# Type-Directed Concurrency

Deepak Garg and Frank Pfenning[⋆]

Carnegie Mellon University
{dg,fp}@cs.cmu.edu

**Abstract.** We introduce a novel way to integrate functional and concurrent programming based on intuitionistic linear logic. The functional core arises from interpreting proof reduction as computation. The concurrent core arises from interpreting proof search as computation. The two are tightly integrated via a monad that permits both sides to share the same logical meaning for the linear connectives while preserving their different computational paradigms. For example, concurrent computation synthesizes proofs which can be evaluated as functional programs. We illustrate our design with some small examples, including an encoding of the pi-calculus.

## 1 Introduction

At the core of functional programming lies the beautiful Curry-Howard isomorphism which identifies intuitionistic proofs with functional programs and propositions with types. In this paradigm, computation arises from proof reduction. One of the most striking consequences is that we can write functions and reason logically about their behavior in an integrated manner.

Concurrent computation has resisted a similarly deep, elegant, and practical analysis with logical tools, despite several explorations in this direction (see, for example, [4, 15]). We believe the lack of a satisfactory Curry-Howard isomorphism is due to the limits inherent in complete proofs: they provide an analysis of constructive truth but not of the dynamics of interaction.

An alternative logical foundation for concurrency is to view computation as proof search [6]. In this paper we show that the two views of computation, via proof reduction and via proof search, are not inherently incompatible, but can coexist harmoniously in a language that combines functional and concurrent computation. We retain the strong guarantees for functional computation without unduly restricting the dynamism of concurrent computation.

In order to achieve this synthesis, we employ several advanced building blocks. The first is linearity: as has been observed [12], the evolution and communication of processes maps naturally to the single-use semantics of assumptions in linear logic. The second is dependency: we use dependent types to model communication channels and also to retain the precision of functional specifications for transmitted values. The third is monads: we use monadic types to encapsulate concurrent computation, so that the linear connectives can retain the same

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

logical meaning on the functional and concurrent side without interference. The fourth is focusing [5]: we use it to enforce the atomicity of concurrent interactions during proof search.

The result is a tightly integrated language in which functional computation proceeds by reduction and concurrent computation proceeds by proof search. Concurrent computation thereby synthesizes proofs which can be evaluated as functional programs. We illustrate the design with some small examples, including an encoding of the $\pi$-calculus to help gauge its expressive power.

There has been significant prior work in combining functional and concurrent programming. One class of languages, including Facile [11], Concurrent ML [18, 19], JOCaml [9], and Concurrent Haskell [13], adds concurrency primitives to a language with functional abstractions. While we share some ideas (such as the use of monadic encapsulation in Concurrent Haskell), the concurrent features in these languages are motivated operationally rather than logically and are only faintly reflected in the type system. Another class of languages start from a rich concurrent formalism such as the $\pi$-calculus and either add or encode some features of functional languages [17]. While operationally adequate, these encodings generally do not have a strong logical component. An interesting intermediate point is the applied $\pi$-calculus [2] where algebraic equations are added to the $\pi$-calculus. However, it is intended for reasoning about specifications rather than as a programming language.

Perhaps most closely related to our work is CLF [21] and the logic programming language LolliMon [14] based on its first-order fragment. Our type system is based on the common logic underlying both these systems. However, these systems are intended as a logical framework and concurrent logic programming language respectively and differ significantly from our language in the operational semantics. Another closely related line of work is Abramsky's computational interpretations of linear logic [3], but the discussion of concurrency there is based on classical rather than intuitionistic linear logic and lacks functional features.

The principal contributions of this paper are conceptual and foundational, although a simple prototype [1] indicates that there is at least some practical merit to the work. Owing to space constraints we omit the linear type constructors $\&$ and $\oplus$, recursive types and all proofs from this paper. These details can be found in the companion technical report [10].

In the remainder of the paper we present our language (called CLL) in three steps. First, we present the functional core ($f$CLL) which integrates linearity and a monad. Second, we present the concurrent core ($l$CLL), which is based on proof search, and which can call upon functional computation. Third, we complete the integration with one additional construct to allow functional computation to call upon concurrent computation. We call the complete language full-CLL. We conclude with some remarks about the limitations of our work.

Our main technical results are as follows. For the functional core $f$CLL, we prove type soundness by proving preservation and progress. For the concurrent core $l$CLL, we only formulate and prove a suitable notion of preservation. For full-CLL, we prove both preservation and progress, but the progress theorem is

| | |
|---|---|
| Sorts | $\gamma ::= \texttt{chan} \mid \ldots$ |
| Index terms | $s, t ::= i \mid f(t_1, \ldots, t_n)$ |
| Index variable contexts | $\Sigma ::= \cdot \mid \Sigma, i : \gamma$ |
| Sorting judgment | $\Sigma \ \vdash \ t \in \gamma$ |
| Kinds | $K ::= \texttt{Type} \mid \gamma \rightarrow K$ |
| Types | $T ::= A \mid S$ |
| Asynchronous types | $A, B ::= C \ t_1 \ldots t_n \mid A \rightarrow B \mid A \multimap B \mid \forall i : \gamma.A(i) \mid \{S\}$ |
| Synchronous types | $S ::= A \mid S_1 \otimes S_2 \mid \mathbf{1} \mid !A \mid \exists i : \gamma.S(i)$ |
| Programs | $P ::= N \mid M \mid E$ |
| Terms | $N ::= x \mid \lambda x : A.N \mid N_1 \ N_2 \mid \hat{\lambda} x : A.N \mid N_1 \ \hat{} \ N_2$ |
| | $\mid \Lambda i : \gamma.N \mid N \ [t] \mid \{E\}$ |
| Monadic-terms | $M ::= N \mid M_1 \otimes M_2 \mid \star \mid !N \mid [t, M]$ |
| Patterns | $p ::= x \mid \star \mid p_1 \otimes p_2 \mid !x \mid [i, p]$ |
| Expressions | $E ::= M \mid \underline{\texttt{let}} \ \{p : S\} = N \ \underline{\texttt{in}} \ E$ |

**Fig. 1.** *f*CLL syntax

weaker than that of *f*CLL. This is because in full-CLL concurrent computations started during functional computation can deadlock.

## 2 *f*CLL: Functional core of CLL

**Syntax.** The functional core of CLL is a first-order dependently typed linear functional language called *f*CLL. It is an extension of a linear lambda calculus with first-order dependent types from DML [22] and a monad. Its type and term syntax is based largely on that of CLF [21]. The syntax of *f*CLL is summarized in figure 1. Types in *f*CLL can depend on index terms (denoted by $s, t$) that are divided into a number of disjoint sorts ($\gamma$). Index terms contain index variables ($i, j, k, \ldots$) and uninterpreted function symbols ($f, g, h, \ldots$). We assume the existence of a *sorting* judgment $\Sigma \ \vdash \ t \in \gamma$, where $\Sigma$ is a context that mentions the sorts of all free index variables in $t$.

Type constructors (denoted by $C$) are classified into kinds. For every *f*CLL program we assume the existence of an implicit signature that mentions the kinds of all type constructors used in the program. An atomic type is formed by applying a type constructor $C$ to index terms $t_1, \ldots, t_n$. If $C$ has kind $\gamma_1 \rightarrow \ldots \rightarrow \gamma_n \rightarrow \texttt{Type}$, we say that the atomic type $C \ t_1 \ldots \ t_n$ is well-formed in the index variable context $\Sigma$ iff for $1 \leq i \leq n$, $\Sigma \ \vdash \ t_i \in \gamma_i$. In the following we assume that all atomic types in *f*CLL programs are well-formed.

Following CLF, types in *f*CLL are divided into two classes - asynchronous ($A, B$) and synchronous ($S$). Asynchronous types can be freely used as synchronous types. However, synchronous types must be coerced explicitly into asynchronous types using a monad $\{\ldots\}$, which is presented in a judgmental style [16].

Programs ($P$) are divided into three syntactic classes – terms ($N$), monadic-terms ($M$) and expressions ($E$). This classification is reminiscent of a similar

$$\Sigma ::= \cdot \mid \Sigma, i : \gamma \qquad \Delta ::= \cdot \mid \Delta, x : A$$
$$\Gamma ::= \cdot \mid \Gamma, x : A \qquad \Psi ::= \cdot \mid \Psi, p : S$$

$\boxed{\mathbf{\Sigma; \Gamma; \Delta \ \vdash \ N : A}}$

$$\frac{}{\Sigma; \Gamma; x : A \ \vdash \ x : A}\,\textsf{Hyp1} \qquad \frac{}{\Sigma; \Gamma, x : A; \cdot \ \vdash \ x : A}\,\textsf{Hyp2}$$

$$\frac{\Sigma; \Gamma, x : A; \Delta \ \vdash \ N : B}{\Sigma; \Gamma; \Delta \ \vdash \ \lambda x : A.N : A \rightarrow B}\,\textsf{→I} \qquad \frac{\Sigma; \Gamma; \Delta, x : A \ \vdash \ N : B}{\Sigma; \Gamma; \Delta \ \vdash \ \hat{\lambda} x : A.N : A \multimap B}\,\textsf{⊸I}$$

$$\frac{\Sigma, i : \gamma; \Gamma; \Delta \ \vdash \ N : A}{\Sigma; \Gamma; \Delta \ \vdash \ \Lambda i : \gamma.N : \forall i : \gamma.A}\,\textsf{∀I} \qquad \frac{\Sigma; \Gamma; \Delta \ \vdash \ E \div S}{\Sigma; \Gamma; \Delta \ \vdash \ \{E\} : \{S\}}\,\textsf{\{\}I}$$

$\boxed{\mathbf{\Sigma; \Gamma; \Delta \ \vdash \ M \Bumpeq S}}$

$$\frac{\Sigma; \Gamma; \cdot \ \vdash \ N : A}{\Sigma; \Gamma; \cdot \ \vdash \ !N \Bumpeq !A}\,\textsf{!R} \qquad \frac{\Sigma; \Gamma; \Delta \ \vdash \ M \Bumpeq S(t) \qquad \Sigma \ \vdash \ t \in \gamma}{\Sigma; \Gamma; \Delta \ \vdash \ [t, M] \Bumpeq \exists i : \gamma.S(i)}\,\textsf{∃R}$$

$$\frac{\Sigma; \Gamma; \Delta_1 \ \vdash \ M_1 \Bumpeq S_1 \qquad \Sigma; \Gamma; \Delta_2 \ \vdash \ M_2 \Bumpeq S_2}{\Sigma; \Gamma; \Delta_1, \Delta_2 \ \vdash \ M_1 \otimes M_2 \Bumpeq S_1 \otimes S_2}\,\textsf{⊗R}$$

$\boxed{\mathbf{\Sigma; \Gamma; \Delta \ \vdash \ E \div S}}$

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

$\boxed{\mathbf{\Sigma; \Gamma; \Delta; \Psi \ \vdash \ E \div S}}$

$$\frac{\Sigma; \Gamma; \Delta \ \vdash \ E \div S}{\Sigma; \Gamma; \Delta; \cdot \ \vdash \ E \div S}\,\div\div \qquad \frac{\Sigma; \Gamma, x : A; \Delta; \Psi \ \vdash \ E \div S}{\Sigma; \Gamma; \Delta; !x :!A, \Psi \ \vdash \ E \div S}\,\textsf{!L} \qquad \frac{\Sigma; \Gamma; \Delta; \Psi \ \vdash \ E \div S}{\Sigma; \Gamma; \Delta; \star : \mathbf{1}, \Psi \ \vdash \ E \div S}\,\textsf{1L}$$

$$\frac{\Sigma; \Gamma; \Delta; p_1 : S_1, p_2 : S_2, \Psi \ \vdash \ E \div S}{\Sigma; \Gamma; \Delta; p_1 \otimes p_2 : S_1 \otimes S_2, \Psi \ \vdash \ E \div S}\,\textsf{⊗L} \qquad \frac{\Sigma, i : \gamma; \Gamma; \Delta; p : S', \Psi \ \vdash \ E \div S}{\Sigma; \Gamma; \Delta; [i, p] : \exists i : \gamma.S', \Psi \ \vdash \ E \div S}\,\textsf{∃L}\ (i\ \textsf{fresh})$$

**Fig. 2.** $f$CLL type system (selected rules)

classification in CLF's objects. Under the Curry-Howard isomorphism, terms are proofs of asynchronous types whereas monadic-terms and expressions are proofs of synchronous types that end with *introduction* rules and *elimination* rules respectively. A $f$CLL program is called closed if it does not contain any free term variables. Closed programs may contain free index variables.

**Typing.** Programs in $f$CLL are type-checked using four contexts – a context of index variables $\Sigma$, a context of linear variables $\Delta$, a context of unrestricted variables $\Gamma$ and a context of patterns $\Psi$. Only the last of these contexts is ordered. There are four typing judgments in the type system. We use the notation $N : A$, $M \Bumpeq S$ and $E \div S$ for typing relations. Some interesting rules from these judgments are shown in figure 2. Type-checking for $f$CLL is decidable.

**Operational Semantics.** We use a call-by-value reduction semantics for $f$CLL. Figure 3 shows the definition of values in $f$CLL and some interesting reduction rules. The substitution relation $P[M_V/p]$ substitutes the monadic-value $M_V$ for a pattern $p$ in the program $P$. It is defined by induction on the pattern $p$. $P[V/x]$ and $P[t/i]$ are the usual capture avoiding substitutions for term and index variables respectively. In $f$CLL, the monad $\{E\}$ is a value because after

$$\begin{array}{lrl}
\text{Term values} & V & ::= \lambda x : A.N \mid \hat{\lambda} x : A.N \mid \{E\} \mid \Lambda i : \gamma.N \\
\text{Monadic values} & M_V & ::= V \mid M_{V_1} \otimes M_{V_2} \mid \star \mid !V \mid [t, M_V] \\
\text{Expression values} & E_V & ::= M_V
\end{array}$$

$\boxed{\mathbf{P[M_V/p]}}$

$$\begin{array}{ll}
P[\star/\star] = P & P[[t, M_V]/[i,p]] = (P[t/i])[M_V/p] \\
P[!V/!x] = P[V/x] & P[M_{V_1} \otimes M_{V_2}/p_1 \otimes p_2] = (P[M_{V_1}/p_1])[M_{V_2}/p_2]
\end{array}$$

$\boxed{\mathbf{N \rightsquigarrow N'}}$

$$\frac{}{(\Lambda i : \gamma.N)\ [t] \ \rightsquigarrow\ N[t/i]} \rightsquigarrow \Lambda \qquad \frac{}{(\lambda x : A.N)\ V \ \rightsquigarrow\ N[V/x]} \rightsquigarrow \lambda$$

$$\frac{}{(\hat{\lambda} x : A.N)\ \hat{}\ V \ \rightsquigarrow\ N[V/x]} \rightsquigarrow \hat{\lambda}$$

$\boxed{\mathbf{M \mapsto M'}}$

$$\frac{N \ \rightsquigarrow\ N'}{N \ \mapsto\ N'} \rightsquigarrow\mapsto \qquad \frac{N \ \rightsquigarrow\ N'}{!N \ \mapsto\ !N'} \mapsto ! \qquad \frac{M \ \mapsto\ M'}{[t, M] \ \mapsto\ [t, M']} \mapsto \exists$$

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

$\boxed{\mathbf{\Sigma; E \hookrightarrow \Sigma; E'}}$

$$\frac{M \ \mapsto\ M'}{\Sigma; M \ \hookrightarrow\ \Sigma; M'} \mapsto\hookrightarrow \qquad \frac{}{\Sigma; \underline{\texttt{let}}\ \{p : S\} = \{M_V\}\ \underline{\texttt{in}}\ E \ \hookrightarrow\ \Sigma; E[M_V/p]} \hookrightarrow \mathit{LETRED}$$

$$\frac{N \ \rightsquigarrow\ N'}{\Sigma; \underline{\texttt{let}}\ \{p : S\} = N\ \underline{\texttt{in}}\ E \ \hookrightarrow\ \Sigma; \underline{\texttt{let}}\ \{p : S\} = N'\ \underline{\texttt{in}}\ E} \hookrightarrow \mathit{LET}_1$$

$$\frac{\Sigma; E \ \hookrightarrow\ \Sigma; E'}{\Sigma; \underline{\texttt{let}}\ \{p : S\} = \{E\}\ \underline{\texttt{in}}\ E_1 \ \hookrightarrow\ \underline{\texttt{let}}\ \Sigma; \{p : S\} = \{E'\}\ \underline{\texttt{in}}\ E_1} \hookrightarrow \mathit{LET}_2$$

**Fig. 3.** *f*CLL operational semantics (selected rules)

we extend the language in section 4, expressions have effects. Reduction of the two components of a $\otimes$ can be interleaved arbitrarily, or it may performed in parallel.

Expressions are reduced in a context of index variables $\Sigma$. This context plays no role in *f*CLL, but when we extend *f*CLL to full-CLL in section 4, the context $\Sigma$ becomes computationally significant. We state preservation and progress theorems for *f*CLL below.

**Theorem 1 (Preservation for *f*CLL).**
1. If $\Sigma; \Gamma; \Delta \ \vdash\ N : A$ and $N \rightsquigarrow N'$, then $\Sigma; \Gamma; \Delta \ \vdash\ N' : A$.
2. If $\Sigma; \Gamma; \Delta \ \vdash\ M \mathrel{\dot{\overset{\sim}{\cdot}}} S$ and $M \rightsquigarrow M'$, then $\Sigma; \Gamma; \Delta \ \vdash\ M' : S$.
3. If $\Sigma; \Gamma; \Delta \ \vdash\ E \div S$ and $\Sigma; E \hookrightarrow \Sigma; E'$, then $\Sigma; \Gamma; \Delta \ \vdash\ E' \div S$.

**Theorem 2 (Progress for *f*CLL).**
1. If $\Sigma; \cdot; \cdot \ \vdash\ N : A$ then either $N = V$ or $N \rightsquigarrow N'$ for some $N'$.
2. If $\Sigma; \cdot; \cdot \ \vdash\ M \mathrel{\dot{\overset{\sim}{\cdot}}} S$ then either $M = M_V$ or $M \mapsto M'$ for some $M'$.
3. If $\Sigma; \cdot; \cdot \ \vdash\ E \div S$ then either $E = E_V$ or $\Sigma; E \hookrightarrow \Sigma; E'$ for some $E'$.

**Example 1 (Fibonacci numbers).** As a simple example of programming in *f*CLL, we describe a function for computing Fibonacci numbers. These numbers are defined inductively as follows.

```
fib: int → {!int} = λn : int.
          if (n = 0 or n = 1) then {!1}
          else
          {
             let {!n₁} = fib (n − 1) in
             let {!n₂} = fib (n − 2) in
                !(n₁ + n₂)
          }
```

**Fig. 4.** The function `fib` in $f$CLL

$$fib(0) = fib(1) = 1 \qquad fib(n) = fib(n-1) + fib(n-2)$$

For implementing this definition as a function in $f$CLL, we assume that $f$CLL terms have been extended with integers having type `int`, named recursive functions and a conditional if-then-else construct. These can be added to $f$CLL in a straightforward manner. Figure 4 shows the $f$CLL function `fib` that computes the $n$th Fibonacci number. It has the type `int` $\rightarrow$ {!`int`}. It is possible to write this function in a manner simpler than the one presented here, but we write it this way to highlight specific features of $f$CLL.

The most interesting computation in `fib`, including recursive calls, occurs inside the monad. Since the monad is evaluated lazily in $f$CLL, computation in `fib` will actually occur only when the caller of `fib` eliminates the monad from the returned value of type {!`int`}. Syntactically, elimination of the monadic constructor can occur only in expressions at the let construct. Hence the program that calls `fib` must be an expression. Here is an example of such a top level program that prints the 5th Fibonacci number: let $\{!x\}$ = `fib` 5 in `print`$(x)$.

## 3 $l$CLL: Concurrent core of CLL

The concurrent core of CLL is called $l$CLL. It embeds the functional language $f$CLL directly. In the structure of concurrent computations $l$CLL is similar to the $\pi$-calculus. However it is different in other respects. First, it allows a direct representation of functional computation inside concurrent ones, as opposed to the use of complex encodings for doing the same in the $\pi$-calculus [20]. Second, the semantics of $l$CLL are directed by types, not terms. This, we believe, is a new idea that has not been explored before.

**Syntax**. We present $l$CLL as a chemical abstract machine (CHAM) [7]. $l$CLL programs are called configurations, denoted by $\mathcal{C}$. Figure 5 shows the syntax of $l$CLL configurations. Each configuration is made of four components, written $\Sigma; \hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}$. $\Sigma$ is a context of index variables, as defined in section 2. $\hat{\sigma}$ is a sorted substitution mapping index variables to index terms. $\hat{\Gamma}$ is a set of closed $f$CLL term values along with their types. $\hat{\Delta}$ is a multiset of closed $f$CLL programs together with their types. We require that whenever $N : A \in \hat{\Delta}$, $N$ have the type $A[\hat{\sigma}]$, where $A[\hat{\sigma}]$ is the result of applying the substitution $\hat{\sigma}$ to the type $A$. Similar conditions hold for monadic-terms and expressions in $\hat{\Delta}$ and term values in $\hat{\Gamma}$. Formally, a configuration $\Sigma; \hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}$ is said to be well-formed if it satisfies the following conditions.

| | |
|---|---|
| Configurations | $\mathcal{C} ::= \Sigma; \hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}$ |
| Global index names | $\Sigma ::= \cdot \mid \Sigma, i : \gamma$ |
| Local name substitutions | $\hat{\sigma} ::= \cdot \mid \hat{\sigma}, t/i : \gamma$ |
| Unrestricted solutions | $\hat{\Gamma} ::= \cdot \mid \hat{\Gamma}, V : A$ |
| Linear solutions | $\hat{\Delta} ::= \cdot \mid \hat{\Delta}, N : A \mid \hat{\Delta}, M \Bumpeq S \mid \hat{\Delta}, E \div S$ |

**Fig. 5.** $l$CLL syntax

1. If $(t/i : \gamma) \in \hat{\sigma}$, then $i \notin \mathtt{dom}(\Sigma)$ and $\Sigma \vdash t \in \gamma$.
2. If $P$ is a program in $\hat{\Gamma}$ or $\hat{\Delta}$, then $\mathtt{fv}(P) \cap \mathtt{dom}(\hat{\sigma}) = \phi$.
3. If $V : A \in \hat{\Gamma}$, then $\Sigma; \cdot; \cdot \vdash V : A[\hat{\sigma}]$.
4. If $N : A \in \hat{\Delta}$, then $\Sigma; \cdot; \cdot \vdash N : A[\hat{\sigma}]$.
5. If $M \Bumpeq S \in \hat{\Delta}$, then $\Sigma; \cdot; \cdot \vdash M \Bumpeq S[\hat{\sigma}]$.
6. If $E \div S \in \hat{\Delta}$, then $\Sigma; \cdot; \cdot \vdash E \div S[\hat{\sigma}]$.

We assume that all our configurations are well-formed. Programs in $\hat{\Delta}$ and values in $\hat{\Gamma}$ are collectively called processes. Intuitively, we view programs in $\hat{\Delta}$ as concurrent processes that are executing simultaneously. $\hat{\Delta}$ is called a linear solution because these processes are single-use in the sense that they can neither be replicated, nor destroyed. Term values in $\hat{\Gamma}$ are viewed as irreducible processes (like functional abstractions) that are replicable. For this reason $\hat{\Gamma}$ is also called an unrestricted solution. The context $\Sigma$ can be viewed as a set of global index names, that are known to have specific sorts. The domain of the substitution $\hat{\sigma}$ can be viewed as a set of local (private) index names that are created during the evaluation of the configuration. The substitution $\hat{\sigma}$ maps these local index names to index terms that depend only on the global names (see condition (1) for well-formedness above).

## 3.1 Semantics of $l$CLL

The semantics of $l$CLL are rewrite rules that allow a configuration to step to other configuration(s). The specific rules that apply to a particular configuration are determined by the *types* of processes in that configuration. In this sense, these rules are type-directed. We classify rewrite rules into three classes – functional, structural and synchronization.

**Functional rules**. Functional rules allow reduction of programs in the linear solution $\hat{\Delta}$. We denote them using the arrow $\twoheadrightarrow$. Figure 6 shows the functional rewrite rules for $l$CLL configurations. There are three rules, one for reducing programs in each of the three syntactic classes of $f$CLL. Reductions of different programs in $\hat{\Delta}$ can be performed in parallel. This supports the idea that programs in $\hat{\Delta}$ can be viewed as processes executing simultaneously.

**Structural rules**. Structural rules apply to those irreducible programs in $\hat{\Delta}$ that have synchronous types. These are exactly the monadic values $M_V$. A structural rule decomposes a monadic value into smaller monadic values. We denote structural rules with the arrow $\rightharpoonup$. All structural rules for rewriting $l$CLL configurations are shown in figure 7. Unlike most CHAMs, our structural rules are not reversible.

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

$$\frac{M \mapsto M'}{\Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}, M \mathrel{\dot{\asymp}} S \twoheadrightarrow \Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}, M' \mathrel{\dot{\asymp}} S}\ {\twoheadrightarrow}{\mapsto}$$

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

**Fig. 6.** Functional rewrite rules for $l$CLL configurations

$$\begin{array}{rcll}
\Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}, (M_{V_1} \otimes M_{V_2}) \mathrel{\dot{\asymp}} (S_1 \otimes S_2) & \rightharpoonup & \Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}, M_{V_1} \mathrel{\dot{\asymp}} S_1, M_{V_2} \mathrel{\dot{\asymp}} S_2 & (\rightharpoonup \otimes) \\
\Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}, \star \mathrel{\dot{\asymp}} \mathbf{1} & \rightharpoonup & \Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta} & (\rightharpoonup \mathbf{1}) \\
\Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}, [t, M_V] \mathrel{\dot{\asymp}} \exists i : \gamma.S(i) & \rightharpoonup & \Sigma;\hat{\sigma}, t/i : \gamma \triangleright \hat{\Gamma} \parallel \hat{\Delta}, M_V \mathrel{\dot{\asymp}} S(i) & (\rightharpoonup \exists) \\
 & & (i \text{ fresh}) & \\
\Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}, !V \mathrel{\dot{\asymp}} !A & \rightharpoonup & \Sigma;\hat{\sigma} \triangleright \hat{\Gamma}, V : A \parallel \hat{\Delta} & (\rightharpoonup !) \\
\Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}, V \mathrel{\dot{\asymp}} A & \rightharpoonup & \Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}, V : A & (\rightharpoonup \mathrel{\dot{\asymp}}) \\
\Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}, M_V \div S & \rightharpoonup & \Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta}, M_V \mathrel{\dot{\asymp}} S & (\rightharpoonup \div)
\end{array}$$

**Fig. 7.** Structural rewrite rules for $l$CLL configurations

The rule $\rightharpoonup \otimes$ splits the monadic value $M_{V_1} \otimes M_{V_2}$ of type $S_1 \otimes S_2$ into two monadic values $M_{V_1}$ and $M_{V_2}$ of types $S_1$ and $S_2$ respectively. Intuitively, we can view $M_{V_1} \otimes M_{V_2}$ as a parallel composition of the processes $M_{V_1}$ and $M_{V_2}$. The rule $\rightharpoonup \otimes$ splits this parallel composition into its components, allowing each component to rewrite separately.

In the rule $\rightharpoonup \exists$, there is a side condition that $i$ must be fresh i.e. it must not occur anywhere except in $S(i)$. Some $\alpha$-renaming may have to be performed to enforce this. In $l$CLL, the $\exists$ type acts as a local index name creator. The rule $\rightharpoonup \exists$ creates the new index name $i$ and records the fact that $i$ is actually bound to the index term $t$ in the substitution $\hat{\sigma}$.

The rule $\rightharpoonup !$ moves a program of type $!A$ to the unrestricted solution, thus allowing multiple uses of this program. For this reason, the type $!A$ serves as a replication construct in $l$CLL. The rules $\rightharpoonup \mathrel{\dot{\asymp}}$ and $\rightharpoonup \div$ change the type ascription for programs that have been coerced from one syntactic class to another.

**Synchronization Rules**. Synchronization rules act on values in $\hat{\Gamma}$ and $\hat{\Delta}$ having asynchronous types. These are exactly the term values $V$. Synchronization rules are denoted by the arrow $\longrightarrow$. Figure 8 shows the two synchronization rules. The rule $\longrightarrow \{\}$ eliminates the monadic constructor $\{\}$ from values $\{E\}$ of asynchronous type $\{S\}$.

The second rule $\longrightarrow\Longrightarrow$ performs synchronization of several term values at the same time. It uses an auxiliary judgment $\Sigma;\hat{\sigma} \triangleright \hat{\Gamma} \parallel \hat{\Delta} \Longrightarrow N : A$, which we call the sync judgment. The rules of this judgment are also shown in figure 8. The sync judgment links values in $\hat{\Gamma}$ and $\hat{\Delta}$ to form a more complex program $N$.

$$\boxed{\textbf{Synchronization rules}, \boldsymbol{\Sigma}; \hat\sigma \triangleright \hat{\boldsymbol{\Gamma}} \parallel \hat{\boldsymbol{\Delta}} \;\longrightarrow\; \boldsymbol{\Sigma}; \hat\sigma \triangleright \hat{\boldsymbol{\Gamma}} \parallel \hat{\boldsymbol{\Delta}}'}$$

$$\frac{}{\Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta, \{E\} : \{S\} \;\longrightarrow\; \Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta, E \div S} \longrightarrow \{\}$$

$$\frac{\Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta \;\Longrightarrow\; N : \{S\}}{\Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta, \hat\Delta' \;\longrightarrow\; \Sigma; \hat\sigma \triangleright \hat\Gamma \parallel N : \{S\}, \hat\Delta'} \longrightarrow\Longrightarrow$$

$$\boxed{\textbf{Sync judgment}, \boldsymbol{\Sigma}; \hat\sigma \triangleright \hat{\boldsymbol{\Gamma}} \parallel \hat{\boldsymbol{\Delta}} \;\Longrightarrow\; \mathbf{N} : \mathbf{A}}$$

$$\frac{}{\Sigma; \hat\sigma \triangleright \hat\Gamma \parallel V : A \;\Longrightarrow\; V : A} \Longrightarrow_{HYP1} \qquad \frac{}{\Sigma; \hat\sigma \triangleright \hat\Gamma, V : A \parallel \cdot \;\Longrightarrow\; V : A} \Longrightarrow_{HYP2}$$

$$\frac{\Sigma \cup \mathtt{dom}(\hat\sigma) \;\vdash\; t \in \gamma \qquad \Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta \;\Longrightarrow\; N : \forall i : \gamma. A(i)}{\Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta \;\Longrightarrow\; N\ [t[\hat\sigma]] : A(t)} \Longrightarrow \forall$$

$$\frac{\Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta_1 \;\Longrightarrow\; N_1 : A \qquad \Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta_2 \;\Longrightarrow\; N_2 : A \multimap B}{\Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta_1, \hat\Delta_2 \;\Longrightarrow\; N_2 \,\hat{}\, N_1 : B} \Longrightarrow\multimap$$

$$\frac{\Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \cdot \;\Longrightarrow\; N_1 : A \qquad \Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta \;\Longrightarrow\; N_2 : A \rightarrow B}{\Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta \;\Longrightarrow\; N_2\ N_1 : B} \Longrightarrow\rightarrow$$

**Fig. 8.** Synchronization rewrite rules for $l$CLL configurations

We call this process synchronization. Synchronization uses values in $\hat\Delta$ exactly once, while those in $\hat\Gamma$ may be used zero or more times.

In the rule $\longrightarrow\Longrightarrow$ shown in figure 8, $\hat\Delta$ denotes a subset of the linear solution that participates in the synchronization. The remaining solution $\hat\Delta'$ is kept as is. Some backward reasoning is performed in the judgment $\Longrightarrow$ to produce the linked program $N$ of type $\{S\}$. This is the essential point here – the result of a synchronization must be of type $\{S\}$.

The semantic rewriting relation for $l$CLL is defined as $\rightrightarrows = \twoheadrightarrow \cup \rightharpoonup \cup \longrightarrow$. It satisfies the following type preservation theorem.

**Theorem 3 (Preservation for $l$CLL).** If $\mathcal{C}$ is a well-formed configuration and $\mathcal{C} \rightrightarrows \mathcal{C}'$, then $\mathcal{C}'$ is also well-formed.

**Concurrent computation as proof search**. Given a $l$CLL configuration $\mathcal{C} = \Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta$, types in $\hat\Delta[\hat\sigma]$ and $\hat\Gamma[\hat\sigma]$ can be viewed as propositions that are *simultaneously true*, in a linear and unrestricted sense respectively. Using the Curry-Howard isomorphism, the corresponding programs in $\hat\Delta[\hat\sigma]$ and $\hat\Gamma[\hat\sigma]$ can be seen as specific proofs of these propositions. The sync judgment (figure 8) is actually a linear entailment judgment – if $\Sigma; \hat\sigma \triangleright \hat\Gamma \parallel \hat\Delta \Longrightarrow N : A$, then from the unrestricted assumptions in $\hat\Gamma[\hat\sigma]$ and linear assumptions in $\hat\Delta[\hat\sigma]$, $A[\hat\sigma]$ can be proved in linear logic. The term $N$ synthesized by this judgment is a proof of the proposition $A[\hat\sigma]$. As a result, each use of the synchronization rule $\longrightarrow\Longrightarrow$ can be viewed as a step of proof search in linear logic that uses several known facts to conclude a new fact, together with its proof term. By the Curry-Howard isomorphism, the proof term is a well-typed program that can be functionally reduced again.

More specifically, each use of $\longrightarrow\!\!\Longrightarrow$ corresponds to a single focusing step for eliminating asynchronous constructors from a proposition that has $\{S\}$ in the head position. For a detailed description of this see [10].

**Example 2 (Client-Server Communication).** We illustrate concurrent programming in *l*CLL with an example of a client-server interaction. The server described here listens to client requests to compute Fibonacci numbers. Each request contains an integer $n$. Given a request, the server computes the $n$th Fibonacci number and returns this value to the client.

We model communication through asynchronous message passing. Assume that all clients and the server have unique identities, which are index names from a special sort called `procid`. The identity of the server is *serv*. A message from one process to another contains three parts – the identity of the sender, the identity of the recipient and an integer, which is the content of the message. Messages are modeled using a type constructor `mess` and a term constructor `message` having the kind and type shown in figure 9. For every pair of index terms $i$ and $j$ of sort `procid` and every integer $n$, we view the value (`message` $[i]$ $[j]$ $n$) of type (`mess` $i$ $j$) as a message having content $n$ from the process with identity $i$ to the process with identity $j$. In order to extract the integer content of a message, we use the destructor `fetchmessage` that has the reduction rule `fetchmessage` $[i]$ $[j]$ ^ (`message` $[i]$ $[j]$ $n$) $\rightsquigarrow$ $\{!n\}$.

The server program called `fibserver` is shown in figure 9. It waits for a message $m$ from any client $i$. Then it extracts the content $n$ from the message, computes the $n$th Fibonacci number using the function `fib` defined in example 1 and returns this computed value to the client $i$ as a message. `fibserver` has the type `fibservtype` $= \forall i :$ `procid`. `mess` $i$ *serv* $\multimap$ $\{$`mess` *serv* $i\}$.

A sequence of rewrite steps in *l*CLL using `fibserver` is shown in figure 9. The initial configuration contains `fibserver` and a message to `fibserver` containing the integer 6 from a client having identity $k$. For brevity, we omit the client process. The crucial rewrite in this sequence is the first one, where the synchronization rule $\longrightarrow\!\!\Longrightarrow$ is used to link the `fibserver` program with the message for it. Rewriting ends with a message containing the value of the 6th Fibonacci number (namely 13) from `fibserver` to the requesting client $k$.

## 3.2 An encoding of the $\pi$-calculus in *l*CLL

We describe a translation of a variant of the asynchronous $\pi$-calculus [8] to *l*CLL. The syntax and semantics of this variant are shown in figure 10. It extends the asynchronous $\pi$-calculus with a nil process 0. The replication operator ! is restricted to actions only.

Two translations $\ulcorner\cdot\urcorner$ and $\ulcorner\!\ulcorner\cdot\urcorner\!\urcorner$ are shown in figure 11. They map $\pi$-calculus entities to programs and types of *fCLL* respectively. We model channels as index terms of a specific sort `chan`. In order to translate $\bar{x}y$, which is an output message, we introduce a type constructor `out` and a related term constructor `output`, whose kind and type are shown in figure 11. The translations of $\bar{x}y$ to terms and types are `output` $[x]$ $[y]$ and `out` $x$ $y$ respectively.

**Additional Signature**

$\texttt{procid}$: sort
$serv : \texttt{procid}$
$\texttt{mess}$: $\texttt{procid} \rightarrow \texttt{procid} \rightarrow \texttt{Type}$
$\texttt{message}$: $\forall i : \texttt{procid}.\ \forall j : \texttt{procid}.\ \texttt{int} \rightarrow \texttt{mess}\ i\ j$
$\texttt{fetchmessage}$: $\forall i : \texttt{procid}.\ \forall j : \texttt{procid}.\ \texttt{mess}\ i\ j \multimap \{!\texttt{int}\}$
$\texttt{fetchmessage}\ [i]\ [j]\ \hat{}\ (\texttt{message}\ [i]\ [j]\ n) \ \rightsquigarrow\ \{!n\}$

**Fibonacci Server**

$\texttt{fibservtype = } \forall i : \texttt{procid}.\ \texttt{mess}\ i\ serv \multimap \{\texttt{mess}\ serv\ i\}$
$\texttt{fibserver: fibservtype = } \Lambda i : \texttt{procid}.\ \hat{\lambda} m : \texttt{mess}\ i\ serv.$

```
{
   let {!n} = fetchmessage [i] [serv] ^ m in
   let {!v} = fib (n) in
      (message [serv] [i] v)
}
```

**Sample Execution** ($\Sigma = serv : \texttt{procid},\ k : \texttt{procid}$)

$$\begin{array}{ll}
 & \Sigma; \cdot \rhd \cdot \Vert\ \texttt{fibserver} : \texttt{fibservtype},\ (\texttt{message}\ [k]\ [serv]\ 6) : \texttt{mess}\ k\ serv \\
\longrightarrow & \Sigma; \cdot \rhd \cdot \Vert\ \texttt{fibserver}\ [k]\ \hat{}\ (\texttt{message}\ [k]\ [serv]\ 6) : \{\texttt{mess}\ serv\ k\} \\
\twoheadrightarrow^* & \Sigma; \cdot \rhd \cdot \Vert \left( \begin{array}{l} \{\ \ \underline{\texttt{let}}\ \{!n\} = \texttt{fetchmessage}\ [k]\ [serv]\ \hat{}\ (\texttt{message}\ [k]\ [serv]\ 6)\ \underline{\texttt{in}} \\ \quad \underline{\texttt{let}}\ \{!v\} = \texttt{fib}\ (n)\ \underline{\texttt{in}}\ (\texttt{message}\ [serv]\ [k]\ v) \\ \}\ :\ \{\texttt{mess}\ serv\ k\} \end{array} \right) \\
\longrightarrow & \Sigma; \cdot \rhd \cdot \Vert \left( \begin{array}{l} (\ \ \underline{\texttt{let}}\ \{!n\} = \texttt{fetchmessage}\ [k]\ [serv]\ \hat{}\ (\texttt{message}\ [k]\ [serv]\ 6)\ \underline{\texttt{in}} \\ \quad \underline{\texttt{let}}\ \{!v\} = \texttt{fib}\ (n)\ \underline{\texttt{in}}\ (\texttt{message}\ [serv]\ [k]\ v) \\ )\ \div\ \texttt{mess}\ serv\ k \end{array} \right) \\
\twoheadrightarrow^* & \Sigma; \cdot \rhd \cdot \Vert\ (\underline{\texttt{let}}\ \{!v\} = \texttt{fib}\ (6)\ \underline{\texttt{in}}\ (\texttt{message}\ [serv]\ [k]\ v)) \div \texttt{mess}\ serv\ k \\
\twoheadrightarrow^* & \Sigma; \cdot \rhd \cdot \Vert\ (\texttt{message}\ [serv]\ [k]\ 13) \div \texttt{mess}\ serv\ k \\
\rightharpoonup^2 & \Sigma; \cdot \rhd \cdot \Vert\ (\texttt{message}\ [serv]\ [k]\ 13) : \texttt{mess}\ serv\ k
\end{array}$$

**Fig. 9.** Server for computing Fibonacci numbers in *l*CLL

**Syntax**

| | |
|---|---|
| Actions | $A ::= \bar{x}y \mid x(y).P$ |
| Processes | $P, Q ::= A \mid !A \mid P\vert P \mid \nu x.P \mid 0$ |
| Molecules | $m ::= P \mid \nu x.S$ |
| Solutions | $S ::= \phi \mid S \uplus \{m\}$ |

**Equations on terms and solutions**

$$\nu x.P = \nu y.P[y/x] \quad (y \notin P) \qquad\qquad \nu x.S = \nu y.S[y/x] \quad (y \notin S)$$

**CHAM semantics**

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}l}
P_1 | P_2 & \rightleftharpoons & P_1, P_2 & x(y).P\ ,\ \bar{x}z & \rightarrow & P[z/y] & \\
0 & \rightleftharpoons & & \nu x.P & \rightleftharpoons & \nu x.\{P\} & \\
!A & \rightleftharpoons & !A, A & (\nu x.P)|Q & \rightleftharpoons & \nu x.(P|Q) & (x \notin Q)
\end{array}$$

**Reduction semantics**

$$P \equiv P' \ \Leftrightarrow\ P \rightleftharpoons^* P' \qquad\qquad P \rightarrow P' \ \Leftrightarrow\ P \rightleftharpoons^* \rightarrow \rightleftharpoons^* P'$$

**Fig. 10.** A variant of the asynchronous $\pi$-calculus

To translate $x(y).P$, we introduce a term destructor `destroyout` corresponding to the constructor `output`. Its type and reduction rule are shown in figure 11. The translation $\ulcorner x(y).P \urcorner$ waits for two inputs – the channel name $y$ and a mes-

**Additional Signature**

$\texttt{chan}$: sort
$\texttt{out}: \texttt{chan} \to \texttt{chan} \to \texttt{Type}$
$\texttt{output}: \forall x : \texttt{chan}.\ \forall y : \texttt{chan}.\ \texttt{out}\ x\ y$
$\texttt{destroyout}: \forall x : \texttt{chan}.\ \forall y : \texttt{chan}.\ \texttt{out}\ x\ y \multimap \{\mathbf{1}\}$
$\texttt{destroyout}\ [x]\ [y]\ \hat{}\ (\texttt{output}\ [x]\ [y]) \rightsquigarrow \{\star\}$
$c_{\texttt{chan}} : \texttt{chan}$

| A/P | $f$CLL Type, $\ulcorner\!\ulcorner A/P \urcorner\!\urcorner$ | $f$CLL Program, $\ulcorner A/P \urcorner$ |
|---|---|---|
| $\bar{x}y$ | $\texttt{out}\ x\ y$ | $\texttt{output}\ [x]\ [y]$ |
| $x(y).P$ | $\forall y : \texttt{chan}.\ \texttt{out}\ x\ y \multimap \{\ulcorner\!\ulcorner P \urcorner\!\urcorner\}$ | $\Lambda y : \texttt{chan}.\ \hat{\lambda} m : \texttt{out}\ x\ y.$ { <u>let</u> $\{\star\} = \texttt{destroyout}\ [x]\ [y]\ \hat{}\ m$ <u>in</u> $\ulcorner P \urcorner$ } |
| $0$ | $\mathbf{1}$ | $\star$ |
| $!A$ | $!\ \ulcorner\!\ulcorner A \urcorner\!\urcorner$ | $!\ \ulcorner A \urcorner$ |
| $P_1 \vert P_2$ | $\ulcorner\!\ulcorner P_1 \urcorner\!\urcorner \otimes \ulcorner\!\ulcorner P_2 \urcorner\!\urcorner$ | $\ulcorner P_1 \urcorner \otimes \ulcorner P_2 \urcorner$ |
| $\nu x.P$ | $\exists x : \texttt{chan}.\ulcorner\!\ulcorner P \urcorner\!\urcorner$ | $[c_{\texttt{chan}},\ (\ulcorner P \urcorner [c_{\texttt{chan}}/x])]$ |

**Fig. 11.** Translation of the $\pi$-calculus

sage $m$ that corresponds to the translation of $\bar{x}y$. It then discards the message $m$ and starts the process $P$.

Translations of $!A$, $P_1|P_2$ and 0 are straightforward. We translate $\nu x.P$ to the type $\exists x : \texttt{chan}.\ulcorner\!\ulcorner P \urcorner\!\urcorner$. To translate $\nu x.P$ to a program, we assume that there is an index constant $c_{\texttt{chan}}$ of sort $\texttt{chan}$. Then we translate $\nu x.P$ to $[c_{\texttt{chan}},\ (\ulcorner P \urcorner [c_{\texttt{chan}}/x])]$, which has the type $\exists x : \texttt{chan}.\ulcorner\!\ulcorner P \urcorner\!\urcorner$.

For any $\pi$-calculus process $P$, $\texttt{fn}(P) : \texttt{chan}; \cdot; \cdot \vdash \ulcorner P \urcorner \div \ulcorner\!\ulcorner P \urcorner\!\urcorner$. The translation of a $\pi$-calculus process $P$ to $l$CLL is defined as the configuration $\langle P \rangle = \texttt{fn}(P) : \texttt{chan}; \cdot \rhd \cdot \parallel \ulcorner P \urcorner \div \ulcorner\!\ulcorner P \urcorner\!\urcorner$. Although we have not formally proved it, we believe that the following correctness result holds for this translation: $P \to^* P'$ iff there is a $l$CLL configuration $\mathcal{C}$ such that $\langle P \rangle \rightrightarrows^* \mathcal{C}$ and $\langle P' \rangle \rightharpoonup^* \mathcal{C}$.

## 4 Full-CLL: The complete language

Full-CLL is an extension of $f$CLL that allows $l$CLL's concurrent computations inside functional ones. This is done by extending $f$CLL expressions by a single construct – <u>link</u> $E \div S$ <u>to</u> $G$. $G \in \{A, !A, \mathbf{1}\}$ is called a goal type. Additional syntax and semantics for this construct are shown in figure 12. Other than the <u>link</u> construct, full-CLL inherits all of $f$CLL's syntax, typing rules and semantics.

<u>link</u> $E \div S$ <u>to</u> $G$ is evaluated in a context of index variables $\Sigma$ as follows. First, the $l$CLL configuration $\mathcal{C} = \Sigma; \cdot \rhd \cdot \parallel E \div S$ is created and allowed to rewrite according to the relation $\rightrightarrows$ till it reaches a *quiescent* configuration $\mathcal{C}'$. By quiescent we mean that no rewrite rule applies to $\mathcal{C}'$ i.e. $\mathcal{C}'$ is in $\rightrightarrows$-normal form. After $\mathcal{C}'$ is obtained, the result of evaluating <u>link</u> $E \div S$ <u>to</u> $G$ depends on the goal type $G$.

**Syntax**

$$\begin{array}{l} \text{Expressions } E ::= \ldots \mid \underline{\texttt{link}}\ E \div S\ \underline{\texttt{to}}\ G \\ \text{Goal Types } G ::= A \mid !A \mid \mathbf{1} \end{array}$$

**Typing rules**

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

**Operational Semantics**

$$\frac{\Sigma; \cdot \rhd \cdot \parallel E \div S \ \rightrightarrows^* \ \Sigma; \hat{\sigma} \rhd \hat{\Gamma} \parallel V : A}{\Sigma; \underline{\texttt{link}}\ E \div S\ \underline{\texttt{to}}\ A \ \hookrightarrow \ \Sigma; V} \hookrightarrow_1 \qquad \frac{\Sigma; \cdot \rhd \cdot \parallel E \div S \ \rightrightarrows^* \ \Sigma; \hat{\sigma} \rhd \hat{\Gamma}, V : A \parallel \cdot}{\Sigma; \underline{\texttt{link}}\ E \div S\ \underline{\texttt{to}}\ A \ \hookrightarrow \ \Sigma; V} \hookrightarrow_2$$

$$\frac{\Sigma; \cdot \rhd \cdot \parallel E \div S \ \rightrightarrows^* \ \Sigma; \hat{\sigma} \rhd \hat{\Gamma}, V : A \parallel \cdot}{\Sigma; \underline{\texttt{link}}\ E \div S\ \underline{\texttt{to}}\ !A \ \hookrightarrow \ \Sigma; !V} \hookrightarrow_3 \qquad \frac{\Sigma; \cdot \rhd \cdot \parallel E \div S \ \rightrightarrows^* \ \Sigma; \hat{\sigma} \rhd \hat{\Gamma} \parallel \cdot}{\Sigma; \underline{\texttt{link}}\ E \div S\ \underline{\texttt{to}}\ \mathbf{1} \ \hookrightarrow \ \Sigma; \star} \hookrightarrow_4$$

**Fig. 12.** Full-CLL syntax and semantics

1. If $G = A$ and $\mathcal{C}' = \Sigma; \hat{\sigma} \rhd \hat{\Gamma} \parallel V : A$ or $\mathcal{C}' = \Sigma; \hat{\sigma} \rhd \hat{\Gamma}, V : A \parallel \cdot$, then $\underline{\texttt{link}}\ E \div S\ \underline{\texttt{to}}\ G$ evaluates to $V$.
2. If $G = !A$ and $\mathcal{C}' = \Sigma; \hat{\sigma} \rhd \hat{\Gamma}, V : A \parallel \cdot$, then $\underline{\texttt{link}}\ E \div S\ \underline{\texttt{to}}\ G$ evaluates to $!V$.
3. If $G = \mathbf{1}$ and $\mathcal{C}' = \Sigma; \hat{\sigma} \rhd \hat{\Gamma} \parallel \cdot$, then $\underline{\texttt{link}}\ E \div S\ \underline{\texttt{to}}\ G$ evaluates to $\star$.

All these conditions are summarized in figure 12. If none of these conditions hold, evaluation of the $\underline{\texttt{link}}$ construct fails and computation deadlocks. We call this condition *link failure*. Since expressions are coerced into terms through a monad, link failure never occurs during evaluation of terms and monadic-terms. As a result, full-CLL has the following progress theorem.

**Theorem 4 (Progress for full-CLL).**
1. If $\Sigma; \cdot; \cdot \ \vdash \ N : A$ then either $N = V$ or $N \rightsquigarrow N'$ for some $N'$.
2. If $\Sigma; \cdot; \cdot \ \vdash \ M \mathrel{\dot{\div}} S$ then either $M = M_V$ or $M \mapsto M'$ for some $M'$.
3. If $\Sigma; \cdot; \cdot \ \vdash \ E \div S$ then either $E = E_V$ or $\Sigma; E \hookrightarrow \Sigma; E'$ for some $E'$ or reduction of $\Sigma; E$ deadlocks due to link failure.

Link failure is easy to detect at runtime and can be handled, for example, by throwing an exception. For all practical problems that we encountered, we found it possible to write programs in which link failure never occurs. $f$CLL's preservation theorem (theorem 1) holds for full-CLL also.

**Example 3 (Fibonacci numbers in full-CLL).** Figure 13 shows a concurrent implementation of Fibonacci numbers in full-CLL. The function `fibc` uses the additional signature from example 2 and assumes that the sort `procid` contains at least three constants $k_1, k_2$ and $k$. `fibc` has the type `int` $\rightarrow$ {`!int`}. Given an input integer $n \geq 2$, `fibc` computes the $n$th Fibonacci number using a $\underline{\texttt{link}}$ construct that starts concurrent computation with a tensor of three processes having identities $k_1$, $k_2$ and $k$ respectively. The first two processes recursively compute $fib(n-1)$ and $fib(n-2)$ and send these values as messages to the third process. The third process waits for these messages ($m_1$ and $m_2$), extracts their integer contents and adds them together to obtain $fib(n)$. This becomes the result of evaluation of the $\underline{\texttt{link}}$ construct.

During the evaluation of `fibc`, each of the two recursive calls can encounter a $\underline{\texttt{link}}$ construct and create a nested $l$CLL concurrent computation. Since the two

```
fibc = λn : int.
   if (n = 0 or n = 1) then {!1}
   else
   {  link
      (
            {let {!n₁} = fibc (n − 1) in (message [k₁] [k] n₁)}
         ⊗  {let {!n₂} = fibc (n − 2) in (message [k₂] [k] n₂)}
         ⊗  λ̂m₁ : mess k₁ k. λ̂m₂ : mess k₂ k.
            {
               let {!x} = fetchmessage [k₁] [k] ^ m₁ in
               let {!y} = fetchmessage [k₂] [k] ^ m₂ in
                  !(x + y)
            }
      ) ÷ {mess k₁ k} ⊗ {mess k₂ k} ⊗ (mess k₁ k ⊸ mess k₂ k ⊸ {!int})
      to !int
   }
```

**Fig. 13.** The function `fibc` in full-CLL

recursive calls can be executed simultaneously, there may actually be more than one nested $l$CLL configuration at the same time. However, these configurations are distinct – processes in one configuration cannot synchronize with those in another. In general, full-CLL programs can spawn several nested concurrent computations that are completely disjoint from each other.

## 5 Conclusion

We have presented a language that combines functional and concurrent computation in a logically motivated manner. It requires linearity, a restricted form of dependent types, a monad, and focusing, in order to retain the desirable properties of each paradigm in their combination.

Perhaps the biggest limitation of our work is that the logic underlying the type system is not strong enough to express many useful properties of concurrent programs like deadlock freedom. This is clearly visible in the fact that full-CLL does not have a progress theorem as strong as that of its functional core $f$CLL. Our types represent only basic structural properties of concurrent processes. At the same time, due to the presence of dependent and linear types, the type system can be used to express very strong functional guarantees about various components of a concurrent program. Finding a logic that can express useful properties of both functional and concurrent computation and converting it to a programming language using the Curry-Howard isomorphism is a challenge at present. Another challenge is to build a realistic implementation of CLL, including a more complete functional language and type reconstruction to see if our ideas scale in practice. Since concurrency in CLL is somewhat low-level, it will be important to build up libraries of common idioms in order to write large programs conveniently.

## References

1. CLL implementation. Available electronically from http://www.cs.cmu.edu/~dg.
2. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. of POPL'01*, pages 104–115, 2001.
3. S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1–2):3–57, 1993.
4. S. Abramsky, S. Gay, and R. Nagarajan. Specification structures and propositions-as-types for concurrency. In *Logics for Concurrency: Structure vs. Automata—Proc. of the VIIIth Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
5. J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.
6. J.-M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. Technical Report ECRC-91-12, European Computer-Industry Research Centre, 1991.
7. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
8. G. Boudol. Asynchrony and the pi-calculus. Technical Report RR-1702, INRIA SofiaAntipolis, 1992.
9. S. Conchon and F. L. Fessant. Jocaml: Mobile agents for objective-caml. In *Proc. of ASAMA'99*. IEEE Computer Society, 1999.
10. D. Garg. CLL: A concurrent language built from logical principles. Technical Report CMU-CS-05-104, Computer Science Department, Carnegie Mellon University, January 2005.
11. A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.
12. J.-Y. Girard. Linear logic. In *Theoretical Computer Science*, volume 5, 1987.
13. S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of POPL'96*, 1996.
14. P. López, F. Pfenning, J. Polakow, and K. Watkins. Monadic concurrent linear logic programming. In *Proc. of PPDP'05*, 2005. To appear.
15. M. Nygaard and G. Winskel. Domain theory for concurrency. *Theor. Comput. Sc.*, 316(1-3), 2004.
16. F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Math. Struc. in Comp. Sci.*, 11(4):511–540, 2001.
17. B. C. Pierce and D. 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.
18. J. H. Reppy. CML: A higher-order concurrent language. In *Proc. of PLDI'91*, 1991.
19. J. H. Reppy. *Concurrent programming in ML*. Cambridge University Press, 1999.
20. D. Sangiorgi and D. Walker. *The $\pi$-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001. Chapters 15–17.
21. K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgements and properties. Technical Report CMU-CS-02-101, Computer Science Department, Carnegie Mellon University, May 2003.
22. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. of POPL'99*, 1999.