes\ \ M\}
\end{array}
$$

Intuitively, the above macro should be read as "send the result of evaluating N on channel k and continue with the process M".[6] If we define the type $\mathtt{Asyncsend}(k, B, S) = \mathtt{queue}_B\ k \multimap \{\mathtt{queue}_B\ k\ \otimes\ S\}$, then the derived typing rule for **asyncsend** is

$$
\frac{\Sigma \vdash k : \mathtt{chan} \qquad \Sigma; \Gamma; \cdot; \Psi \vdash N : B \qquad \Sigma; \Gamma; \Delta; \Psi \vdash M \# S}{\Sigma; \Gamma; \Delta; \Psi \vdash (\textbf{asyncsend}(k, N : B); M \# S) \# \mathtt{Asyncsend}(k, B, S)}\ \textbf{asyncsend}
$$

The corresponding receive macro is harder to create. Suppose we want to bind x to a value received on the channel k in the monadic term M. Then we need to wait till there is a message pending on the message queue for channel k. This we do by repeatedly synchronizing with the associated queue and checking for non-emptiness. If the queue is empty, we leave the queue and keep waiting. If it is non-empty, we pop the queue, bind the value popped to x and return the popped queue to the solution. The following receive macro implements this.

$$
\begin{array}{l}
\textbf{asyncrecv}\ \ x : B\ \ \mathtt{on}\ \ k\ \ \mathtt{in}\ \ M\ \ \#\ \ S\ \ = \\
\quad \mu u.\ \ \underline{\mathtt{fold}}_{\mathtt{Asyncrecv}(k,B,S)}.\ \ \hat{\lambda} q : \mathtt{queue}_B\ \ k. \\
\quad \{ \\
\qquad \underline{\mathtt{let}}\ \ \{!b \otimes q'\} = \mathtt{isempty}\ \ [k]\ \ \hat{}\ \ q\ \ \underline{\mathtt{in}} \\
\qquad\quad \underline{\mathtt{if}}\ \ b\ \ \underline{\mathtt{then}}\ \ \underline{\mathtt{inl}}\ \ (u \otimes q') \\
\qquad\quad \underline{\mathtt{else}} \\
\qquad\qquad \underline{\mathtt{let}}\ \ \{!x \otimes q''\} = \mathtt{pop}\ \ [k]\ \ \hat{}\ \ q'\ \ \underline{\mathtt{in}} \\
\qquad\qquad\quad \underline{\mathtt{inr}}\ \ (M\ \ \otimes\ \ q'') \\
\quad \}
\end{array}
$$

The type Asyncrecv is defined as follows

$$
\mathtt{Asyncrecv}(k, B, S) = \mu\alpha.\ \mathtt{queue}_B\ k \multimap \{(\alpha \otimes \mathtt{queue}_B\ k) \oplus (S \otimes \mathtt{queue}_B\ k)\}
$$

[6] N may not be a value and might be evaluated in parallel with M itself. Since evaluation of pure terms has no side effects, the exact point of evaluation of N does not matter.

The derived typing rule for this macro is:

$$\frac{\Sigma \vdash k : \texttt{chan} \qquad \Sigma; \Gamma, x : B; \cdot; \Psi \vdash M \# S}{\Sigma; \Gamma; \cdot; \Psi \vdash (\textbf{asyncrecv}\ x : B \ \texttt{on}\ k\ \texttt{in}\ M : S) \# \texttt{Asyncrecv}(k, B, S)}\ \textbf{asyncrecv}$$

Next we define a macro to actually create a private channel name for communication. This macro uses the previously defined macro `priv` . In addition to creating the private channel name, it also creates a new queue to be used for communication on the channel. This is done by a call to the function `empty` from the specification of the type $\texttt{queue}_B$.

privasyncchan k `in` M $\#$ $S(k)$ =
  `priv` $k : \texttt{chan}$ `in` $(M \ \otimes \ (\texttt{empty}\ [k]))\ \#\ (S(k) \ \otimes \ (\texttt{queue}_B\ \ k))$

If we define the type $\texttt{Privasyncchan}(B, k.S(k)) = \exists k : \texttt{chan}.(S \otimes \texttt{queue}_B\ k)$, then the typing rule for the above construct is[7]

$$\frac{\Sigma, k : \texttt{chan}; \Gamma; \Delta; \Psi \vdash M \# S(k)}{\Sigma; \Gamma; \Delta; \Psi \vdash (\textbf{privasyncchan}\ k\ \texttt{in}\ M \# S(k)) \# \texttt{Privasyncchan}(B, k.S(k))}\ \textbf{privasyncchan}$$

Finally we define a cleanup macro that destroys the message queue associated with a channel. This macro is used when the channel is no longer needed for communication. Once this macro is used on a channel, subsequent attempts to send or receive on the channel will deadlock.

destroyasyncchan $k; M\ \#\ S$ =
  $\hat{\lambda} q : \texttt{queue}_B\ \ k.\ \{\ \underline{\texttt{let}}\ \{1\} = \texttt{destroy}\ \ [k]\ \ \hat{}\ \ q\ \ \underline{\texttt{in}}\ \ M\ \ \}$

If we define $\texttt{Destroyasyncchan}(k, B, S) = \texttt{queue}_B\ k \multimap \{S\}$, then we have the following derived typing rule

$$\frac{\Sigma \vdash k : \texttt{chan} \qquad \Sigma; \Gamma; \Delta; \Psi \vdash M \# S}{\Sigma; \Gamma; \Delta; \Psi \vdash (\textbf{destroyasyncchan}\ \ k; M \# S) \# \texttt{Destroyasyncchan}(k, B, S)}\ \textbf{destroyasyncchan}$$

All the above constructs are summarized in figure 28. We often omit type annotations from these constructs if they are clear from the context.

5.3 Example: Sieve of Eratosthenes

We build a concurrent version of the sieve for Eratosthenes for filtering prime numbers from a sequence $[2, \ldots, n]$. This example uses the asynchronous message passing mechanism described earlier. For this example, the messages we send on channels are integers and hence the queue data structure described earlier uses $B = \texttt{int}$. We omit the type annotation `int` from the type $\texttt{queue}_{\texttt{int}}$. We begin with a function that sends all numbers from 2 to N on channel k. Let us assume we have a special integer called END which we use to signal end of data on a (message) queue. This function called `integersupto` is shown in figure 29. It has the type $\forall k : \texttt{chan}.\ \texttt{int} \rightarrow \{\mu\alpha.\texttt{Asyncsend}(k, \texttt{int}, 1 \oplus \{\alpha\})\}$.

If we allow `integersupto` $[k]$ N to execute in a CHAM, then each recursive call of the loop adds a new integer to the queue associated with channel k. Eventually the condition $n > N$ succeeds and `integersupto`

[7] We use the notation $k.S(k)$ to indicate that k is bound in the type $\texttt{Privasyncchan}(B, k.S(k))$.

Types

$\texttt{Asyncsend}(k, B, S) = \texttt{queue}_B\ k \multimap \{\texttt{queue}_B\ k\ \otimes\ S\}$
$\texttt{Asyncrecv}(k, B, S) = \mu\alpha.\ \texttt{queue}_B\ k \multimap \{(\alpha \otimes \texttt{queue}_B\ k) \oplus (S \otimes \texttt{queue}_B\ k)\}$
$\texttt{Privasyncchan}(B, k.S(k)) = \exists k : \texttt{chan}.(S(k) \otimes \texttt{queue}_B\ k)$
$\texttt{Destroyasyncchan}(k, B, S) = \texttt{queue}_B\ k \multimap \{S\}$

Macros

$\textbf{asyncsend}(k, N : B); M\ \#\ S$ =
$\quad\hat{\lambda}q : \texttt{queue}_B\ \ k.\ \ \{(\texttt{push}\ \ [k]\ \ \hat{}\ \ q\ \ N)\ \ \otimes\ \ M\}$

$\textbf{asyncrecv}\ \ x : B\ \ \texttt{on}\ \ k\ \ \texttt{in}\ \ M\ \ \#\ \ S$ =
$\quad\mu u.\ \ \underline{\texttt{fold}}_{\texttt{Asyncrecv}(k,B,S)}.\ \ \hat{\lambda}q : \texttt{queue}_B\ \ k.$
$\quad\{$
$\qquad\underline{\texttt{let}}\ \ \{!b \otimes q'\} = \texttt{isempty}\ \ [k]\ \ \hat{}\ \ q\ \ \underline{\texttt{in}}$
$\qquad\quad\underline{\texttt{if}}\ \ (b = true)\ \ \underline{\texttt{then}}\ \ \underline{\texttt{inl}}\ \ (u \otimes q')$
$\qquad\quad\underline{\texttt{else}}$
$\qquad\qquad\underline{\texttt{let}}\ \ \{!x \otimes q''\} = \texttt{pop}\ \ [k]\ \ \hat{}\ \ q'\ \ \underline{\texttt{in}}$
$\qquad\qquad\quad\underline{\texttt{inr}}\ \ (M\ \ \otimes\ \ q'')$
$\quad\}$

$\textbf{privasyncchan}\ \ k\ \ \texttt{in}\ \ M\ \ \#\ \ S(k)$ =
$\quad\texttt{priv}\ \ k : \texttt{chan}\ \ \ \texttt{in}\ \ (M\ \ \otimes\ \ (\texttt{empty}\ \ [k]))\ \ \#\ \ (S(k)\ \ \otimes\ \ (\texttt{queue}_B\ \ k))$

$\textbf{destroyasyncchan}\ \ k; M\ \ \#\ \ S$ =
$\quad\hat{\lambda}q : \texttt{queue}_B\ \ k.\ \ \{\ \ \underline{\texttt{let}}\ \ \{1\} = \texttt{destroy}\ \ [k]\ \ \hat{}\ \ q\ \ \underline{\texttt{in}}\ \ M\ \ \}$

Typing Rules

$$\frac{\Sigma\ \vdash\ k : \texttt{chan} \qquad \Sigma; \Gamma; \cdot; \Psi\ \vdash\ N : B \qquad \Sigma; \Gamma; \Delta; \Psi\ \vdash\ M \# S}{\Sigma; \Gamma; \Delta; \Psi\ \vdash\ (\textbf{asyncsend}(k, N : B); M \# S) \# \texttt{Asyncsend}(k, B, S)}\ \text{asyncsend}$$

$$\frac{\Sigma\ \vdash\ k : \texttt{chan} \qquad \Sigma; \Gamma, x : B; \cdot; \Psi\ \vdash\ M \# S}{\Sigma; \Gamma; \cdot; \Psi\ \vdash\ (\textbf{asyncrecv}\ x : B\ \text{on}\ k\ \text{in}\ M : S) \# \texttt{Asyncrecv}(k, B, S)}\ \text{asyncrecv}$$

$$\frac{\Sigma, k : \texttt{chan}; \Gamma; \Delta; \Psi\ \vdash\ M \# S(k)}{\Sigma; \Gamma; \Delta; \Psi\ \vdash\ (\textbf{privasyncchan}\ k\ \text{in}\ M \# S(k)) \# \texttt{Privasyncchan}(B, k.S(k))}\ \text{privasyncchan}$$

$$\frac{\Sigma\ \vdash\ k : \texttt{chan} \qquad \Sigma; \Gamma; \Delta; \Psi\ \vdash\ M \# S}{\Sigma; \Gamma; \Delta; \Psi\ \vdash\ (\textbf{destroyasyncchan}\ \ k; M \# S) \# \texttt{Destroyasyncchan}(k, B, S)}\ \text{destroyasyncchan}$$

Figure 28: Macros for asynchronous communication

```
(* integersupto:  ∀k : chan. int → {μα.Asyncsend(k, int, 1 ⊕ {α})} *)

fun integersupto [k:chan] (N:int) =

   (* loop:  int → {μα.Asyncsend(k, int, 1 ⊕ {α})} *)
   let val loop (n : int) =
   {
      fold_{μα.Asyncsend(k,int,1⊕{α})}.
         if (n > N) then
            asyncsend(k, END);  inl 1
         else
            asyncsend(k, n);  inr (loop(n + 1))
   }
   in
      loop 2
   end
```

Figure 29: The function `integersupto`

terminates with <u>inl</u> 1. Note that other **asyncsend** and **asyncrecv** calls on the channel k can be interleaved. For example, if at some point of time, the message queue on k has integers $2 \ldots 10$, then some other process may use the macro **asyncrecv** up to nine times on channel k before any more integers are sent by `integersupto`. Next we write a filter function which given an input channel inp, an output channel out and a prime p, filters the integers on inp for numbers *not divisible* by p and writes the output to channel out. This function is shown in figure 30. It has the type $\forall inp : \texttt{chan}.\ \forall out : \texttt{chan}.\ \texttt{int} \rightarrow \{F(inp, out)\}$.

Next we come to the program sieve which takes an input channel inp and an output channel out and filters the input channel for all integers that are relatively prime to their predecessors on the same channel. These filtered integers are written to the channel out. This program is shown in figure 31. It has the type $\{\forall inp : \texttt{chan}.\ \forall out : \texttt{chan}.\ \{R(inp, out)\}\}$ where $R(inp, out)$ is the type:

```
type R(inp, out) =
       Asyncrecv(inp, int,
          Destroyasyncchan(inp, int, Asyncsend(out, int, 1))
          ⊕ Asyncsend(out, int, Privasyncchan(int, k.({F(inp, k)} ⊗ R(k, out))))
          )
```

The type $R(inp, out)$ is not a regular recursive type since it cannot be expressed using the standard recursive construct $\mu\alpha.S$. Instead, it requires recursive definitions or recursive type binders at kinds higher than `Type`. Either may be added to the language without much technical difficulty. Recursive definitions, in particular, can be added using the standard <u>fold</u> construction as follows. Under the assumption that we have a definition $R(i_1 \ldots i_n) = S$, where the synchronous type S may mention R again, we have the following typing rules:

$$\frac{\Sigma; \Gamma; \Delta; \Psi \vdash M \# S}{\Sigma; \Gamma; \Delta; \Psi \vdash \underline{\texttt{fold}}_{R(i_1 \ldots i_n)}(M) \# R(i_1 \ldots i_n)}\ \underline{\texttt{fold}}\text{-R}'$$

```
type F(inp, out) =
      μα. Asyncrecv(inp, int, Destroyasyncchan(inp, int, Asyncsend(out, int, 1))⊕
      (α ⊕ Asyncsend(out, int, α)))

(* filter : ∀inp : chan. ∀out : chan. int → {F(inp, out)} *)

fun filter [inp] [out] p =
{
   μu. fold_F(inp,out)
      asyncrecv n : int on inp in
         if (n = END)
         then inl
            destroyasyncchan inp in asyncsend(out, END); 1
         else inr
            if (n mod p = 0) then inl u
            else inr (asyncsend(out, n); u)
}
```

Figure 30: The function `filter`

$$\frac{\Sigma; \Gamma; \Delta, p : S; \Psi \;\vdash\; \gamma}{\Sigma; \Gamma; \Delta, \underline{\texttt{fold}}_{R(i_1 \ldots i_n)}(p) : R(i_1 \ldots i_n); \Psi \;\vdash\; \gamma}\ \underline{\texttt{fold}}\text{-L}'$$

The statement $\underline{\texttt{let}}\ \{!f\} = \{!u\}\ \underline{\texttt{in}}\ \ldots$ in the body of `sieve` binds f to a pure term which has the same behavior and type as the recursive variable u. We integrate all the functions together to produce a single function `primes` that takes a channel name *out* and an integer N and produces as output a single queue of type queue *out* containing all primes up to N. This function is shown in figure 32. It has the type $\forall out : \texttt{chan}.\ \texttt{int} \rightarrow \{\texttt{queue}\ out\}$.

5.4 Implementing Buffered Asynchronous Message Passing using Functions

As mentioned in the introduction to section 5, it is possible to rewrite all the macros for buffered asynchronous message passing presented in section 5.2 as functions. In this section we present the functional equivalents of all the macros of section 5.2. Similar transformations can be applied to all macros presented in later sections. The purpose of doing this is to establish that the library code presented here can be represented using the abstraction mechanisms in CLL, and the use of macros is merely a convenience rather than a necessity. We start by writing an equivalent functional representation of the macro **asyncsend**(see figure 28). As can be seen, this macro requires three arguments - a channel name k, a value N of type B to send on the channel and a continuation M of type S. The type of **asyncsend** $(k, N : B); M \# S$ is $\texttt{Asyncsend}(k, B, S)$. This suggests the type for the corresponding functional abstraction: $\forall k : \texttt{chan}.\ B \rightarrow \{S\} \multimap \texttt{Asyncsend}(k, B, S)$. We observe three facts here. First, the argument of type B is unrestricted because we want values passed on channels to be unrestricted. Second, we have to pass M after enclosing it in a monad because due to syntactic restrictions in CLL, we cannot pass monadic terms as arguments. Hence the second argument of the functional abstraction is of type $\{S\}$ instead of S. Third, since we do not have polymorphism in CLL, we need a separate function for each pair of types (B, S). All these functions

```
type R(inp, out) =
    Asyncrecv(inp, int,
        Destroyasyncchan(inp, int, Asyncsend(out, int, 1))
        ⊕ Asyncsend(out, int, Privasyncchan(int, k.({F(inp, k)} ⊗ R(k, out))))
        )

(* sieve : {∀inp : chan. ∀out : chan. {R(inp, out)}} *)

sieve =
  {
    μu. Λinp : chan. Λout : chan.
    {
      let {!f} = {!u} in
        fold_R(inp,out)
          asyncrecv p : int on inp in
            if (p = END)
            then inl
                (destroyasyncchan inp in asyncsend(out, END) in 1)
            else inr
              asyncsend(out, p); privasyncchan k in
                (filter [inp] [k] p) ⊗ (f [k] [out])
    }
  }
```

Figure 31: The program `sieve`

```
(* primes : ∀out : chan. int → {queue out} *)

fun primes [out : chan] (N : int) =
{
  let {f} = sieve in
    link
    (
      privasyncchan k in (integersupto [k] N) ⊗ (f [k] [out])
       ÷ Privasyncchan(k, int, {μα.Asyncsend(k, int, 1 ⊕ {α})} ⊗ {R(k, out)})
    ) to queue out
}
```

Figure 32: The function `primes`

look exactly the same, except that they have different types. Assuming fixed types B and S, the function `asyncsend'` is shown below. It has the type $\forall k : \texttt{chan}.\ B \rightarrow \{S\} \multimap \texttt{Asyncsend}(k, B, S)$.

```
fun asyncsend' [k : chan] (N : B) (M : {S}) =
    λ̂q : queue_B k.
    {
        let {m'} = M in
            (push [k] ^ q N) ⊗ m'
    }
```

Now we consider the macro **asyncrecv** . This macro takes two arguments - a channel name k on which input is to be received and a monadic term M of type S that has a free variable x of type B that is to be bound to the input value received on the channel k. We can represent the second argument, M, as a function of type $B \rightarrow \{S\}$. This gives us the type of the functional abstraction corresponding to **asyncrecv** : $\forall k : \texttt{chan}.\ (B \rightarrow \{S\}) \rightarrow \{\texttt{Asyncrecv}(k, B, S)\}$. We observe that the return type of this function is $\{\texttt{Asyncrecv}(k, B, S)\}$ instead of $\texttt{Asyncrecv}(k, B, S)$ because $\texttt{Asyncrecv}(k, B, S)$ is a synchronous type and owing to syntactic restrictions in CLL, it cannot be returned directly by a function. The functional abstraction `asyncrecv'` is shown below. It has the type $\forall k : \texttt{chan}.\ (B \rightarrow \{S\}) \rightarrow \{\texttt{Asyncrecv}(k, B, S)\}$.

```
fun asyncrecv' [k : chan] (M : B → {S}) =
    {
        μu. fold_Asyncrecv(k,B,S). λ̂q : queue_B k.
        {
            let {!b ⊗ q'} = isempty [k] ^ q in
                if (b = true) then inl (u ⊗ q')
                else
                    let {!x ⊗ q''} = pop [k] ^ q'
                    let {m'} = M x in
                        inr (m' ⊗ q'')
        }
    }
```

Next we come to the macro **privasyncchan** . This macro takes as argument a monadic term M of type $S(k)$ where k is a parameterized channel name (see the typing rule for **privasyncchan** in figure 28). In terms of abstractions, such a monadic term can be represented by the type $\forall k : \texttt{chan}.\{S(k)\}$. The functional abstraction `privasyncchan` that corresponds to the macro **privasyncchan** ′ is shown below. It has the type $(\forall k : \texttt{chan}.\{S(k)\}) \multimap \{\exists k : \texttt{chan}.\{S(k) \otimes \texttt{queue}_B\ k\}\}$. It is instructive to compare this function and its return type to the macro **privasyncchan** and the type $\texttt{Privasyncchan}(B, k.S(k))$ respectively.

```
fun privasyncchan' (M : (∀k : chan.{S(k)})) =
    {
        priv k : chan in
        {
            let {m'} = M [k] in
                (m' ⊗ (empty [k])) # (S(k) ⊗ (queue_B k))
        }
    }
```

Finally we consider the macro **destroyasyncchan** . This macro takes two arguments - a channel name k and a continuation M of type S. Writing an equivalent functional representation for this macro is straightforward and is shown below. The function `destroyasyncchan'` shown below has the type $\forall k : \texttt{chan}. \{S\} \multimap \texttt{Destroyasyncchan}(k, B, S)$.

$$
\begin{array}{l}
\texttt{fun destroyasyncchan'}\ [k : \texttt{chan}]\ (M : \{S\})\ = \\
\quad \hat{\lambda} q : \texttt{queue}_B\ \ k. \\
\quad \{ \\
\quad\quad \underline{\texttt{let}}\ \ \{1\} = \texttt{destroy}\ \ [k]\ \ \hat{}\ \ q \\
\quad\quad \underline{\texttt{let}}\ \ \{m'\} = M\ \ \underline{\texttt{in}}\ \ m' \\
\quad \}
\end{array}
$$

Thus the abstraction mechanisms in CLL are expressive enough to allow us to write all the macros presented so far as functions. However, the bodies and types of these functions are more complicated than those of the corresponding macros. For the sake of conciseness and clarity we present the remaining library code only as macros. It should, however, be kept in mind that all these macros can be represented as functions as well.

5.5 Programming Technique: Synchronous Message Passing

The communication primitive in CLL is inherently asynchronous. The basic communication primitive is to use the theorem prover to link together a function of type $P \multimap B$ and a value of the input type P using the rule $\Longrightarrow - \multimap$. In this case the value itself is consumed and hence senders have no continuation i.e. they are asynchronous. In section 5.2, we built a library of macros to extend this communication primitive to allow queuing of messages on a channel. However, communication was asynchronous in the sense that senders received no confirmation that the message sent had been received before they were allowed to continue evaluation. Now we build a library of macros to implement synchronous communication, where senders receive confirmation that their message has been received before they are allowed to continue execution. As expected, this requires implementation of a protocol over the primitive asynchronous communication. The protocol we choose is based on a protocol in [6] to implement the synchronous π-calculus (without choices) in the asynchronous π-calculus. It works as follows. Suppose a sender S wants to sends a value V to receiver R on channel k. S and R create a private channel each. Let us call these u and t respectively. First, S sends the channel name u to R on channel k. Once R knows the channel name u, it sends back the channel name t on the channel u to S. S now forks - in one thread it sends V to R on t and in the other it resumes execution with its continuation. R on receiving V on t resumes its own execution. In π-calculus notation, this translation is represented as follows.

$$
\begin{array}{lcll}
\langle\!\langle\, \bar{k}V.\, P' \,\rangle\!\rangle & = & \nu u.\, (\bar{k}u \mid u(t).\, (\bar{t}V \mid \langle\!\langle\, P' \,\rangle\!\rangle)) & (1) \\
\langle\!\langle\, k(y).\, P \,\rangle\!\rangle & = & \nu t.\, k(u).\, (\bar{u}t \mid t(y).\, \langle\!\langle\, P \,\rangle\!\rangle) & (2)
\end{array}
$$

In order to implement this protocol, we assume that we have the constructor-destructor pairs $(\underline{\texttt{out}_0}, \underline{\texttt{destroyout}_0})$ and $(\underline{\texttt{out}_1}, \underline{\texttt{destroyout}_1})$ and the corresponding kinds $\texttt{out}_0$ and $\texttt{out}_1$ from section 3.3. The signature for these constants is reproduced below.

$$
\begin{array}{lcl}
\texttt{out}_0 & : & \texttt{chan} \rightarrow \texttt{Type} \\
\texttt{out}_1 & : & \texttt{chan} \rightarrow \texttt{chan} \rightarrow \texttt{Type} \\
\underline{\texttt{out}_0} & : & \forall x : \texttt{chan}.\ \texttt{out}_0\ x \\
\underline{\texttt{out}_1} & : & \forall x : \texttt{chan}.\ \forall y : \texttt{chan}.\ \texttt{out}_1\ x\ y \\
\underline{\texttt{destroyout}_0} & : & \forall x : \texttt{chan}.\ \texttt{out}_0\ x \multimap \{1\} \\
\underline{\texttt{destroyout}_1} & : & \forall x : \texttt{chan}.\ \forall y : \texttt{chan}.\ \texttt{out}_1\ x\ y \multimap \{1\}
\end{array}
$$

We also need a datatype to encode data being sent on channel k. Our signature for this datatype is

$$\begin{array}{lcl} \mathtt{data}_B & : & \mathtt{chan} \rightarrow \mathtt{Type} \\ \underline{\mathtt{data}}_B & : & \forall x : \mathtt{chan}.\ B \multimap \mathtt{data}_B\ x \\ \underline{\mathtt{undata}}_B & : & \forall x : \mathtt{chan}.\ \mathtt{data}_B\ x \multimap B \end{array}$$

The corresponding reduction rules are:

$$\frac{}{\underline{\mathtt{undata}}_B\ [k]\,\hat{}\,(\underline{\mathtt{data}}_B\ [k]\,\hat{}\,V) \ \rightarrow\ V}$$

$$\frac{N \ \rightarrow\ N'}{\underline{\mathtt{data}}_B\ [k]\,\hat{}\,N \ \rightarrow\ \underline{\mathtt{data}}_B\ [k]\,\hat{}\,N'}$$

$$\frac{N \ \rightarrow\ N'}{\underline{\mathtt{undata}}_B\ [k]\,\hat{}\,N \ \rightarrow\ \underline{\mathtt{undata}}_B\ [k]\,\hat{}\,N'}$$

The actual implementation of the send and receive macros uses the same encoding as in section 3.3. The synchronous send macro, called **syncsend** is defined below.

$$\begin{array}{l}
\textbf{syncsend}\ \ (k, N : B); M\ \ \#\ \ S\ \ = \\
\quad \mathtt{priv}\ \ u : \mathtt{chan}\ \ \mathtt{in} \\
\qquad (\underline{\mathtt{out}}_1\ \ [k]\ \ [u])\ \ \otimes \\
\qquad \Lambda t : \mathtt{chan}.\ \ \hat{\lambda} c : \mathtt{out}_1\ \ u\ \ t. \\
\qquad \{ \\
\qquad\quad \underline{\mathtt{let}}\ \ \{1\} = \underline{\mathtt{destroyout}}_1\ \ [u]\ \ [t]\ \ \hat{}\ \ c \\
\qquad\quad \underline{\mathtt{in}} \\
\qquad\qquad (\underline{\mathtt{data}}_B\ \ [t]\ \ \hat{}\ \ N)\ \ \otimes\ \ M \\
\qquad \}
\end{array}$$

Let us define the type $\mathtt{Syncsend}(k, B, S)$ as follows.

$$\mathtt{Syncsend}(k, B, S) = \exists u : \mathtt{chan}.\ ((\mathtt{out}_1\ k\ u)\ \otimes\ (\forall t : \mathtt{chan}.\ \mathtt{out}_1\ u\ t \multimap \{(\mathtt{data}_B\ t)\ \otimes\ S\}))$$

The derived typing rule for this construct is

$$\frac{\Sigma\ \vdash\ k : \mathtt{chan} \qquad \Sigma; \Gamma; \Delta; \Psi\ \vdash\ M \# S \qquad \Sigma; \Gamma; \Delta'; \Psi\ \vdash\ N : B}{\Sigma; \Gamma; \Delta, \Delta'; \Psi\ \vdash\ (\textbf{syncsend}(k, N : B); M \# S) \# \mathtt{Syncsend}(k, B, S)}\ \textbf{syncsend}$$

The definitions **syncsend** and $\mathtt{Syncsend}$ correspond to the translation of the right hand side of equation (1) according to the rules in figure 22. The corresponding synchronous receive macro is the following.

$$\begin{array}{l}
\textbf{syncrecv}\ \ y : B\ \ \mathtt{on}\ \ k\ \ \mathtt{in}\ \ M\ \ \#\ \ S\ \ = \\
\quad \mathtt{priv}\ \ t : \mathtt{chan}\ \ \mathtt{in} \\
\qquad \Lambda u : \mathtt{chan}.\ \ \hat{\lambda} c : \mathtt{out}_1\ \ k\ \ u. \\
\qquad \{ \\
\qquad\quad \underline{\mathtt{let}}\ \ \{1\} = \underline{\mathtt{destroyout}}_1\ \ [k]\ \ [u]\ \ \hat{}\ \ c\ \ \underline{\mathtt{in}} \\
\qquad\qquad (\underline{\mathtt{out}}_1\ \ [u]\ \ [t])\ \ \otimes \\
\qquad\qquad \hat{\lambda} y' : \mathtt{data}_B\ \ t.
\end{array}$$

```
type Buf_B(read, write) = μα.Syncrecv(write, B, Syncsend(read, B, α))

(* oneCellBuffer_B : ∀read : chan. ∀write : chan. {Buf_B(read, write)} *)

fun oneCellBuffer_B [read : chan] [write : chan] =
{
  μu. fold_{Buf_B(read,write)}
      syncrecv x : B on write in
         syncsend(read, x); u
}
```

Figure 33: The function `oneCellBuffer`

```
            {
                let {y} = {undata_B [t] ^ y'}
                in M
            }
        }
```

We define the type $\texttt{Syncrecv}(k, B, S)$ as follows.

$$\texttt{Syncrecv}(k, B, S) = \exists t : \texttt{chan}.\ \forall u : \texttt{chan}.\ \texttt{out}_1\ k\ u \multimap \{(\texttt{out}_1\ u\ t)\ \otimes\ (\texttt{data}_B\ t \multimap \{S\})\}$$

Then the derived typing rule for **syncrecv** is

$$\frac{\Sigma \vdash k : \texttt{chan} \qquad \Sigma; \Gamma; \Delta, y : B; \Psi \vdash M \# S}{\Sigma; \Gamma; \Delta; \Psi \vdash (\textbf{syncrecv}\ y : B\ \texttt{on}\ k\ \texttt{in}\ M \# S) \# \texttt{Syncrecv}(k, B, S)}\ \textbf{syncrecv}$$

Again, this encoding is actually the translation of the right hand side of equation (2) according to the rules in figure 22.

5.6 Example: One Cell Buffer

Using the synchronous send and receive methods defined earlier, we define a one cell buffer[8]. This buffer operates on two channels $read$ and $write$ which are used to read and write to the buffer. When the buffer is empty, sending a value on $write$ has the effect of storing this value in the buffer. Subsequently, attempts to write to the buffer block, until some process reads the buffer on channel $read$. After the buffer is read, attempts to read block until the buffer is written to again. This implementation is shown in figure 33.

5.7 Programming Technique: Synchronous Choices

Choice in the context of concurrent programming refers to a primitive that allows the system to non-deterministically choose from one of several possibilities. The candidates for the choice may be values, events (like send and receive) or processes. Usually, the choice is based on some criteria i.e. not all of the

[8] A one cell buffer is also called an M-structure.

possibilities are considered as possible candidates for selection. The simplest notion of choice is *internal* choice, where the executing process spontaneously selects from several possible alternatives and continues with one of these. In our system, the type $S_1 \oplus S_2$ represents internal choice between processes. A monadic term of this type may evaluate to a monadic value of type S_1 or S_2. The environment in which the process computes plays no role in this selection. Thus this kind of choice is internal. Another very useful kind of choice is *external*. This is a choice resolved by the environment, based on some selection criteria. In process-calculi like π, several variants of external choice have been suggested. Most of these are based on selecting some input or output action. For example, in the synchronous π-calculus [24, 25], there is an associative and commutative (AC) choice operator $[]$ and a syntactic class C to represent external choice between input and output actions[9].

$$C ::= x(y).P \mid \bar{x}y.P \mid C_1[]C_2$$

The semantics of this operator are as follows.

$$(x(y).P \; [] \; C_1) \mid (\bar{x}z.P' \; [] \; C_2) \;\rightarrow\; P[z/y] \mid P'$$

A choice may be resolved by the environment in favor of an action if there is a corresponding co-action. In [27] it is shown that this kind of choice is strictly more expressive than internal choice and *primitive* in the sense that it cannot be implemented in a system without some similar construct. The concurrent programming language CML provides similar constructs called `choose` and `select`.

In the case of asynchronous process-calculi (concurrent systems where senders have no continuation) like the asynchronous π-calculus, mention of external choice operators in literature is rather limited. Most of these choice operators allow choice between input processes only.

$$C ::= x(y).P \mid C_1 + C_2 \qquad (3)$$

$$(x(y).P + C_1) \mid \bar{x}z \;\rightarrow\; P[z/y]$$

As shown in [27], this choice operator is also strictly less expressive than the external choice operator in synchronous calculi mentioned earlier. *l*CLL is also an asynchronous language. As seen in section 3.5, the pairing construct $\langle N_1, N_2 \rangle$ and the associated type constructor & act as an external choice primitive in *l*CLL because forward chaining can project out either N_1 or N_2 from a pair $\langle N_1, N_2 \rangle$, if it can be used to complete a reaction step. From the translation in section 3.3, we see that our choice construct corresponds to the following choice operator in the asynchronous π-calculus.

$$C ::= x(y).P \mid \bar{x}y \mid C_1 + C_2$$

$$(x(y).P + C_1) \mid (\bar{x}z + C_2) \;\rightarrow\; P[z/y]$$

Clearly this operator is at least as expressive as the input-only choice operator in equation (3). We now show that this operator can be used to implement a complete synchronous external choice operator $[]$ in CLL. The encoding is not obvious and we present it case by case. Throughout this section, we use analogy with the π-calculus to describe constructions abstractly.

[9]This choice operator is called $+$ in the original paper. We call it $[]$ to avoid syntactic ambiguity.

5.7.1 Input-input Choice

We implement a choice between two receivers. Suppose we have two synchronous receivers, $k_1(y_1).P_1$ and $k_2(y_2).P_2$. Using equation (2), the translations of these two receivers into the asynchronous π-calculus are

$$\begin{array}{rcl}\langle\!\langle\, k_1(y_1).\, P_1 \,\rangle\!\rangle & = & \nu t_1.\, k_1(u_1).\, (\bar{u}_1 t_1 \mid t_1(y_1).\, \langle\!\langle\, P_1 \,\rangle\!\rangle) \\ \langle\!\langle\, k_2(y_2).\, P_2 \,\rangle\!\rangle & = & \nu t_2.\, k_2(u_2).\, (\bar{u}_2 t_2 \mid t_2(y_2).\, \langle\!\langle\, P_2 \,\rangle\!\rangle)\end{array}$$

This suggests the following translation for synchronous input-input choices.

$$\begin{array}{rcl}\langle\!\langle\, k_1(y_1).\, P_1 \,[]\, k_2(y_2).\, P_2 \,\rangle\!\rangle & = & \nu t_1.\nu t_2. \\ & & (\\ & & \quad (k_1(u_1).\, (\bar{u}_1 t_1 \mid t_1(y_1).\, \langle\!\langle\, P_1 \,\rangle\!\rangle)) + \\ & & \quad (k_2(u_2).\, (\bar{u}_2 t_2 \mid t_2(y_2).\, \langle\!\langle\, P_2 \,\rangle\!\rangle)) \\ & &)\end{array}$$

We can now define the operator $\mathbf{syncchoice_{rr}}$ that allows us to choose synchronously between two receivers. It is just a translation of the above term into CLL. We use notation from the language `PICT` [30] to denote events in a choice. $k?(y : B).M$ denotes the event of receiving a value of type B on channel k and binding it to y in M. Observe that the *event* $k?(y : B).M$ differs from the process $\mathbf{syncrecv}\ y : B\ \mathtt{on}\ k\ \mathtt{in}\ M$ in that the latter will execute on its own. The former, on the other hand, is notation for a potential communication.

$$\begin{array}{l}
\mathbf{syncchoice_{rr}}\ [k_1?(y_1 : B_1).M_1\ \#\ S_1,\ \ k_2?(y_2 : B_2).M_2\ \#\ S_2] = \\
\quad \mathtt{priv}\ t_1 : \mathtt{chan}\ \mathtt{in} \\
\quad \mathtt{priv}\ t_2 : \mathtt{chan}\ \mathtt{in} \\
\quad \langle \\
\quad\quad \Lambda u_1 : \mathtt{chan}.\ \hat{\lambda} c : \mathtt{out_1}\ \ k_1\ \ u_1. \\
\quad\quad \{ \\
\quad\quad\quad \underline{\mathtt{let}}\ \{1\} = \underline{\mathtt{destroyout}}_1\ \ [k_1]\ \ [u_1]\ \ \hat{}\ \ c\ \ \underline{\mathtt{in}} \\
\quad\quad\quad\quad (\underline{\mathtt{out}}_1\ \ [u_1]\ \ [t_1])\ \otimes \\
\quad\quad\quad\quad \hat{\lambda} y_1' : \mathtt{data}_{B_1}\ \ t_1. \\
\quad\quad\quad\quad \{ \\
\quad\quad\quad\quad\quad \underline{\mathtt{let}}\ \{y_1\} = \{\underline{\mathtt{undata}}_{B_1}\ \ [t_1]\ \ \hat{}\ \ y_1'\} \\
\quad\quad\quad\quad\quad \underline{\mathtt{in}}\ M_1 \\
\quad\quad\quad\quad \} \\
\quad\quad \}\ , \\
\quad\quad \Lambda u_2 : \mathtt{chan}.\ \hat{\lambda} c : \mathtt{out_1}\ \ k_2\ \ u_2. \\
\quad\quad \{ \\
\quad\quad\quad \underline{\mathtt{let}}\ \{1\} = \underline{\mathtt{destroyout}}_1\ \ [k_2]\ \ [u_2]\ \ \hat{}\ \ c\ \ \underline{\mathtt{in}} \\
\quad\quad\quad\quad (\underline{\mathtt{out}}_1\ \ [u_2]\ \ [t_2])\ \otimes \\
\quad\quad\quad\quad \hat{\lambda} y_2' : \mathtt{data}_{B_2}\ \ t_2. \\
\quad\quad\quad\quad \{ \\
\quad\quad\quad\quad\quad \underline{\mathtt{let}}\ \{y_2\} = \{\underline{\mathtt{undata}}_{B_2}\ \ [t_2]\ \ \hat{}\ \ y_2'\} \\
\quad\quad\quad\quad\quad \underline{\mathtt{in}}\ M_2 \\
\quad\quad\quad\quad \} \\
\quad\quad \} \\
\quad \rangle
\end{array}$$

The typing rule for this macro is

$$\frac{\Sigma \vdash k_i : \mathtt{chan} \qquad \Sigma;\Gamma;\Delta, y_i : B_i;\Psi \vdash M_i \# S_i \qquad i = 1,2}{\Sigma;\Gamma;\Delta;\Psi \vdash \begin{array}{c}(\mathbf{syncchoice_{rr}}\ [k_1?(y_1 : B_1).M_1 \# S_1,\ k_2?(y_2 : B_2).M_2 \# S_2]) \\ \# \mathtt{Syncchoice_{rr}}(k_1, B_1, S_1, k_2, B_2, S_2)\end{array}}\ \mathbf{syncchoice_{rr}}$$

where $\mathtt{Syncchoice_{rr}}$ is defined as

$$\begin{array}{l}\mathtt{Syncchoice_{rr}}(k_1, B_1, S_1, k_2, B_2, S_2) \ = \\ \quad \exists t_1 : \mathtt{chan}.\ \exists t_2 : \mathtt{chan}. \\ \qquad (\forall u_1 : \mathtt{chan}.\ \mathtt{out_1}\ k_1\ u_1 \multimap \{(\mathtt{out_1}\ u_1\ t_1) \otimes (\mathtt{data}_{B_1}\ t_1 \multimap \{S_1\})\}) \ \& \\ \qquad (\forall u_2 : \mathtt{chan}.\ \mathtt{out_1}\ k_2\ u_2 \multimap \{(\mathtt{out_1}\ u_2\ t_2) \otimes (\mathtt{data}_{B_2}\ t_2 \multimap \{S_2\})\})\end{array}$$

5.7.2 Output-output Choice

Now we implement a synchronous choice between two senders. As for the case of receivers, we begin by considering two senders $\bar{k}_1 N_1.P_1$ and $\bar{k}_2 N_2.P_2$ in the synchronous π-calculus. Their translations to the asynchronous π-calculus are

$$\begin{array}{lcl}\langle\!\langle\ \bar{k}_1 N_1.\ P_1\ \rangle\!\rangle & = & \nu u_1.\ (\bar{k}_1 u_1 \mid u_1(t_1).\ (\bar{t}_1 N_1 \mid \langle\!\langle\ P_1\ \rangle\!\rangle)) \\ \langle\!\langle\ \bar{k}_2 N_2.\ P_2\ \rangle\!\rangle & = & \nu u_2.\ (\bar{k}_2 u_2 \mid u_2(t_2).\ (\bar{t}_2 N_2 \mid \langle\!\langle\ P_2\ \rangle\!\rangle))\end{array}$$

From these we obtain the following translation for output-output choice.

$$\begin{array}{lcl}\langle\!\langle\ (\bar{k}_1 N_1.P_1)\ []\ (\bar{k}_2 N_2.P_2)\ \rangle\!\rangle & = & \nu u_1.\ \nu u_2. \\ & & (\\ & & \quad (\bar{k}_1 u_1\ +\ \bar{k}_2 u_2) \mid \\ & & \quad (u_1(t_1).(\bar{t}_1 N_1 \mid \langle\!\langle\ P_1\ \rangle\!\rangle)\ +\ u_2(t_2).(\bar{t}_2 N_2 \mid \langle\!\langle\ P_2\ \rangle\!\rangle)) \\ & &)\end{array}$$

This encoding works because in order for communication to proceed with the term on the right, the first communication must occur with $\bar{k}_1 u_1$ or $\bar{k}_2 u_2$. Once this has happened, the other option is eliminated from the choice. As an example, suppose that a receiver receives u_1 on k_1 before a receiver communicates on k_2. Then the term $\bar{k}_2 u_2$ is eliminated and since u_2 is private, no process can communicate with the term $u_2(t_2).(\bar{t}_2 N_2 \mid \langle\!\langle\ P_2\ \rangle\!\rangle)$. This term gets eliminated when the receiver on k_1 replies on u_1. Thus we can define a synchronous choice macro for output as shown below. As before, we use PICT notation to denote events in the choice. $k!(N : B); M$ denotes the event of sending N of type B on channel k and continuing with the process M.

$$\begin{array}{l}\mathbf{syncchoice_{ss}}\ [k_1!(N_1 : B_1); M_1\ \#\ S_1,\ \ k_2!(N_2 : B_2); M_2\ \#\ S_2] = \\ \quad \mathtt{priv}\ u_1 : \mathtt{chan}\ \mathtt{in} \\ \quad \mathtt{priv}\ u_2 : \mathtt{chan}\ \mathtt{in} \\ \qquad \langle \underline{\mathtt{out}_1}\ [k_1]\ [u_1],\ \underline{\mathtt{out}_1}\ [k_2]\ [u_2]\rangle \otimes \\ \qquad \langle \\ \qquad\quad \Lambda t_1 : \mathtt{chan}.\ \hat{\lambda} c : \mathtt{out_1}\ u_1\ t_1. \\ \qquad\quad \{ \\ \qquad\qquad \underline{\mathtt{let}}\ \{1\} = \underline{\mathtt{destroyout}_1}\ [u_1]\ [t_1]\ \hat{}\ c \\ \qquad\qquad \underline{\mathtt{in}}\end{array}$$

$$
\begin{array}{l}
\qquad\qquad (\underline{\mathtt{data}}_{B_1}\ [t_1]\ \hat{}\ N_1)\ \otimes\ M_1 \\
\quad \}\ , \\
\quad \Lambda t_2 : \mathtt{chan}.\ \hat{\lambda} c : \mathtt{out_1}\ u_2\ t_2. \\
\quad \{ \\
\qquad \underline{\mathtt{let}}\ \{1\} = \underline{\mathtt{destroyout}_1}\ [u_2]\ [t_2]\ \hat{}\ c \\
\qquad \underline{\mathtt{in}} \\
\qquad\qquad (\underline{\mathtt{data}}_{B_2}\ [t_2]\ \hat{}\ N_2)\ \otimes\ M_2 \\
\quad \} \\
\rangle
\end{array}
$$

The typing rule for this macro is

$$
\frac{\Sigma \vdash k_i : \mathtt{chan} \qquad \Sigma; \Gamma; \Delta; \Psi \vdash M_i \# S_i \qquad \Sigma; \Gamma; \Delta'; \Psi \vdash N_i : B_i}{\Sigma; \Gamma; \Delta, \Delta'; \Psi \vdash \begin{array}{c}(\mathbf{syncchoice}_{\mathtt{ss}}\ [k_1!(N_1 : B_1); M_1 \# S_1,\ k_2!(N_2 : B_2); M_2 \# S_2]) \\ \# \ \mathtt{Syncchoice_{ss}}(k_1, B_1, S_1, k_2, B_2, S_2)\end{array}} \ \mathbf{syncchoice}_{\mathtt{ss}}
$$

where the type $\mathtt{Syncchoice_{ss}}(k_1, B_1, S_1, k_2, B_2, S_2)$ is defined as

$$
\begin{array}{l}
\mathtt{Syncchoice_{ss}}(k_1, B_1, S_1, k_2, B_2, S_2)\ = \\
\quad \exists u_1 : \mathtt{chan}.\ \exists u_2 : \mathtt{chan}. \\
\qquad ((\mathtt{out_1}\ k_1\ u_1\ \&\ \mathtt{out_1}\ k_2\ u_2)\ \otimes \\
\qquad ((\forall t_1 : \mathtt{chan}.\ \mathtt{out_1}\ u_1\ t_1 \multimap \{(\mathtt{data}_{B_1}\ t_1)\ \otimes\ S_1\})\ \& \\
\qquad (\forall t_2 : \mathtt{chan}.\ \mathtt{out_1}\ u_2\ t_2 \multimap \{(\mathtt{data}_{B_2}\ t_2)\ \otimes\ S_2\})))
\end{array}
$$

5.7.3 Input-output Choice

Consider a receiver $k_1(y_1).P_1$ and a sender $\bar{k}_2 N_2.P_2$. The translations of these to the asynchronous π-calculus are

$$
\begin{array}{lcl}
\langle\!\langle\ k_1(y_1).\ P_1\ \rangle\!\rangle & = & \nu t_1.\ k_1(u_1).\ (\bar{u}_1 t_1 \mid t_1(y_1).\ \langle\!\langle\ P_1\ \rangle\!\rangle) \\
\langle\!\langle\ \bar{k}_2 N_2.\ P_2\ \rangle\!\rangle & = & \nu u_2.\ (\bar{k}_2 u_2 \mid u_2(t_2).\ (\bar{t}_2 N_2 \mid \langle\!\langle\ P_2\ \rangle\!\rangle))
\end{array}
$$

We can combine these two terms in a choice as follows.

$$
\begin{array}{lcl}
\langle\!\langle\ (k_1(y_1).\ P_1)\ []\ (\bar{k}_2 N_2.\ P_2)\ \rangle\!\rangle & = & \nu t_1.\ \nu u_2. \\
& & (\\
& & \quad (k_1(u_1).\ (\bar{u}_1 t_1 \mid t_1(y_1).\ \langle\!\langle\ P_1\ \rangle\!\rangle)\ +\ \bar{k}_2 u_2) \mid \\
& & \quad u_2(t_2).\ (\bar{t}_2 N_2 \mid \langle\!\langle\ P_2\ \rangle\!\rangle) \\
& &)
\end{array}
$$

Though this encoding is correct in the π-calculus, we cannot implement it in CLL because we encode choices using the type connective $\&$, and hence the two components of a choice must use the same linear resources. This is not the case here since there is a choice between $(k_1(u_1).\ (\bar{u}_1 t_1 \mid t_1(y_1).\ \langle\!\langle\ P_1\ \rangle\!\rangle)$ and $\bar{k}_2 u_2$ in the above equation. An alternate encoding that balances all resources is shown below. This encoding is incorrect because it has an atomicity problem, which is described after the encoding.

$$
\begin{array}{lcl}
\langle\!\langle\ (k_1(y_1).P_1)\ []\ (\bar{k}_2 N_2.P_2)\ \rangle\!\rangle & = & \nu t_1.\ \nu u_2. \\
& & (\\
& & \quad \bar{k}_2 u_2 \mid \\
& & \quad (k_1(u_1).\ (\bar{u}_1 t_1 \mid t_1(y_1).\ \langle\!\langle\ P_1\ \rangle\!\rangle) \\
& & \qquad +\ u_2(t_2).\ (\bar{t}_2 N_2 \mid \langle\!\langle\ P_2\ \rangle\!\rangle)) \\
& &)
\end{array}
$$

The atomicity problem in this encoding is the following. Consider the scenario where there is a receiver on k_2 and a sender on k_1 i.e. both actions in the choice can be selected. Since there is a receiver on k_2, the term $\bar{k}_2 u_2$ can communicate with it. If the process $k_1(u_1).\ (\bar{u}_1 t_1 \mid t_1(y_1).\ \langle\!\langle\ P_1\ \rangle\!\rangle)$ communicates with the sender on k_1 *before* the receiver on k_2 can reply on u_2, the choice is resolved and the continuation $u_2(t_2).\ (\bar{t}_2 N_2 \mid \langle\!\langle\ P_2\ \rangle\!\rangle)$ is eliminated. This deadlocks the partial communication on k_2.

One way to eliminate this problem is to deactivate the input process $k_1(u_1).\ (\bar{u}_1 t_1 \mid t_1(y_1).\ \langle\!\langle\ P_1\ \rangle\!\rangle)$ once $\bar{k}_2 u_2$ has communicated. This can be done by creating a private channel w_1 and requiring the input process to obtain a signal on that. One such encoding is shown below. As we shall see later, an internal communication can occur in this encoding, and hence it does not work in the π-calculus. However, in CLL, we can implement this encoding using 3-way synchronization.

$$
\begin{array}{rcl}
\langle\!\langle\ (k_1(y_1).P_1)\ []\ (\bar{k}_2 N_2.P_2)\ \rangle\!\rangle & = & \nu t_1.\ \nu u_2.\ \nu w_1. \\
& & (\\
& & \quad (\bar{w}_1\langle\rangle\ +\ \bar{k}_2 u_2) \mid \\
& & \quad (w_1().\ k_1(u_1).\ (\bar{u}_1 t_1 \mid t_1(y_1).\ \langle\!\langle\ P_1\ \rangle\!\rangle) \\
& & \qquad\quad +\ u_2(t_2).\ (\bar{t}_2 N_2 \mid \langle\!\langle\ P_2\ \rangle\!\rangle)) \\
& &)
\end{array}
$$

As mentioned earlier, the atomicity problem does not arise in this encoding because once $\bar{k}_2 u_2$ communicates, $\bar{w}_1\langle\rangle$ is eliminated and hence the input process cannot communicate. However, this encoding suffers from an internal communication problem. The term on the right side above can perform a communication within itself and reduce, thus resolving the choice internally.

$$
\begin{array}{lcl}
\nu t_1.\ \nu u_2.\ \nu w_1. & & \\
(& & \\
\quad (\bar{w}_1\langle\rangle\ +\ \bar{k}_2 u_2) \mid & & \nu t_1.\ \nu u_2.\ \nu w_1. \\
\quad (w_1().\ k_1(u_1).\ (\bar{u}_1 t_1 \mid t_1(y_1).\ \langle\!\langle\ P_1\ \rangle\!\rangle) & \longrightarrow & \quad (\bar{k}_2 u_2 \mid u_2(t_2).\ (\bar{t}_2 N_2 \mid \langle\!\langle\ P_2\ \rangle\!\rangle)) \\
\qquad\quad +\ u_2(t_2).\ (\bar{t}_2 N_2 \mid \langle\!\langle\ P_2\ \rangle\!\rangle)) & & \\
) & &
\end{array}
$$

Thus this encoding *does not* work in the π-calculus. However, in CLL, we can chain reactions together. In particular, successive inputs can be chained together i.e. we can force two senders to synchronize simultaneously with a receiver. If we chain together the two inputs in $w_1().\ k_1(u_1).\ \ldots$, then this internal communication on w_1 cannot occur without the presence of a sender on k_1. Conversely, no sender on k_1 can communicate with this term unless $\bar{w}_1\langle\rangle$ is also present. Thus this encoding works for CLL. We build a macro based on this encoding as follows.

$$
\begin{array}{l}
\mathbf{syncchoice_{rs}}\ \ [k_1?(y_1 : B_1).M_1\ \ \#\ \ S_1,\ \ k_2!(N_2 : B_2); M_2\ \ \#\ \ S_2] = \\
\quad \texttt{priv}\ \ t_1 : \texttt{chan}\ \ \texttt{in} \\
\quad \texttt{priv}\ \ u_2 : \texttt{chan}\ \ \texttt{in} \\
\quad \texttt{priv}\ \ w_1 : \texttt{chan}\ \ \texttt{in} \\
\qquad \langle \underline{\texttt{out}_0}\ \ [w_1],\ \ \underline{\texttt{out}_1}\ \ [k_2]\ \ [u_2]\rangle\ \otimes \\
\qquad \langle \\
\qquad\quad \hat{\lambda} c' : \texttt{out}_0\ \ w_1. \\
\qquad\quad \Lambda u_1 : \texttt{chan}.\ \ \hat{\lambda} c : \texttt{out}_1\ \ k_1\ \ u_1. \\
\qquad\quad \{
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad \underline{\texttt{let}}\ \{1\} = \underline{\texttt{destroyout}_1}\ [k_1]\ [u_1]\ \hat{}\ c\ \underline{\texttt{in}} \\
\qquad\qquad \underline{\texttt{let}}\ \{1\} = \underline{\texttt{destroyout}_0}\ [w_1]\ \hat{}\ c'\ \underline{\texttt{in}} \\
\qquad\qquad\quad (\underline{\texttt{out}}_1\ [u_1]\ [t_1])\ \otimes \\
\qquad\qquad\quad \hat{\lambda} y_1' : \texttt{data}_{B_1}\ t_1. \\
\qquad\qquad\quad \{ \\
\qquad\qquad\qquad \underline{\texttt{let}}\ \{y_1\} = \{\underline{\texttt{undata}}_{B_1}\ [t_1]\ \hat{}\ y_1'\} \\
\qquad\qquad\qquad \underline{\texttt{in}}\ M_1 \\
\qquad\qquad\quad \} \\
\qquad \}\ , \\
\qquad \Lambda t_2 : \texttt{chan}.\ \hat{\lambda} c : \texttt{out}_1\ u_2\ t_2. \\
\qquad \{ \\
\qquad\qquad \underline{\texttt{let}}\ \{1\} = \underline{\texttt{destroyout}_1}\ [u_2]\ [t_2]\ \hat{}\ c \\
\qquad\qquad \underline{\texttt{in}} \\
\qquad\qquad\quad (\underline{\texttt{data}}_{B_2}\ [t_2]\ \hat{}\ N_2)\ \otimes\ M_2 \\
\qquad \} \\
\rangle
\end{array}
$$

The typing rule for this construct is

$$
\frac{\begin{array}{cc} \Sigma\ \vdash\ k_i : \texttt{chan} & \Sigma; \Gamma; \Delta'; \Psi\ \vdash\ N_2 : B_2 \\ \Sigma; \Gamma; \Delta, \Delta', y_1 : B_1; \Psi\ \vdash\ M_1 \# S_1 & \Sigma; \Gamma; \Delta; \Psi\ \vdash\ M_2 \# S_2 \end{array}}{\Sigma; \Gamma; \Delta, \Delta'; \Psi\ \vdash\ \begin{array}{c} (\mathbf{syncchoice_{rs}}\ [k_1?(y_1 : B_1).M_1 \# S_1,\ k_2!(N_2 : B_2); M_2 \# S_2]) \\ \# \ \texttt{Syncchoice}_{\texttt{rs}}(k_1, B_1, S_1, k_2, B_2, S_2) \end{array}}\ \mathbf{syncchoice_{rs}}
$$

where the type $\texttt{Syncchoice}_{\texttt{rs}}(k_1, B_1, S_1, k_2, B_2, S_2)$ is defined as follows.

$$
\begin{array}{l}
\texttt{Syncchoice}_{\texttt{rs}}(k_1, B_1, S_1, k_2, B_2, S_2)\ = \\
\quad \exists t_1 : \texttt{chan}.\ \exists u_2 : \texttt{chan}.\ \exists w_1 : \texttt{chan} \\
\qquad ((\texttt{out}_0\ w_1\ \&\ \texttt{out}_1\ k_2\ u_2)\ \otimes \\
\qquad ((\texttt{out}_0\ w_1 \multimap \forall u_1 : \texttt{chan}.\ \texttt{out}_1\ k_1\ u_1 \multimap \{(\texttt{out}_1\ u_1\ t_1)\ \otimes\ (\texttt{data}_{B_1}\ t_1 \multimap \{S_1\})\})\ \& \\
\qquad (\forall t_2 : \texttt{chan}.\ \texttt{out}_1\ u_2\ t_2 \multimap \{(\texttt{data}_{B_2}\ t_2)\ \otimes\ S_2\})))
\end{array}
$$

The input-output choice construct described here can be generalized to an arbitrary number of senders and receivers. The extension is straightforward and we elide the details here. We also observe that the choice macros presented here can be used in conjunction with the macros **syncsend** and **syncrecv** defined earlier. However, separate channels must be used for synchronous and asynchronous communication i.e. channels used for calls on **syncsend** or **syncrecv** must not be used for calls on **asyncsend** or **asyncrecv** and vice-versa.

5.8 Example: Read-Write Memory Cell

We construct a read-write memory cell to illustrate the choice mechanism designed above. A read-write cell is a process that remembers one single value. It listens to requests to read the value stored on channel $read$ and to write (change) the value in the cell on channel $write$. Since a single write can be followed by several reads, the value stored in the cell has to be non-linear. Further, we assume that the cell is always created with

```
type Cell_B(read, write) = μα.Syncchoice_rs(write, {!B}, {{α}}, read, B, {α})

(* memoryCell_B : ∀read : chan. ∀write : chan. B → {Cell_B(read, write)} *)

fun memoryCell_B [read : chan] [write : chan] (v : B) =
{
    fold_{Cell_B(read,write)}
        syncchoice_rs
        [
            write?(x : {!B}).
            {
                let {!y} = x in
                    memoryCell_B [read] [write] y
            },
            read!(v : B); memoryCell_B [read] [write] v
        ]
}
```

Figure 34: The function `memoryCell`

a value stored in it.[10] Figure 34 describes a function `memoryCell` that creates a memory cell on channels $read$ and $write$ and initializes it with the value v.

6 Discussion

CLL is a concurrent language designed from logical principles. In the process of designing CLL, we have accomplished four main objectives. First, we have shown that proof-search in logic has an interesting computational interpretation - it can viewed as a procedure to link together programs to form larger programs. This may be viewed as an extension of the Curry-Howard isomorphism to include proof-search procedures. Second, we have obtained a symmetric integration between functional and logic programming. *f*CLL is purely functional. *l*CLL introduced in section 3 embeds this functional language in a concurrent logic programming language that performs proof-search on types of programs and then links programs together. In section 4 we embed the *l*CLL back into *f*CLL, making the integration between functional and logic programming symmetric. Execution of programs in full-CLL proceeds in interleaving phases of functional evaluation of programs and proof-search to link parts of programs. To the best of our knowledge, this is the first time that functional and logic programming have been integrated in this manner.

CLL is also a symmetric integration of functional and concurrent programming in a typed setting. *l*CLL in section 3 adds concurrency to the functional language *f*CLL. Full-CLL allows *l*CLL CHAMs to be created and nested inside functional evaluation through the <u>`link`</u> construct, thus making the integration symmetric. The idea of integrating functional and concurrent programming is not new. The blue-calculus [8], CML

[10] This is in sharp contrast with memory cells called I-structures which are created empty and have a write-once, read-many semantics. See [32] for a description of I-structures.

[31, 32], JoCAML [15], PICT [30] and Facile [17] all integrate functional and concurrent programming. All these languages have both functional and concurrent features and are typed. However, there are several differences between these languages and CLL. First, all these languages have a "flat" model for concurrent processes, i.e. there is a *single* global configuration in which all parallel processes execute simultaneously. When a function creates a sub-process, the process is automatically lifted and placed in this global configuration. This process can then freely communicate with all other processes. Thus communication and synchronization cannot be localized to specific parts of programs. In sharp contrast, each call to the `link` construct in CLL creates a separate configuration for concurrent processes.[11] Processes within a configuration can communicate and synchronize with each other, but processes in separate configurations cannot (for an illustration, see the example of Fibonacci numbers in section 5.1). Another consequence of having a single configuration for processes in existing concurrent functional languages is that concurrent computations (processes) do not return values to functional terms directly. This has to be done indirectly through the message passing mechanism of the language. In CLL, on the other hand, a concurrent computation started using the `link` construct directly returns a result that can be used in the remainder of the functional computation. This results in a significant difference in the structure of programs written in CLL and other languages. It also makes the integration between functional and concurrent programming more symmetric in CLL. The third difference between CLL and blue-calculus, CML, JoCAML, PICT and Facile is that every process in CLL has a distinct type that provides definite information about the behavior of the process. For example, a process of type $S_1 \otimes S_2$ is a parallel composition of two processes of types S_1 and S_2. On the other hand, typing for processes in the other concurrent languages mentioned above is weak and process types provide no information about the behavior of processes. In Facile, PICT and JoCAML processes have no types at all. The type system only checks that each individual functional term in a process has a type. In the blue-calculus, all processes in the global configuration must have the same type. In CML, processes are not explicitly visible; they are only observable through side-effects like communication. We believe that having informative types on processes will make it easier to reason about correctness of CLL programs.

The fourth contribution of CLL is an exploration of connections between process-calculi constructs and connectives of linear logic in the context of programming language design. As seen in section 3.5, the linear logic connectives $\otimes$, $\exists$, $\&$, $\oplus$, $\multimap$ and atomic propositions correspond to process-calculi constructs of parallel composition, name restriction, external choice, internal choice, input prefixing and asynchronous output respectively. Further, communication channels can be simulated using index refinements and synchronization and communication between processes can be performed using proof-search. Thus there is a correspondence between linear logic connectives and process-calculi constructs and proof-search in linear logic and communication in process-calculi. Abramsky's work on computational interpretations of linear logic [1] and the MSR framework [11] also explore similar connections between linear logic and concurrent computation but as opposed to CLL they do not use this correspondence to construct a programming language. As far as we know, this is the first time that such connections have been used explicitly in a programming language.

Acknowledgment

The author expresses his sincere thanks to Frank Pfenning for his guidance, comments, suggestions and ideas that have been invaluable to the creation of CLL and this report.

[11] The ambient calculus [9] allows creation of several separate nested configurations for processes. However, the ambient calculus lacks functional programming and emphasizes process mobility and cannot be compared to CLL directly.

References

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1–2):3–57, 1993.

[2] Samson Abramsky, Simon Gay, and Rajagopal Nagarajan. Specification structures and propositions-as-types for concurrency. In G. Birtwistle and F. Moller, editors, *Logics for Concurrency: Structure vs. Automata—Proceedings of the VIIIth Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[3] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[4] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 81–94. ACM, January 1990.

[5] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[6] Gérard Boudol. Asynchrony and the pi-calculus. Technical Report RR-1702, INRIA SofiaAntipolis, 1992.

[7] Gérard Boudol. Some chemical abstract machines. In *A Decade of Concurrency*, volume 803 of *LNCS*, pages 92–123. Springer-Verlag, 1994.

[8] Gérard Boudol. The π-calculus in direct style. *Higher Order Symbol. Comput.*, 11(2):177–208, 1998.

[9] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Types for the ambient calculus. *Inf. Comput.*, 177(2):160–194, 2002.

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

[11] Iliano Cervesato. The logical meeting point of multiset rewriting and process algebra. Unpublished manuscript. 2004. Available electronically from http://theory.stanford.edu/˜iliano/forthcoming.html.

[12] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Computer Science Department, Carnegie Mellon University, May 2003.

[13] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131, Computer Science Department, Carnegie Mellon University, 2003.

[14] Kaustuv Chaudhuri. Focusing the inverse method for linear logic. Unpublished Manuscript. 2005. Available electronically from http://www.cs.cmu.edu/˜kaustuv/papers/lics05.pdf.

[15] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for objective-caml. In *ASAMA '99: Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, page 22. IEEE Computer Society, 1999.

[16] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ML for the join-calculus. In *Proceedings of the 8th International Conference on Concurrency Theory*, pages 196–212. Springer-Verlag, 1997.

[17] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[18] Jean-Yves Girard. Linear logic. In *Theoretical Computer Science*, volume 5, pages 1–102, 1987.

[19] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In *Proceedings of TAP-SOFT'87, vol 2*, volume 250 of *Lecture Notes in Computer Science*, pages 52–66, 1987.

[20] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–141. ACM Press, 2001.

[21] Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. Technical Report TR03-0007, Department of Computer Science, Tokyo Institute of Technology, October 2003.

[22] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.

[23] C. Mierowsky, S. Taylor, E. Shapiro, J. Levy, and S. Safra. The design and implementation of flat concurrent prolog. Technical Report CS85-09, Department of Computer Science, Weiszmann Institute of Science, 1985.

[24] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes part 1. Technical Report ECS-LFCS-89-85, Edinburgh University, 1989.

[25] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes part 2. Technical Report ECS-LFCS-89-86, Edinburgh University, 1989.

[26] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[27] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Symposium on Principles of Programming Languages (POPL)*, pages 256–265, 1997.

[28] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308. ACM Press, 1996.

[29] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical. Structures in Comp. Sci.*, 11(4):511–540, 2001.

[30] Benjamin C. Pierce and David N. Turner. Pict: a programming language based on the pi-calculus. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[31] John H. Reppy. CML: A higher concurrent language. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 293–305. ACM Press, 1991.

[32] John H. Reppy. *Concurrent programming in ML.* Cambridge University Press, 1999.

[33] Ehud Shapiro, editor. *Concurrent Prolog: Collected Papers*, volume 1-2. MIT Press, Cambridge, MA, 1987.

[34] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, 1989.

[35] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgements and properties. Technical Report CMU-CS-02-101, Computer Science Department, Carnegie Mellon University, May 2003.

[36] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM Press, 1999.