Testing concurrent systems: An interpretation of intuitionistic logic

Radha Jagadeesan[1], Gopalan Nadathur[2], and Vijay Saraswat[3]

[1] School of CTI, DePaul University
[2] Digital Technology Center and Department of CSE, University of Minnesota
[3] IBM T.J. Watson Research Center

Abstract. We present a natural confluence of higher-order hereditary Harrop formulas (HH formulas), Constraint Logic Programming (CLP, [JL87]), and Concurrent Constraint Programming (CCP, [Sar93]) as a fragment of (intuitionistic, higher-order) logic. This combination is motivated by the need for a simple executable, logical presentation for static and dynamic semantics of modern programming languages. The power of HH formulas is needed for higher-order abstract syntax, and the power of constraints is needed to naturally abstract the underlying domain of computation. Underpinning the combination is a sound and complete operational interpretation of a two-sided sequent presentation of (a large fragment of) intuitionistic logic in terms of *behavioral testing of concurrent systems*. Formulas on the left hand side of a sequent style presentation are viewed as a system of concurrent agents, and formulas on the right hand side as *tests* against this evolving system. The language permits recursive definitions of agents and tests, allows tests to augment the system being tested and allows agents to be contingent on the success of a test. We present a condition on proofs, *operational derivability* (OD), and show that the operational semantics generates only operationally derivable proofs. We show that a sequent in this logic has a proof iff it has an operationally derivable proof.

1 Introduction

The investigations in this paper are driven by an interest in logical frameworks for program manipulation. This interest has a twofold motivation.

First, the recent emergence of extremely successful program development environments such as Eclipse [ecl], has highlighted the power of advanced program manipulation techniques (such as refactorings, [FTK04]), particularly for modern, concurrent, object-oriented programming languages such as JAVA. At the same time the complexity of internal programming APIs in Eclipse – and the brittleness in extending them to languages other than JAVA – has highlighted the importance of developing a coherent conceptual framework for programs that manipulate programs. The holy grail of this work is to make it possible for end-users to define their own refactorings. This requires that there be a simple declarative framework in which the user can compose the refactoring, and there be a way to determine if the proposed refactoring is semantics-preserving.

A second motivation for such a framework is to ease the task of writing and extending compilers. Object-oriented (OO) compiler frameworks such as Polyglot [NCM03] ease some of the burden of compiler-writing for new OO languages. A compiler-writer has to provide a parser for the new language, define new Abstract Syntax Tree (AST) nodes to represent the parsed program, and implement different passes for various compiler tasks such as disambiguation, type-checking, translation to an intermediate representation (IR), static analysis and code-generation. Several of these passes can be thought of as generating and checking constraints on the AST. However this structure is hidden in the current conceptualization in terms of *procedural* code to implement "visitors" that build up the context for a node as they traverse the path from the root to the node, and rewrite the AST based on this context. Thus, it becomes difficult to extend the underlying IR and perform new analyses for new programming constructs. Similar difficulties have been reported in other frameworks that aim to make it easy for programmers to specify and plug new optimization rules into a compiler, while guaranteeing that these rules preserve program correctness [LMRC05].

This leads us to enunciate the following desiderata for the kind of framework we investigate. Below we will find it convenient to distinguish the (hypothetical) *object language* $\mathcal{O}$ and the programming language $\mathcal{F}$ to be used to write programs that manipulate $\mathcal{O}$-programs.

Programs as data. $\mathcal{F}$ should be able to express programs in modern languages (e.g. Java, Co-Array Fortran, Prolog) as data in such a way that object programs can be decomposed into their constituent parts and new object programs can be created from program parts, while respecting scoping constructs. A central requirement is that $\mathcal{O}$ scoping constructs (such as method parameter declaration, local variable introduction) subject to "alpha renaming" should in fact be represented by $\mathcal{F}$ scoping constructs so that the programmer does not have to worry about the book-keeping involved in explicitly implementing alpha renaming, substitution, generating "new" constants etc. This is the idea of *higher-order abstract syntax* [PE88].

Constraint-based. There is a large body of work establishing the centrality of constraints to the static and dynamic analysis of programs e.g. [Hei92,OSW99,Pal95,PS94], [Aik99,PW98,RMR01]. Thus, the framework should support the compositional generation of constraints from program structures. Constraints may be simple (no embedded quantifiers) or polymorphic (universally and existentially quantified). Programs should be able to query these constraints and take further action (such as generating more constraints or checking more constraints), based on the success or failure of such a query. Furthermore, we demand *extensibility*. It should be possible to extend the constraint system with analysis-specific constraints with the same ease with which new analyses can be written.

Declarative. It should be possible to view $\mathcal{F}$ programs as logical formulas so that properties of these programs (such as: they preserve the semantics of the object program they are manipulating) can be established through logical reasoning involving other $\mathcal{F}$ programs (e.g. representing the static and dynamic semantics of O). The declarative framework should be expressive enough to allow the static and dynamic semantics of $\mathcal{O}$ to be expressed (and implemented) in terms of declarative rules over program structures in $\mathcal{F}$.

2 Basic Paradigm

In searching for a programming language framework for dealing with programs, it is natural to start with λProlog, and its underlying conceptual basis, higher-order hereditary Harrop formulas [MNPS91] (henceforth called HH). Consider the basic syntactic structure of definite clause programs. Starting with the base

$$\begin{array}{l} \textbf{(Agent)}\ D ::= \texttt{true} \mid D \wedge D \mid \forall x\, D \\ (\textbf{Test})\ \ G ::= \texttt{true} \mid G \wedge G \mid G \vee G \mid \exists x\, G \end{array} \tag{1}$$

we may obtain definite clause logic programming LP by adding

$$D ::= G \supset A \qquad\qquad G ::= A \tag{2}$$

That is, a program is formulated in terms of universally quantified implications, whose head contains an atom and whose body contains *goals*, which may be atomic or conjunctive, disjunctive or existential formulas. We assume a higher-order language so the arguments of atomic formulas may be (typed) lambda terms. To keep matters simple, we exclude quantification over predicates; this condition may be relaxed as in λProlog.

To LP, HH adds the notion of *universal* and *implicational goals*:

$$G ::= D \supset G \mid \forall x\, G \tag{3}$$

Computationally, implicational goals or *extensible tests* permit the extension of the current database of programs before answering a specific query. Universal goals permit the introduction of *scoped constants*. These additional constructs complement the use of typed lambda-calculus to represent object level binding notions with devices for realizing recursion over such structure [NM98]. The practical benefits of these capabilities in syntax manipulation have been discussed in several places in the literature.

A limitation of λProlog for the applications of interest is the absence of a treatment of constraints; as we have noted earlier, constraint systems find many uses in static and dynamic analyses over program structure. This leads us to the integration of constraint programming with HH. The addition of constraints to goals and agents has been proposed by [FFL03,GDN04,LNRA01] through the further syntax rules:

$$D ::= c \qquad\qquad G ::= c \tag{4}$$

An implicational goal (e.g. $c \supset G$) can be used to add constraints to the *store* (the LHS); a constraint goal c may check that a constraint follows from the store.

The language with syntax described by rules (1)–(4) still has a shortcoming: it does not permit (recursive) computations on the LHS of a sequent. For example, consider the computation of the goal $D \supset (G_1 \wedge G_2)$ in the context of an LHS given by Λ. Solving this goal requires the addition of D to Λ. In HH, the consequences that emerge from the addition of D to Λ must be computed separately while solving G_1 and G_2. Permitting recursive computation in the LHS could eliminate this redundancy: the consequences can then be computed once, and used in showing both G_1 and G_2. The following excerpt from the type checking of Java programs in the context of a class hierarchy is an example of the utility of this idea.

Example 1 (Java type checking). The type checking of a method body in a JAVA program must be done in the context of type assertions generated by examining the classes referenced in the code. These assertions are built using predicates such as *extends* between class names that captures the subtype relationship. It is desirable that the parsing of referenced classes and the elaboration of type assertions (based on the inheritance hierarchy, method signatures, field signatures, etc) be done only once. Thus, conceptually, one wishes to define:

$$\begin{array}{l} \forall \mathit{ClassName}\ \forall \mathit{Code} \\ \quad (((\mathit{parse}\ \mathit{ClassName}\ \mathit{Code}) \wedge \\ \quad\ ((\mathit{referencedClassTypes}\ \mathit{Code}) \supset (\mathit{typed\_code}\ \mathit{Code}))) \\ \qquad \supset (\mathit{typed\_class}\ \mathit{ClassName})) \end{array}$$

The definition of the predicate for *typed_code* assumes that the type information for each referenced class is already available in the store and may simply be queried (e.g. using the constraint `subType`):

$$\begin{array}{l} \forall \mathit{LExp}\ \forall \mathit{RExp}\ \forall \mathit{LT}\ \forall \mathit{RT} \\ \quad ((\mathit{isType}\ \mathit{RExp}\ \mathit{RT}) \wedge (\mathit{isType}\ \mathit{LExp}\ \mathit{LT}) \wedge (\mathit{subType}\ \mathit{RT}\ \mathit{LT})) \\ \qquad \supset (\mathit{typed}\ (\mathit{assign}\ \mathit{LExp}\ \mathit{RExp}))). \end{array}$$

Thus, we expect *referencedClassTypes* to be a user-defined (agent) predicate here that operates on the LHS of a sequent and that walks the AST *Code*, determining referenced classes and for each such class generating type assertions based on the type hierarchy. Running the agent *referencedClassTypes Code* to quiescence on the LHS would thus elaborate the type information in *Code* once and for all, sharing this computation among all subsequent RHS queries.

This motivates us to take the fundamental step underlying this paper: combining the power of HH with CCP. CCP is organized around the notion of (deterministic) agents working together in parallel to produce constraints on a shared store.

$$\begin{array}{l} \textbf{(Agent)}\ D ::= \texttt{true} \mid c \mid D \wedge D \mid E \mid G \supset D \mid E \supset D \mid \exists x\, D \\ (\textbf{Test})\ \ G ::= \texttt{true} \mid c \mid G \wedge G \end{array}$$

One views `true` as the vacuous agent, c as the agent which adds the constraint c to the *store*, $D_1 \wedge D_2$ as the parallel composition of D_1 and D_2, E (an atomic formula) as a recursively defined agent, (whose rules of behavior are specified by the formulas $E \supset D$), $G \supset D$ as a *deep guard ask agent* which checks whether the store entails G, and if so, reduces to D, and $\exists x\, D$ as the agent that introduces a new local variable x and then behaves like D. [LS93] has shown that the logical view of CCP (in the subcase of flat guards $c \supset D$) corresponds to *computation on the left* in a sequent based presentation. Conceptually the purpose of the computation is to determine the (strongest) set of constraints c (on the variables in D) that follow from D. Thus on termination we have a c and a proof tree for $D \vdash c$ such that for any other c_1, if $D \vdash c_1$ then $c \vdash c_1$.

This motivates us to add to the syntax rules (1)–(4) the rules:

$$D ::= E \mid G \supset D \mid E \supset D \mid \exists x\, D \tag{5}$$

We call the resulting language framework λRCC (RCC for the sub-language with first-order terms). Notice that the second rule should be thought of as permitting fully recursive asks ("deep guards" in concurrent logic programming terminology), thus allowing a symmetric interplay between goals and agents (cf the production $G{::=}D \supset G$).

An important restriction in λRCC is that the vocabulary of predicate names used for goals, agents and constraints are pairwise disjoint—we refer to this as the *Disjoint Vocabulary* condition. The Rules (1)–(5) can be consolidated as:

$$\begin{array}{ll}\textbf{(Agent)} & D ::= \texttt{true} \mid c \mid E \mid D \wedge D \mid G \supset A \mid G \supset D \mid E \supset D \mid \exists x\, D \mid \forall x\, D \\ (\textbf{Test}) & G ::= \texttt{true} \mid c \mid A \mid G \wedge G \mid D \supset G \mid G \vee G \mid \exists x\, G \mid \forall x\, G\end{array} \tag{6}$$

Clearly this includes LP, HH, CLP and CCP. The results of this paper may be extended to support disjunctive agents as well; but we omit their treatment for lack of space.

3 Operational Semantics for λRCC

How should we understand computation in λRCC? We propose that *behavioral testing of concurrent systems* provides a suitable framework. Let us think of a configuration in our system as being given by a multiset of *predications* of the form (Λ, G) in which Λ is a multiset of D (agent) formulas. Informally, we would like to view such a pair as posing the question "Does the concurrent system Λ pass the test G?" We expect the operational semantics of the language to be described by a transition relation $\longrightarrow$ on configurations that allows us to address such a question in an incremental fashion. To indicate success, we introduce the configuration ϵ; thus the question (Λ, G) is considered to be one that has a successful answer iff $(\Lambda, G) \xrightarrow{\star} \epsilon$.

The testing notion is behavioral in the sense that it merely examines the behavior, i.e. the potential to produce certain results, treating the structure of the system as opaque. Even simple structural queries such as "Does the system contain the agent A?" are not permitted (thanks to the Disjoint Vocabulary condition). Permitting such queries would interfere with the understanding of goal-predicates and agent-predicates as recursive procedure calls. One would have to account for the possibility that a query A can be answered not only by unrolling A into the body G of a clause defining A but by the mere presence of the atom A on the LHS. Similarly, there is no possibility of formulating a query which is able to decompose the system into the parallel composition of two agents A_1 and A_2 and ask whether A_1 satisfies G_1 and A_2 satisfies G_2 (cf bunched implication logics [OP99]).

3.1 The underlying intuitions

Let us write $\Lambda \mid\vdash G$ (read: "Λ passes G" or "Λ has the potential to answer G") to represent the condition $(\Lambda, G) \xrightarrow{\star} \epsilon$.

Structural principles. A question to ask is: When should $\Lambda \mid\vdash c$ succeed? The operational interpretation of CCP suggests a natural answer: it should succeed iff it is possible for Λ to evolve in such a way that the resulting store entails c. Thus the question being asked is: does Λ have the *potential* to generate c? Even before we get specific about the evolution process, the viewpoint that it only serves to "actualize" potential leads to certain structural principles that our operational semantics should satisfy:

Potential preservation $(\Lambda, c) \xrightarrow{\star} (\Lambda', c)$ and $\Lambda \mid\vdash c$ implies $\Lambda' \mid\vdash c$.
Structural Rules $\Lambda \mid\vdash c'$ and $\Lambda, c' \mid\vdash c$ implies $\Lambda \mid\vdash c$; $\Lambda, D \mid\vdash c$ if $\Lambda, D, D \mid\vdash c$; $\Lambda, D_1, D_2 \mid\vdash c$ if $\Lambda, D_2, D_1 \mid\vdash c$; and $\Lambda, D \mid\vdash c$ if $\Lambda \mid\vdash c$.

Agent combinators. To address the issue of evolution itself, when should Λ, D pass a test c? This should happen if (a) Λ passes the test by itself or (b) D interacts with Λ in such a way that the system reaches a state in which c can be answered. To specify this precisely, we need *agent interaction rules*:

Vacuous agent $\Lambda, \mathtt{true} \mid\vdash c$ iff $\Lambda \mid\vdash c$.
Parallel agent $\Lambda, D_1 \wedge D_2 \mid\vdash c$ iff $\Lambda, D_1, D_2 \mid\vdash c$.
Recursive agent $\Lambda, E \mid\vdash c$ iff $\Lambda \mid\vdash c$ or there is a Λ' and a rule $E \supset D \in \Lambda'$ s.t. $((\Lambda, E), c) \xrightarrow{\star} ((\Lambda', E), c)$ and $\Lambda', E, D \mid\vdash c$.
Deep guard agent $\Lambda, G \supset D \mid\vdash c$ iff for some Λ': $((\Lambda, G \supset D), c) \xrightarrow{\star} ((\Lambda', G \supset D), c)$, and (i) $\Lambda' \mid\vdash c$ or (ii) $\Lambda', G \supset D \mid\vdash G$ and $\Lambda', D \mid\vdash c$.
Existential agent $\Lambda, \exists x\, D \mid\vdash c$ iff $\Lambda, D[i/x] \mid\vdash c$ for some new parameter i.
Universal agent $\Lambda, \forall x\, D \mid\vdash c$ iff $\Lambda, \forall x\, D, D[t/x] \mid\vdash c$.

The first two rules have already been discussed in conjunction with CCP. For recursive agents, Λ, E passes the test c if Λ passes the test by itself or if Λ can evolve to Λ' in which a rule $E \supset D$ is revealed such that Λ', E, D passes the test. The agent $\exists x\, D$ interacts with Λ by producing a previously unknown instance of D that it runs in parallel. The case for $\forall x\, D$ keeps $\forall x\, D$ around to produce other instances that might be needed. Finally, $G \supset D$ interacts with Λ by testing whether Λ passes G (using $G \supset D$ as a resource if needed) and, if so, by running D in parallel with Λ. Thus we require $\Lambda, G \supset D \mid\vdash c$ iff for some Λ': $((\Lambda, G \supset D), c) \xrightarrow{\star} ((\Lambda', G \supset D), c)$, and (i) $\Lambda' \mid\vdash c$ or (ii) $\Lambda', G \supset D \mid\vdash G$ and $\Lambda', D \mid\vdash c$. Notice that, in contrast to E, G functions as a deep guard in this kind of agent formula. Further, the evolution of $((\Lambda, G \supset D), c)$ may itself involve a recursive use of $G \supset D$, but this time in the context of establishing G' for a different $G' \supset D'$ in the current configuration.

Test combinators. Of course, tests may themselves have a complex, non-primitive structure and the operational semantics must specify behavior with respect to such structure as well. Here we rely on the usual interpretation of atomic goals A as recursively defined tests and of $G_1 \wedge G_2$ (resp. $G_1 \vee G_2$, $D \supset G$, $\forall x\, G$) should be viewed as a conjunctive (resp. disjunctive, conditional, generic) test, consistent with their "search reading" formalized by uniform proofs [MNPS91]. There is, however, a subtle difference in the interpretation of existential tests: While the test $\exists x\, G$ succeeds when there is some term t such that the test $G[t/x]$ succeeds, existential agents are allowed to evolve and introduce new constants that can be used to construct t.

Vacuous query $\Lambda \mid\vdash \mathtt{true}$ always holds.
Recursive query $\Lambda \mid\vdash A$ iff there is some Λ' s.t. $(\Lambda, A) \xrightarrow{\star} (\Lambda', A)$, and there is a $G \supset A \in \Lambda'$ and $\Lambda' \mid\vdash G$.
Conjunctive query $\Lambda \mid\vdash G_1 \wedge G_2$ iff $\Lambda \mid\vdash G_1$ and $\Lambda \mid\vdash G_2$.
Disjunctive query $\Lambda \mid\vdash G_1 \vee G_2$ iff $\Lambda \mid\vdash G_1$ or $\Lambda \mid\vdash G_2$.
Extensible query $\Lambda \mid\vdash D \supset G$ iff $\Lambda, D \mid\vdash G$.

Universal query $\Lambda \mid\vdash \forall x\, G$ iff $\Lambda \mid\vdash G[i/x]$ for some new parameter i.
Existential query $\Lambda \mid\vdash \exists x\, G$ iff there is some Λ' s.t. $(\Lambda, \exists x\, G) \overset{\star}{\longrightarrow} (\Lambda', \exists x\, G)$ and $\Lambda' \mid\vdash G[t/x]$, for some t built using the constants in Λ', G.

From a programmer's point of view, the notion of behavioral testing of a concurrent system provides an account of the operational behavior of various combinators. λRCC can be thought of as building on the basic query of the underlying constraint system, $c_0, \ldots, c_n \vdash c$, by permitting complex, recursively defined agents on the LHS of the $\vdash$, and complex recursively defined queries on the RHS. The purpose of the complex formulas on the LHS and RHS in this context is to construct appropriate queries of the underlying constraint system (which may be viewed as a replacement for the axiom case in the usual inference systems).

3.2 A formal presentation

We formalize these ideas via a transition system specified in the tradition of Plotkin's SOS. The transition relation builds on some unknown but fixed underlying constraint system $\mathcal{C}$ satisfying the properties described in [Sar92,PSSS92] that formalizes a derivability relation of the form $c_0, \ldots, c_k \vdash_{\mathcal{C}} c$. In particular, the properties include the admissibility of CUT, i.e., if $c_0, \ldots, c_{k-1} \vdash_{\mathcal{C}} c_k$ and $c_0, \ldots, c_k \vdash_{\mathcal{C}} c$ then $c_0, \ldots, c_{k-1} \vdash_{\mathcal{C}} c$, the admissibility of *Contraction*, i.e., if $\Gamma, c, c \vdash_{\mathcal{C}} c'$ then $\Gamma, c \vdash_{\mathcal{C}} c'$, and closure under substitution for parameters, i.e., if $\Gamma \vdash_{\mathcal{C}} c$ and Γ' and c' result from Γ and c by replacing a parameter i by a term t then $\Gamma' \vdash_{\mathcal{C}} c'$. We augment $\mathcal{C}$ with the inference rule CONST

$$\frac{c_0, \ldots, c_k \vdash_{\mathcal{C}} c}{\Lambda, c_0, \ldots, c_k \vdash_{\mathcal{C}} c}(\textsc{Const}) \tag{7}$$

in which Λ ranges over multisets of D-formulas. The configurations of the machine are multisets Γ of predications (Λ, G). We use ϵ for the empty multiset. The inference rules of the transition system are:

$$((\Lambda, E, E \supset D), G) \longrightarrow ((\Lambda, E, D), G)\ (\text{FC}) \qquad \frac{\Lambda \vdash_{\mathcal{C}} c}{(\Lambda, c) \longrightarrow \epsilon}(\text{C})$$

$$\frac{((\Lambda, G \supset D), G) \overset{\star}{\longrightarrow} \epsilon}{((\Lambda, G \supset D), G') \longrightarrow ((\Lambda, D), G')}(\text{DG}) \qquad (\Lambda, G \vee G') \longrightarrow (\Lambda, G)\ (\text{R-Or-1})$$

$$((\Lambda, D \wedge D'), G) \longrightarrow ((\Lambda, D, D'), G)\ (\text{L-And}) \qquad (\Lambda, G \vee G') \longrightarrow (\Lambda, G')\ (\text{R-Or-2})$$

$$((\Lambda, \exists x\, D), G) \longrightarrow ((\Lambda, D[i/x]), G)\ (\text{L-E(*)}) \qquad (\Lambda, D \supset G) \longrightarrow ((\Lambda, D), G)\ (\text{R-Imp})$$

$$((\Lambda, \forall x\, D), G) \longrightarrow ((\Lambda, \forall x\, D, D[t/x]), G)\ (\text{L-U}) \qquad (\Lambda, \exists x\, G) \longrightarrow (\Lambda, G[t/x])\ (\text{R-E})$$

$$(\Lambda, \texttt{true}) \longrightarrow \epsilon\ (\text{R-True}) \qquad (\Lambda, (\forall x) G) \longrightarrow (\Lambda, G[i/x])\ (\text{R-U(*)})$$

$$((\Lambda, G \supset A), A) \longrightarrow ((\Lambda, G \supset A), G)\ (\text{BC}) \qquad \frac{(\Lambda, G) \overset{\star}{\longrightarrow} \Gamma'}{\Gamma, (\Lambda, G) \longrightarrow \Gamma, \Gamma'}(\text{Struc})$$

$$(\Lambda, G \wedge G') \longrightarrow (\Lambda, G), (\Lambda, G')\ (\text{R-And})$$

The symbol "," is used to denote multiset union in these rules. In determining the applicability of any rule to a given configuration, we assume that a notion of equality modulo the rules of λ-conversion is used. In the rules L-E and R-U, i must be a parameter that does not already appear in the predication on the LHS of the transition rule.

The semantics described above accurately models *successful termination* leveraging don't know non-determinism inherent in the application of BC (which of many applicable rules should be chosen?), R-Or-1/2 (which branch should be chosen?), and R-E (when the rule should be used and with which term?). (See Theorem 8 which establishes that the nondeterminism in the application of the remaining rules is don't care.) The first two can be handled via or-parallel search or backtracking in the usual Prolog style. Once the point of use of the R-E rule has been determined, the actual instantiation for the quantifier may be incrementally generated, using techniques such as those described in [Sha92] to encode quantifier dependency information that constrains the instantiation. A more detailed operational semantics could also replace the "coarse step" evaluation of deep guards above with an incremental evaluation based on maintaining and propagating partial state (cf AKL [HJ90]). Such a detailed operational semantics is beyond the scope of this paper and will be presented in subsequent work.

The proof of the following theorem relies on Theorem 3, Theorem 7, and known properties of intuitionistic derivability.

Theorem 2 (Operational Characterization). *The operational semantics formalized above validates the structural principles and the agent and test combinator conditions described in Section 3.1.*

4 Proof-Theoretic Semantics for λRCC

We show the declarative semantics of λRCC to be given by provability in intuitionistic logic augmented by a fixed constraint system $\mathcal{C}$ of the kind described in Section 3. Specifically we assume that the derivability relation is characterized by a standard sequent system that may additionally use as axioms

$$\frac{}{\Lambda \vdash c}(\textsc{Const})$$

whenever $\Lambda \vdash_{\mathcal{C}} c$ is a valid judgement. We differentiate these axioms from the usual ones in a sequent calculus below by annotating the latter as ($\textsc{Id}$).

4.1 Operational Derivability

We are interested in (cut-free) proofs of sequents of the form $\Lambda \vdash G$ where Λ is a multiset of D formulas. Observe that if Ξ is any sequent that appears in a proof, then the LHS of Ξ contains only D and A formulas and the RHS of Ξ contains either a G or an E formula. One consequence of this observation is that we do not have a need for the $\vee$-L rule in constructing proofs for the sequents under consideration. We would also like to restrict the use of the $\supset$-L rule as follows.
Chaining condition: Every instance of $\supset$-L in which the principal formula is $G \supset A$ (resp. $E \supset D$) is of the form on the left (resp. right):

$$\dfrac{\dfrac{\begin{matrix}\Pi' \\ \vdots\end{matrix}}{\Lambda, G \supset A \vdash G} \qquad \dfrac{}{\Lambda, A \vdash A}(\textsc{Id})}{\Lambda, G \supset A \vdash A}(\supset\text{-L}) \qquad \dfrac{\dfrac{}{\Lambda, E, E \supset D \vdash E}(\textsc{Id}) \qquad \dfrac{\begin{matrix}\Pi' \\ \vdots\end{matrix}}{\Lambda, E, D \vdash G}}{\Lambda, E, E \supset D \vdash G}(\supset\text{-L})$$

Constraint-condition: Every instance of $\supset$-L in which the principal formula is $c \supset D$ is of the form:

$$\dfrac{\dfrac{}{\Lambda, c \supset D \vdash c}(\textsc{Const}) \qquad \dfrac{\overset{\Pi'}{\vdots}}{\Lambda, D \vdash G}}{\Lambda, c \supset D \vdash G}(\text{L-}\textsc{Imp})$$

There are no restrictions on the use of $\supset$-L on $G \supset D$ formulas where G is not c.

We say that a sequent $D \vdash G$ is *operationally derivable* iff it has a proof in which the $\vee$-L rule is not used and each occurrence of the $\supset$-L rule satisfies the above restrictions. We indicate the existence of such a proof by writing $D \vdash_o G$. In such proofs, goal rules are used only to determine what to do next when trying to prove an atomic goal (thus goal rules define the behavior of goal-predicates); agent rules are used only to determine which agents follow from atomic agents (thus agent rules define the behavior of agent-predicates); a constraint query can be proven only if sufficiently powerful constraints are explicitly present in the constraint store. In particular, operational derivability forces proofs to have a "straight line" structure. In a proof the only nodes which have two deep subtrees (i.e. subtrees of depth > 1) and which correspond to the application of a left rule are those whose principal formula is $(G \supset D)$ (where G is not c).

Operational derivability corresponds to the transition system of Section 3.

Theorem 3 (Faithfulness Theorem). *$\Lambda \vdash_o G$ iff $(\Lambda, G) \xrightarrow{\star} \epsilon$*

The proof in one direction proceeds by induction on the size of a derivation and in the other by induction on the length of the transition sequence.

4.2 Correspondence with Intuitionistic Logic

Operational derivability is intended as a bridge between the transition semantics and intuitionistic provability. In one direction, the connection is immediate since operational proofs are intuitionistic proofs with additional structure.

Theorem 4 (Soundness Theorem). *$D \vdash_o G$ implies $D \vdash G$.*

For the other direction, we have to show that the provability relation is unaltered even though we may lose some proofs. We proceed towards this goal via a couple of lemmas. The first lemma is modelled on results in Dyckhoff [Dyc92]. Call a proof *sensible* if whenever $A \supset B$ is the principal formula of an $\supset$-L rule in an intuitionistic derivation and A is atomic, then A also appears on the LHS of the lower sequent. Then the following holds for Intuitionistic Logic (with constraints, as developed in this paper):

Lemma 5. *A proof exists for a sequent if and only if a sensible proof exists.*

Proof. (Sketch) Associate with a proof an *insensibility* measure that counts the number of places where $\supset$-L is applied in a way that violates the notion of sensibility, i.e., where it pertains to a formula of the form $A \supset B$ where A is atomic and A does not appear in the antecedent. We then prove the lemma by induction on the insensibility measure, essentially showing that the first occurrence of such a rule in the derivation along any path starting from the leaves (axioms) can be eliminated.

Lemma 5 shows that we can restrict attention to intuitionistic derivations satisfying the forward chaining condition. By a similar argument, we can show also that the constraint condition can be respected without loss of completeness. We now want to show that if the RHS of the sequent is an atom then we can require it to be proved by backchaining. Define a *clause instance* based on the structure of a D formula as follows:
(1) Any clause instance of $D'[i/x]$ for a new constant i is a clause instance of $\exists x\, D'$.
(2) Any clause instance of $D'[t/x]$, for a closed term t is a clause instance of $\forall x\, D'$.
(3) Any clause instance of D_1 or D_2 is a clause instance of $D_1 \wedge D_2$.
(4) Let $D = G \supset D$: If $G' \supset A$ is a clause instance of D then $((G \wedge G') \supset A$ is a clause instance of D.

Lemma 6. *If A is an atomic formula and Λ is a multiset of D formulas, then $\Lambda \vdash A$ has a derivation if and only if there is a clause instance $G \supset A$ of some D formula in Λ such that $\Lambda \vdash G$ has a derivation.*

Proof. (Sketch) The proof proceeds by induction on the height of the derivation. The last rule in the derivation must pertain to the LHS. The definition of clause instances is modelled to address the non-trivial cases, namely $\exists$-L, $\forall$-L and $\supset$-L.

Lemmas 5 and 6 provide the basis for the proof of the desired result:

Theorem 7 (Completeness Theorem). *$D \vdash G$ implies $D \vdash_o G$.*

The results of this section show that entailment in intuitionistic logic provides an alternative semantics for λRCC. Apart from underpinning the declarative semantics of this language, this property also allows us to use known properties of the intuitionistic calculus to understand characteristics of our transition relation. As one example, known permutation properties for this calculus reveal that some aspects of non-determinism in the transition relation are inconsequential:

Theorem 8 (Local Confluence Theorem). *Let $(\Lambda, A) \longrightarrow (\Lambda_1, A_1)$ by any rule except* R-OR-2, R-OR-1, R-E *or* BC. *Let $(\Lambda, A) \longrightarrow (\Lambda_2, A_2)$ by any rule. Then there exists a Λ_3 such that $(\Lambda_1, A_1) \xrightarrow{\star} (\Lambda_3, A')$ and $(\Lambda_2, A_1) \xrightarrow{\star} (\Lambda_3, A')$.*

5 Conclusions

This paper establishes the semantic foundations for a logical approach to program manipulation, λRCC, which satisfies the desiderata laid out in Section 1. λRCC endows a very rich subset of intuitionistic logic with a (complete) computational interpretation based on testing determinate concurrent systems. Operationally, the programmer may use recursive agents to generate constraints from a representation of an object program, and recursive queries to test these constraints.

From a practical point of view, we are currently developing a concrete extension of λProlog along these lines. We intend to develop an integration of such a language into JAVA-like languages along the lines of jcc[SJG03], and use it as the basis for AST-rewrites in Polyglot and Eclipse.

On the theoretical front, extending the basic conception of this paper to sub-structural logics such as linear logic remains open. In contrast to LolliMon [LPPW05] that associates backward (resp. forward) chaining with asynchronous (resp. synchronous) connectives of linear logic, this paper explores forward and backward chaining mostly (except existentials) in the asynchronous fragment. The detailed integration of these seemingly different approaches remains open to further investigations.

Acknowledgements. We gratefully acknowledge discussions with Robert Fuhrer and Mandana Vaziri on the topic of logical representation of program refactorings. Radha Jagadeesan was supported in part by NSF 0430175. Gopalan Nadathur has received support for this work from NSF Grant CCR-0429572 and the Digital Technology Center and the Department of Computer Science at the University of Minnesota.

References

[Aik99] Alexander Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2-3):79–111, 1999.

[Dyc92] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3), September 1992.

[ecl] The eclipse project. www.eclipse.org.

[FFL03] Stacy E. Finkelstein, Peter Freyd, and James Lipton. A new framework for declarative programming. *Theor. Comput. Sci.*, 300(1-3):91–160, 2003.

[FTK04] Robert Fuhrer, Frank Tip, and Adam Kiezun. Advanced refactorings in eclipse. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 8–8, New York, NY, USA, 2004. ACM Press.

[GDN04] Miguel Garcia-Diaz and Susana Nieva. Providing declarative semantics for hh extended constraint logic programs. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 55–66, New York, NY, USA, 2004. ACM Press.

[Hei92] Nevin Charles Heintze. *Set based program analysis*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[HJ90] S. Haridi and S. Janson. Kernel andorra Prolog and its computation model. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46, Jerusalem, 1990. The MIT Press.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL'87), Munich, Germany*, pages 111–119. ACM Press, New York (NY), USA, 1987.

[LMRC05] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 364–377, New York, NY, USA, 2005. ACM Press.

[LNRA01] Javier Leach, Susana Nieva, and Mario Rodrguez-Artalejo. Constraint logic programming with hereditary harrop formulas. *Theory Pract. Log. Program.*, 1(4):409–445, 2001.

[LPPW05] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In Pedro Barahona and Amy P. Felty, editors, *PPDP*, pages 35–46. ACM, 2005.

[LS93] Patrick Lincoln and Vijay Saraswat. Proofs as concurrent processes. Technical report, PARC, 1993.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of the Conference on Compiler Construction (CC'03)*, pages 1380–152, April 2003.

[NM98] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In C. Hogger D. Gabbay and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 499–590. Oxford University Press, 1998.

[OP99] P.W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.

[Pal95] Jens Palsberg. Closure analysis in constraint form. *ACM Trans. Program. Lang. Syst.*, 17(1):47–62, 1995.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-oriented type systems*. John Wiley and Sons Ltd., Chichester, UK, UK, 1994.

[PSSS92] Prakash Panangaden, Vijay A. Saraswat, P. J. Scott, and R. A. G. Seely. A hyperdoctrinal view of concurrent constraint programming. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 666 of *Lecture Notes in Computer Science*, pages 457–476. Springer, 1992.

[PW98] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Trans. Program. Lang. Syst.*, 20(3):635–678, 1998.

[RMR01] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 43–55, New York, NY, USA, 2001. ACM Press.

[Sar92] Vijay A. Saraswat. The category of constraint systems is cartesian closed. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1992.

[Sar93] V. Saraswat. *Concurrent Constraint Programming*. Doctoral Dissertation Award and Logic Programming. MIT Press, 1993.

[Sha92] Natarajan Shankar. Proof search in the intuitionistic sequent calculus. In Deepak Kapur, editor, *Automated Deduction – CADE-11*, number 607 in Lecture Notes in Computer Science, pages 522–536. Springer Verlag, June 1992.

[SJG03] V Saraswat, R Jagadeesan, and V Gupta. jcc: Integrating timed default concurrent constraint programming into Java. Number 2902 in Lecture Notes in Computer Science, pages 156–170. Springer Verlag, 2003.