Higher-order, linear, concurrent constraint programming

Vijay Saraswat Xerox PARC
3333 Coyote Hill Road
Palo Alto Ca 94304
(saraswat@parc.xerox.com)

Patrick Lincoln
Department of Computer Science
Stanford University
Stanford Ca 94305
(lincoln@cs.stanford.edu)*

July 1992

Abstract

We present a very simple and powerful framework for indeterminate, asynchronous, higher-order computation based on the *formula-as-agent* and *proof-as-computation* interpretation of (higher-order) linear logic [Gir87]. The framework significantly refines and extends the scope of the concurrent constraint programming paradigm [Sar89] in two fundamental ways: (1) by allowing for the *consumption* of information by agents it permits a direct modelling of (indeterminate) state change in a logical framework, and (2) by admitting simply-typed λ-terms as data-objects, it permits the construction, transmission and application of (abstractions of) programs at run-time.

Much more dramatically, however, the framework can be seen as presenting higher-order (and if desired, constraint-enriched) versions of a variety of other asynchronous concurrent systems, including the asynchronous (''input guarded'') fragment of the (first-order) π-calculus, Hewitt's actors formalism, (abstract forms of) Gelernter's Linda, asynchronous assignment-based languages, and Petri nets. It can also be seen as smoothly layering around the functional programming style provided by the λ-calculus a minimal amount of extra logical machinery needed to obtain concurrency, synchronization and indeterminism declaratively. Additionally, there are remarkably simple and direct translations of the *untyped* λ-calculus into the higher-order linear cc (HLcc) programming paradigm.

We give (1) a simple operational semantics for HLcc, (2) establish several connections between proof-theory and operational semantics, (3) develop the notion of bisimulation for HLcc, along the lines of [Tho89], (4) establish that logical

*An abstract of this paper has been submitted for publication. Please send comments to the authors.

equivalence implies bisimilarity, (5) show how to obtain the effect of recursion for parametrized processes, (6) demonstrate embeddings of the (asynchronous) π-calculus and untyped λ-calculus into HLcc.

In summary, this paper draws on ideas from logic and proof theory to present a framework for the design and analysis of concurrent programming languages.

1 Introduction and overview

Our objective is to develop a logical foundation for concurrent programming that transparently integrates constraint programming, functional programming and process algebras.

Our starting point is the concurrent constraint (cc) programming paradigm (see, e.g., [Sar89,SR90,SRP91,JSS91]; we summarize here briefly. Imperative languages may be thought of as based on the *store-as-valuation* principle: a state of the computation is described by a valuation which assigns a unique value to each variable of interest. In *constraint-based* computation, this notion is replaced by that of *store-as-constraint*–-the store is seen to contain pieces of partial information (e.g., first-order formulas such as $X \geq Y + Z$) which specify a set of (possibly infinite) admissible values for the variables of interest. The notions of Write and Read underlying imperative programming languages no longer make sense, and are replaced here by the notions of *Tell* and *Ask*. Computation progresses via a monotonic accumulation of information, i.e., via Tell operations that add more constraints to the store; and its progress is monitored by Ask operations which check whether enough information has accumulated to entail a given constraint. *Concurrency* is accommodated naturally by conceiving of multiple agents interacting with a shared store via such ask and tell transactions. *Synchronization* is achieved by having an ask request *block* until there is enough information (if ever) to entail the given constraint, thereby allowing other processes an opportunity to unblock the computation by adding more information to the store. *Search non-determinism* is introduced by allowing computation to fork into two distinct branches, preserving only the (possibly disjunctive) information obtained on *both*. While simple, these ideas are being used for the design of both programming languages for distributed systems [SKL90] and very powerful languages for symbolic reasoning [HS91,HJ90].

From a computational perspective, the cc paradigm may be seen to generalize (and arose out of) the areas of concurrent logic programming and constraint logic programming. But where are the logical foundations of the (very operationally motivated) cc paradigm? In [LS91], we have shown that a logical semantics may be given using a *formula-as-agent* and *proof-as-computation* interpretation of intuitionistic logic.[1] The main underlying idea is to regard the steps of computation as expanding the frontier of a partial proof tree, whose leaves may be regarded as nonlogical axioms describing the residual computation, and whose root describes the initial state of the system.

[1]This is to be contrasted with the usual logic programming notion of *formula-as-goal* and *computation-as-proof-search*. Both of these notions can be combined fruitfully through the notion of *testing* [LS91].

Each step of the computation unfolds a (leaf) nonlogical axiom into perhaps several other nonlogical axioms and some concrete proof structure. A completed computation corresponds to a completed proof.

More generally, these ideas lead to the formula-as-agent interpretation of a larger class of logics. A subset of the formulas of the logic are viewed as computational agents. The left-rules in a sequent-style presentation of the logic are taken to specify the operational ‘‘heating” rules (in the style of [BB90]) which describe how complex agents may be decomposed into simpler agents, and the basic interactions betwen these agents. Thus, the operational derivability relation between configurations *qua* agents is related to the entailment relation between configurations *qua* logical formulas. In general, the logical counterpart of parallel composition is (seen to be) conjunction, of (ask-)prefixing is (a quantified version of) implication, and of hiding is existential quantification. Communication occurs (essentially) via constraint-imposition on first-order variables.

In this paper, we show that a very large area of concurrency is opened up to this perspective by moving to the setting of a higher-order version of Girard's *linear logic* [Gir87]. Linear logic may be seen as arising from classical logic by dropping certain structural rules that allow formulas to be arbitrarily duplicated and eliminated during the course of a derivation. This causes the familiar conjunction and disjunction operations to split into two: the so-called *additive* versions that copy formulas, and the *multiplicative* versions which do not. This gives formulas the nature of *resources* that must be accounted for carefully during proof (computation). The lost power of the structural rules can be recovered locally by means of the ‘‘modalities” ! and ?. In particular, the formula $!A$ is allowed to be duplicated any number of times, derelicted to obtain A, and dropped altogether.[2]

Informal review of HLcc. The linear logic *formulas-as-resources* viewpoint may be reinterpreted as *formulas-as-communications*. Multiplicative conjunction ($\otimes$) is viewed as parallel composition. An (unbanged) atomic formula is regarded as an indivisible piece of information, a *message* placed in a pool of messages and agents. (Since $A \otimes A$ is not linearly equivalent to A, multiplicities of messages matter.) A banged atomic formula is regarded as a permanant (non-consumable), globally valid piece of information, that is, a constraint (multiplicities of constraints do not matter, logically $!c$ is linearly equivalent to $!c \otimes !c$). Linear implication ($\multimap$) is viewed as a *method* for transforming a communication into a collection of agents. Intuitionistic implication provides a more restricted form of method which can only be invoked with constraints. Additive Conjunction (&) provides external choice –- the environment can cause the combination ($p \multimap A \,\&\, q \multimap B$) to reduce to A or to B by supplying either p or q respectively. Universal quantification provides a parameterization mechanism: the method $\forall X.(p\ X) \multimap (a\ X)$ can reduce to $a\ (Y\ 3)$ in the presence of the message $p\ (Y\ 3)$. Existential quantification provides *hiding* –- the ability to generate a new communication channel distinct from all others in the configuration. (Though it is

[2]See [Sce90,Lin92] for more motivation and tutorial introductions to linear logic.

outside the scope of this paper, a form of parallel search can be obtained through the additive disjunction $\oplus$.)

Linear communication. The presence of unbanged atomic formulas (''linear tells'') and linear implication (''linear asks'') now allows for a direct representation of indeterminacy and state change: the configuration $p \otimes (p \multimap r) \otimes (p \multimap q)$ can evolve to (entail) either $a \equiv r \otimes (p \multimap q)$ or $b \equiv q \otimes (p \multimap r)$, but *not* to their tensor product $a \otimes b$. In Hewitt's terminology, this kind of indeterminism is called *arrival indeterminism*, since it depends on the order in which messages (here, p) are delivered to methods (here, $p \multimap r$, and $p \multimap q$).

Indeed linear asks and tells allow for a direct, powerful form of communication. An example should suffice to convey the flavor. A natural way to model Actor-style languages within the cc framework (see [SKL90,KS90]) is to represent actor mailboxes as logical variables equated to *bags* of values. Sending a message, say 5, to an actor with associated channel X, then, corresponds to posting the constraint $5 \in X$. The actor suspends, asking whether there is a message on X; when activated, it pops a message off the bag and recurs with the tail of the bag.

However, such a translation does not work: two different actors may wish to communicate the *same* message to an actor — hence they would both post, say, the constraint $5 \in X$. Operationally we would like *both* messages to be delivered to (the actor) X; in (intuitionistic/classical) logic, however, this situation cannot be distinguished from that in which *only one* message was sent, since in these logics $a \wedge a$ is logically equivalent to a.

The problem does not arise in linear logic since multiplicities matter, and the above translation works correctly. Since a configuration is *itself* a multiset of agents, we need not use constraints such as $5 \in X$. Rather, just atomic formulas of the form $X : 5$ suffice — here '':'' is a binary predicate that has no special properties (built-in inference mechanism). An atom $(b \equiv) X : 5$ is, by convention, read as ''5 is a message on channel X''. An agent can read a message on the channel, and forward it on another channel Y by means of the method: $(a \equiv) \forall M. X : M \multimap Y : M$. As we shall see, the agent $a \otimes b$ can evolve into the agent $Y : 5$, as expected. Note that simultaneously *many* agents can send to the same channel, and many can read from the same channel (as with $p \multimap q$ and $p \multimap r$ above): the underlying notion of communication is *many-to-many* as in Linda-like languages, rather than many-to-one, as in Actor languages. Summarizing: Agents have access to a communication channel. On this channel, an agent can either post a *message* or a *method*. A method has the ability to suspend until a message matching a specified pattern arrives on the channel. Once activated, a method may, recursively, create new channels, and post messages and methods on new or old channels.

Note that *state* and *state change* is just the flip side of this ability to receive messages and consume them. The following example will demonstrate a number of programming techniques simultaneously. One may set up a `counter_creator` agent as follows:

$$
\begin{aligned}
\mathtt{A} \equiv \forall \mathtt{counter}.\big(&\mathtt{counter\_creator} : \mathtt{counter} \multimap \\
&\exists \mathtt{value}\big(\mathtt{value} : \mathtt{0} \otimes \\
&\quad \forall \mathtt{m}\big(\mathtt{counter} : \mathtt{m} \multimap \\
&\qquad \big(\forall \mathtt{val}\ \big(\mathtt{m} : \big(\mathtt{inc\ val}\big) \multimap \forall \mathtt{v}\ \big(\mathtt{value} : \mathtt{v} \multimap \mathtt{value} : \big(\mathtt{v} + \mathtt{val}\big)\big)\big) \\
&\qquad \ \&\forall \mathtt{val}\ \big(\mathtt{m} : \big(\mathtt{value\ val}\big) \multimap \forall \mathtt{v}\ \big(\mathtt{value} : \mathtt{v} \multimap \mathtt{value} : \mathtt{v} \otimes \mathtt{val} : \mathtt{v}\big)\big) \\
&\qquad \ \&\forall \mathtt{val}\ \big(\mathtt{m} : \big(\mathtt{dec\ val}\big) \multimap \forall \mathtt{v}\ \big(\mathtt{value} : \mathtt{v} \multimap \mathtt{value} : \big(\mathtt{v} - \mathtt{val}\big)\big)\big)\big)\big)\big)\big).
\end{aligned}
$$

Informally, such an agent responds to a message `counter` sent on the `counter_creator` channel as follows: it sets up a new local communication channel, `value`, initializes it with the message 0, and installs a method on the *input* message channel, which suspends until a message `m` arises, and then suspends until `m` (which is itself a channel), contains a parametrized `inc` or `dec` or `val` message. In the `inc` case, it reads the value `M` on its private channel (thereby deleting the value), and sends the message $\mathtt{M} + \mathtt{val}$ on `value`. Similarly, in the other cases.[3] Note: an assignable memory location is merely a channel with a single outstanding message; incoming channels are just like any other channels . . . it is possible to both read and write (post messages and methods on) them; the effect of lexically scoped, shared, encapsulated, concurrently updatable state variables is created through the alternation of universal and existential quantifiers; the usual logic programming ideas of ‘‘incomplete” messages are still available; and in the spirit of Hewitt’s slogan, control-structures emerge from patterns of communications.

Essentially the system we have presened hitherto (*without* any built-in constraint system) is the basis of the programming language Linear Janus [Tse92]. Linear Janus is a cleaned up and considerably simplified version of Lucy, the ‘‘missing link” between concurrent logic programming and actor languages described in [KS90]. As such it is in a long tradition of languages arising from the connection between concurrent logic programming languages and Actor languages. From a computational perspective, the system is very close to the asynchronous fragment of the π-calculus [MPW89], and one can also show that it is possible to embed the untyped λ-calculus into it through a very simple translation. (The translation effectively shows how to interprete the λ-calculus in a very simple fragment of first-order linear or intuitionistic logic.)

The higher-order system. For all its power, however, it is not possible in the first-order Lcc to pass *processes* as parameters. At best one can pass names of (channels to) processes. One cannot obtain unknown processes from communication channels, destructure them, combine them, run them, apply them to other unknown processes or otherwise treat them as first-class citizens. This motivates our final move, then, to higher-order linear logic.

The higher-order version of linear logic we consider is a straightforward adaption of a presentation of higher-order intuitionistic logic using Church’s simple theory of types (see Section 2.1 for details). Intuitively, one may understand such a logic as obtained from first-order linear logic by replacing first-order terms with simply typed λ-terms.

[3] As described here, all these actions will happen *exactly once*. We will describe below how recursion and repetition are programmable in the higher-order language.

One assumes that the set of types comes equipped with a type o for propositions, and a type ι for individuals; the linear connectives can then all be introduced as logical constants (functions) over o. (Similarly, for each type α, the existential quantifier $\Sigma^{\alpha}_{(\alpha\to o)\to o}$ and the universal quantifier $\Pi^{\alpha}_{(\alpha\to o)\to o}$ are also provided. This allows, then, the expression of terms such as

$$(t \equiv)\lambda a_\iota \lambda b_\iota \lambda c_\iota . \Pi \lambda y_o (a : y \multimap \Pi \lambda z_o (b : z \multimap c : (y \otimes z)))$$

which can be viewed as an abstraction which when applied to three communication channels a, b and c yields a process that will accept an arbitrary process y on a and z on b, and send their parallel composition $y \otimes z$ on to c. Of course t is itself just a term and can be passed around in messages from agent to agent; in that case the receiving agent would need to use a universal quantifier at a functional (higher-order) type to receive the message.

In general, higher-order quantification is an exceedingly powerful mechanism, and developing practical programming languages with such features is a delicate task. The main restriction placed on these quantifiers in HLcc involves limitations on the use of Π, restricting occurrences of Π to immediately precede $\multimap$, with further restrictions which effectively ensure that it is not possible to universally quantify on the first argument (the ‘‘channel”) in a message $x : t$, and no universally quantified unguarded agents will ever appear during execution. (An agent of the form $\Pi : \lambda\, x.x$ is an agent which can perform *any* task, provide *any* information to other agents, etc.; such an agent is computationally disastrous.)

For pragmatic reasons, it may be desirable to impose syntactic restrictions which disallow existential quantification at functional types and/or to disallow in the head logical constants or λ-abstractions or multiple occurrences of variables (see, e.g., [Wad91]). This can obviate the need for higher-order unification or matching at run-time; however, the higher-order structure left in the programming language is still adequate to construct, assemble, communicate and use higher-order expressions.

Related work. The desire for a clean, higher-order, indeterminate language framework is a long-standing one, both within theoretical and applied areas in concurrency.

Perhaps the most well-developed body of work related to this paper is that on the π-calculus [MPW89,Mil90,Mil91]. A primary reason for the interestingness of constraint languages is the notion of constraint-based communication, which subsumes the notion of mobility of communication channels. It seemed clear to us, therefore, that there should be ‘‘simple” variants of cc languages which exhibit the computational characteristics of the π-calculus. Just so. If certain aspects of the π-calculus (e.g., unrestricted sums, ‘‘tell”-prefixing) are ignored, the π-calculus *is* almost identical to Linear Janus. However, the logical perspective underlying the cc languages has a lot to offer to work on process algebras: it has provided a ready guide to the introduction into concurrent programming languages of complex data-structures and systems of partial information, search non-determinism and first-order and higher-order quantifiers. Additionally, logical equivalence emerges as even a finer congruence on processes than bisimulation (see Section 2.5).

Other related work includes that of [HT92], who presents an asynchronous, first-order fragment of the π-calculus closely related to Linear Janus, and studies its semantics. No logical interpretation of the calculus is given however. [Tho89] presents a higher-order version of CCS, including a notion of higher-order bisimulation. In contrast to the system of this paper, CHOCS uses dynamic binding, does not allow the transmission of *abstractions* of programs (indeed, it has no notion of application –- it is really second-order rather than higher, i.e., ω-order). [Bou89] presents the (asynchronous) γ-calculus, which abdandons CCS' parallel composition in favor of two operators –- *interleaving* and *cooperation*. The proof-theoretic analogue of cooperation is unclear to us. We do not know whether either of HLcc and the γ-calculus can be embedded in the other. [Nie89] presents a system extending the typed λ-calculus with CCS-like processes; the focus of the work is in using types to record the possible communications of a process. [dBKPR91] present a general class of (first-order) asynchronous programming languages; these languages should be describable within first-order Lcc.

There has been considerable work on the integration of logic programming and functional programming languages [JP91,Lin85,DL86]. However, before the advent of linear logic, it has not been possible to build such languages centrally on indeterministic concurrency. The paradigm proposed here –- build a little layer of sub-structural logic and constraints around the simply-typed lambda-calculus –- seems to be very simple and rich. Indeed, it should be possible to design powerful asynchronous extensions of languages like ML based on these ideas.

A number of proposals have been made for programming on top of linear logic. [AP90] have explored very novel concurrent constructs that communicate by instantiating the endsequent, which is left unspecified in the original goal. The expression of concurrency in HLcc is more direct. [HP91] adapt the idea of ''uniform proofs", underlying a view of logic programming, to the setting of linear logic. [HM91] extend λ-Prolog to include a restricted class of linear features. The main computational mechanism in both cases is backward-chaining on Horn clauses; it is not clear to us how ''ask synchronization" can be represented in this setting, and how indeterminate computations can be reflected into the logic. The relationship between the two approaches should be examined further.

[Mil92] independently discusses connections between the π-calculus and linear logic; parallel composition is mapped to multiplicative *dis*junction ($\wp$), and hiding to universal quantification. Non-logical constants are used to represent prefixing, nondeterministic choice, and ''match" guards. A ''dual" translation is mentioned, and this is very close to the translation we give in Section 3. Indeed, despite some technical differences it seems clear that this work shares a common perspective with [Mil92].

[Mes90,Mes92] have recently introduced a general theory of concurrent objects based on concurrent rewriting, and a specific language, Maude. Although semantically based on completely different logics (rewriting logic versus linear logic), there is a close connection between first-order, constraint-less HLcc and Maude's system modules. HLcc generalizes this nearly-common sublanguage by adding a constraint system and moving to higher-order. Maude generalizes this sublanguage by adding certain kinds of parametricity and object-oriented features such as inheritance.

Rest of this paper. The next section sketches out the formal system –- the logic underlying HLcc, the language HLcc (with its syntactic restrictions) and a transition system underlying its operational semantics. The close connection between the operational semantics and provability in linear logic is discussed. The operational semantics can be viewed as specifying an incomplete theorem-prover for a fragment of linear logic, and the theorems in that section show precisely how to extend the operational semantics to recover completeness. We also discuss a coarser notion of equivalence motivated from the formula-as-agent viewpoint. Roughly, two formulas are regarded as indistinguishable if they can engage in the same (potentially infinite) tree of basic interactions (the asking and telling linear atoms and constraints) with their environments. Some care is necessary, however, to define the notion of bisimulation ($\simeq$) in this asynchronous, logical setting in a simple and smooth manner, and to ensure that two agents are not distinguished because they are communicating programs that, while inequivalent logically, are bisimilar. Because of the power of the language, it is not possible to provide a complete axiomitization; we show however, that bisimulation respects logical equivalence.

Next we discuss how to obtain the effect of recursion via concurrency and communication[4]; in particular, we show that it is possible to specify a fixed-point combinator **fix** at every type $\alpha \to o$ which satisfies the property that $\mathbf{fix}\ f \simeq f(\mathbf{fix}\ f)$. This can be used to get the effect of recursion, and the ‘‘replication” operator of [Mil90].

Section 3 briefly compares the HLcc framework with other asynchronous computing frameworks, emphasizing the embeddings of the π-calculus and λ-calculus in HLcc.

2 The basic paradigm

2.1 The basic logic, HLL.

The set of types, ranged over by α is taken to consist of at least the base types o and i, and is closed under function-space construction:

$$\alpha ::= o \mid \iota \mid \alpha \to \alpha \tag{1}$$

In a concrete language, other basic types (e.g., `real`, `int`) may be given; for the present paper we take ι as representative.

Assume given denumerably many variables and constants at each type, and the logical constants $\mathbf{1}_o$, $!_{o \to o}$, $\&, \otimes, \multimap_{o \to o \to o}$, and for each type α, the constants $\Sigma^\alpha, \Pi^\alpha$ of type $(\alpha \to o) \to o$. (Constants other than these will also be called *parameters*.) We shall adopt the usual syntactic convention of writing ! prefix, $\&, \otimes, \multimap$ infix, and $\Sigma^\alpha \lambda x_\alpha.M$ as $\exists x_\alpha.M$ and $\Pi^\alpha \lambda x_\alpha.M$ as $\forall x_\alpha.M$. As usual, the terms are given by:

$$t ::= v_\alpha \mid k_\alpha \mid (\lambda x_\alpha.t_\beta)_{\alpha\to\beta} \mid (t_{\alpha\to\beta} t_\alpha)_\beta \tag{2}$$

[4]Note that HLcc has no explicit recursion, and the underlying λ-calculus is typed, so strongly terminating.

We define two terms s_α and t_α to be λ-*equivalent*, and write $s_\alpha =_\lambda t_\alpha$ if they can be shown equivalent using (α, β, η) rules. For background on such a treatment of terms, the reader may refer to [NM90]. As shown there, under such conversion rules a λ-term M has a unique normal form; we shall denote it by $\rho(M)$.

A *sequent* is of the form $\Gamma \vdash \Delta$ where Γ, Δ are multisets of formulas (terms of type o). The inference figures for HLL are the expected ones for a higher-order logic in the style of Church; they are given in full in Appendix A.

A theorem fundamental to the study of logic is the cut-elimination theorem, which states that whatever can be proven in a system, such as HLL, can be proven without the use of the cut rule. This theorem is used to prove consistency and in first order and propositional system yields a very useful subformula property. Cut-elimination is often demonstrated by providing a terminating cut-elimination procedure which slowly eliminates cuts from a given proof.

Theorem 2.1 (Cut-elimination) *All instances of the (Cut) rule can be eliminated from a proof of a sequent* $\Gamma \vdash \Delta$ *in HLL.*

This theorem may be proven by providing equivalent systems with extra rules, and showing that the extra rules and cut can be eliminated simultaneously. The extra rules stand for several applications of other rules, and serve to simplify the proof of termination of cut-elimination.

In what follows, we will assume that the underlying logic has been augmented by a user-specified constraint system $\mathcal{C}$. For our present purposes, it suffices to consider that the user has provided certain ‘‘built-in" functions, and non-logical axioms of the form:

$$!c_1, \ldots, !c_n \vdash !c$$

where c is an atom using the given built-in functions. Henceforth, when we talk of linear derivability, $\vdash$, we shall assume that some such constraint system $\mathcal{C}$ is already built into the system.[5]

2.2 HLcc

A certain sub-class of HLL formulas are isolated as HLcc *processes*. In order to more crisply state the syntactic restrictions on HLcc, we make a few modifications. First, we introduce the types c (for constraints) and m (for methods, i.e., implications). Second, we eliminate the constants $!, \Pi$ and $\multimap$ from our vocabulary, and instead introduce two new ‘‘constants", $\mathbf{A}$ and $\mathbf{B}h$.[6] The combinator $\mathbf{A}_{c \to o \to m}$ is a simple abbreviation: if the constants $!_{c \to o}$ and $\multimap_{o \to o \to m}$ are available, it can be understood

[5]Of course, a particular HLcc language may have a vacuous constraint system –- Linear Janus is an example. This need not cripple the language –- linear ask and tell operations are extremely powerful in their own right.

[6]In concrete syntax, we allow the user to use $!, \Pi$ and $\multimap$; we merely require that the program be λ-equivalent to a program in the restricted syntax. Also, we will sometimes write m instead of $\&^1 m$, in a context where an o-term is expected.

as $\mathbf{A} = \lambda d_c.\lambda a_o.!d \multimap a$. For $h_{\alpha_1 \to \ldots \to \alpha_n \to o}$, a term of certain shape (see below) we introduce a term $(\mathbf{B}h)_{(\alpha_1 \to \ldots \to \alpha_n \to o) \to m}$. Intuitively, $\mathbf{B}ha$ stands for

$$\forall {x^1}_{\alpha_1} \ldots {x^n}_{\alpha_n}.(hx^1 \ldots x^n) \multimap (a\ x^1\ \ldots\ x^n)$$

We require that (1) h be in λ-normal form, say $\lambda x_1.\ldots.x_n.k\ t_1\ \ldots\ t_m$, (2) that k be a parameter, and (3) that each x_i have at least one *strict* occurrence in $k\ t_1\ \ldots\ t_n$ (i.e., occurrences which ''cannot disappear'' from h by β reduction). This is ensured by requiring that at least one occurrence of each variable x_i in $k\ t_1\ \ldots\ t_n$ is not in the application-scope of a variable other than the x_i; also specific rules are given for each constant that can appear in h (e.g., cons, if-then-else).

We note another important restriction on h. In order to operationally regard $\exists X\ A$ as treating X as a ''private'' channel in A, it is necessary to curtail the power of universal quantification. The simplest way to achieve this is to require: (4) if k is the constant '':'', then t_1 is a variable distinct from $x_1, \ldots, x_n$. This ensures that no method can universally quantify on all communication channels: agent such as $\mathbf{B}(\lambda x \lambda y\ (x : y))a$, which logically can accept *any* message y on *any* communication channel x (even one newly created using existential quantification) are banned. While this solution is not general, it is very adequate in practice.

HLcc processes are now merely terms of type o. For simplicity we assume given a family of k-ary &-operations, one for each $k > 0$, rather than a single binary &-operation. Also, we assume that existential quantification is provided only at type ι. The built-in ''logical'' constants in HLcc then are: [7]

$\mathbf{1}_o$	– Nil	
$\otimes_{o \to o \to o}$	– Parallel composition	
$\Sigma^{\iota}_{(\iota \to o) \to o}$	– Local channels	(3)
${\&^k}_{m \to \ldots \to m \to o}$	– Guarded choice	
$(\mathbf{B}h_{\alpha_1 \to \ldots \to \alpha_n \to o})_{(\alpha_1 \to \ldots \to \alpha_n \to o) \to m}$	– Linear Ask method	
$\mathbf{A}_{c \to o \to m}$	– Ask method	

The set of HLcc terms $\mathcal{H}$ is then the set of lambda-terms generated from these logical constants with the use of variables, parameters, and abstraction and application.

2.3 Operational semantics.

As configurations we take multisets of processes, ranged over by Γ, Δ. For Γ such a multiset, we let $\sigma(\Gamma)$ stand for the submultiset of formulas in Γ of the form $!d$, for d a constraint. Also, for V a set of variables, and c a constraint, we use the notation $\delta V.c$ to mean the existential closure of c on all variables other than V.

We take the transition relation $\longrightarrow$ to be the smallest binary relation on configurations closed under the following five simple and intuitive inference rules. First,

[7] Strictly speaking, existential quantification and tensor are also provided for constraints, that is, we have another constant $\Sigma^c_\iota : (\iota \to c) \to c$, and $\otimes^c_{c \to c \to c}$. However, we shall not be pedantic, and shall usually avoid distinguishing between Σ^ι and $(\Sigma^c)^\iota$, and between $\otimes$ and $\otimes^c$.

we assume that λ-equivalence for typed terms is transparently built into the reduction mechanism:

$$\frac{F =_\lambda F' \;\; (\Gamma, F') \longrightarrow (\Delta, G') \;\; G' = \lambda G}{(\Gamma, F) \longrightarrow (\Delta, G)} \tag{4}$$

Second, **1** disappears silently:

$$\frac{\Gamma \longrightarrow \Delta}{(\Gamma, 1) \longrightarrow \Delta} \tag{5}$$

The top-level multiset of agents represents their parallel composition:

$$\frac{(\Gamma, F, G) \longrightarrow \Delta}{(\Gamma, F \otimes G) \longrightarrow \Delta} \tag{6}$$

The execution of a Σ term requires the creation of a new communication channel:

$$\frac{(\Gamma, Fy) \longrightarrow \Delta \;\; y_\alpha \notin \mathbf{var}(\Gamma, \Delta)}{(\Gamma, \Sigma_\alpha F) \longrightarrow \Delta} \tag{7}$$

A Linear Ask can consume a message A provided that it is possible to find values for its universally quantified variables such that A matches its template:

$$\frac{\sigma(\Gamma), A \vdash ht_1 \ldots t_n}{(\Gamma, A, \&^j m_1 \ldots (\mathbf{B}hF) \ldots m_j) \longrightarrow (\Gamma, Ft_1 \ldots t_n)} \tag{8}$$

An (Intuitionistic) Ask operation checks to see if the current store is strong enough to entail the given constraint:

$$\frac{\sigma(\Gamma) \vdash !c}{(\Gamma, \&^j m_1 \ldots (\mathbf{A}cF) \ldots m_j) \longrightarrow (\Gamma, F)} \tag{9}$$

This completes the definition of the operational semantics of HLcc. Below we point out interesting restricted sublanguages of HLcc, and then we discuss connections between HLcc and the logic HLL. The logical connections discussed for full HLcc also apply to the more restricted cases.

Restricted HLcc languages. Propositional Lcc is obtained by admitting, in addition to the logical constants, only constants at base types. First-order Lcc is obtained by allowing existential and universal quantification only at base types.

For pragmatic reasons, it may be desirable to not allow existential quantification at functional types. In such cases, there can be no possibility of expressing constraint-solving at higher-types (higher-order unification is not needed). In addition, the user may not be allowed to have λ-abstractions in the heads of clauses. This would rule out the need for higher-order matching as well: however, the higher-order structure left in the programming language is still adequate to construct, assemble, communicate and use higher-order expressions. (One cannot, however, decompose higher-order expressions, or incrementally construct them using constraints, as can be done in λ-Prolog.)

2.4 Connection between operational and logical interpretations.

There is a tight connection between the operational and logical properties of HLcc. Essentially, both soundness and a form of completeness can be established. (Details of the following section may be found in the Appendix.)

This operational semantics given above is sound with respect to HLL, as may be seen by induction on the operational derivation. Soundness effectively means that one will never get a wrong answer from the operational semantics.

Theorem 2.2 (Soundness) $\Gamma \vdash \Delta$ *whenever* $\Gamma \longrightarrow \Gamma'$ *and* $\Gamma' \vdash \Delta$.

Cut-elimination in HLL is not sufficient to guarantee the *subformula property*: any formula appearing in a cut-free proof must also appear in the conclusion of the proof. (This property is violated by the β reductions allowed in HLL.) If it held, the subformula property could be used to show that if the formulas in the conclusion satisfied some property closed under subformulas, then so would formulas in any cut-free proof. Nevertheless, we can show the following preservation property:

Theorem 2.3 (Well-Formed Formulas) *If F, G are well-formed* HLcc *processes, then so is any formula appearing in a proof of* $F \vdash G$.

Though the above operational semantics is incomplete in general, a very useful form of completeness can be shown to hold. We define a *forward* proof to be a proof where there are no applications of right rules below any applications of left rules. For the purposes of this definition, we consider identity to be a right rule. In other words, a forward proof is one where all the ''action" happens on the left of a sequent until the very end, where the right hand side is unwound to the axioms. In the following, let G be a meta-variable ranging over *goals*, that is, possibly existentially quantified tensor conjunctions of atomic formulas or constraints.

Lemma 2.4 (Forward Proofs) *If there is a proof of* $\Gamma \vdash G \otimes \top$ *in HLL, where* Γ *is an* HLcc *configuration, and G a goal, then there is a forward cut-free proof of this sequent.*

We define a *sequentialized* proof to be a proof where there are no applications of left rules in the left hand proof branch of an ($\multimap$ Left) inference. For the purposes of this definition, we consider identity to be a right rule. This property will be of interest mainly for proofs already heavily normalized.

Lemma 2.5 (Sequential Proofs) *If there is a proof of* $\Gamma \vdash G \otimes \top$ *in HLL, for* Γ *an* HLcc *configuration, and G a goal, then there is a sequentialized, forward cut-free proof of this sequent.*

Given the existence of sequentialized, forward, cut-free proofs for any provable sequent, we can show:

Theorem 2.6 (Completeness) $\Gamma \vdash G \otimes \top$ *implies that for some* $\Delta, \Gamma \longrightarrow^\star (\Delta, A_1, \ldots, A_n)$, *and* $A_1, \ldots, A_n \vdash G$.

Finally, we show how $\longrightarrow$ can be extended in a very simple way to obtain a complete proof procedure for the HLcc fragment. We define the relation $\mid\!\leadsto$ as the least relation containing the right hand side inference rules and axioms of $\vdash$, and satisfying the ''cut'' inference rule:

$$\frac{\Gamma \longrightarrow \Delta \quad \Delta \mid\!\leadsto \Theta}{\Gamma \mid\!\leadsto \Theta} \tag{10}$$

Theorem 2.7 ($\vdash = \ \mid\!\leadsto$) *If Γ and Δ satisfy the syntactic restrictions on* HLcc *programs, then $\Gamma \vdash \Delta$ is provable in HLL iff $\Gamma \mid\!\leadsto \Delta$.*

2.5 Bisimulation semantics

In the previous section we presented logical equivalence of processes. This is quite a powerful notion –- indeed it already gives us a number of equational laws to work with (see Table 1), laws that are established in the setting of proces algebras via operational arguments.

However, logical equivalence is not able to capture many essential aspects of HLcc *qua* concurrent programming language. For instance, we are unable to use the underlying logic to talk about liveness properties, or to reason about non-terminating computations. Moreover, if some of the logical features are used in a limited way (e.g., universal quantified variables are prevented from occurring in certain places in certain atoms), then operational equivalence may not imply logical equivalence. For, fewer contexts are available to operationally distinguish between processes, hence operational equivalence is coarser.[8]

Therefore, it seems appropriate to view HLcc as a formalism for concurrent computation, and analyze it using the well-developed set of techniques from process algebra. A number of equivalences have by now been studied for concurrent systems — here we focus on one of the finest such equivalences, bisimulation.

We move away from the traditional SOS-style of giving semantics to concurrent languages. In particular that style is too tied to the individual steps made by a process, and to labels carrying information at each step. In logical asynchronous computation of the kind described here, it is much more natural to allow pieces of information to accumulate in the store and then be used as appropriate. This breaks a fundamental conflict in the granularity of transitions and the granularity of constraint imposition. In order to define higher-order bisimulation, we need to extend a binary relation on processes to a binary relation defined at all types.

Definition 2.1 Let R be a binary relation on $\mathcal{H}$. Then $[R]$ is the least relation on $\mathcal{H}$ extending $R \cup =_\lambda$ and closed under the inference rules:

- $(\lambda x_\alpha\ M_\beta)\ [R]\ (\lambda x_\alpha\ N_\beta)$ if $M_\beta\ [R]\ N_\beta$,

[8] Indeed, this can already be seen in the (intuitionistic) determinate cc case. The closure operator semantics given in [SRP91] does not give us all I.L. connectives: certain first-order formulas are operationally indistinguishable while being logically distinct.

- $(M_{\alpha\to\beta} N_\alpha)\ [R]\ (M'_{\alpha\to\beta} N'_\alpha)$ if $M_{\alpha\to\beta}\ [R]\ M'_{\alpha\to\beta}$ and $N_\alpha\ [R]\ N'_\alpha$.

$\square$

Definition 2.2 Let $\Longrightarrow$ be the least reflexive and transitive relation containing $\longrightarrow$ and closed under the rules:

- $(\Gamma, F \otimes G) \Longrightarrow (\Gamma, F, G)$,
- $(\Gamma, \Sigma_\alpha F) \Longrightarrow (\Gamma, FY_\alpha)$, provided that Y_α is not free in Γ, F,
- $(\Gamma, F) \Longrightarrow (\Gamma, G)$, if $F =_\lambda G$.

$\square$

Corresponding to the notion of weak bisimulation in CCS, we have:

Definition 2.3 (Reactive equivalence) Reactive equivalence on the variables V (written $\simeq_V$) is the largest binary, symmetric relationship over configurations such that $\Gamma \simeq_V \Delta$ implies

1. Whenever $\Gamma \Longrightarrow \Gamma'$ and $\sigma(\Gamma') \vdash !c$, where $\mathbf{var}(c) \subseteq V$, there is a Δ' such that $\Delta \Longrightarrow \Delta'$, $\sigma(\Delta') \vdash !c$, and $\Gamma' \simeq_V \Delta'$.
2. Whenever $\Gamma \Longrightarrow (\Gamma', kt_1 \dots t_n)$, and either $k \not\equiv :$ or $\mathbf{var}(t_1) \subseteq V$, there is a Δ', B such that $\Delta \Longrightarrow (\Delta', B)$, and, $\Gamma' \simeq_W \Delta'$ and $A\ [\simeq_W]\ B$, where $W = V \cup \mathbf{var}(\rho(A)) \cup \mathbf{var}(\rho(B))$
3. For all A with $\mathbf{fvar}(A) \subseteq V$, $(\Gamma, A) \simeq_V (\Delta, A)$.
4. For all c with $\mathbf{fvar}(c) \subseteq V$, $(\Gamma, !c) \simeq_V (\Delta, !c)$.
5. $\Gamma \longrightarrow$ iff $\Delta \longrightarrow$ (where by $\Gamma \longrightarrow$ we mean that there is a configuration Γ' such that $\Gamma \longrightarrow \Gamma'$).

$\square$

We will write $P \simeq Q$ (for processes P, Q) to mean $P \simeq_{\mathbf{var}(P) \cup \mathbf{var}(Q)} Q$.

Theorem 2.8 *Reactive equivalence is a congruence for* HLcc. *That is.*

- $P_1 \otimes P_2 \simeq Q_1 \otimes Q_2$,
- $\exists x.P_1 \simeq \exists x.Q_1$,
- $\&^1 \mathbf{A} c P_1 \simeq \& \mathbf{A} d Q_1$,
- $\&^1 \mathbf{B} h(\lambda x_1 \dots \lambda x_n.P_1) \simeq \& \mathbf{B} h(\lambda x_1 \dots \lambda x_n.Q_1)$,
- $\&^k m_1 \dots m_k \simeq \&^k m'_1 \dots m'_k$

(1)	$\&^k m_1 \ldots m\ m' \ldots m_k \simeq \&^k m_1 \ldots m'\ m \ldots m_k$
(2)	$\&^k m_1 \ldots m\ m \ldots m_k \simeq \&^{k-1} m_1 \ldots m \ldots m_k$
(3)	$F_1 \otimes F_2 \simeq F_2 \otimes F_1$
(4)	$(F_1 \otimes F_2) \otimes F_3 \simeq F_1 \otimes (F_2 \otimes F_3)$
(5)	$F \otimes \mathbf{1} \simeq F$
(6)	$\exists x \exists y F \simeq \exists y \exists x F$
(7)	$\exists x \exists x.F \simeq \exists x.F$
(8)	$(\exists x F_1) \otimes F_2 \simeq \exists x (F_1 \otimes F_2) \quad x \notin fv(q)$
(9)	$\exists x.F \simeq F \quad x \notin fv(F)$
(10)	$F_1 \simeq F_2 \quad (F =_\lambda G)$

Table 1: Laws for reactive congruence

whenever $P_i \simeq Q_i$, *for* $i = 1, \ldots k$, $\vdash !c \multimap\!\!\circ\ !d$, *and* $\&^1 m_i \simeq \&^1 m_i'$, *for* $i = 1, \ldots, k$.

Thus, synchronization trees give rise to a model for (this fragment of) linear logic.

Theorem 2.9 $F \simeq G$ *whenever* $\vdash F \multimap\!\!\circ\ G$.

The proof is pretty direct, using the cut-elimination theorem. The theorem immediately leads to the various laws in Table 1

2.6 Generating recursive behaviors

Parametrized processes can be defined recursively in HLcc –- for any type α, a fixed-point operator can be defined at type $\beta = \alpha \rightarrow o$:

$$\begin{aligned} \mathbf{fix} &\stackrel{d}{=} \lambda p_{\beta\rightarrow\beta} \lambda x_\alpha (\exists b_\iota.(\mathbf{M} p b x) \otimes (b : (\mathbf{M} p)) \\ \mathbf{M} &\stackrel{d}{=} \lambda p_{\beta\rightarrow\beta} \lambda b_\iota \lambda x_\alpha . \forall z_{\iota\rightarrow\beta}. b : z \multimap p(\lambda x_\alpha \exists b_\iota\ (z b x \otimes b : z)) x \end{aligned} \tag{11}$$

Proposition 2.10 *Let β be the type $\alpha \rightarrow o$. Then* $\mathbf{fix}_{(\beta\rightarrow\beta)\rightarrow\beta)}\ p\ t \simeq p(\mathbf{fix}\ p)t$ *for any terms* $p_{\beta\rightarrow\beta}, t_\alpha$.

Example 2.1 It is now clear how to program a counter that can accept more than one message:

$$\begin{aligned} &\mathtt{c} \equiv \lambda\mathtt{counter}\exists\mathtt{value}\big(\mathtt{value}:\mathtt{0} \\ &\qquad\qquad \otimes\ \mathtt{rep\_method\ counter} \\ &\qquad\qquad\quad \lambda\mathtt{m}\ \big(\forall\mathtt{val}\ \big(\mathtt{m}:\big(\mathtt{inc\ val}\big)\multimap\forall\mathtt{v}\ \big(\mathtt{value}:\mathtt{v}\multimap\mathtt{value}:\big(\mathtt{v}+\mathtt{val}\big)\big)\big) \\ &\qquad\qquad\qquad \&\forall\mathtt{val}\ \big(\mathtt{m}:\big(\mathtt{value\ val}\big)\multimap\forall\mathtt{v}\ \big(\mathtt{value}:\mathtt{v}\multimap\mathtt{value}:\mathtt{v}\otimes\mathtt{val}:\mathtt{v}\big)\big) \\ &\qquad\qquad\qquad \&\forall\mathtt{val}\ \big(\mathtt{m}:\big(\mathtt{dec\ val}\big)\multimap\forall\mathtt{v}\ \big(\mathtt{value}:\mathtt{v}\multimap\mathtt{value}:\big(\mathtt{v}-\mathtt{val}\big)\big)\big)\big)\big). \\ &\mathtt{rep\_method} \equiv \big(\mathbf{fix}\ \big(\lambda\mathtt{p}\lambda\mathtt{channel}\lambda\mathtt{body}\forall\mathtt{m}\ \big(\mathtt{channel}:\mathtt{m}\big)\multimap\big(\mathtt{p\ channel\ body}\big)\otimes\big(\mathtt{body\ m}\big)\big)\big). \end{aligned}$$

□

3 Connection with other paradigms

Petri nets. HLL provides simple encodings of Petri net reachability. Consider the following formula: $x : (\{a, c\} \bigcup S) \multimap x : (S \bigcup b)$. This may be seen as an encoding of a Petri net transition which takes a token from place a and another from place c and replaces them with one token on place b. This encoding enforces a kind of interleaving model of concurrency for the Petri net. This and other connections between linear logic and Petri nets have been well-studied [Asp87,GG89,MOM89,AFG90,GG90], and extended in various ways to cover other models of concurrency [Laf90,Pra92]. In fact, the Lcc languages can be seen as a ‘‘first-order” version of Petri nets augmented with constraints, and HLcc as a ‘‘Higher-order” version.

In the above encoding, the atoms a, b, and c contain no internal structure. However, one may consider ‘‘colored tokens” which contain some fixed amount of internal information, or even more expressive ‘‘first order tokens” which may carry arbitrary amounts of information. Such tokens may be encoded easily in HLcc.

Actors and Linda. As discussed in the introduction, the basic actor model corresponds to the first-order, constraint-less version of HLcc, with ‘‘:” as the only atomic formula. Indeed, the system is more powerful in that it allows for the dynamic installation of methods on *pre-existing* mailboxes — hence some restriction in expressiveness is needed to get actors exactly. Since the basic communication mechanism in Lcc can be seen as ‘‘many-to-many” rather than ‘‘many-to-one,” HLcc also immediately provides a constraint-based, ‘‘higher-order” version of the Linda computation model. Indeed, *each channel* can be thought of as a Linda tuple-space –- the ‘‘in” operation corresponds to posting a message, and the ‘‘out” operation corresponds to reading a message.

π-calculus. Table 2 demonstrates, how the ‘‘asynchronous” subset of the π-calculus [MPW89] may be embedded into (first-order) Lcc. Channels are treated as first-order variables, and the rest of the translation follows naturally. The seemingly mysterious notions of ‘‘extrusion” and ‘‘intrusion” of scope are seen as well-known operational manifestations of the logical properties of the first-order quantifiers. It is possible to translate the entire language (i.e., tell-prefixing as well) –- at the cost of implementing a synchronous π-calculus transmission with a few-step transmit/acknowledge protocol.

Lambda calculus. Perhaps the most remarkable connection is with the lambda calculus: there are at least two direct and simple translation of the (untyped) lambda-calculus into Lcc languages. (We focus here on the lazy version of the lambda-calculus [AO89].) The first is an analog of the translation to the π-calculus in [Mil90]; indeed it can be

$$
\begin{array}{rcl}
[0] & = & 1 \\
[(y)P] & = & \exists y_\iota.[P] \\
[\bar{x}y.0] & = & x : y \\
[x(y).P] & = & \forall y_\iota.x : y \multimap [P] \\
[P \mid Q] & = & [P] \otimes [Q] \\
[[x = y]P] & = & !(x = y) \multimap [P] \\
[A(X_1, \ldots, X_n)] & = & A : X_1 \ldots X_n \\
[A(X_1, \ldots, X_n) = P] & = & \mathbf{fix}(\lambda r_o\ (\forall X_{1\,\iota}\ \ldots \forall X_{n\,\iota} A : X_1 \ldots X_n \multimap r \otimes [P]))
\end{array}
$$

Choice non-determinism (''$+$'') is not handled; a guarded version can be translated into methods, as for the cc languages below.
Note that the structural rules of [Mil90] are immediately verified. That is, $P \equiv Q$ implies $[P] \circ\!\!-\!\!\circ [Q]$.

Table 2: Translation of the input-guarded π-calculus to Lcc

seen as providing a direct embedding of the λ-calculus in a very simple fragment of first-order intuitionistic logic.

Direct translation. The first translation can be seen as the Lcc analogue of the translation in [Mil90]. Intuitively, an application is treated as a parallel composition, and the operand and the argument are treated as separate processes communicating on a channel hidden from the outside world. A λ-abstraction is seen as a server waiting for a message to come down its specific communication channel (such messages are generated by application). β-reduction is reflected by the universal instantiation and (linear) *modus ponens* underlying the operational semantics of the Lcc languages. For M a lambda-term we define $\langle M \rangle_{\iota \to o}$ by:[9]

$$
\begin{array}{l}
\langle x \rangle = \lambda z_\iota\ : x\ z \\
\langle \lambda x\ M \rangle = \lambda z_\iota\ \forall x_\iota \forall y_\iota\ : z\ x\ y \multimap \langle M \rangle\ y \\
\langle MN \rangle = \lambda z_\iota\ \exists x_\iota \exists y_\iota\ (\langle M \rangle\ x \otimes : x\ y\ z \otimes (\mathbf{fix}\ \lambda p_o \forall s_\iota\ (: y\ s \multimap p \otimes \langle N \rangle\ s))
\end{array}
\tag{12}
$$

The translation can be read thus: the value of M is z iff $\langle M \rangle z$, where $\langle M \rangle z$ is read as a linear formula, with $(:\ x\ y\ z)$ being read as ''the value of x applied to y is z'', and $(:\ x\ z)$ as ''the value of x is z''. For instance, the clause for $\langle MN \rangle$ is read as: ''the value of $\langle MN \rangle$ is z iff there exists an x and y such that the value of M is x and the value of x applied to y is z, and for any s, the value of y is s implies that the value of N is s''.

Concretely, the agent $\langle \lambda x\ x \rangle z$ is just:

$$\forall x \forall b\ (:\ z\ x\ b) \multimap (:\ x\ b)$$

[9]Note that we use two constants for message transmission: $:_{\iota \to \iota \to o}$ and $:_{\iota \to \iota \to \iota \to o}$.

That is, $\lambda x\ x$ is viewed as a server that waits for an argument x, and then asserts that the result b is just x. Logically, one reads it as saying that the value of $\lambda x\ x$ is z iff for all x and b, the value of applying z to x is b implies that the value of x is b.

Continuation-based translation. We concentrate here on a second translation which is cleaner (does not leave behind residual ''argument"-servers) and tighter (it mimicks one λ-reduction step by *one* process reduction step). The basic idea is to use continuations of type $k \equiv d \rightarrow d$, where $d \equiv (\iota \rightarrow o)$. We define an auxiliary map that translates a λ-term M to an HLcc term $[M]_{(d \rightarrow d) \rightarrow d}$, and define the translation $[\![M]\!]$ of a λ-term M to HLcc to be $[M]\mathbf{I}_k$. We use a constant $:_{\iota \rightarrow d \rightarrow \iota \rightarrow o}$ for message transmission.

$$\begin{array}{rcl} [u] & = & \lambda p_k . pu \\ [\lambda\, x.M] & = & \lambda p_k . \lambda c_\iota . \forall x_d \forall r_\iota ((:\ c\ x\ r) \multimap (p\ [\![M]\!]\ r)) \\ [MN] & = & \lambda p_k . \lambda c_\iota \exists z_\iota\ (([M]\ p\ z) \otimes (:\ z\ [\![N]\!]\ c)) \end{array}$$

The term $[u]$ takes a continuation, and invokes it on u. The term $[\lambda x\ M]$ takes a continuation p, a channel c waits for a message pair x, r (the argument, and return channel), and then invokes p on $[\![M]\!]$ and r. The term $[MN]$ takes a continuation p and a return channel c, creates a new local channel z, invokes $[M]$ on p and z, and feeds it the argument and the return channel on z.

Example 3.1 Some example translations:

$$\begin{array}{rcl} [\lambda\, x.x] & = & \lambda p \lambda c \forall x \forall r ((:\ c\ x\ r) \multimap (p\ x\ r)) \\ [\lambda\, x.xx] & = & \lambda p \lambda c \forall x \forall r ((:\ c\ x\ p\ r) \multimap p (\lambda z \exists c ((x\ c) \otimes (:\ c\ x\ z))) r) \end{array}$$

β reduction occurs as follows:

$$\begin{array}{rcl} [\![(\lambda x\ x)(\lambda x\ x)]\!] u & =_\lambda & \exists z\, (\forall x \forall r ((:\ z\ x\ r) \multimap (\mathbf{I}\ x\ r)) \otimes (:\ z\ [\![\lambda x\ x]\!]\ u) \\ & \longrightarrow & \mathbf{I} [\![\lambda x\ x]\!]\ u \\ & =_\lambda & [\![\lambda x\ x]\!] u \end{array}$$

□

The translation represents a very tight connection between the lazy lambda-calculus and HLcc. We state the main theorem:[10]

Lemma 3.1 $[\![(\lambda x\ M)N]\!]u \longrightarrow [\![M[x := N]]\!]u$

Proof 3.1 Expand LHS. □

Lemma 3.2 $[MN]Pu \longrightarrow [M'N]Pu$ *whenever* $[M]Pu \longrightarrow [M']Pu$

[10] The final version of the paper will have a much more detailed discussion, including connections with Sangiorgi's characterization of the equivalence induced by the π-calculus translation [San92].

Proof 3.2 By structural induction on M. $\square$

Corollary 3.3 *$[\![MN]\!]u \longrightarrow [\![M'N]\!]u$ whenever $[\![M]\!]u \longrightarrow [\![N]\!]u$.*

Corollary 3.4 *$[\![M]\!]u \longrightarrow [\![N]\!]u$ whenever $M \longrightarrow N$.*

Lemma 3.5 *If $[MN]Pu \longrightarrow S$, then $MN \rightarrow Q$ and $S =_\lambda [Q]Pu$, for some lambda-term Q.*

Lemma 3.6 *If $[\![M]\!]u \rightarrow S$, then there exist lambda-terms A, B such that $M \equiv AB$.*

Putting these together, we get:

Theorem 3.7 (Encoding theorem) *If $M \rightarrow N$, then $[\![M]\!]u \rightarrow [\![N]\!]u$; if $[\![M]\!]u \rightarrow P$, then there exists a lambda-term N such that $P =_\beta [\![N]\!]u$ and $M \longrightarrow N$.*

First-order indeterminate cc languages. The first-order indeterminate cc languages discussed in [SRP91] (and hence the concurrent logic programming languages) are special cases of HLcc. Essentially, each goal in cc is treated as an (unbanged) atomic formula, each constraint as a banged formula, and each guarded command is translated into the corresponding method. Recursion is obtained using **fix**. Assuming the same underlying constraint system, programs and agents in the syntax of [SRP91] are translated into first-order HLcc by:

$$\begin{array}{rcl}
[p(X) :: A] & = & \mathbf{fix}(\lambda r_o\ (\forall X_\iota\ p : X \multimap r \otimes [A])) \\
[D_1.D_2] & = & [D_1] \otimes [D_2] \\
 & & \\
[c] & = & !c \\
[\,\square_{i\in I} c_i \rightarrow A_i] & = & \&^n(!c_1 \multimap [A_1]) \ldots (!c_n \multimap [A_n]) \\
[A_1 \wedge A_2] & = & [A_1] \otimes [A_2] \\
[\exists X.A] & = & \exists X_\iota.[A] \\
[p(X)] & = & p : X
\end{array}$$

4 Future work

This paper opens up work in a variety of areas. Many aspects of sub-structural logics (e.g., non-commutativity and non-associativity, modalities for ‘‘mobility” in a sequent) promise to be very interesting to explore computationally. Also, one may move to a richer logic, e.g., using a lambda-calculus with dependent types, as in [HHP,Pfe91], or using impredicative features, as in the calculus of constructions [CH88].

Much work needs to be done in developing a coherent semantic framework for HLcc, exploiting, for example, the now well-established ideas of testing equivalences, and developing proof procedures for reasoning about programs. Indeed, it seems feasible that a variety of different operational notions can be captured by adopting the simple idea of taking the denotation of a process to be a *subset* of the formulas logically entailed by it. More generally, the connection between the model-theoretic semantics of linear logic and appropriate denotational semantics of HLcc will need to be explored. (The connection between the ''game-theoretic'' semantics for linear logic due to Blass, and the bisimulation semantics discussed here should be explored.)

Concrete programming languages are already being designed and implemented in this framework; as a specific task it would be interesting to develop an asynchronous, concurrent version of ML based on these ideas.

Acknowledgements. The authors would like to thank Dale Miller, Tim Winkler, Jose Meseguer and Radha Jagadeesan for helpful discussions during the development of this paper. This paper is based in part on talks given by the authors at Montreal, CMU and Imperial College. The work on the design and implementation of Linear Janus is being done in collaboration with Cliff Tse. Patrick Lincoln was partially supported by AT&T Fellowship and SRI internal funding.

References

[AFG90] A. Asperti, G.-L. Ferrari, and R. Gorrieri. Implicative formulae in the 'proofs as computations' analogy. In *Proc. 17-th ACM Symp. on Principles of Programming Languages, San Francisco*, pages 59–71, January 1990.

[AO89] S. Abramsky and C.-H.L. Ong. Full abstraction in the lazy lambda-calculus. Technical report, Dept. of Computing, Imperial College, 1989. To appear in Information and Computation.

[AP90] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In *Proc. 7-th International Conference on Logic Programming, Jerusalem*, May 1990.

[Asp87] A. Asperti. A logic for concurrency. Technical report, Dipartimento di Informatica, Universitá di Pisa, 1987.

[BB90] Gerard Boudol and G. Berry. The Chemical Abstract Machine. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 81–94. ACM, 1990.

[Bou89] Gerard Boudol. Towards a lambda-calculus for concurrent and communicating systems. In *TAPSOFT 1989*, volume 351 of *Lecture Notes in Computer Science*, pages 141–169, 1989.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[dBKPR91] Frank S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm for asynchronous communication. In *Proceedings of CONCUR '91*, 1991.

[DL86] Doug DeGroot and Gary Lindstrom, editors. *Logic Programming: Functions, Relations and Equations*. Prentice Hall, 1986.

[GG89] C.A. Gunter and V. Gehlot. Nets as tensor theories. In G. De Michelis, editor, *Proc. 10-th International Conference on Application and Theory of Petri Nets, Bonn*, pages 174–191, 1989.

[GG90] V. Gehlot and C.A. Gunter. Normal process representatives. In *Proc. 5-th IEEE Symp. on Logic in Computer Science, Philadelphia*, June 1990.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[HHP] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of ACM*. To appear.

[HJ90] Seif Haridi and Sverker Janson. Kernel andorra prolog and its computation model. In *Proceedings of the Seventh International Conference on Logic Programming*, June 1990.

[HM91] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. In *Proc. 6th IEEE Symp. on Logic in Computer Science, Amsterdam*, 1991.

[HP91] James Harland and David Pym. The uniform proof-theoretic foundation of linear logic programming. In V. Saraswat and K. Ueda, editors, *Logic Programming (Proceedings of the 1991 International Symposium)*, pages 304–320. MIT Press, 1991.

[HS91] Pascal Van Hentenryck and Vijay A. Saraswat. cc(FD): Constraint programming over finite domains. Technical report, Computer Science Department, Brown University, forthcoming 1991.

[HT92] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. Presented at the Abingdon Workshop on Mathematical Aspects of Information Systems, April 1992.

[JP91] Radha Jagadeesan and Keshav Pingali. Abstract semantics for higher-order functional language with logic variables. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*. ACM, 1991.

[JSS91] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, System Sciences Laboratory, Xerox PARC, January 1991.

[KS90] Kenneth Kahn and Vijay Saraswat. Actors as a special case of concurrent constraint programming. In *Proceedings of the Joint Conference on Object-Oriented Programming: Systems, Languages, and Applications and the European Conference on Object-Oriented Programming*. ACM Press, October 1990.

[Laf90] Lafont. Interaction nets. In *Conf. Proceedings of Seventeenth POPL, forthcoming*, 1990.

[Lin85] Gary Lindstrom. Functional programming and the logical variable. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 266–280, January 1985.

[Lin92] P. Lincoln. Linear logic. *SIGACT Notices*, 23(2):29–37, Spring 1992.

[LS91] Patrick Lincoln and Vijay Saraswat. Proofs as Concurrent Processes: A Logical Interpretation for Concurrent Constraint Programming. Technical report, Systems Sciences Laboratory, Xerox PARC, November 1991.

[Mes90] J. Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA '90*, October 1990.

[Mes92] J. Meseguer. A logical theory of concurrent objects and its realization in the maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*. 1992.

[Mil90] R. Milner. Functions as processes. In M. S. Paterson, editor, *The Seventeenth International Colloquium On Automata Languages And Programming*, pages 167–180. Springer-Verlag, 1990. Lecture Notes In Computer Science 443.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. LFCS Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[Mil92] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. Draft, Febrruary 1992.

[MOM89] N. Marti-Oliet and J. Meseguer. From Petri nets to linear logic. In: Springer LNCS 389, ed. by D.H. Pitt et al., 1989. 313-340.

[MPW89] R. Milner, J. G. Parrow, and D. J. Walker. A calculus for mobile processes, part i and ii. LFCS Report ECS-LFCS-89-85, University of Edinburgh, 1989.

[Nie89] Flemming Nielson. The typed λ-calculus with first-class processes. In *Proceedings of PARLE '89*, number 366 in LNCS, pages 357 – 373. Springer Verlag, 1989.

[NM90] Gopalan Nadathur and Dale Miller. Higher-order horn clauses. *Journal of the Association for Computing Machinery*, 37(4):777–814, October 1990.

[Pfe91] Frank Pfenning. Logic programming in the lf logical framework. Technical Report ERGO-91-098, School of Computer Science, Carnegie Mellon University, February 1991.

[Pra92] V.R. Pratt. Event spaces and their linear logic. In *Proc. Second International Conference on Algebraic Methodology and Software Technology*. Springer-Verlag, 1992.

[San92] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario (extended abstract). In A. Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 102 – 109. IEEE Computer Society Press, 1992.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. To appear, Doctoral Dissertation Award and Logic Programming Series, MIT Press, 1990.

[Sce90] A. Scedrov. A brief guide to linear logic. *Bulletin of the European Assoc. for Theoretical Computer Science*, 41:154–165, June 1990.

[SKL90] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In *Proceedings of the North American Conference on Logic Programming*, October 1990.

[SR90] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages, San Fransisco*, January 1990.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of Eighteenth ACM Symposium on Principles of Programming Languages, Orlando*, January 1991.

[Tho89] Bent Thomsen. A calculus of higher-order communicating systems. In *POPL 89*. ACM, 1989.

[Tse92] Clifford Tse. Linear janus: A concurrent programming language. Technical report, Xerox PARC, 1992.

[Wad91] William Wadge. Higher-order horn logic programming. In V. Saraswat and K. Ueda, editors, *Logic Programming (Proceedings of the 1991 International Symposium)*, pages 289–303. MIT Press, 1991.

A Higher-order linear logic

Identity	$\dfrac{}{F \vdash F}$	$\dfrac{\Gamma_1 \vdash F, \Sigma_1 \qquad \Gamma_2, F \vdash \Sigma_2}{\Gamma_1, \Gamma_2 \vdash \Sigma_1, \Sigma_2}$	**Cut**
Exch. Left	$\dfrac{\Gamma_1, F, G, \Gamma_2 \vdash \Sigma}{\Gamma_1, G, F, \Gamma_2 \vdash \Sigma}$	$\dfrac{\Gamma \vdash \Sigma_1, F, G, \Sigma_2}{\Gamma \vdash \Sigma_1, G, F, \Sigma_2}$	**Exch. Right**
λ **Left**	$\dfrac{\Gamma, F' \vdash \Delta \qquad F \rightarrow_\lambda F'}{\Gamma, F \vdash \Delta}$	$\dfrac{\Gamma \vdash \Delta, F' \qquad F \rightarrow_\lambda F'}{\Gamma \vdash \Delta, F'}$	λ **Right**
$\otimes$ **Left**	$\dfrac{\Gamma, F, G \vdash \Sigma}{\Gamma, (F \otimes G) \vdash \Sigma}$	$\dfrac{\Gamma_1 \vdash F, \Sigma_1 \qquad \Gamma_2 \vdash G, \Sigma_2}{\Gamma_1, \Gamma_2 \vdash (F \otimes G), \Sigma_1, \Sigma_2}$	$\otimes$ **Right**
$\multimap$ **Left**	$\dfrac{\Gamma_1 \vdash F, \Sigma_1 \qquad \Gamma_2, G \vdash \Sigma_2}{\Gamma_1, \Gamma_2, (F \multimap G) \vdash \Sigma_1, \Sigma_2}$	$\dfrac{\Gamma, F \vdash G, \Sigma}{\Gamma \vdash (F \multimap G), \Sigma}$	$\multimap$ **Right**
$\wp$ **Left**	$\dfrac{\Gamma_1, F \vdash \Sigma_1 \qquad \Gamma_2, G \vdash \Sigma_2}{\Gamma_1, \Gamma_2, (F \wp G) \vdash \Sigma_1, \Sigma_2}$	$\dfrac{\Gamma \vdash F, G, \Sigma}{\Gamma \vdash (F \wp G), \Sigma}$	$\wp$ **Right**
$\&$ **Left**	$\dfrac{\Gamma, F \vdash \Sigma}{\Gamma, (F \& G) \vdash \Sigma} \qquad \dfrac{\Gamma, G \vdash \Sigma}{\Gamma, (F \& G) \vdash \Sigma}$	$\dfrac{\Gamma \vdash F, \Sigma \qquad \Gamma \vdash G, \Sigma}{\Gamma \vdash (F \& G), \Sigma}$	$\&$ **Right**
$\oplus$ **Left**	$\dfrac{\Gamma, F \vdash \Sigma \qquad \Gamma, G \vdash \Sigma}{\Gamma, (F \oplus G) \vdash \Sigma}$	$\dfrac{\Gamma \vdash F, \Sigma}{\Gamma \vdash (F \oplus G), \Sigma} \qquad \dfrac{\Gamma \vdash G, \Sigma}{\Gamma \vdash (F \oplus G), \Sigma}$	$\oplus$ **Right**
! W	$\dfrac{\Gamma \vdash \Sigma}{\Gamma, !F \vdash \Sigma}$	$\dfrac{\Gamma, !F, !F \vdash \Sigma}{\Gamma, !F \vdash \Sigma}$	**! C**
! D	$\dfrac{\Gamma, F \vdash \Sigma}{\Gamma, !F \vdash \Sigma}$	$\dfrac{!\Gamma \vdash F, ?\Sigma}{!\Gamma \vdash !F, ?\Sigma}$	**! S**
? W	$\dfrac{\Gamma \vdash \Sigma}{\Gamma \vdash ?F, \Sigma}$	$\dfrac{\Gamma \vdash ?F, ?F, \Sigma}{\Gamma \vdash ?F, \Sigma}$	**? C**
? D	$\dfrac{\Gamma \vdash F, \Sigma}{\Gamma \vdash ?F, \Sigma}$	$\dfrac{!\Gamma, F \vdash ?\Sigma}{!\Gamma, ?F \vdash ?\Sigma}$	**? S**
$^\perp$ **Left**	$\dfrac{\Gamma \vdash F, \Sigma}{\Gamma, F^\perp \vdash \Sigma}$	$\dfrac{\Gamma, F \vdash \Sigma}{\Gamma \vdash F^\perp, \Sigma}$	$^\perp$ **Right**
0 Left	$\Gamma, 0 \vdash \Sigma$	$\Gamma \vdash \top, \Sigma$	$\top$ **Right**
$\perp$ **Left**	$\perp \vdash$	$\dfrac{\Gamma \vdash \Sigma}{\Gamma \vdash \perp, \Sigma}$	$\perp$ **Right**
1 Left	$\dfrac{\Gamma \vdash \Sigma}{\Gamma, 1 \vdash \Sigma}$	$\vdash 1$	**1 Right**
Π **Left**	$\dfrac{\Gamma, Pt \vdash \Sigma}{\Gamma, \Pi P \vdash \Sigma}$	$\dfrac{\Gamma \vdash Py, \Sigma}{\Gamma \vdash \Pi P, \Sigma}$	$\forall$ **Right**
Σ **Left**	$\dfrac{\Gamma, Py \vdash \Sigma}{\Gamma, \Sigma P \vdash \Sigma}$	$\dfrac{\Gamma \vdash Pt, \Sigma}{\Gamma \vdash \Sigma P, \Sigma}$	Σ **Right**

The Π **Right** and Σ **Left** rules only apply if y is not free in Γ, Σ, and any nonlogical theory axioms.

B Proofs sketches of some theorems

Theorem B.1 (Well-Formed Formulas) *If the conclusion sequent $\Gamma \vdash \Sigma$ is a syntactically well formed HLcc program, then so is every formula which appears in a normal proof.*

Proof B.1 Rough Sketch. This theorem depends on several permutability of inference arguments which will not be formally stated or repeated here.

The main argument in the proof of this theorem is that at applications of $\multimap$ Left, Π Left, and λ Left, if there is a proof, then there exists a proof with the above property. This normal proof is obtained by permuting inferences found in the left hypothesis of $\multimap$ Left until that proof branch is trivial. In such a normal form proof, the instantiations of Π Left quantifiers always appear immediately below an application of $\multimap$ Left with trivial left hand branch. Such a fragment of a proof may be trivially modified to satisfy the well-formedness condition above. The remaining problematic rule of inference is the λ Left rule, which by the restrictions on the quantifiers over left hand sides of $\multimap$ enforced by the **B** and **A** macros, may never lead to ill-formed formulas. $\square$

Lemma B.2 (Forward Proofs) *If there is a proof of $\Gamma \vdash G \otimes \top$ in HLL, where Γ and G satisfy the syntactic restrictions given above for HLcc, then there is a forward cut-free proof of this sequent.*

Proof B.2 It suffices to consider a cut-free proof of $\Gamma \vdash G \otimes \top$. If the proof is already forward, then we are done. If there is an application of a left rule above an application of a right rule, we find an application of a right rule immediately below an application of a left rule, and perform case analysis on these two rules. In effect, we are performing induction on the depth of cut-free proofs.

By studying the polarities of formulas, one can see that only syntactic entities corresponding to G or $G \otimes \top$ can appear on the right hand side of a proof. Further, the only right rules which apply to sequents $\Gamma \vdash G$ are axioms, constraint axioms, $\top$ or $\otimes$ Right. In the first three cases, there is no hypothesis to the application of axioms, thus there is no left rule applied above the right rule. Thus the right rule in question must be $\otimes$ Right:

$$\frac{\Gamma_1 \vdash G_1 \qquad\qquad \Gamma_2 \vdash G_2}{\Gamma_1, \Gamma_2 \vdash G_1 \otimes G_2} \otimes \mathbf{R}$$

And now we perform case analysis on the left rule which is applied above this application of $\otimes$ Right. If that left rule is $\otimes$ Left, we may simply permute the inferences in the obvious manner: first perform the $\otimes$ Left rule, then the $\otimes$ Right, making sure that both subformulas broken by the $\otimes$ Left rule appear in the same branch of the proof.

This same permutability reasoning applies the cases of: Exchange Left, λ Left, 1 Left, and & Left. For the cases of: $\wp$ Left, $\multimap$ Left, and $\oplus$ Left, the $\multimap$ Left case is typical:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma_1 \vdash F\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma_2, E \vdash G_1\end{array}}{\Gamma_1, \Gamma_2, F \multimap E \vdash G_1}{\multimap}\mathbf{L} \qquad \begin{array}{c}\vdots\\ \Gamma_3 \vdash G_2\end{array}}{\Gamma_1, \Gamma_2, F \multimap E, \Gamma_3 \vdash G_1 \otimes G_2}{\otimes}\mathbf{R}$$

$$\Downarrow$$

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma_1 \vdash F\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Gamma_2, E \vdash G_1\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma_3 \vdash G_2\end{array}}{\Gamma_2, \Gamma_3, E \vdash G_1 \otimes G_2}{\otimes}\mathbf{R}}{\Gamma_1, \Gamma_2, F \multimap E, \Gamma_3 \vdash G_1 \otimes G_2}{\multimap}\mathbf{L}$$

This same sort of transformation is possible also for $\wp$ Left and $\oplus$ Left.

The Σ Left and Π Left rules are handled in the same way, that is, by permutation, although the Σ Left rule may require that the new variable (y in the proof rule display in the Appendix) be renamed throughout that proof branch to avoid name clashes. Formally, the soundness of this technique is stated in the following proposition: if $\Gamma, \Sigma P \vdash \Delta$ is provable by application of Σ Left with variable y, then this sequent is provable by application of Σ Left with variable z, for some variable z not appearing below that application of Σ Left in the proof tree.

The remaining left rules of HLcc are excluded by the syntactic conventions of HLcc, and thus this completes the case analysis. □

Lemma B.3 (Sequential Proofs) *If there is a proof of $\Gamma \vdash G \otimes \top$ in HLL, where Γ and G satisfy the syntactic restrictions given above for HLcc, then there is a sequentialized, forward cut-free proof of this sequent.*

Proof B.3 By the previous lemma, it suffices to consider a forward, cut-free proof of $\Gamma \vdash G \otimes \top$. If the proof is already sequentialized, then we are done. If there is an application of a $\multimap$ Left rule with applications of other left rules in its left-hand branch, then we find such an application with the smallest (in terms of depth) left hand branch. In effect, we are performing induction on the size of the left hand branch of unsequentialized $\multimap$ Left rules in forward, cut-free proofs.

Since the proof is forward, all right hand rules appear above all applications of left rules, so we need only consider the case when the rule applied immediately in the left proof branch is a left rule. We perform case analysis on this rule. In each case, we permute the application of the left rule to below the application of $\multimap$ Left in question, thus reducing the unsequentializedness of that proof branch. For each case, simple permutability arguments apply. □

Theorem B.4 (Completeness) $\Gamma \vdash G \otimes \top$ *implies that for some* Δ, $\Gamma \longrightarrow^{\star} (\Delta, G)$.

Proof B.4 This theorem may be proven by induction on the sequentialized, forward, cut-free proof of $\Gamma \vdash G \otimes \top$. Essentially, the work has been done by the previously stated normalization theorems. All that remains is the case analysis on each rule applied in the normalized proof to see that a corresponding operational rule applies. $\square$

Theorem B.5 ($\vdash \equiv\ |\!\!\leadsto$) *If Γ and Δ satisfy the syntactic restrictions on HLcc programs, then $\Gamma \vdash \Delta$ is provable in HLL, if and only if $\Gamma\ |\!\!\leadsto \Delta$.*

Proof B.5 In the only-if direction, we perform induction on the size of sequentialized cut-free $\vdash$ proof. Formally, we first need to show that any $\vdash$ proof can be transformed into a sequentialized cut-free proof. First perform cut-elimination, then make each left hand branch of $\multimap$ left inferences forward. This transformation will succeed because the class of formulas allowed to appear in the antecedent of $\multimap$ is restricted in HLcc. Finally, apply the same procedure used above to generate a sequentialized cut-free proof.

We then perform induction on the size of that normalized proof. If the last proof step was a right hand rule or an axiom, there is a corresponding proof step in $|\!\!\leadsto$ by definition. The hypotheses of this $|\!\!\leadsto$ inference may be constructed by induction. If the last proof step in the $\vdash$ proof was a left hand rule, then we perform case analysis on the rules. Exactly as in the proof of completeness, every left hand rule applied to a Γ satisfying the HLcc restrictions corresponds to an operational step of execution. Thus one may read the final proof step as a step of execution, $\Gamma \longrightarrow \Gamma'$, and by induction $\Gamma'\ |\!\!\leadsto \Delta$, so by definition of $|\!\!\leadsto$, we have that $\Gamma\ |\!\!\leadsto \Delta$.

In the other direction, we perform induction on the derivation that $\Gamma\ |\!\!\leadsto \Delta$. In the case that the derivation ends in a right hand side rule, the same rule may be applied in the construction of the $\vdash$ proof, and the remaining proof may be constructed by induction. If the $|\!\!\leadsto$ derivation comes from $\Gamma \longrightarrow \Gamma'$ and $\Gamma'\ |\!\!\leadsto \Delta$, then by induction one may construct a proof of $\Gamma' \vdash \Delta$, and the proof of this may be extended to a proof of $\Gamma \vdash \Delta$, by the same argument as the soundness result above, that is, by induction on the derivation $\Gamma \longrightarrow \Gamma'$, and in each case of single operational step, constructing the corresponding $\vdash$ proof step. It happens that the $\vdash$ proof constructed in this way will be sequentialized and cut free. $\square$