Concurrency in the λ-calculus: Higher-order Concurrent Constraint Programming

Vijay A. Saraswat*

(Extended Abstract of Preliminary Report)

December 4, 1991

Abstract

We introduce higher-order versions of the determinate and non-determinate concurrent constraint (cc) programming languages [Sar89]. Thereby, we propose a simple conceptual and semantic framework for the joint exploration of (concurrent) functional and (concurrent) (higher-order) (constraint) logic programming languages.

1 Introduction and overview

The concurrent constraint (cc) programming paradigm is based on a set of very simple ideas, arising from concurrent logic programming and constraint logic programming. The basic references are [Sar89, SR90, SRP91, JSS91]; we summarize here briefly. Imperative languages may be thought of as based on the notion of *store-as-valuation*: 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 which specify a set of (possibly infinite) admissible values for the variables of interest. The notions of Read and Write for 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.

*Address: Xerox PARC, 3333 Coyote Hill Road, Palo Alto Ca 94304. E-mail: saraswat@parc.xerox.com.

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*.

Schematically, then, such (first-order) agents in a cc language may be described by the little grammar:

$$A ::= c \mid c \rightarrow A \mid A \wedge A \mid A \vee A \tag{1}$$

The first agent adds the constraint c (e.g., $X \geq Y + 5$) to the store, the second blocks until c is entailed, and then behaves like A, the third spawns two concurrent sub-computations conjunctively, the fourth spawns them disjunctively.

Thus, in this set-up, processes are thought of as *information transducers*: they examine pieces of partial information (constraints) and produce other such pieces. Through very general considerations, it is not difficult to argue that processes should be monotone and continuous in the underlying information ordering. In fact, at a meta-level, such a view of computation underlay the original motivation for Dana Scott's introduction of various lattice-theoretic models for the untyped lambda-calculus [Sco70]. Perhaps not surprisingly, this paper establishes those lattice-theoretic models as the right semantic framework in which to characterize cc languages.

I want to remark in passing that the framework I have sketched out above, while apparently simple, is powerful enough to account (among others) for data-flow, (first-order) functional and concurrent logic programming languages. All this is discussed in some detail in my thesis [Sar89], and we shall not re-make that case here.

The purpose of this paper is to explore the higher-order analog of the cc languages. We are motivated in part by the obvious need to understand the interaction between concurrency and higher-order calculi (see for example [Bou89, Tho90, MPW89, BB90]), in part by the need to understand at a more fundamental level the relationship between the cc *programming* framework, and Scott's conception of computation, and in part by the need to get a more powerful framework for computing with constraints than provided by the first-order cc languages. For example, we would like to be able to pass processes as parameters (as in writing a scheduler, or passing comparison routines to sort programs etc.), be able to recover from inconsistent sub-computations, store constraints and processes in data-structures, etc.

The basic proposal. Our starting point is to view the first-order cc processes as functions over the underlying constraint system (viewed, for the time being, as an information system in the sense of [Sco82]). In [SRP91] we showed that such processes could be taken to denote *closure operators* over the underlying constraint system, with parallel composition corresponding in a natural way to conjunction (in a manner reminescent of [Hoa89]). The key question to ask is: how should such a calculus of closures be embedded within a calculus of functionality? For this we appeal to the extensive use of closures to characterize data-types in [Sco76]. In that paper, Scott defined in LAMBDA the combinator $\mathbf{V}$, which when applied to any (continuous) function f yielded the least closure operator $cl(f)$ above f. The basic idea is simple: $cl(f)$ is obtained by applying f repeatedly to its input, adding the input at each iteration. That is, $\mathbf{V} \stackrel{def}{=} \lambda f.\lambda s.\mathbf{Y}(\lambda a.s \sqcup (fa)))$.

To be able to define $\mathbf{V}$ therefore, we are led to borrow from the design of LAMBDA[1] and add to the standard syntax of the lambda-calculus

$$A ::= x \mid \lambda x\, A \mid A\ A \tag{2}$$

a combinator $\wedge$ via the production:

$$A ::= A \wedge A \tag{3}$$

We have in mind the intuition that the process $A_1 \wedge A_2$ provides the conjunction of the information (constraints) provided by A_1 and A_2. In the information-order, therefore, $\wedge$ should be interpreted by *joins*,[2] and therefore we must require that the underlying domain of denotations be a *lattice* (rather than, say, a consistently complete dcpo).

Operationally, the result of applying $A \wedge B$ to any argument C is the conjunction of the result of applying A to C and B to C:

$$(A \wedge B)C = AC \wedge BC \tag{4}$$

Note that such a parallel composition must be *fair*: thinking of each agent *qua* producer of constraints—and of computation as a mechanism for making *explicit* constraints that are implicit in the original formulation of the program—it is not permissible to schedule one agent perpetually while another may be waiting to produce constraints. This is reflected in the semantic domain by the equation: $M \wedge \bot = M$. That is, the "divergent process" is equivalently thought of as the process that produces no constraints. In the following, to emphasize the logical nature of $\bot$ as the identity of conjunction, we shall write it as `true`.

However, parallel composition yields just a notion of *disjoint parallelism*–an argument is merely copied to all the concurrent processes "examining" it. In order to get a notion of *synchronization*, it has to be possible for an abstraction

[1] While joins are not provided as a basic combinator in LAMBDA, they are definable in LAMBDA via $\wedge \equiv \lambda x \lambda y.(\lambda z.0) \supset x, y$ [Sco76, P.532].

[2] We are following the usual domain-theoretic terminology in which $\top$ is considered to have "more information" than $\bot$, even though this unfortunately clashes with the tradition in logic of regarding conjunctions as meets and disjunctions as joins.

to explicitly *test* the information-content of its argument, while the argument is being evaluated. How best should this be done? In the *graph-models* of the λ-calculus [Plo72, Sco80], the denotation of an abstraction $\lambda x.M$ is obtained by probing M with all possible pieces of partial information u for x, in a manner very reminescent of the notion of asks. So a simple way suggests itself: introduce the notion of abstractions which test their body only with pieces of information that are *above a particular (finite) threshold.* Thus we propose adding to the lambda-calculus the notion of *ask-abstraction*, via the production:

$$A ::= \lambda x : u.M \tag{5}$$

where u will be defined shortly to range over finite pieces of partial information. Intuitively, an argument is evaluated until—if ever—it can be determined that it satisfies u. If it does, β-reduction can be performed; otherwise no piece of information can be gleaned from this application. As with parallel composition, we require that testing be done fairly so that when enabled a β-reduction occurs after at most finitely many other evaluation steps.

What is the notion of partial information for the lambda-calculus? First, clearly `true` should be an admissible piece of partial information—it is the vacuous piece true of every object.[3] Another piece of information about a function would be a finite portion of its graph—the observation that when it was fed a finite piece of information u_1, it produced a finite piece of information u_2. Similarly, if u_1 is a finite piece of information about a function, and we know that u_2 is true of its argument, we should be able to get back a finite piece of information $u_1 u_2$ about its result. And finally, of course, if $u_1 \wedge u_2$ should be a piece of partial information if u_1 and u_2 are. To summarize, we can take our tests to be of the form:

$$u ::= \texttt{true} \mid \lambda x : u.u \mid u\ u \mid u \wedge u \tag{6}$$

This grammar is identical to that for λ-terms—except that the production $A ::= x$ has been replaced by $u ::= \texttt{true}$. Not surprisingly, it will turn out that the grammar captures precisely the (algebraically) finite elements in the intended domain of interpretation for such terms.[4]

The usual call-by-name and call-by-value abstractions of the λ-calculus are special cases of ask-abstraction. Call-by-name abstraction is just $\lambda x : \texttt{true}.M$—test that the input satisfies the trivial test. Call-by-value abstraction [Plo75] (in the pure case) is just: $\lambda x : (\lambda y.\texttt{true}).M$—test that your input is a function, before doing the β-reduction. Within the pure λ-calculus it turns out that call-by-name and call-by-value abstractions are sufficient to get the effect of general ask-abstractions (as we shall demonstrate in Section 2.3, using a construction

[3] I am being methodologically sloppy here—taking `true` to stand for both a *term* in the logic, and the object in the domain that it denotes! Mea culpa.

[4] In fact, it is possible to do away with tests of the form $u\ u$ without losing any expressive power, since an expression $u\ u$ can always be normalized completely at compile-time. For the rest of the paper, we shall consider u to range over such a simplified vocabulary.

by Boudol which was crucial in establishing full abstraction in [Bou91])—but the added generality of ask-abstraction is essential when other objects are introduced into the domain of computation.

Finally, we note that Scott's combinator $\mathbf{V}$ is now expressible within the programming language. In conjunction with ask-abstractions, this provides a simple logical way to express synchronization within the language. For, the agent $\mathbf{V}(M \wedge N)S$ expresses the idea that the agents M and N should be run (fairly!) in parallel—in any order—on the "shared store" S. On a particular invocation, M may not be able to produce any information when applied to S—-but the "next time around", N may have acted on S to raise its information content, thereby allowing the test that M may have been making to succeed. Computation quiesces (that is, cannot produce any more constraints) exactly when both M and N quiesce on the current state of the store. This is exactly the higher-order analogue of the ask-and-tell synchronization of the cc languages.

The study of this basic idea of higher-order concurrent constraint programming is the purpose of the rest of the paper.

Outline of paper. We proceed as follows. After discussing some remaining preliminary issues in this section, we study the denotational semantics for λcc. First, generalizing related ideas of [AO89, Bou91], we show that the initial solution of the domain equation $D \cong A + (D \to D)$ in the category $\omega\mathbf{LAT}$ of ω-algebraic lattices provides a fully-abstract semantics for $\lambda\mathsf{cc}(A)$, where A is the underlying *propositional* constraint system. Our approach to the semantics is via the "domain logics" of [Abr91], which are particularly pleasing in this setting because of the logical nature of the programming language. The close symmetry between syntax and semantics of λcc makes the model seem natural and the proofs easy. For example, the entailment relation $\mapsto$ between programs M and "finite" programs u (Appendix B) is described using an intuitionistic sequent calculus closely related to the calculus used to establish the denotational semantics by describing the relation between programs M and formulas ϕ of the underlying domain logic (Table 1).

A significant area of applicability of constraint programming is in the specification of non-deterministic search problems of the kind that underlie computations in problem-solving, diagnosis and other AI settings [Hen89, Sar89, SRH91]. To account for such computational phenomena, it is necessary to introduce a form of disjunction into the programming language. We discuss the extension of the operational and denotational semantics for this construct (Section 3). Essentially, the extension can be done smoothly in the same way as was possible for the basic cc languages [JSS91]. A fully abstract model is obtained by taking the initial solution of $E \cong \mathcal{P}_S(A+(E \to E))$, for $\mathcal{P}_S(\_)$ the Smyth power-domain construction.

The above two sections establish the semantic foundations for the *propositional* λcc languages. In these languages, any notion of a variable on which constraints are imposed is *internal* to the built-in information system A. In particular, while full constraints can be passed around as arguments to λ-terms, the variables on which these constraints are imposed, cannot. This limits the expressiveness of these languages—essentially, constraint-based computations "occur within" the domain A and are kept isolated from the higher-order machinery of D.

To solve this problem, we move to a more general setting. The domains used in denotational semantics correspond to *propositional* geometric theories [Vic88, Abr91]: each token is taken to be a proposition about some object of interest. In constraint programming, however, there is a need for *first-order* systems of information [Sar89, SR90, SRP91] : there may simultaneously be several objects of interest, and a need to express pieces of partial information simultaneously relating several of these simultaneously. This leads to the idea of considering *indexed* information systems, in which tokens can be thought of as *parametrized* by a sequence of variables taken from some underlying (fixed, unstructured) set of *variables*. (Tokens can hence be legitimately construed as *constraints* on these variables). The entailment relation is required to be *generic* with respect to the variables (i.e., the name of a particular variable is not of semantic importance). The resulting category **ConstSys** of *constraint systems* [5], are shown to be closed under all the usual domain-theoretic constructions (products, sums, lifting, function-space), and to admit of initial solutions to recursive "domain equations". Hence, they can serve as the setting in which to seek denotations for various *first-order* λcc-languages. We also show how to obtain from any domain D (and an underlying set of variables) the constraint system $\mathcal{L}(D)$ whose elements can be thought of as constraints (in finitely many variables) over $|D|$.

As an aside, let me remark that the need for such an extension should not be surprising. The development of domain theory was driven by the demands of obtaining models for the λ-calculus, for which propositional systems of information suffice. Our current enterprise, however, is about *logic* programming, that is, programming *with* logic, that is, programming in a set-up where the state of the computation *is* a logical formula. It is not surprising, therefore, that some (rather modest!) extra logical machinery needs to be imported into the foundations of the theory used to describe such computations.

This then is the setting for the *first-order* λcc languages discussed in the next section (Section 5). Models for such languages are solutions (in **ConstSys**) to the domain equation $D \cong A + \mathcal{L}(D) + (D \to D)$, for A an underlying system of information. The programming language is obtained by augmenting propositional λcc with some new combinators on constraints, reflecting the structure of $\mathcal{L}(D)$. In particular, in such languages variables can be passed freely as arguments to λ-terms "sitting inside" $\mathcal{L}(D)$. The results in the propositional case (discussed above) extend without much difficulty to the first-order case.

[5] We prefer this name to the more descriptive though more cumbersome *indexed information systems*.

Finally, in Section 6, we present a few concrete λcc languages to fix the reader's programming intuitions, and briefly discuss how they allow the representation and manipulation of partially-formed processes.

The rest of this paper is designed to fill in some key conceptual details. Most of the major constructions are summarized. Some more details are provided in Appendices.

1.1 Contributions of this paper

We briefly summarize the major contributions of the paper. They are five.

(1) We introduce the basic conception of higher-order cc programming languages. This is done in a uniform manner, for *any* underlying constraint system, by introducing the notion of parallel composition and ask-abstraction. Furthermore, we explain how ask-and-tell synchronization arises through the use of **V**—essentially through the idea of repeated application until quiescence. (2) We study the operational and denotational semantics of the λcc calculus and show that the "obvious" model is fully abstract. This lends some credence to the naturalness of the λcc programming paradigm. (3) To simulate the "backtracking" of angelic cc languages (and their special case, the logic programming languages), we add to the calculus a disjunction operation. Correspondingly, we provide a fully abstract model for this enriched language. (4) We introduce the category of constraint systems, and show that recursive domain equations may be solved over it. We also show how to construct constraint systems from arbitrary information systems—though simple, the construction is crucial in making it possible to smoothly embed constraint-solving with the functional calculus. (5) We exhibit the design of a specific λcc language, based on the Herbrand constraint system. In this language, processes can be stored in data-structures on which constraints are being imposed, extracted, and applied to other objects (including the constraint they were extracted from).

In summary, this paper is intended to establish the foundations for higher-order constraint programming.

We now quickly describe the extensions to the basic functional programming and cc framework brought about by λcc.

λcc qua functional programming. This work is related to the lazy functional programming languages such as Miranda [Tur85] via its close relationship with the lazy λ-calculus of [AO89]. To functional programming, it adds the notion of parallel composition and synchronization in a purely "declarative" setting. It also shows how to include constraint-solving within the lambda-calculus.

λcc qua constraint programming. Unlike the basic cc languages, there is no longer any *explicit* notion of a store. In any application $\mathbf{V}(M_0 \wedge \ldots \wedge M_{k-1})N$, N may be regarded as the store via which the concurrent processes $M_0, \ldots, M_{k-1}$

are interacting. Indeed, the notion of *store-as-constraint* has given way here to the notion of *argument-as-constraint*.

Second, "failure" — a store becoming inconsistent — can be trapped, and some action can be taken based on it. This takes care of one of the major difficulties of cc languages, in which inconsistency of the shared store can abort the entire computation, since there is no entity examining the store "from the outside". Indeed it seems that several rather complex notions of "meta-calls" investigated in the concurrent logic programming community can now be considerably simplified in this setting.

Third, processes can be passed as parameters, as is quite usual in a higher-order functional calculus. Indeed, a process itself may be produced as the result of a computation. This leads to considerable expressive power.

Fourth, sequential function composition is possible now, whereas this was not possible in the basic calculus, which was based purely on closure operators. That is, it is possible to write in λcc programs which check whether some other *given* process produces some constraints, before taking some action.

Furthermore, for certain kinds of constraint systems (see the discussion on λcc(Herbrand) in Section 6) we show that even inconsistency can be localized to the concerned variables, thus greatly simplifying the conception of distributed cc programming languages (cf. [SKL90]).

1.2 Comparison with related work

The notion that lattice models of the λ-calculus have something to do with concurrency is part of the folklore of domain theory (see, e.g., [Mey88], [Blo90]). However, to our knowledge the appropriate notions of concurrent computation has not been worked out in such detail before, and the notion of join as *fair* parallel composition, and agents as information transducers, not been exposed before.

This work is most closely related to [AO89] and [Bou91]. The former discusses in detail the semantics of the lazy λ-calculus, and shows that the least solution of $D \cong (D \rightarrow D)_\perp$ is fully abstract for the lazy λ-calculus, provided that it is extended with "parallel or", and a "convergence tester". [Bou91] re-did the work (and examined a "strict" calculus), pointing out that from a programming language point of view a much simpler idea was to introduce "call-by-value" and parallel composition, and called the resulting calculus λ_v^{nj}. Essentially, λ_v^{nj} is pure, deterministic, propositional λcc: this work adds the more general notion of ask-abstraction, explicates the notion of synchronization, adds non-determinism, and considers "first-order" languages over arbitrary *constraint* systems. We have borrowed the basic approach liberally from [Bou91], however, particularly the structure of the full-abstraction proofs.

Our work is close to [Tur85], except that (1) it is done in an untyped setting (2) the built-in constraint-system over which asks are being answered could be completely arbitrary, and is indeed far more general (e.g., in permitting logical variables) than what is allowed by "algebraic data-types" and, hand in hand (3) the explicit notion of parallel composition allows for finitary cooperation between concurrently executing agents. Laws in Miranda [Tho86] do allow the definition and use of un-free data-types, but are not as general as first-order λcc languages.

I have seen an early version of a paper being prepared by Jagadeesan and Pingali on a higher-order version of Id [JP91]. They discuss a typed, determinate higher-order calculus, and use a fairly specific construction for which they do not seem to have full abstraction. They do not seem to have explored a general ask-operation, or the introduction of non-determinism, or provided a general mechanism for integrating constraint-systems into the programming language (e.g., via the $\mathcal{L}(D)$ construction).

Because our work provides automatically a higher-order version of Prolog, it is related to other proposals for higher-order logic programming, e.g., [CKW89],[NM90]. In particular, the connection with [NM90] seems to be close (though that study is proof-theoretic in nature and a notion of models has not been elucidated) and bears further investigation. It should be noted, however, that first-order λcc languages do *not* require higher-order unification to be implemented—in fact any underlying constraint-system will do, and is augmented by essentially just λ-terms.

Investigations of the semantics of the first-order language have been carried out by [GL90, dBP90, GMS89, MR91, PSSS91] etc. But to a large extent that work is not relevant to the issues of this paper.

Recently, I learnt of Hassan Ait-Kaci's work on this topic [AK91]. It is too early to make detailed technical comparisons; however, it seems that the underlying intuitions behind the two approaches are quite similar.

2 Propositional λcc

2.1 Denotational semantics

We give the denotational semantics using the "logical form of domain theory" as developed extensively in [Abr91], and illustrated in [Bou91]. The basic point is this: as Scott repeatedly emphasized in his introduction of the concrete information systems [Sco82], elements of a Scott domain can be thought of as just filters of formulas of a logic (the "domain logic", whose formulas are given by the tokens of the domain and whose axioms are given by those defining the entailment relation of the domain.) Therefore, to give the denotational semantics for a programming language, it is sufficient to fix a domain of denotations D and give a "syntax-directed" *proof* system for the language, specifying which formulas in the domain logic for D are derivable for which programs. The definition is given in Table 1.

Syntax.

$$\begin{array}{l} M ::= x \mid \lambda x : u.M \mid M\ M \mid M \wedge M \mid a \\ u ::= \mathtt{true} \mid \lambda x : u.u \mid u \wedge u \mid a \end{array} \tag{7}$$

Denotational semantics. Programs are interpreted in the domain $D(A)_\star$, the least solution of the equation $D \cong A + (D \to D)$ via the definition:

$$[\![M]\!]\sigma = \{\phi \mid M \vdash_\sigma \phi\} \tag{8}$$

Here, M is a program, ϕ a formula of the logic for $D(A)_\star{}^a$, and σ a mapping from variables to filters of formulas. The entailment relation used is the least relation satisfying the following axioms:

$$M \vdash \mathtt{true} \quad \frac{M \vdash \phi \quad M \vdash \psi}{M \vdash (\phi \wedge \psi)} \quad \frac{M \vdash \phi \quad \phi < \psi}{M \vdash \psi} \tag{9}$$

$$x \vdash_\sigma \sigma(x) \quad \frac{M \vdash_{\sigma[x:=\psi]} \phi \quad \psi \leq [\![u]\!]\sigma}{\lambda x : u.M \vdash_\sigma \mathtt{true} \oplus (\psi, \phi)} \tag{10}$$

$$\frac{M \vdash \mathtt{true} \oplus (\psi, \phi) \quad N \vdash \psi}{MN \vdash \phi} \quad \frac{M \vdash \phi \quad N \vdash \psi}{M \wedge N \vdash (\phi \wedge \psi)} \tag{11}$$

$$a \vdash a' \oplus \mathtt{true} \quad (a \leq_A a') \tag{12}$$

Operational Semantics.

$$\lambda x : u.M \Downarrow \lambda x : u.M \tag{13}$$

$$\frac{M \Downarrow \lambda x : u.R \quad N \Downarrow J \quad M[J/x] \Downarrow K \quad N \mapsto u}{MN \Downarrow K} \quad \frac{M \Downarrow (M_0 \wedge M_1) \quad (M_0 N \wedge M_1 N) \Downarrow K}{MN \Downarrow K} \tag{14}$$

$$\frac{M \Downarrow K}{\begin{array}{c} M \wedge N \Downarrow K \wedge N \\ N \wedge M \Downarrow N \wedge K \end{array}} \quad \frac{M \Downarrow K \quad N \Downarrow J}{M \wedge N \Downarrow K \wedge J} \quad a \Downarrow a \tag{15}$$

[a] The syntax of the formulas, and the relevant axioms of the domain logic are given in Appendix A.

Table 1: Syntax and semantics for λcc.

Note that since the above specifies a continuous model (in the sense of [Bar84, Chapter ??], the least fixed point operator is denotable in the language by **Y**, as usual. Also, `true`, the bottom of the (domain-) lattice is definable as usual by $\Omega = (\lambda x.xx)(\lambda x.xx)$.

A sound and complete set of laws for proving equality between finite programs u will be given in the full paper.

Because of the "syntax-directed" nature of the proof system for the denotational semantics, it is not hard to see that:

Lemma 2.1 *$\sqsubseteq$ is contextual, that is, for all terms M, N, $M \sqsubseteq N$ iff for all contexts $C[\_]$, $C[M] \sqsubseteq C[N]$.*

2.2 Operational semantics

Following [Bou91] we formalize the relation $M \Downarrow K$ which should be read as: K is a value of M. Here, K, the type of values, is given by the grammar:

$$K ::= a \mid \lambda x : u.M \mid K \wedge M \mid M \wedge K \tag{16}$$

The rules are adapted and extended versions of those in [Bou91] and are given in Table 1. The rules make use of an *entailment relation* $\mapsto \subseteq A \times u$ between agents and finite tests defined in Appendix B via the axioms of (an extended form of) an intuitionistic sequent calculus.

We say that a term M *diverges* iff there is no K such that $M \Downarrow K$. An operational pre-order is defined by: $M \sqsubseteq_{\mathcal{O}} N$ iff in all context $C[\_]$, $C[M]$ diverges whenever $C[N]$ does. A divergent computation should be thought of as a computation that does not produce any information. Thus an agent capable only of engaging in an endless sequence of reductions, without producing any information, is considered a divergent computation. Similarly the agent $(\lambda x : u.M)N$ where N is such that it does not entail u is considered to be divergent even though, in some sense, it is a "deadlocked" configuration, with no possibility of progress.

While the operational rules are non-deterministic, and can give rise to several values for a term, all the values are compatible in the obvious sense:

Lemma 2.2 (Church-Rosser property) *If $M \Downarrow K_0$ and $M \Downarrow K_1$, then there is a K such that $K_0 \Downarrow K$ and $K_1 \Downarrow K$.*

Intuitively, different values can arise only because different branches of a conjunctive computation are evaluated at different "speeds". By carrying along the computation on both branches enough, we can make sure that the values re-converge to the same expression.

We can now state the adequacy theorem:

Theorem 2.3 (Adequacy) *$M \sqsubseteq N$ implies $M \sqsubseteq_{\mathcal{O}} N$.*

It proof relies on a *convergence lemma* that establishes that if $M \Downarrow$, then there is a formula ϕ of the form $a \oplus \mathtt{true}$ or $\mathtt{true} \oplus (\mathtt{true}, \mathtt{true})$ such that $M \vdash \phi$.

2.3 Full abstraction

The proof of full-abstraction basically follows the lines of that in [Bou91]. The main point is that it is necessary to show that every finite finite element u of the semantic domain is representable explicitly by a program $\mathbf{M}_u$ (that is, we have $[\![\mathbf{M}_u]\!] = u$). But this is easily done via the induction:

$$\begin{array}{l} \mathbf{M}_{\mathtt{true}} = \Omega \\ \mathbf{M}_{\phi\wedge\psi} = \mathbf{M}_\phi \wedge \mathbf{M}_\psi \\ \mathbf{M}_{\mathtt{true}\oplus(\phi,\psi)} = \lambda x : \mathbf{M}_\phi.\mathbf{M}_\psi \\ \mathbf{M}_{a\oplus\mathtt{true}} = a \end{array} \tag{17}$$

Obviously, the presence of both conjunction and ask-abstraction makes this result very easy. The direct correspondence between the syntax of the $\lambda\mathtt{cc}$ languages and semantics is quite pleasing (though not surprising considering that $\lambda\mathtt{cc}$ is a *logical* language).

In passing, we note that Boudol has shown that some tests can be defined in terms of others. In particular, it is sufficient to take ask-abstractions of the form $\lambda x : w.M$ where w is given by:

$$w ::= \mathtt{true} \mid \lambda x : \mathtt{true}.\mathtt{true} \mid a \tag{18}$$

He uses the clever idea of defining not just a recognizer for every formula ϕ but also a *tester* T_ϕ which evaluates to I when applied to an argument N iff $N \vdash \phi$. The explicit construction is:

$$\begin{array}{ll} \mathbf{M}_{\mathtt{true}} = \Omega & \mathbf{T}_{\mathtt{true}} = \lambda x.\lambda y.y \\ \mathbf{M}_{\phi\wedge\psi} = \mathbf{M}_\phi \wedge \mathbf{M}_\psi & \mathbf{T}_{\phi\wedge\psi} = \lambda x.(\mathbf{T}_\phi x)(\mathbf{T}_\psi) \\ \mathbf{M}_{\mathtt{true}\oplus(\phi,\psi)} = \lambda x.(\mathbf{T}_\phi x)(\mathbf{M}_\psi) & \mathbf{T}_{\mathtt{true}\oplus(\phi,\psi)} = \lambda x : (\lambda x : \mathtt{true}.\mathtt{true}).\mathbf{T}_\psi(x\mathbf{M}_\phi) \\ \mathbf{M}_{c\oplus\mathtt{true}} = c & \mathbf{T}_{c\oplus\mathtt{true}} = \lambda x : c.\lambda y.y \end{array} \tag{19}$$

Theorem 2.4 *$\mathcal{D}(A)_\star$ is fully abstract for $\lambda\mathtt{cc}(A)$.*

Via parallel composition, both parallel or, and the parallel convergence tester of [AO89] can be programmed.

2.4 V revisited

The operational behavior of V leaves something to be desired, since every use of a process would spawn an infinite computation, only a portion of which would usually be useful. It is rather easy to repair this however, by using some derived laws involving the $\mathbf{V}$ combinator to produce new "symbolic" evaluation rules for expressions involving the $\mathbf{V}$ combinator. These rules exploit the idea that

$\mathbf{V}(M \wedge N)P$ can be thought of as two processes M and N running (fairly) in parallel on a shared store P.

The complete set of about half-a-dozen laws will be presented in the final version of the paper, and proven to be sound.

2.5 Other issues

Typed λcc. It is straightforward to develop a typed version of λcc. Each base type is taken to be an ω-algebraic lattice, and the parallel composition operations at a base type can be easily lifted up to higher functional types. The finite elements at the higher-types can obviously be generated from the finite elements of the base types. I believe (but have not checked in detail) that a similar full-abstraction result should hold.

3 Disjunctive λcc

To obtain non-determinism, agents are allowed to be disjunctive. We add to the syntax the production:

$$A ::= A \vee A \tag{20}$$

Intuitively, an agent to normalize $(A \vee B)$, A and B are normalized separately, and the (possibly disjunctive) information *common* to both is returned. First we extend the definition of "value" by adding the production $K ::= K \vee K$. The rules added to the definition of the relation $\Downarrow$ are:

$$\frac{M \Downarrow (M_1 \vee M_2) \quad (M_1\ N \vee M_2\ N) \Downarrow K}{M\ N \Downarrow K} \qquad \frac{M_1 \Downarrow K_1 \quad M_2 \Downarrow K_2}{(M_1 \vee M_2) \Downarrow (K_1 \vee K_2)} \tag{21}$$

Note the similarity with conjunction. A difference lies in the treatment of the entailment relation $\mapsto$ (see Appendix B): essentially, $(M \vee N) \mapsto u$ iff $M \mapsto u$ and $N \mapsto u$.

The semantic domain is now taken to be the least solution (in the category of lattices) of: $E \cong \mathcal{P}_S(C + E \rightarrow E)$. Non-determinism is handled by the union operation of the power-domain. As in [JSS91], this kind of non-determinism is angelic because it is *failure* avoiding (that is, the equation $A = \mathtt{false} \vee A$ is true in the model) not because it is deadlock avoiding.

To get full abstraction, the key issue again is to ensure that all finite elements in the semantic domain are finitely denotable. But for this we do not have to add to the power of the permissible ask operations, for $\lambda x : (u_0 \vee \ldots u_{k-1}).M$ is semantically identical to $(\lambda x : u_0.M \wedge \ldots \wedge \lambda x : U_{k-1}.M)$.

4 From information systems to constraint systems

4.1 Constraint systems

A V-constraint system (for V an unstructured infinite set of objects called variables) is essentially taken to be a structure $(D, \vdash)$ where D is a ranked alphabet of tokens, and $\vdash$ is a finitary entailment relation on *constraints*, which are of the form $(d, \langle X_1, \ldots, X_n \rangle)$, for $d \in D$ with arity n, and $X_1, \ldots, X_n \in V$. The entailment relation must satisfy a genericity condition: $u \vdash e$ iff for all variable-renamings μ, $\mu u \vdash \mu e$. (The application of a substitution to a constraint, and to sets of constraints is defined in the obvious way.) An operation $\exists X$ is defined that takes a set s of constraints to the subset consisting of those constraints c such that for every variable Y, $[Y/X]c \in s$. A variable X is said to be independent of s if $\exists X s = s$, and dependent otherwise. $|(D, \vdash)|$, the set of *elements* of the constraint system, is taken to be the collection of all sets of constraints closed under entailment, and dependent on only finitely many variables. As morphisms we take the approximable maps between constraint systems. It is straightforward to show the resulting structure forms a category that is closed under lifting, products, sums, function-space and the upper and lower- power-domains, *all* defined in a manner exactly parallel to the definitions for information systems. Furthermore, the techniques in [LW84] for constructing a "large" cpo of information systems and solving recursive domain equations via a least fixed point calculation, carry over exactly into this richer setting.

4.2 Constraints over an information system

We now show how to generate certain kinds of constraint systems from an information system. Let $\mathcal{D} = (D, \vdash_D)$ be an information system. The tokens for the V-constraint system $\mathcal{L}(D) = (E, \vdash_{\mathcal{L}})$ are given by:

$$e ::= X : d \mid e \wedge e \mid e \Rightarrow e \tag{22}$$

for $d \in D$ and $X \in \texttt{Var}$. The entailment relation $\vdash_{\mathcal{L}}$ is defined from $\vdash_D$ in the obvious way: $\{e_0, \ldots, e_{n-1}\} \vdash_{\mathcal{L}} e$ if every valuation θ (mapping from variables to $|D|$) realizing each of the e_i, realizes e. (θ realizes $X : d$ if $d \in \theta(X)$; it realizes $e \wedge e'$ if it realizes e and e', and it realizes $e \Rightarrow e'$ if it realizes e' whenever it realizes e.)

The existential quantifier has already been defined for an arbitrary constraint system above. Other operations on $|\mathcal{L}(D)|$ are also of interest in what follows. For every finite element of the domain u, we define the approximable relation $(u \rightarrow \_) \subseteq pE \times E$ by $v(u \rightarrow \_)e$ iff $(u \Rightarrow v) \vdash_E e$. Finally, we define the approximable map $(X \textbf{ where } \_) : pE \times D$ (for X a variable) by: $u\ (X \textbf{ where } \_)\ d$ iff $\{a \mid X : a \in u\} \vdash_D d$.

Members of $|\mathcal{L}(D)|$ can be denoted by terms involving free variables and function-symbols interpreted by continuous functions on D. For example, in the domain $\mathcal{L}(D \to E)$, the element $X : (MN)$ consists of tokens of the form $(M : (u, v), N : u') \Rightarrow X : v'$ for u, v, u', v' finite subsets of tokens in D and E respectively such that $u' \vdash_D u$ and $v \vdash_E v'$.

In general, $\mathcal{L}$ provides a recipe for constructing ask-and-tell constraint systems, in which (some) limit elements may be finitely denotable. The associated constraint solver must be powerful enough, of course, to determine when a (finite) conjunction of limit elements entails a basic token (since operationally one views a limit element as a possibly infinite generator of tokens that are true of it). Note that if basic constraints can be told (as we have assumed) then the only constraints that can be asked effectively are basic constraints. It must be kept in mind that the end-user would often prefer to see a symbolic representation of limit elements (e.g. $X \geq Y + Z$) rather than the infinite set of tokens that it represents (e.g., $(Y : [1, 1] \wedge Z : [1, 1]) \Rightarrow X : [2, \infty], \ldots$).

Thus there is a general idea here of generating constraint systems uniformly from information systems.

5 First-order λcc

The domain of denotations is given by:

$$D \cong A + \mathcal{L}(D) + (D \to D) \tag{23}$$

for some underlying information system A.

In the following, we shall follow the convention of syntactically distinguishing between λ-variables, which shall begin with a lower-case letter, and $\exists$-variables, which shall begin with an upper-case letter.

The abstract syntax of programs is given by:

$$\begin{array}{lll} M ::= & a & \text{— } A \\ & \mid X : T \mid \exists X.M \mid X \textbf{ where } M \mid u \to M & \text{— } \mathcal{L}(D) \\ & \mid x \mid \lambda x : u.M \mid M\ M & \text{— } D \to D \\ & \mid M \wedge M & \end{array} \tag{24}$$

where the terms T are built up using the same set of productions as M (with all occurrences of the meta-variable M replaced by T), augmented with the production $T ::= X$ (a symbolic variable is a term). The productions for finite programs (u) and finite terms (v) are obtained uniformly from those for M and T respectively, by replacing all occurrences of M with u and T with v, and by replacing the productions $M ::= x$ and $T ::= x$ by $u ::=$ `true` and $v ::=$ `true` respectively. The interpretation of the new constructs is intended to be that given in the previous section.

5.1 Denotational semantics

All the rules in Table 1 carry-over to the first-order λcc setting, provided that the appropriate "re-formatting" of formulas of the domain logic is done. The new operations are defined by:

$$\frac{t \vdash \phi}{X : t \vdash \mathtt{true} \oplus X : \phi \oplus \mathtt{true}} \qquad \frac{d \vdash X : \phi}{X \textbf{ where } d \vdash \phi}$$
$$\frac{d \vdash \mathtt{true} \oplus \psi \oplus \mathtt{true}}{u \rightarrow d \vdash \mathtt{true} \oplus \phi_u \rightarrow \psi \oplus \mathtt{true}} \qquad \frac{M[Y/X] \vdash \phi}{\exists X.M \vdash \phi} \tag{25}$$

In the last rule, Y is not free for X in ϕ. The characteristic formula ϕ_u of a finite agent u — as used above — is defined inductively by:

$$\begin{array}{ll} \phi_a = (a \oplus \mathtt{true} \oplus \mathtt{true}) & \phi_{(u \wedge u')} = \phi_u \wedge \phi_{u'} \\ \phi_{X:v} = \mathtt{true} \oplus X : \phi_v \oplus \mathtt{true} & \phi_{\exists X.h} = \mathtt{true} \oplus \exists X.\phi_h \oplus \mathtt{true} \\ \phi_{\lambda x:u.u'} = \mathtt{true} \oplus \mathtt{true} \oplus (\phi_u, \phi_{u'}) & \phi_{\mathtt{true}} = \mathtt{true} \end{array} \tag{26}$$

5.2 Operational semantics

The following new rules for $\Downarrow$ are added to those in Table 1:

$$\frac{d \Downarrow_t K}{X : d \Downarrow_t X : K} \qquad \frac{d \Downarrow K}{X \textbf{ where } d \Downarrow X \textbf{ where } K} \qquad \frac{d \Downarrow K}{\exists X.d \Downarrow \exists X.K} \tag{27}$$

Above, the definition of $\Downarrow_t$ is identical to that of $\Downarrow$ with the additional axiom $X \Downarrow_t X$.

Note that there is no evaluation rule for an ask-agent $u \rightarrow M$. Such an agent can, however, participate in entailing a primitive constraint, provided that its assumption (u) is discharged. This is described by the additional axioms for the first-order case in Appendix B — essentially, the connective "$\rightarrow$" is interpreted as intuitionistic implication.

The results in the propositional case carry over to the richer setting considered here. Specifically, we have that the denotational semantics is computationally adequate and fully abstract for the operational semantics.

6 Specific λcc languages

Our first example — the higher-order version of Kahn networks — is being omitted due to shortage of space.

λcc(Herbrand). The underlying constraint system C is taken to include a flat lattice of atomic objects (A), and finite products of the domain H being constructed:

$$\begin{array}{l} H \cong C + \mathcal{L}(H) + (H \rightarrow H) \\ C \cong A + (H \times H) \end{array} \tag{28}$$

Thus, to the abstract syntax of programs is added:

$$h ::= a \mid \langle h, h\rangle \mid \textbf{first } h \mid \textbf{rest } h \quad — (H \times H) \tag{29}$$

This allows for a smooth and uniform representation of *constraints* on programs as data *for* programs: a partially defined object

$$\alpha \equiv \lambda x : (\lambda y : \texttt{true.true}).M(M\ x)$$

can be treated as (1) a function (hence applied to other data–$(\alpha\ R)$), (2) as a constraint in itself (hence imposed on a variable, $(R : \alpha)$), (3) as data, (hence stored in (communicable) data-structures, $R : \langle\alpha, \_\rangle$). All this, while its definition is simultaneously being augmented, as can happen in conjunctive contexts in which other agents are imposing constraints on M.

A simple example program is the usual `foldr` program:

$$\begin{array}{l}\texttt{Foldr} : \lambda\texttt{fun}.\lambda\texttt{l}.\\ \quad \mathbf{Y}(\lambda\texttt{foldr}_1.\lambda\texttt{x.R } \textbf{where } \exists\texttt{Q}.(\texttt{Q} : \texttt{x}\\ \qquad\qquad\qquad\qquad \wedge (\texttt{nullQ}) \rightarrow \texttt{R} : \texttt{l}\\ \qquad\qquad\qquad\qquad \wedge (\texttt{consQ}) \rightarrow \texttt{R} : (\texttt{fun } (\textbf{first } \texttt{Q})(\texttt{foldr}_1\ (\textbf{rest } \texttt{Q}))))).\end{array}$$

(Here `(null Q)` and `(cons Q)` are being used as evident shorthands for the corresponding finite tests.) The usual append program can be obtained by specializing with *cons*:

$$\texttt{Append} : \texttt{Foldr}(\lambda\texttt{x}.\lambda\texttt{y}.\langle\texttt{x}, \texttt{y}\rangle)$$

A typical application would be the agent

$$\texttt{R} : \texttt{Append}\langle 1, \langle 2, \langle\rangle\rangle\rangle\langle 3, \langle\rangle\rangle$$

running in parallel with the previous constraint. Note that the function supplied as argument does *not* have to be supplied all at once—it could contain symbolic variables on which constraints are simultaneously being imposed:

$$\texttt{Map\_M} : \texttt{Foldr}(\lambda\texttt{x}.\lambda\texttt{y}.\langle(\texttt{M x}), \texttt{y}\rangle)$$

An invocation $\texttt{Result} : \texttt{Map\_M}\langle 1, \langle 2, []\rangle\rangle$ will entail the constraint $\texttt{Result} : \langle(\texttt{M 1}), \langle(\texttt{M 2}), []\rangle\rangle$, without knowing anything about `M`. Subsequently, any information produced on `M` (by a conjunctively executing agent) will be transmitted to this value, causing further evaluation.

In a similar manner, we have explored the design of λcc(FS), λcc over *feature-structures*, which can be thought of as particular kinds of record structures (see, for example, [AKN86]) and of λcc(FD), the higher-order version of the cc language over the constraint system of "finite domains" [HS91]. More details will be contained in the full paper.

In a similar manner, we have explored the design of λcc(FS), λcc over *feature-structures*, which can be thought of as particular kinds of record structures (see, for example, [AKN86]) and of λcc(FD), the higher-order version of the cc language over the constraint system of "finite domains" [HS91].

7 Conclusions and Future work

This work establishes simple foundations for higher-order constraint programming languages. This is done by extending the λ-calculus with notions of parallel composition, and ask-abstractions, and disjunction (to get angelic non-determinism), by introducing the notion of indexed information systems (constraint systems), by showing how to design some very powerful programming languages using these ideas, and by establishing some basic connections between the operational and denotational semantics for these languages. In particular, it is shown that there is a close harmony between such programming languages and the lattice-theoretic information systems of Scott. Throughout the logical nature of the programming language is emphasized (e.g., in the presentation of the operational and denotational semantics).

The single biggest topic that has not been tackled above is the development of the notion of *indeterminate* higher-order cc languages. Models for the first-order theory are developed in [SRP91] and [dBP90], but much work remains to be done to extend these ideas to the higher-order setting. In particular, it would be interesting to connect the theory of "sequential" closure operators (viz., trace operators) developed in [SRP91] with the notions of "stable" functions developed by Berry et al (see, e.g.,[NR85]).

Another task to be tackled is the integration of *polymorphism* into the programming language. One approach would be to examine the relationship between the roles played by closures in the models of the polymorphic λ-calculus and the first-order cc-calculus.

I have recently realized that computation in cc languages can be viewed as deduction in a fragment of intuitionistic logic, though the setting is somewhat different from that in Dale Miller's work [MNS87]. There is a connection here with higher-order (untyped) intuitionistic logics that should be examined carefully.

Jointly with Spiro Michaylov, a prototype implementation is underway in Prolog based on extending the Krivine machine presented in [ACCL90]. However, not much of the power of Prolog can be used directly. For example, even an implementation of λcc(Herbrand) cannot directly use Prolog's unification mechanism, since Prolog does not have any notion of scoped execution, i.e. the possibility of representing multiple stores, each of which talk of possibly different constraints on the same variables.

Acknowledgements. This work has been long in the making. I owe a special debt to Mark Miller and Eric Dean Tribble who have actively pushed me since fall 1987 to develop a higher-order version of the cc paradigm. The basic ideas underlying the current treatment were in place in May 1991. Work since then has focussed on tying up the connection with the operational semantics. I have benefited considerably from discussions with Dana Scott, Prakash Panangaden, John Lamping, Martin Rinard, Spiro Michaylov, Wm. Cook, Ken Kahn, Ugo Montanari, Francesca Rossi and Gordon Plotkin.

References

[Abr91] Samson Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 1991.

[ACCL90] M. Abadi, P.L. Curien, L. Cardelli, and J.-J. Levy. Explicit substitutions. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1990.

[AK91] Hassan Ait-Kaci. Scoop: Solving constraints with object-oriented programming. Slides from talk at Workshop on Constraint Programming at the International Conference on Logic Programming, Paris, June 1991, 1991.

[AKN86] Hassan Ait-Kaci and Roger Nasr. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[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.

[Bar84] H.P. Barendregt. *The Lambda Calculus–Its syntax and semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[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.

[Blo90] Bard Bloom. Can LCF be topped? *Information and Computation*, 87:264–301, 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.

[Bou91] Gerard Boudol. Lambda-calculii for (strict) parallel functions. Technical report, INRIA Sophia-Antipolis, 1991. To appear in Information and Computation.

[CKW89] Weidong Chen, Michael Kifer, and David S. Warren. Hilog: A first-order semantics for higher-order logic programming constructs. In *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114, October 1989.

[dBP90] Frank de Boer and Catuscia Palamidessi. A fully abstract model for concurrent constraint programming. June 4 1990.

[Gal87] Jean H. Gallier. *Logic for Computer Science*. Harper and Row, 1987.

[GL90] Maurizio Gabbrielli and G. Levi. Unfolding and fixpoint semantics of concurrent constraint logic programs. Technical report, University of Pisa, 1990.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proof and Types*. Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[GMS89] Haim Gaifman, Michael J. Maher, and Ehud Shapiro. Reactive behavior semantics for concurrent constraint logic programs. In *North American Logic Programming Conference*. MIT Press, October 1989.

[Hen89] Pascal Van Hentenryck. *Constraint satisfaction in logic programming*. Logic Programming Series. MIT Press, 1989.

[Hoa89] C.A.R. Hoare. A theory of conjunction and concurrency. Oxford PRG, May 1989.

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

[JP91] Radha Jagadeesan and Keshav Pingali. Abstract semantics for higher-order functional language with logic variables. Forthcoming Technical Report, 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.

[LW84] Kim G. Larsen and Glynn Winskel. Using information systems to solve recursive domain equations effectively. In G. Kahn, D.B.MacQueen, and G.Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia Antipolis*, LNCS 173. Springer Verlag, 1984.

[Mey88] Albert Meyer. Semantical paradigms: Notes for an invited lecture. In *IEEE LICS*, pages 236–253, 1988.

[MNS87] Dale Miller, Gopalan Nandathur, and Andre Scedrov. Hereditary Harrop Formulas and Uniform Proof Systems. In *Symposium on Logic in Computer Science*, pages 98 — 105. IEEE Computer Society, IEEE Computer Society Press, June 1987.

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

[MR91] Ugo Montanari and Francesca Rossi. True concurrency semantics for concurrent constraint programminmg. In Vijay Saraswat and Kazunori Ueda, editors, *Proc. of the 1991 International Logic Programming Symposium*, 1991.

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

[NR85] Maurice Nivat and John C. Reynolds, editors. *Algebraic methods in semantics*, chapter Full abstraction for sequential languages: the state of the art. Cambridge University Press, 1985.

[Plo72] Gordon Plotkin. A set-theoretical definition of application. Memorandum MIP-R-95, 17th March, 1972.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PSSS91] Prakash Panangaden, Vijay Saraswat, Phil Scott, and Robert Seely. What is a constraint system? a hyperdoctrinal account. Forthcoming Technical Report, 1991.

[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.

[Sco70] Dana S. Scott. Outline of a mathematical theory of computation. In *Proc. of the Fourth Annual Princeton Conf. on Info. Sciences and Systems*, pages 169–176, 1970.

[Sco76] Dana S. Scott. Data types as lattices. *SIAM*, 5(3):522–587, 1976.

[Sco80] Dana S. Scott. *The Kleene Symposium*, chapter Lambda-calculus: some models, some philosophy, pages 223–265. North-Holland, 1980.

[Sco82] Dana S. Scott. Domains for denotational semantics. In *Proceedings of ICALP*, 1982.

[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.

[SRH91] Vijay A. Saraswat, Fransesca Rossi, and Pascal Van Hentenryck. Towards a general framework for constraint programming. Technical report, System Sciences Laboratory, Xerox PARC, February 1991.

[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.

[Tho86] Simon Thompson. Laws in Miranda. In *Proceedings of the 4th ACM International Conference on LISP and Functional Programming*, August 1986.

[Tho90] Bent Thomsen. In *Conf. Proceedings of Seventeenth POPL*, 1990.

[Tur85] David Turner. Miranda: a non-strict functional language with polymorphic types. In *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Sprinter Lecture Notes in Computer Science*, 1985.

[Vic88] S.J. Vickers. *Topology via Logic.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

A Domain theory in logical form

The following definitions are based on those in [Abr91]. We have adapted his development for **SFP** to the category ω**LAT** (and **ConstSys**) of interest in this paper. The type-expressions of interest to us are given by:

$$\sigma ::= \mathbf{1} \mid \sigma \times \tau \mid \sigma \rightarrow \tau \mid \sigma \oplus \tau \mid (\sigma)_\perp \mid \mathcal{P}_S(\sigma) \mid \mathtt{rec}\, t.\sigma \tag{30}$$

for t ranging over type variables, and σ, τ over type expressions.

A *lattice domain logic* for σ is a structure

$$\mathcal{A}(\sigma) = (A(\sigma), \leq_\sigma, =_\sigma, \mathtt{true}_\sigma, \wedge_\sigma) \tag{31}$$

where $A(\sigma)$ is a set of formulae (formed per the syntax rules given below), $\leq_\sigma$ and $=_\sigma$ are the relations of logical implication and equivalence between formulas, $\wedge$ is the connective for conjunction, and $\mathtt{true}$ is a constant for nullary conjunction.

The formation rules, axioms and inference rules for the type-constructors used in this paper are given below:

Formation rules.

$$\phi_\sigma ::= \mathtt{true}_\sigma \mid \phi_\sigma \wedge \phi_\sigma$$

$$\begin{array}{ll} \phi_{\sigma\times\tau} ::= \phi_\sigma \times \tau_\sigma & \phi_{\sigma\to\tau} ::= \phi_\sigma, \phi_\tau \\ \phi_{\sigma\oplus\tau} ::= \phi_\sigma \oplus \mathtt{true} \mid \mathtt{true} \oplus \phi_\tau & \phi_{\sigma_\perp} ::= (\phi_\sigma)_\perp \\ \phi_{(\mathtt{rec}\ t.\sigma)} ::= \phi_{\sigma[\mathtt{rec}\ t.\sigma/t]} & \end{array} \tag{32}$$

Axioms. These are of two types. The logical axioms give $(A(\sigma), \leq)$ the structure of a meet-semilattice, with a weakest element $\mathtt{true}$. The *type-specific* axioms are given here in a polymorphic style, with the types elided:

$$\wedge_{i\in I}(\phi_i \times \psi_i) = (\wedge_{i\in I}\phi_i) \times (\wedge_{i\in I}\psi_i) \quad (\phi, \wedge_{i\in I}\psi_i) = \wedge_{i\in I}(\phi, \psi_i) \tag{33}$$

$$(\wedge_{i\in I}\phi) \oplus \mathtt{true} = \wedge_{i\in I}(\phi \oplus \mathtt{true}) \quad (\mathtt{true} \oplus \wedge_{i\in I}\phi) = \wedge_{i\in I}(\mathtt{true} \oplus \phi) \tag{34}$$

$$(\phi \wedge \psi)_\perp = \phi_\perp \wedge \psi_\perp \quad \Box \wedge_{i\in I} \phi_i = \wedge_{i\in I}\Box\phi_i \quad \frac{\phi' \leq \phi, \psi \leq \psi'}{(\phi, \psi) \leq (\phi', \psi')} \tag{35}$$

$$\frac{\phi \leq \psi}{\phi_\perp \leq \psi_\perp} \quad \frac{\phi \leq \psi}{\Box\phi \leq \Box\psi} \quad \frac{\phi \leq \psi}{\mathtt{true} \oplus \phi \leq \mathtt{true} \oplus \phi} \quad \frac{\phi \leq \psi}{\phi \oplus \mathtt{true} \leq \phi \oplus \mathtt{true}} \tag{36}$$

We also give the rules for the $\mathcal{L}(\sigma)$ construction:

$$\phi_{\mathcal{L}(\sigma)} ::= X : \phi_\sigma \mid (\phi_{\mathcal{L}(\sigma)} \to \phi_{\mathcal{L}(\sigma)}) \tag{37}$$

The following axioms specify the interaction of the generators with the logical structure of formulas:

$$\phi_1 \to (\phi_2 \to \psi) = (\phi_1 \wedge \phi_2 \to \psi) \tag{38}$$

$$\phi \to \wedge_{i\in I}\psi_i = \wedge_{i\in I}(\phi \to \psi_i) \tag{39}$$

$$\frac{\phi' \leq \phi \quad \psi \leq \psi'}{\phi \to \psi \leq \phi' \to \psi'} \tag{40}$$

$$(X : \phi) \wedge (X : \psi) = X : (\phi \wedge \psi) \tag{41}$$

B Entailment rules for λcc-languages

The rules are given in the style of the sequent calculus. A *sequent* (for current purposes) is a formula:

$$M_0, \ldots, M_{k-1} \mapsto u$$

where the M_i are λcc terms, u is a *finite* λcc-term. Intuitively, the rules formalize the idea that any information derivable from u is derivable from the conjunction of $M_0, \ldots, M_{k-1}$.

As far as the standard logical connectives $\wedge, \to$ and quantifiers (we only treat $\exists$) are concerned, the treatment is standard. The novelty is in the treatment of the extra structure (abstraction, application, **where**, _ : _ etc.). For background information, the reader should consult textbooks such as [GLT89, Gal87].

As usual, proof figures are divided into several groups. First, we present the rules for propositional λcc; and then we give the additional rules for each extension.

Structural rules. These are the standard exchange, weakening and contraction rules, [GLT89, P.32–33].

Identity rules. Here we also include rules from the underlying constraint system.

$$u \mapsto u \quad \text{(Identity)} \qquad M \mapsto \mathtt{true} \quad \text{(Truth)} \qquad \frac{B \mapsto_A a}{\underline{A} \mapsto a} \quad \text{(Primitive Rule)}$$

Conjunction. These rules are standard, as given e.g. by Rules $L1\wedge, L2\wedge, R\wedge$ in [GLT89, P.32–33].

Abstraction and application. There is a single rule for abstraction, which introduces abstraction simultaneously on the left and the right.

$$\frac{M_0[u/x], \ldots, M_{k-1}[u/x] \mapsto u' \qquad u \mapsto u_i \quad (i < k)}{\lambda x : u_0.M_0, \ldots, \lambda_x : u_{k-1}.M_{k-1} \mapsto \lambda x : u.u'} \quad \text{(Abstraction)}$$

There is no right-introduction rule for application, since formulas on the right are not taken to be closed under application:

$$\frac{M \mapsto \lambda x : u.u' \qquad N \mapsto u}{MN \mapsto u'} \quad \text{(Application)}$$

Disjunctive λcc. The standard intuitionistic rules for disjunction (e.g., $L\vee$, $R1\vee$, $R2\vee$, *ibid.*) are added.

First-order λcc. Existential quantification is axiomatized via the usual intuitionistic sequent calculus rules $L\exists$, $R\exists$ *ibid.*. (The only minor difference is that in Rule $R\exists$, a must be taken to be a variable not free in A, rather than a constant.) The ask-arrow is axiomatized via the entailment rules for the intuitionistic sequent calculus ($L \Rightarrow, R \Rightarrow$). Finally, the rules for **where** clauses and basic constraints are straightforward:

$$\frac{M \mapsto X : u}{\underline{A}, X \textbf{ where } M \mapsto u} \quad \textbf{(where)}$$

$$\frac{t_0, \ldots, t_{k-1} \mapsto u}{\underline{A}, X : t_0, \ldots, X : t_{k-1} \mapsto X : u} \qquad \frac{\underline{A}, X : t[t'/Y] \mapsto u}{\underline{A}, Y : t', X : t \mapsto u} \quad \text{(Basic constraints)}$$