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] Department of CSE, University of Minnesota
[3] IBM T.J. Watson Research Center

Abstract. We present the natural confluence of higher-order hereditary Harrop formulas (HH formulas) as developed concretely in λProlog, Constraint Logic Programming (CLP, [JL87]), and Concurrent Constraint Programming (CCP, [Sar93]) as a fragment of (intuitionistic, higher-order) logic. The 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 this 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

Our primary motivations for investigating logical frameworks for program manipulation are twofold.

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, 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 construct. Similar difficulties have been reported in other frameworks whose goal is to make it extremely 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 investigate a framework for manipulating and analyzing programs. We believe such a framework should satisfy the following desiderata. Below we will find it convenient to distinguish $\mathcal{O}$, the (hypothetical) *object language* and $\mathcal{F}$ the programming language in the proposed framework being used to write programs to 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. [PS94,Pal95,Hei92,OSW99], [PW98,RMR01,Aik99]. 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

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

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.

2.1 Scoped constants and extensible tests

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

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

An operational semantics is provided implicitly by the notion of *uniform proofs* within a sequent calculus [MNPS91]. A proof in this context is uniform if it satisfies the property that if a node in the proof tree contains a non-atomic formula in the RHS, then that node is an application of the right introduction rule for the principal connective (or quantifier) of the formula. This gives a natural interpretation for RHS connectives: a query $D \vdash G_1 \wedge G_2$ succeeds iff the queries $D \vdash G_1$ and $D \vdash G_2$ succeed; $D \vdash G_1 \vee G_2$ succeeds iff $D \vdash G_1$ or $D \vdash G_2$ succeeds; $D \vdash D_1 \supset G$ succeeds iff $D, D_1 \vdash G$ succeeds; $D \vdash \forall x\, G$ succeeds iff $D \vdash G$ succeeds, where x is not free in D; $D \vdash \exists x G$ succeeds iff $D \vdash G[t/x]$ succeeds, for some term t.

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 work synergistically with the idea of using the typed lambda-calculus to represent programming language objects as the following program in the concrete λProlog syntax illustrates.

Example 1 (Monotypes in λProlog). Consider a higher-order encoding of pure λ-terms in which an abstraction of the form $\lambda x M$ is represented by `(fn x\ M')` *and an application of the form* $(M\ N)$ *is represented by* `(M' @ N')` *where* `M'` *and* `N'` *are representations of* M *and* N *respectively. The following remarkably simple logic program then provides a declarative encoding of the type assignment problem for λ-terms:*

```
mono (fn M) (A --> B) :- pi x\ tyvar x A => mono (M x) B.
mono T A              :- tyvar T A.
mono (M @ N) B        :- mono M (A --> B), mono N A.
```

We note that `(pi x\ ...)` *is λProlog syntax for universal quantification and* `=>` *represents implication. Now, the first clause specifies that the type of an abstraction is* `A --> B` *if the type of* `(M x)` *can be shown to be* `B` under the assumption *that the type of* `x` *is* `A`*, for* `x` *a "fresh constant" (eigenvariable). The assertion is captured by augmentation of the "left hand side" of the query. The second clause specifies that the type of* `T` *is* `A` *if this can be deduced from the asserted facts about the types of variables. The third case is straightforward. Thus the query* `mono (fn x (fn y x)) Type` *yields the answer:*

```
Type = A --> B --> A
```

The central limitation of λProlog for the applications we have discussed above is the lack of constraints. For instance suppose we wish to extend the program above to infer *polymorphic* types, e.g. types of the form `all d \ all e \ d --> e --> d`. Here it is natural to require that `all` be interpreted, that is satisfy the following properties:

```
all d \ P = P
all d \ all e \ P d e = all e \ all d \ P d e
```

The first says that vacuous quantifications can be dropped, the second that the order of quantifications does not matter. Simple unification – the implementation of existential quantifiers on the RHS – cannot be used in this context. Many other examples of constraint systems of use in a type-checking context may be found in [PS94,OSW99,PR04].

2.2 Adding constraints to Harrop formulas

This leads us to the integration of constraint programming with HH. The addition of constraints to goals and agents has has been proposed by [LNRA01,GDN04,FFL03] through the 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.

2.3 Adding concurrent constraint programming

The language with syntax described by rules (1)–(4) suffers from a major 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 redundant computation: the consequences are computed once, and used to show 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 2 (Java type checking). Consider a JAVA *type checker. The code for every method in a class must be type-checked. Such a type check must be made in the context of type assertions generated by examining the classes referenced in the code. These are assertions built using types and predicates such as:*

```
kind cName type.  // class names; similarly iName, fName, etc.
type extends cName -> cName -> o. // inheritance hierarchy
```

It is desirable that the parsing of referenced classes and the generation of type assertions (inheritance hierarchy, method signatures, field signatures) is done only once. Thus, conceptually, one wishes to define:

```
typed ClassName :-
  parse ClassName Code,
  referencedClassTypes Code => typed Code.
```

The definition of the predicate for `typed` *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`*):*

```
typed (assign LExp RExp) :-
 isType RExp RT, isType LExp, LT,  subType RT LT.
```

`referencedClassTypes` *is a user-defined (agent) predicate, operating on the LHS of a sequent, that walks the AST* `Code`*, determines referenced classes, and for each generates type assertions (e.g.* `point extends object, msig point void move int::int::nil` *etc). Running the agent* `referenceClassTypes 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: combine 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.

$$\textbf{(Agent)}\ D ::= \mathtt{true} \mid c \mid D \wedge D \mid E \mid G \supset D \mid E \supset D \mid \exists x\, D$$
$$(\textbf{Test})\ \ G ::= \mathtt{true} \mid c \mid G \wedge G$$

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 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 satisfying the property $D \vdash c_1$ it is the case that $c \vdash c_1$.

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:

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

The results of this paper may be extended to support disjunctive agents as well; but we omit their treatment for lack of space. With this extension, λRCC contains all intuitionistic formulas (over constraints) satisfying the Disjoint Vocabulary condition. Clearly this includes LP, HH, CLP and CCP. Figure 1 in the Appendix shows λRCC and its sublanguages in detail.

Rest of this paper. Section 3 presents an interpretation of λRCC formulas in terms of *behavioral testing*, and describe an operational semantics consistent with this interpretation. Section 4 shows that (higher-order) intuitionistic logic provides an alternate semantics for the same language. Thus the operational semantics shows that a test succeeds against a given agent iff this is provable in logic. Appendix B summarizes the standard notion of intuitionistic provability. Appendix D shows some paradigmatic λRCC programming examples.

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 wreak havoc 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).

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 $(\Lambda, G) \xrightarrow{\star} \epsilon$. Our semantics has to contend with two different possibilities: Λ may contain complex (non-primitive) agents that evolve, and G may be a complex (non-primitive) query. Thus the transition rules must specify the operational behavior of agent and test combinators. The conditions discussed in this section are enumerated formally in Appendix C for the reader's convenience.

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 basic query being asked is: does Λ have the *potential* to generate c? The operational semantics should guarantee that this potential should be maintained as Λ evolves. That is, if $(\Lambda, c) \xrightarrow{\star} (\Lambda', c)$ and $\Lambda \mid\vdash c$ then it should be the case that $\Lambda' \mid\vdash c$. Evolution should merely serve to "actualize" some of the potential, in that answers to some queries may be "read out" (resolved by querying the built-in constraint solver) instead of requiring additional computation; it should not *discard* any potential.

Structural rules. This reading can be seen to verify certain "structural" principles. (Cut) If Λ has the potential to generate c' and Λ, c' it has the potential to generate c, then Λ should have by itself the potential to generate c, i.e. if $\Lambda \mid\vdash c'$ and $\Lambda, c' \mid\vdash c$ then $\Lambda \mid\vdash c$. (Weak) If a system Λ passes a test, then running more agents in parallel will not cause the test to not pass, i.e. if $\Lambda \mid\vdash c$ then $\Lambda, D \mid\vdash c$. (Dup) If two copies of an agent can be used to answer a query, then one copy should suffice, because two copies can be used merely to generate the same constraints twice, and constraint entailment is not concerned with multiplicities i.e. $\Lambda, D, D \mid\vdash c$ implies $\Lambda, D \mid\vdash c$. (Exch) Finally, the very notion of parallel composition means that order does not matter, i.e. $\Lambda, D, D' \mid\vdash c$ implies $\Lambda, D', D \mid\vdash c$.

Agent combinators. When should Λ, D pass a test c? Clearly, this should happen if (a) Λ passes c – thus D was not needed, or (b) D interacts with Λ in some way to cause the system to evolve and reach a state in which c can be answered. Thus to answer this question we need to specify the *agent interaction rules*.

`true`. Clearly, it should be regarded as an inert, vacuous agent, incapable of contributing anything to answering a question. Hence we should require $\Lambda, \texttt{true} \mid\vdash c$ iff $\Lambda \mid\vdash c$.

E. When should Λ, E pass the test c? One possibility is that Λ passes the test by itself. Another alternative is that evolution of Λ to Λ' reveals a rule $E \supset D \in \Lambda'$ such that Λ', E, D passes the test. Thus E is a (potentially recursively defined) agent with a behavior rule $E \supset D$.

$G \supset D$. This 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 should require

$\Lambda, G \supset D \mid\vdash c$ iff $\Lambda \mid\vdash c$ or ($\Lambda, G \supset D \mid\vdash G$ and $\Lambda, D \mid\vdash c$). Notice that, in contrast to E, G functions as a deep guard in this kind of agent formula.

$D_1 \wedge D_2$. This interacts with Λ by running D_1 and D_2 in parallel. Thus we should require $\Lambda, D_1 \wedge D_2 \mid\vdash c$ iff $\Lambda, D_1, D_2 \mid\vdash c$.

$\exists x\, D$. This interacts with Λ by producing a previously unknown instance of D that it runs in parallel with it. Thus we should require that $\Lambda, \exists x\, D \mid\vdash c$ iff $\Lambda, D[i/x] \mid\vdash c$ for some constant i that does not appear in Λ or D.

$\forall x\, D$. This interacts with Λ by producing as many instances of D as needed, with a terms t substituted for x. Thus we should require $\Lambda, \forall x\, D \mid\vdash c$ iff $\Lambda, \forall x\, D, D[t/x] \mid\vdash c$; we keep $\forall x\, D$ around to produce other instances that might be needed.

Test combinators. Consider complex queries. Clearly, $\Lambda \mid\vdash \texttt{true}$ should always hold; `true` is the vacuous test. Atomic goals A should be thought of as recursively defined tests; thus $\Lambda \mid\vdash A$ should hold iff there is some $G \supset A$ in an evolution Λ' of Λ such that $\Lambda' \mid\vdash G$. $G_1 \wedge G_2$ should be viewed as a conjunctive test, so it should be the case that $\Lambda \mid\vdash G_1 \wedge G_2$ iff $\Lambda \mid\vdash G_1$ and $\Lambda \mid\vdash G_2$. $G_1 \vee G_2$ should be viewed as disjunctive test, hence it should be the case that $\Lambda \mid\vdash G_1 \vee G_2$ iff $\Lambda \mid\vdash G_1$ or $\Lambda \mid\vdash G_2$. $D \supset G$ should be a *conditional* test, so it should be the case that $\Lambda \mid\vdash D \supset G$ iff $\Lambda, D \mid\vdash G$. $\forall x\, G$ should be viewed as a *generic* test, so we require $\Lambda \mid\vdash \forall x\, G$ iff $\Lambda \mid\vdash G[i/x]$, for a constant i not appearing in Λ or D.

The operational reading of test combinators is so far consistent with their "search reading" formalized by uniform proofs [MNPS91]. The interpretation of $\exists$ tests is however subtly different. The test $\exists x\, G$ should succeed when there is some term t such that the test $G[t/x]$ succeeds. However, recall that the evolution of existentially quantified agents can introduce new constants, which can be used to construct t. Thus it should be the case that $\Lambda \mid\vdash \exists x\, G$ iff there is some Λ' such that $(\Lambda, \exists x\, G) \xrightarrow{\star} (\Lambda', \exists x\, G)$ 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 can be seen to provides an account of the operational behavior of various combinators. This account generalizes the "search interpretation" of logical connectives for the HH fragment of λRCC, and extends it conservatively in the sense that it reduces to exactly that interpretation when the syntax is limited to that of HH formulas. From a different perspective, λRCC can be thought of as building on the basic query of the underlying constraint system, $c_0, \ldots, c_{n-1} \vdash c$, 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 theses ideas through a high-level transition system specified in the tradition of Plotkin's SOS.

The transition relation builds on some unknown but fixed underlying constraint system $\mathcal{C}$ that formalizes a derivability relation of the form $c_0, \ldots, c_k \vdash_{\mathcal{C}} c$. We assume that $\mathcal{C}$ is closed under 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

$c_0, \ldots, c_{k-1} \vdash_{\mathcal{C}} c$. Further, $\mathcal{C}$ includes the inference rule CONST

$$\frac{c_0, \ldots, c_{k-1} \vdash_{\mathcal{C}} c}{\Lambda, c_0, \ldots, c_{k-1} \vdash_{\mathcal{C}} c}(\text{CONST}) \tag{7}$$

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:

$$\frac{(\Lambda, G) \xrightarrow{\star} \Gamma'}{\Gamma, (\Lambda, G) \longrightarrow \Gamma, \Gamma'}(\text{STRUC}) \qquad ((\Lambda, E, E \supset D), G) \longrightarrow ((\Lambda, E, D), G)\ (\text{FC})$$

$$\frac{((\Lambda, G \supset D), G) \xrightarrow{\star} \epsilon}{((\Lambda, G \supset D), G') \longrightarrow ((\Lambda, D), G')}(\text{DG}) \qquad ((\Lambda, D \wedge D'), G) \longrightarrow ((\Lambda, D, D'), G)\ (\text{L-AND})$$

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

$$(\Lambda, \texttt{true}) \longrightarrow \epsilon\ (\text{R-TRUE}) \qquad ((\Lambda, G \supset A), A) \longrightarrow ((\Lambda, G \supset A), G)\ (\text{BC})$$

$$\frac{\Lambda \vdash_{\mathcal{C}} c}{(\Lambda, c) \longrightarrow \epsilon}(\text{C}) \qquad (\Lambda, G \wedge G') \longrightarrow (\Lambda, G), (\Lambda, G')\ (\text{R-AND})$$

$$(\Lambda, G \vee G') \longrightarrow (\Lambda, G)\ (\text{R-OR-1}) \qquad (\Lambda, G \vee G') \longrightarrow (\Lambda, G')\ (\text{R-OR-2})$$

$$(\Lambda, D \rightarrow G) \longrightarrow ((\Lambda, D), G)\ (\text{R-IMP}) \qquad (\Lambda, \exists x\, G) \longrightarrow (\Lambda', G[t/x])\ (\text{R-E})$$

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

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.

There is a proviso that governs the rules L-E and R-U: i must represent a constant that does not already appear in the predication on the LHS of the transition rule. In the other quantifier rules, i.e., in L-U and R-E, t may be an arbitrary closed term.

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 L-U/R-E (which term should be chosen). (See Theorem 9 which establishes that the nondeterminism in the application of the remaining rules is dont care.) While the first two can be handled via or-parallel search or backtracking in the usual Prolog style, the last requires techniques based on "dynamic Herbrandization" (encoding dependency information for quantifiers [Sha92]) to incrementally generate the terms for L-U and R-E (and avoid nondeterministic 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 4, Theorem 8, and known properties of intuitionistic derivability.

Theorem 3 (Operational Characterization). *The propositions in Section 3.1 (enumerated in Appendix C) are valid.*

4 Proof-Theoretic Semantics for λRCC

We intend the declarative semantics of λRCC to be given by provability in intuitionistic logic extended with constraints. Formally we use the sequent system presentation of Appendix B, augmented with a fixed constraint system $\mathcal{C}$ as described in Section 3 (and its CONST inference rule).

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. We note the following property for such proofs: if Ξ is any sequent that appears in them, 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 $G \supset A$ is of the form on the left, and every instance of $\supset$-L in which the principal formula is $E \supset D$ is of the form on the right:

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

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

$$\dfrac{\dfrac{}{c_1, \ldots, c_k \vdash_{\mathcal{C}} c}(\text{CONST}) \qquad \begin{array}{c}\Pi'\\ \vdots\\ \Lambda, D \vdash G\end{array}}{\Lambda, c \supset D \vdash G}(\text{L-IMP})$$

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

We shall 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 $D \vdash_o G$. The intuition underlying operational derivability is that 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).

There is a natural correspondence between operational derivability and the transition system of Section 3 that is stated in the following theorem:

Theorem 4 (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 5 (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]. Let us say that a proof is *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

Lemma 6. *A proof exists for a sequent if and only if a sensible proof exists.*

Proof. Only a sketch is provided. First, we 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 6 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. Towards this end, let us define a *clause instance* based on the structure of a D formula as follows:

- If D is of the form $\exists x\, D'$ then any clause instance of $D'[i/x]$ for a new constant i is a clause instance of D.
- If D is of the form $\forall x\, D'$ then any clause instance of $D'[t/x]$, for a closed term t is a clause instance of D.
- If D is of the form $D_1 \wedge D_2$ then any clause instance of D_1 or D_2 is a clause instance of D.
- If D is of the form $G \supset D$ and $G' \supset A$ is a clause instance of D then it is the case that $((G \wedge G') \supset A$ is a clause instance of D.

We then have the following:

Lemma 7. *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. The proof proceeds by induction on the height of the derivation. The last rule in the derivation must pertain to the LHS. The cases where this is $\exists$-L, $\forall$-L and $\supset$-L are the only ones that need some argument but the definition of clause instances is modelled to build in the actions of these rules.

Lemmas 6 and 7 provide the basis for the proof of the desired result:

Theorem 8 (Completeness Theorem). *$D \vdash G$ implies $D \vdash_o G$.*

The results of this section have shown 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, the structural properties for this relation discussed in Section 3 follow quickly from the admissibility of weakening, contraction for our sequent calculus.

Similarly, known permutation properties for this calculus lead to the following observation that reveals that some aspects of non-determinism in the transition relation are inconsequential:

Theorem 9 (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, the basic conception of this paper should be extensible to sub-structural logics such as linear logic – in the same way as LCC [SL92,FRS01], extends CCP. However indeterminacy of "agent" computations is likely to pose additional semantic problems.

Acknowledgements. We gratefully acknowledge discussions with Robert Fuhrer and Mandana Vaziri on the topic of logical representation of program refactorings.

Authors' note: Material after the bibliography is intended to be supplementary and should be read at the discretion of the reviewer.

References

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

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[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.

[FRS01] Francois Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: operational and phase semantics. *Journal of Information and Computation*, 165(1):14–41, February 2001.

[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.

[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.

[NM99] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In Harald Ganzinger, editor, *Automated Deduction–CADE-16*, number 1632 in Lecture Notes in Artificial Intelligence, pages 287–291. Springer-Verlag, July 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.

[PR04] F. Pottier and D. Re'my. *Advanced Topics in Types and Programming Languages*, chapter The Essence of ML Type Inference. MIT Press, 2004.

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

[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.

[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.

[SJ05] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Concur '05*, 2005.

[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.

[SL92] Vijay Saraswat and Patrick Lincoln. Higher-order Linear Concurrent Constraint Programming. Technical report, Xerox PARC, 1992.

[tey] The Teyjus distribution of λProlog. Available from http://teyjus.cs.umn.edu.

A λRCC

Figure 1 specifies the syntactic structure of the language λRCC and its well-known sub-languages.

B Intuitionistic Derivability

This appendix characterizes the notion of intuitionistic derivability used in this paper. We assume a logical vocabulary that includes the following symbols: $\top$ representing the tautologous proposition `true`, $\bot$ representing the contradictory proposition, $\wedge$, $\vee$ and $\supset$ representing the usual, infix logical connectives, and $\exists$ and $\forall$ representing the quantifiers. We need to consider substitution of terms for free variables in formulas. We use the expression $P[t/x]$ to depict the logically correct such substitution, i.e., an operation that replaces only free occurrences of x in P and that takes care to make the needed renamings to avoid illegal capture of variables appearing in t. We assume a language that is higher-order in the following sense: our atomic formulas have the form

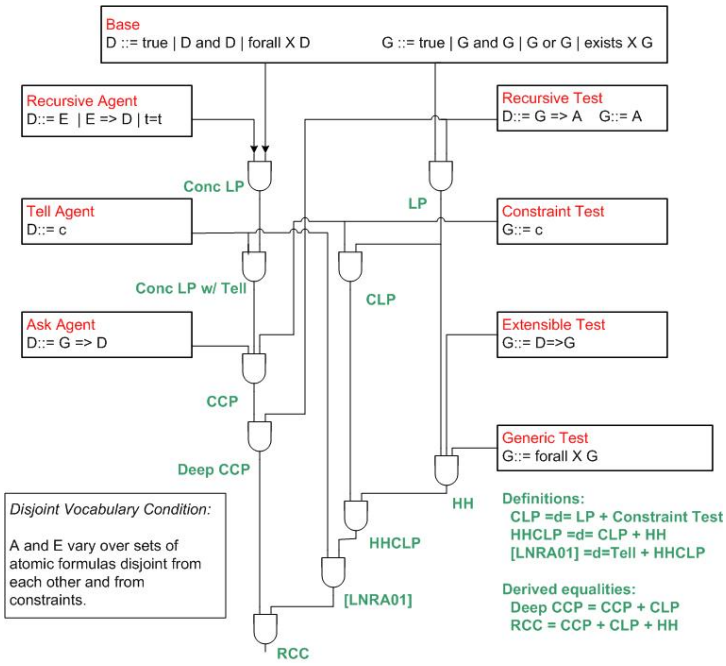

Fig. 1. RCC subsumes LP, HH, CLPand CCP. Similar inclusions hold for λRCC and LP, HH, CLP, CCP w/ higher-order terms.

$(p\, t_1\ \ldots\ t_n)$ where p is a predicate constant and, for $1 \leq i \leq n$, t_i is a lambda term that may contain function variables. We assume that our terms are simply typed [Chu40]. Typing places restrictions on substitution and other related operations; for example, in an expression of the form $P[t/x]$ it is necessary that t and x have the same type. We assume these restrictions implicitly and actually suppress all further mention of types here.

The basic unit of assertion that is of interest is a sequent. This is a pair of finite (possibly empty) multisets of formulas $\langle \Delta, \Theta \rangle$, written $\Delta \vdash \Theta$, where, additionally, Θ contains at most one formula. A proof or derivation for a sequent is a finite tree constructed using the inference rules in Figure 2 and such that the root is labeled with the sequent in question and the leaves are *axioms*, identified as sequents in which $\bot$ appears to the left of the turnstile symbol, $\top$ appears to the right of this symbol or some atomic formula appears both to its left and to its right. In the depiction of an actual proof, we shall use the annotation ID to indicate axiom occurrences. The rules shown in Figure 2 are actually schemas. Particular rules may be obtained from these by instantiating Δ and Θ by specific multisets of formulas, B, D, and P by formulas, x by a variable, t by a term and c by a constant. There is, in addition, a proviso on the choice of constant for c: it should not appear in any formula contained in the lower sequent in the same figure. Also, in the inference figure schemata λ, the sets Δ and Δ' and the sets Θ and Θ' differ only in that zero or more formulas in them are replaced by formulas that can be obtained from them by using the λ-conversion rules. Finally, every sequent that appears in an instantiation of these schemas must satisfy the requirement that at most one formula appears to the right of the turnstile symbol.

$$\frac{B, D, \Delta \vdash \Theta}{B \wedge D, \Delta \vdash \Theta}(\wedge\text{-L}) \qquad \frac{\Delta \vdash B \qquad \Delta \vdash D}{\Delta \vdash B \wedge D}(\wedge\text{-R})$$

$$\frac{B, \Delta \vdash \Theta \qquad D, \Delta \vdash \Theta}{B \vee D, \Delta \vdash \Theta}(\vee\text{-L})$$

$$\frac{\Delta \vdash B}{\Delta \vdash B \vee D}(\vee\text{-R}) \qquad \frac{\Delta \vdash D}{\Delta \vdash B \vee D}(\vee\text{-R})$$

$$\frac{B \supset D, \Delta \vdash B \qquad D, \Delta \vdash \Theta}{B \supset D, \Delta \vdash \Theta}(\supset\text{-L}) \qquad \frac{B, \Delta \vdash D}{\Delta \vdash B \supset D}(\supset\text{-R})$$

$$\frac{P[t/x], \forall x P, \Delta \vdash \Theta}{\forall x P, \Delta \vdash \Theta}(\forall\text{-L}) \qquad \frac{\Delta \vdash P[t/x]}{\Delta \vdash \exists x P}(\exists\text{-R})$$

$$\frac{P[c/x], \Delta \vdash \Theta}{\exists x P, \Delta \vdash \Theta}(\exists\text{-L}) \qquad \frac{\Delta \vdash P[c/x]}{\Delta \vdash \forall x P}(\exists\text{-R})$$

$$\frac{\Delta' \vdash \Theta'}{\Delta \vdash \Theta}(\lambda)$$

Fig. 2. Inference Rules for Intuitionistic Derivability

Expressions of the form B, Δ that are used in the inference figure schemata in Figure 2 are to be treated as abbreviations for multisets which have B as an element and whose remaining elements constitute Δ. Our presentation leaves out the structural rules of thinning, contraction, interchange and cut that are typically found in sequent calculi. Thinning is redundant because of the structure we have assumed for axioms. Our use of multisets obviates interchange and all essential uses of contraction have been assimilated in our presentation into relevant operational rules. Finally, cut is an admissable rule in our calculus: the same sequents are derivable with and without it.

C Operational characterization

The properties of $\mid\!\vdash$ discussed in Section 3.1 are:

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 $\Lambda \mid\!\vdash c$; $\Lambda, D \mid\!\vdash c$ if $\Lambda, D, D \mid\!\vdash c$ $\Lambda, D_1, D_2 \mid\!\vdash c$ iff $\Lambda, D_2, D_1 \mid\!\vdash c$;
Vacuous agent $\Lambda, \mathtt{true} \mid\!\vdash c$ iff $\Lambda \mid\!\vdash c$.
Recursive agent $\Lambda, E \mid\!\vdash c$ iff $\Lambda \mid\!\vdash c$ or there is a Λ' and 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 $\Lambda, G \supset D \mid\!\vdash c$ iff $\Lambda \mid\!\vdash c$ or ($\Lambda, G \supset D \mid\!\vdash G$ and $\Lambda, D \mid\!\vdash c$).
Parallel agent $\Lambda, D_1 \wedge D_2 \mid\!\vdash c$ iff $\Lambda, D_1, D_2 \mid\!\vdash c$.
Existential agent $\Lambda, \exists x\, D \mid\!\vdash c$ iff $\Lambda, D[i/x] \mid\!\vdash c$ for some i not free in Λ, c.
Universal agent $\Lambda, \forall x\, D \mid\!\vdash c$ iff $\Lambda, \forall x\, D, D[t/x] \mid\!\vdash c$.
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 i not in Λ.
Existential query $\Lambda \mid\!\vdash \exists x\, G$ iff there is some Λ' s.t. $(\Lambda, \exists x\, G) \xrightarrow{\star} (\Lambda', \exists x\, G)$ and $\Lambda' \mid\!\vdash G[t/x]$, for some t built using the constants in Λ', G.

D Programming examples

We present programs in a concrete syntax reminescent of the Teyjus implementation of λProlog [NM99].

There are several ways of thinking about the duality of agent and query computation. Fundamentally, λRCC permits user-defined arbitrarily intertwined forward- and backward-chaining computations.

First, one may think of agents as representing a database, and the test as performing queries on the database. As in SQL-like languages, the query language may augment the database (extensible queries of HH). Agent computation may be thought of as permitting the user to specify integrity constraints. The addition of any information by the query may trigger (forward-chaining) agent computation.

Another way is to think of agent computation as augmenting the underlying constraint system. (This is illustrated by the JAVA type-checker example, Example 2.) For instance, an agent predicate may be written to parse a class and generate the type signature of the class for use in the query.

A third major use of agent computations is to deduce and cache facts that may then be repeatedly queried, without recomputation.

D.1 Representing type-checkers in λRCC

Representing HM(X) A *polytype* is an explicit universally quantified type. Writing a program to produce *polytypes* for lambda terms – for instance the solution `Type = all a  all b  a --> b --> a` to the above query is much harder.

Here is code to represent polytypes in λProlog taken from [tey].

```
% Predicate for removing vacuous quantifications
% in poly types.
type  rem_vac   poly -> poly -> o.

rem_vac (c Ty) (c Ty).
rem_vac (all x\P) Q     :- !, rem_vac P Q.
rem_vac (all P) (all Q) :- pi x\ rem_vac (P x) (Q x).

% Representing typing judgements; these are
% essentially term, type pairs
kind pair    type -> type -> type.
type pr      A -> B -> pair A B.

% Inferring a poly type for a term. The main work is done
% in typeof that generates typing judgements and constraints
%  between these, and invokes type unification to solve
% the constraints.
type  tybind      tm -> ty -> o.
type  tyinfer     tm -> poly -> o.

type  typeof      list (pair tm ty) -> list eq -> poly -> o.

tyinfer Term Poly :- typeof (pr Term topvar :: nil) nil Poly.

typeof (pr (M @ N) A ::S) Eqs (all P) :- !,
  pi c\ tvar c => typeof (pr M (c --> A) :: pr N c :: S) Eqs (P c).

typeof (pr (fn M) A :: S) Eqs  (all d\ all e\ P d e) :- !,
  pi d\ tvar d => pi e\ tvar e => pi x\ tybind x d =>
   typeof (pr (M x) e :: S) ((A == d --> e) :: Eqs) (P d e).

typeof (pr C B :: S) Eqs Poly :-
  prim_poly C Ty, !, poly_inst Ty B S Eqs Poly.

typeof (pr X B :: S) Eqs Poly :-
```

```
  tybind X A, typeof S ((A == B) :: Eqs) Poly.
typeof nil Eqs (c Ty) :-  unify Eqs topvar Ty.
% The main predicate. Infer a poly type first and then prune
% vacuous quantifications
type  polyinfer  tm -> poly -> o.

polyinfer Tm Ty :- tyinfer Tm PreTy, rem_vac PreTy Ty.
```

The essential problem is that in order to permit abstraction it is necessary that the type of a (sub)term be represented not as an existentially quantified variable (EQV) but as a universally quantified variable (UQV). However, tests do not permit constraints to be imposed on UQV. Instead the constraints must be collected in the form of a data-structure (the third argument `Eqs` to `typeof` involving UQVs; these equations are solved by a programmer written `unify` and `substitute` routine, and a *representation* of the solution returned.

λRCC permits a slight improvement. Constraints may now be asserted directly in the store using a goal of the form `c => G`. However, it is necessary that constraints generated by multiple goals be collected conjunctively in a single set. This cannot be done by representing these goals using test conjunction: the query `c => G, d => H` results in *two* different queries, one augmented with `c` and the other with `d`. Instead it becomes necessary to represent the subgoals in a datastructure and to progress down the datastructure in a linear fashion.

Example 10. The program is:

```
tyinfer Term Poly :-
  pi top \ typeof (mono Term top :: nil) Poly top.

typeof (mono (M @ N) A ::S) (all P) Top :-
  pi c\ tvar c
    => typeof (mono M (c --> A) :: pr N c :: S) (P c) Top.

typeof (mono (fn M) A :: S)  (all d\ all e\ P d e) Top :-
  pi d\ tvar d
    => pi e\ tvar e
      => pi x\ tybind x d, A = d --> e
          => typeof (mono (M x) e :: S)  (P d e) Top.

typeof (mono X B :: S) Type Top :-
  A = B => tybind X A, typeof S  Type Top.

typeof nil Type Top :-  Ty = Top.

poly Term Type :- typeof Term Type.
```

The query `poly (fn x  (fn y  x)) Type` *generates the final configuration:*

```
top = d --> e, e = d1 --> e, d = e1
  ?- Type = all d \ all e \ P d e, P = all d1 \ all e1 P1 d1 e1,
     P1 d1 e1 = top
```

This has many solutions, including

```
Type = all d \ all e \ all d1 \ all e1 d --> d1 --> d
Type = all d \ all e \ all d1 \ all e1 d --> d1 --> e1
Type = all d \ all e \ all d1 \ all e1 d --> e
```

The last two offending solution can be ruled out by defining:

```
poly Term Type :- typeof Term Type, gen Term Type.
gen Term (all d \ P d) Term :- pi d \ gen Term (P d).
gen Term (D --> R) :- mono Term  (D --> R).
```

An even cleaner solution is possible in λRCC using agent recursion. Here a subgoal can be directly represented as an agent.

Example 11 (Poly, with agents).

```
poly Term Type -:  sigma top \ p Term top Type top.
p (fn M) D Y X -:
  sigma d \ sigma e \ sigma x \ sigma q \
    (Y = (all d \ all e \ q d e),
     tybind x d,
     D = d --> e,
     p (M x) e (q d e) X).
p X R Head Tail -:
   pi D (tybind X D => D = R, Head = Tail).
p (M@N) R Y Tail -:
   sigma d \ pi Mid \ sigma q \
   (Y = (all q),
    p M (d --> R) (P d) Mid,
    p N d Mid Tail).
gen Term (all d \ P d) Term -: pi d \ gen Term (P d).
gen Term (D --> R) -: mono Term  (D --> R).
mono (M @ N) B        -: mono M (A --> B), mono N A.
mono (fn M) (A --> B) -: pi x\ tyvar x A => mono (M x) B.
mono T A              -: tyvar T A.
```

The query `poly (fn x  (fn y  x)) Type` *generates the final configuration:*

```
top = d --> e, e = d1 --> e, d = e1,
Type = all d \ all e \ P d e, P = all d1 \ all e1 P1 d1 e1,
P1 d1 e1 = top,
Type d' e' d1' e1' = A --> B --> A
```

This store entails `Type = all d  all e  all d1  all e1 d --> d1 --> d`

If we add to the constraint system the following properties of `all`:

```
all d\P = P
all d\ all e \ P = all e \ all d \ P
```

then the store will uniquely entail `Type = all d  all e d --> e --> d`, the desired result.

D.2 Representing operational semantics in λRCC.

λRCC is rich enough to directly represent structural operational semantics of modern stateful, object-oriented, concurrent programming languages such as JAVA and X10 using ideas developed in the context of implementing "natural semantics" in Prolog-like languages. A configuration can be represented as a (possibly higher-order) λRCC term and the transition relation as a binary test-predicate, `step`: the relation `step s1 s2` is intended to hold precisely when the configuration represented by `s1` can transition to the configuration represented by `s2`. Using a test-predicate rather than an agent-predicate allows the representation of non-deterministic transition relations.

We illustrate with a presentation of the operational semantics of CCP and λRCC.

Example 12 (Interpreter for CCP). For instance, here is an interpreter for a CCP *language, in* λRCC*:*

```
step A A.
step Agents Final :-
   select Agents (tell C) Rest,
   tell C => (step Rest Final).
step Agents Final :-
   select Agents (ask C A) Rest,
   ask C,
   select New A Rest,
   step New Final.
step Agents Final :-  select Agents true Rest,  step Rest Final.
step Agents Final :-  select Agents (and A1 A2) Rest,
   select Rest1 A1 Rest, select Rest2 A2 Rest, step Rest2 Final.
step Agents Final :-
   select Agents (atomic A), Rest,
   clause A Body,
   select Rest1 Body Rest,
   step Rest1 Final.
step Agents Final :-
   select Agents  (new X A) Rest,
   pi x \ (select Rest1  (A x) Rest),
           step Rest1 Final).
```

It directly represents the constraint store of the object program as the store of the meta-program.

Example 13 (Meta-interpreter for RCC*).*

```
rcc D (some B)  :- rcc D (B _T).
rcc D (all B)   :- pi c \ rcc D (B c).
rcc D (B and C) :- rcc D B, rcc D C.
rcc D (B or C)  :- (rcc D B ; rcc D C).
rcc D (D1 imp G) :- rcc (D1::D) G.
rcc D (goal A)  :- member (G imp (goal A)) D, rcc D G.
rcc D (a (constraint c))  :- store D Store, entails Store c.
rcc D true.
```

```
rcc D G  :- member (agent E) D, member ((agent E) imp D1) D, rcc (D1::D) G.
rcc D G  :- member (G1 imp D1) D, rcc D G1, rcc (D1::D) G.
rcc D G  :- select (D1 and D2) D R, rcc D1::D2::R G.
rcc D G  :- member (all D1) D, rcc (D1 _T)::D G.

store D S :- store D S nil.
store D::D1 S T:- store D S M, store D1 M T.
store nil S S.
store (D1 and D2) S T :- store D1 S M, store D2 M T.
store (t (constraint C)) C::S S.
store (all D) S S.
store (G gImpD D) S S.
store (D dImpD D1) S S.
store (G gImpG G) S S.
```

For operational semantics of languages other than constraint languages, the power of constraints can be usefully applied in two ways. First, operational semantics is typically restricted to using "syntactic" (very concrete) data-structures such as trees of processes and finite functions from cells to values. λRCC permits a higher-level representation of these concrete data-structures using constraints and operations on constraints.

Example 14 (Deadlock-detecting parallel language). For instance, consider a hypothetical concurrent programming language that permits deadlock to be discovered and throws an exception in the oldest activity involved in a deadlock. This may be represented quite simply as:

```
step S1 S2 :-
   pi graph \ waitfor S1 graph => cyclic graph,
   breakDeadlock S1 S2.
```

Here `waitfor` *is an agent-predicate that traverses the state* `S1` *compositionally generating constraints on a graph structure that represents the "wait for" relation between suspended processes. The query* `cyclic` *is a primitive query in the graph constraint language that checks if the graph has a cycle in it.*

Second, constraints may sometimes provide the right level of abstraction to define a programming construct.

Example 15 (Clocks in X10*, [SJ05]). The programming language* X10 *uses constraints in the heap to detect quiescence of barrier-based computations. These $\mathcal{O}$-constraints can be directly represented as λ*RCC *constraints in the store, thereby considerably simplifying the formal presentation of the operational semantics.*

We note in passing that this modeling of operational semantics may not permit the representation of parallel composition in $\mathcal{O}$ by parallel composition in λRCC because $\mathcal{O}$ may require shared mutable state. Instead, the entire set of parallel processes in an $\mathcal{O}$ computation must be represented as a single λRCC term; thus λRCC can provide a "global" view of $\mathcal{O}$-computations. An extension of λRCC based on linear logic may be able to provide such a compositional representation.