A brief introduction to linear concurrent constraint programming

Vijay Saraswat
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto Ca 94304 *

April 22, 1993

Abstract

I briefly review the (linear) concurrent constraint programming framework, and discuss its connection to linear logic.

1 Concurrent Constraint Programming

The concurrent constraint programming (CCP) framework [Sar89] is based on the *proof-as-computation* principle pioneered by logic programming. It is a declarative, high-level paradigm for concurrent computation that is at the same time computationally powerful, conceptually simple and semantically elegant. The following introduction is drawn from [Sar89].

The simplest way of introducing this computational paradigm is to contrast it with imperative programming. Following [Dij76] we may say that in imperative programming, the state of the system is specified by a *store*, which is a vector V of n variables of interest, and a *valuation* assigning to each variable in V a fully-formed (completely known) value in its domain. Thus each store describes a point in the n-dimensional *state space* for the system, obtained by taking the product of all possible values for the variables X_i. Let us define a *constraint* to be a (possibly infinite) subset of the state space, that is, a set of valuations. When designing a (possibly non-deterministic) algorithm, we are concerned with specifying a mechanism, such that when it is initiated in a store that satisfies a given constraint (the "pre-condition") it terminates in a store that satisfies another given constraint (the "post-condition"). Hence in any formal notation ("programming language") used to describe such algorithms, some basic operations are necessary to transform one store to another. In imperative programming languages,

*This paper is available by anonymous ftp on the Internet from 13.1.64.94, as pub/ccp/lcc/lcc-intro.{dvi,ps}.

the basic operations executed by an agent are *Read*, which obtains the value of some variable in the store, and *Write* (or assignment) which translates the store parallel to some axis by *changing* the value of some variable (possibly based on applying some functions to the values obtained from prior reads).

The fundamental difference in concurrent constraint programming is that the *store is itself a constraint*, a set of valuations which provides *partial information* about the possible values for the given variables. The basic operations an agent may execute are *Ask* (instead of read), which checks whether or not the store *entails* a given constraint (that is, whether every valuation permitted by the store is permitted by the constraint), and *Tell* (instead of write), which adds a given constraint to the store, provided that the resultant store is *consistent*.[1] Thus, a basic step does not change the value of a variable, but may rule out certain values that were previously possible; we say that the store is *monotonically refined*, since at any step of the computation, the set of possible values for the variable is contained in the set of possible values at any prior step.

In this way, the entire constraint programming framework is based on extremely general notions of partial information, with its concommitant notions of consistency and entailment.

A fundamental consequence of this radically different view of the store — which one may regard as the essence of logic programming — is that it enables a simple, elegant but very powerful paradigm for concurrent computation. Briefly, one may imagine multiple agents of the kind described above, executing concurrently in a shared store. Synchronization is achieved through a *blocking* Ask: an agent may block if it attempts to ask whether a constraint is entailed by the given store, and it is not yet so; it remains blocked until such time as (if ever) some other concurrently executing agents add constraints to the store such that the resulting store is strong enough to either entail the query or its negation.

Conceptually, this notion of *concurrent constraint programming* provides a theoretical footing for the intuitive, pre-theoretic visions about computing with constraints, evidenced in some of the work of such people as Sutherland, Sussman, Steele, Borning and some of their colleagues. Further, a coherent development of this notion leads to a synthesis and extension of some of the work on *concurrent logic programming*, which has occupied a number of researchers for the better part of the last decade [Sha87,Sar87,Ued86,CG86] and the more recent notion of *constraint logic programming* due to [JL87,Mah87]. Their union properly generalizes both: the former because its basic concepts of communication and synchronization are now seen to be applicable in a far more general setting, and the latter because it introduces a simple but powerful framework for control where earlier there was none. More generally, such a view of computation is interesting (a) in the context of programming languages because of the ability to represent and manipulate partial information about the domain of discourse, (b) in the context of concurrency because of the use of constraints for communication and control, and (c) in the context of Artificial Intelligence research because of

[1]Terminology borrowed from Brachman and Levesque [BFL85] who used these terms in a closely related context — see [Sar89, Ch. 1] for more on the connections.

the availability of simple yet powerful mechanisms for controlling inference, and the promise that very rich hybrid representational/programming languages, sharing the same set of abstract properties, may be possible. In addition, a major promise of this approach is that it will enable a very simple treatment of thorny issues in the semantics of concurrent, non-deterministic programming languages, and that, too, in the context of a versatile and expressive computational framework.

2 Logical foundations of CCP

Where are the logical foundations of the (very operationally motivated) cc paradigm? [LS91] shows that a logical semantics may be given using a *formula-as-agent* and *proof-as-computation* interpretation of intuitionistic logic. 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. 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.

For example, the basic determinate ask-and-tell cc languages can be described in intuitionistic logic thus. There are three syntactic categories of interest: Programs (P), Agents (A) and Goals (G). They are given by the syntax:

$$
\begin{array}{llcll}
(Program) & P & ::= & H \rightarrow A & \text{– Clause.} \\
 & & & \mid \forall X\ P & \text{– Universal Closure.} \\
 & & & \mid P \wedge P & \text{– Multiple Definitions.} \\
 & & & & \\
(Agents) & A & ::= c & \text{– Tell.} & \\
 & & & c \rightarrow A & \text{– Ask.} \\
 & & & H & \text{– Procedure Call.} \\
 & & & A \wedge A & \text{– Parallel Composition.} \\
 & & & \exists X.A & \text{– Hiding.} \\
 & & & & \\
(Goals) & G & ::= & c & \text{– Primitive Test.} \\
 & & & \mid A \rightarrow G & \text{– Test an agent.} \\
 & & & \forall X.G & \text{– Generic Test.}
\end{array}
\tag{1}
$$

In the above syntax $\rightarrow$ is intended to be read as implication, and H ranges over the syntactic category of atomic formulas. Thus the fragment of logic specified above is in some sense “dual” to the usual definite clause logic underlying logic programming: a clause is of the form $H \rightarrow A$, *not* $H \leftarrow A$; one describes *necessary* rather than sufficient conditions. Intuitively, one thinks of specifying for each predicate some pieces of information that must follow (for some of the arguments to the predicate) when some other pieces of information are know to be true about some arguments to the predicate. Each predicate is thus thought of as a transducer of (partial) information.

One thinks of the underlying engine as making explicit pieces of information that are implicit in the original presentation of an agent — as explicit as is needed to answer a (primitive) test.

For example, we can understand the usual directional program for append and naive reverse as corresponding to the following formulas

$$
\begin{array}{l}
\texttt{append}(\texttt{X, Y, Z}) \rightarrow \\
\qquad ((\texttt{X} = [] \rightarrow \texttt{Y} = \texttt{Z}) \\
\qquad\ \wedge (\texttt{cons}(\texttt{X}) \rightarrow \exists\texttt{A}\exists\texttt{B}\exists\texttt{C.X} = [\texttt{A} \mid \texttt{B}],\ \texttt{Z} = [\texttt{A} \mid \texttt{C}],\ \texttt{append}(\texttt{B, Y, C}))). \\
\texttt{nrev}(\texttt{X, Y}) \rightarrow \\
\qquad ((\texttt{X} = [] \rightarrow \texttt{X} = \texttt{Y}) \\
\qquad\ \wedge (\texttt{cons}(\texttt{X}) \rightarrow \exists\texttt{A}\exists\texttt{B}\exists\texttt{C.X} = [\texttt{A} \mid \texttt{B}],\ \texttt{nrev}(\texttt{B, C}),\ \texttt{append}(\texttt{C}, [\texttt{A}], \texttt{Y})).
\end{array}
$$

Computation is initiated when the underlying execution engine is given a program and a goal. The operational semantics of the language is the usual one, with tells being accumulated in the store, and implication being treated as a blocking Ask: a configuration $(\Gamma, c \rightarrow A)$ can "be simplified to" (can make a transition to) the configuration (Γ, A) only if the constrains in Γ are strong enough to entail c. A minor modification is that execution terminates successfully if and when the store associated with the agent is strong enough to entail the constraint in the consequent of the goal.

The basic theorem connecting the operational semantics to logical notions is: a program presentation (P, G) succeeds *if and only if* the sequent $P \vdash G$ is derivable in intuitionistic logic (augmented with the axioms of the constraint system). Thus the usual operational imagery of store-as-constraint with Ask and Tell as basic operations can be understood merely as one way of implementing a complete theorem-prover for this fragment of logic.

More generally, these ideas lead to the formula-as-agent interpretation of a larger fragment of logic [LS91] — for example, one can "internalize" tests into the language by allowing agents of the form $G \rightarrow A$. One can allow for *recursive* testing (i.e., allow goals to be atoms, and allow programs to contain definite clauses specifying how such tests should be decomposed into more primitive tests). This shows how the notion of testing — a staple from the theory of concurrent systems — can be used to re-admit ("backward-chaining", definite-clause, constraint) logic programming into the CCP conceptual framework. In such a hybrid logical system, the "action" happening on the LHS of a sequent (in basic CCP) and on the RHS (in logic programming) can be combined smoothly. (From the perspective of AI-based reasoning system, what we have here is a smooth combination of constraint-based "backward-chaining" and "forward-chaining" systems.) With some care, it is possible to prove a much richer set of soundness and completeness theorems relating intuitionistic derivability to a simply motivated operational semantics.

3 Linear concurrent constraint programming.

As discussed above, the logical foundation for the CCP paradigm is intuitionistic logic. Linear logic [Gir87], one of the most significant advances in logic in several decades, is making possible a spectacular expansion in the scope of computational phenomena that can be analyzed logically.

The shift in perspective can be described in very simple proof-theoretic terms. Essentially, the role of the structural rules in Gentzen-style proof systems for (classical, intuitionistic) logic is examined carefully. The rules for weakening and contraction are dropped — this means that in an inference it is not possible to copy or discard arbitrary formulas. A system of *resource conscious* logic emerges in which the multiplicity of formulas is important: logical formulas can now be regarded as *dynamic resources* that may be consumed in a derivation. Two distinct notions of conjunction emerge, the multiplicative (tensor, $\otimes$) and the additive ("with", &), corresponding to two distinct but hitherto conflated notions of conjunction: the first corresponds to the notion of simultaneously possessing two resources *both* of which can be consumed in an interaction, and the second to possessing the potential to use any *one* of two resources. As the usual adjoint of the (tensor) conjunction arises the notion of *linear implication* $\multimap$, which captures the notion of "consumption" of resources. Thus, from $p \otimes q \otimes (p \multimap r)$ one can derive (entail) $q \otimes r$, but *not* $p \otimes q \otimes r$ (the single occurrence of p is "used up" in discharging the antecedent of $p \multimap r$). Interestingly, classical and intuitionistic logic can be recovered from linear logic by the introduction of an "of course" modality (!) — formulas marked with "!" can be weakened and contracted arbitrarily. For further details, the reader is referred to tutorial expositions by Scedrov [Sce90], a very readable description of the underlying ideas by Vaughan Pratt [Pra92]. The inference rules underlying this logic are given in the Appendix.

These ideas can be directly transferred to the CCP framework. We can think of the state of a computation as a multiset (tensor conjunction) of certain kinds of first-order linear logic formulas. As in the intuitionistic case, a configuration Γ can transition to another Δ by "simplifying" Δ, i.e. generally by finding an implication that can be discharged. The syntax and operational semantics of the resultant notion of linear concurrent constraint programming are shown in Table 1.

The resulting computational system seems extremely powerful, going well beyond usual constraint-based computation to incorporate directly aspects of state change as well. The following hint of its power will have to suffice. 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: the problem is that two different actors may wish to communicate the *same* message to an actor — hence they would both post, say, the constraint $5 \in X$. Now, operationally we would like both messages to be delivered to (the actor) X; in (intuitionistic/classical) logic, however, this situation

Syntax.

$$
\begin{array}{llll}
(Agents) & a ::= & \mathbf{1} & \text{– Termination} \\
 & & \mid p(t_1, \ldots, t_n) & \text{– Procedure call} \\
 & & \mid a \otimes a & \text{– Parallel composition} \\
 & & \mid \forall \bar{X}\ (q \multimap a) & \text{– Linear asks (methods)} \\
 & & \mid \exists X.a & \text{– New channels} \\
 & & \mid !\, c & \text{– Tell} \\
 & & \mid c \to a & \text{– Ask} \\
 & & \mid !\forall \bar{X}\ (q \multimap a) & \text{– Reusable methods}
\end{array}
\tag{2}
$$

$$
(Asks) \qquad q ::= \quad \mathbf{1} \mid p(t_1, \ldots, t_n) \mid q \otimes q
$$

The *store* $s(\Gamma)$ associated with a configuration Γ is defined to be the subset of all atomic formulas in Γ.
As usual, the language is presumed to have an entailment relation $\vdash_{\mathcal{C}}$ between finite sets of constraints and singleton constraints, capturing the interaction between constraints.

Transition System. The operational semantics is defined via a transition system on configurations Γ, Δ which are just multisets of agents. The binary relation "$\longrightarrow$" on configurations is the smallest relation satisfying:

$$
\begin{array}{l}
\dfrac{\Gamma \longrightarrow \Delta}{(\Gamma, \mathbf{1}) \longrightarrow \Delta}\ (\mathbf{1}) \\[2ex]
\dfrac{(\Gamma, a_1, a_2) \longrightarrow \Delta}{(\Gamma, a_1 \otimes a_2) \longrightarrow \Delta}\ (\otimes) \\[2ex]
\dfrac{(\Gamma, a) \longrightarrow \Delta \quad X \notin \mathbf{var}(\Gamma) \cup V}{(\Gamma, \exists X.a) \longrightarrow \Delta}\ (Hiding) \\[2ex]
\dfrac{c_1, \ldots, c_n \vdash_{\mathcal{C}} c}{(\Gamma, !\, c_1, \ldots, !\, c_n, c \ \to\ a) \longrightarrow (\Gamma, !\, c_1, \ldots, !\, c_n, a)}\ (\to) \\[2ex]
\begin{array}{l} (\Gamma, p_0[\bar{t}/\bar{x}], \ldots, p_{n-1}[\bar{t}/\bar{x}], (\forall \bar{X}.p_0 \otimes \ldots \otimes p_{n-1} \multimap a)) \\ \qquad \longrightarrow (\Gamma, a[\bar{t}/\bar{X}]) \end{array} \quad (\multimap) \\[2ex]
\begin{array}{l} (\Gamma, p_0[\bar{t}/\bar{x}], \ldots, p_{n-1}, [\bar{t}/\bar{x}], !\,(\forall \bar{X}.p_0 \otimes \ldots \otimes p_{n-1} \multimap a)) \\ \qquad \longrightarrow (\Gamma, !\,(\forall \bar{X}.p_0 \otimes \ldots \otimes p_{n-1} \multimap a), a[\bar{t}/\bar{X}]) \end{array} \quad (!)
\end{array}
\tag{3}
$$

Table 1: Syntax and semantics for linear cc

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, however, since $p \otimes p$ is *not* (linearly) equivalent to p (multiplicities matter). Therefore, the above translation of Actor-like languages into linear cc works directly. Interestingly, 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 $X : 5$ (using a binary predicate ":" with *no* built-in inference mechanism) suffice.

Our preliminary investigation has shown that linear cc (lcc) languages unify and synthesize a number of extant asynchronous formalisms. Clearly, Petri nets can be seen as just the propositional version of lcc, following the work of Asperti, Gunter and Gehlot, Marti-Oliet and Meseguer[Asp87,GG89,Mar89]. Indeed, the lcc languages can be seen as a "first-order" version of Petri nets (augmented with constraints). The (indeterminate, intuitionistic) cc languages are obviously embedded in the lcc framework. Similarly, an "asynchronous" subset of Milner's π-calculus can be directly embedded — 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. Since the basic communication mechanism in lcc can be seen as "many-to-many" rather than "many-to-one," lcc also immediately provides a constraint-based, "first-order" version of the Linda computation model.

Perhaps the most remarkable connection is with the lambda calculus: there is a direct and simple translation of the (untyped) lambda-calculus into lcc languages, as outlined briefly in Table 2. 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). Beta-reduction is reflected by the universal instantiation and (linear) *modus ponens* underlying the operational semantics of the lcc languages.

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 (see [BB90b]) for the programming language, thus relating the operational derivability relation between configurations *qua* agents to the entailment relation between configurations qua logical formulas. In general, the logical counterpart of parallel composition is conjunction, of (ask-)prefixing is (a quantified version of) implication, and of hiding is existential quantification. Communication occurs (essentially) via first-order variables. Indeterminism will arise, as in linear logic, if the entailment relation of the underlying logic is indeterminate.

References

[Asp87] A. Asperti. A Logic For Concurrency. Technical report, Dipartimento di Informatica, Universitá di Pisa, 1987.

Let M be an arbitrary lambda-term, and Z an arbitrary variable. We define an lcc agent M_Z by structural induction on M:

$$
\begin{array}{ll}
X_Z & !(X = Z) \\
(\lambda X\ M)_Z & !\,(\forall X \forall Y\ Z : X.Y \multimap M_Y) \\
(MN)_Z & \exists X\, \exists Y.(M_X \otimes N_Y \otimes X : Y.Z)
\end{array}
$$

The translation can be read thus: the value of M is X iff M_X, where M_X is read as a linear formula.

It makes use of the following correspondences:

lambda-parameters	$\longleftrightarrow$	communication channels
abstraction	$\longleftrightarrow$	reusable, universally quantified linear entailments
application	$\longleftrightarrow$	parallel composition on a hidden channel
beta-reduction	$\longleftrightarrow$	universal instantiation, and linear modus ponens

For example, the agent $(\lambda x.x)_Y$ is just:

$$
!\,\forall B\ (Y : X.B \multimap !(X = B))
$$

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.

Table 2: Translating the λ-calculus into lcc.

[BB90a] 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.

[BB90b] 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.

[BFL85] Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. *Readings in Knowledge Representation*, chapter KRYPTON: A functional approach to Knowledge Representation, pages 411 – 429. Morgan Kaufmann Publishers, 1985.

[CG86] K. L. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *TOPLAS*, 8(1):1–49, January 1986.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[GG89] C.A. Gunter and V. Gehlot. Nets as Tensor Theories. In *10-th International Conference on Application and Theory of Petri Nets*, Bonn, Germany, 1989.

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

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, Munich, W. Germany, January 1987.

[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.

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

[Mah87] Michael Maher. Logic Semantics for a Class of Committed-Choice Programs. In *4th International Conference on Logic Programming*. MIT Press, May 1987.

[Mar89] Marti-Oliet, N. and J. Meseguer. From Petri Nets to Linear Logic. In *LNCS 389*, 1989.

[Pra92] Vaughan Pratt. Response to Jon Barwise. Appeared on the Linear Logic mailing list, linear@cs.stanford.edu., February 1992.

[Sar87] Vijay A. Saraswat. The Concurrent Logic Programming Language **CP**: definition and operational semantics. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 49–62. ACM, January 1987.

[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, 1991.

[Sce90] Andre Scedrov. A Brief Introduction to Linear Logic. Appeared in the EATCS Bulletin, 1990.

[Sha87] Ehud Shapiro, editor. *Concurrent Prolog, Volume* 1 *and* 2. MIT Press, December 1987.

[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.

[Ued86] K. Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, 1986.

A Linear Logic Proof Rules

Identity	$\dfrac{}{A \vdash A}$	$\dfrac{\Gamma_1 \vdash A, \Sigma_1 \qquad \Gamma_2, A \vdash \Sigma_2}{\Gamma_1, \Gamma_2 \vdash \Sigma_1, \Sigma_2}$	**Cut**
Exch. Left	$\dfrac{\Gamma_1, A, B, \Gamma_2 \vdash \Sigma}{\Gamma_1, B, A, \Gamma_2 \vdash \Sigma}$	$\dfrac{\Gamma \vdash \Sigma_1, A, B, \Sigma_2}{\Gamma \vdash \Sigma_1, B, A, \Sigma_2}$	**Exch. Right**
$\otimes$ **Left**	$\dfrac{\Gamma, A, B \vdash \Sigma}{\Gamma, (A \otimes B) \vdash \Sigma}$	$\dfrac{\Gamma_1 \vdash A, \Sigma_1 \qquad \Gamma_2 \vdash B, \Sigma_2}{\Gamma_1, \Gamma_2 \vdash (A \otimes B), \Sigma_1, \Sigma_2}$	$\otimes$ **Right**
$\multimap$ **Left**	$\dfrac{\Gamma_1 \vdash A, \Sigma_1 \qquad \Gamma_2, B \vdash \Sigma_2}{\Gamma_1, \Gamma_2, (A \multimap B) \vdash \Sigma_1, \Sigma_2}$	$\dfrac{\Gamma, A \vdash B, \Sigma}{\Gamma \vdash (A \multimap B), \Sigma}$	$\multimap$ **Right**
$\wp$ **Left**	$\dfrac{\Gamma_1, A \vdash \Sigma_1 \qquad \Gamma_2, B \vdash \Sigma_2}{\Gamma_1, \Gamma_2, (A \wp B) \vdash \Sigma_1, \Sigma_2}$	$\dfrac{\Gamma \vdash A, B, \Sigma}{\Gamma \vdash (A \wp B), \Sigma}$	$\wp$ **Right**
$\&$ **Left**	$\dfrac{\Gamma, A \vdash \Sigma}{\Gamma, (A \& B) \vdash \Sigma} \qquad \dfrac{\Gamma, B \vdash \Sigma}{\Gamma, (A \& B) \vdash \Sigma}$	$\dfrac{\Gamma \vdash A, \Sigma \qquad \Gamma \vdash B, \Sigma}{\Gamma \vdash (A \& B), \Sigma}$	$\&$ **Right**
$\oplus$ **Left**	$\dfrac{\Gamma, A \vdash \Sigma \qquad \Gamma, B \vdash \Sigma}{\Gamma, (A \oplus B) \vdash \Sigma}$	$\dfrac{\Gamma \vdash A, \Sigma}{\Gamma \vdash (A \oplus B), \Sigma} \qquad \dfrac{\Gamma \vdash B, \Sigma}{\Gamma \vdash (A \oplus B), \Sigma}$	$\oplus$ **Right**
! W	$\dfrac{\Gamma \vdash \Sigma}{\Gamma, !A \vdash \Sigma}$	$\dfrac{\Gamma, !A, !A \vdash \Sigma}{\Gamma, !A \vdash \Sigma}$	**! C**
! D	$\dfrac{\Gamma, A \vdash \Sigma}{\Gamma, !A \vdash \Sigma}$	$\dfrac{!\Gamma \vdash A, ?\Sigma}{!\Gamma \vdash !A, ?\Sigma}$	**! S**
? W	$\dfrac{\Gamma \vdash \Sigma}{\Gamma \vdash ?A, \Sigma}$	$\dfrac{\Gamma \vdash ?A, ?A, \Sigma}{\Gamma \vdash ?A, \Sigma}$	**? C**
? D	$\dfrac{\Gamma \vdash A, \Sigma}{\Gamma \vdash ?A, \Sigma}$	$\dfrac{!\Gamma, A \vdash ?\Sigma}{!\Gamma, ?A \vdash ?\Sigma}$	**? S**
$^{\perp}$ **Left**	$\dfrac{\Gamma \vdash A, \Sigma}{\Gamma, A^{\perp} \vdash \Sigma}$	$\dfrac{\Gamma, A \vdash \Sigma}{\Gamma \vdash A^{\perp}, \Sigma}$	$^{\perp}$ **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**
$\forall$ **Left**	$\dfrac{\Gamma, A\{t/x\} \vdash \Sigma}{\Gamma, \forall x.A \vdash \Sigma}$	$\dfrac{\Gamma \vdash A\{y/x\}, \Sigma}{\Gamma \vdash \forall x.A, \Sigma}$	$\forall$ **Right**
$\exists$ **Left**	$\dfrac{\Gamma, A\{y/x\} \vdash \Sigma}{\Gamma, \exists x.A \vdash \Sigma}$	$\dfrac{\Gamma \vdash A\{t/x\}, \Sigma}{\Gamma \vdash \exists x.A, \Sigma}$	$\exists$ **Right**

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