Representing and reasoning with operational semantics

Dale Miller

INRIA & LIX, École Polytechnique

Abstract. The operational semantics of programming and specification languages is often presented via inference rules and these can generally be mapped into logic programming-like clauses. Such logical encodings of operational semantics can be surprisingly declarative if one uses logics that directly account for term-level bindings and for resources, such as are found in linear logic. Traditional theorem proving techniques, such as unification and backtracking search, can then be applied to animate operational semantic specifications. Of course, one wishes to go a step further than animation: using logic to encode computation should facilitate formal reasoning directly with semantic specifications. We outline an approach to reasoning about logic specifications that involves viewing logic specifications as theories in an object-logic and then using a meta-logic to reason about properties of those object-logic theories. We motivate the principal design goals of a particular meta-logic that has been built for that purpose.

1 Roles for logic in the specification of computations

There are two broad approaches to using logic to specify computational systems. In the *computation-as-model* approach, computations are encoded as mathematical structures, containing such items as nodes, transitions, and state. Logic is used in an external sense to make statements *about* those structures. That is, computations are used as models for logical expressions. Intensional operators, such as the modals of temporal and dynamic logics or the triples of Hoare logic, are often employed to express propositions about the change in state. This use of logic to represent and reason about computation is probably the oldest and most broadly successful use of logic for specifying computation.

The *computation-as-deduction* approach uses pieces of logic's syntax (formulas, terms, types, and proofs) directly as elements of the specified computation. In this much more rarefied setting, there are two rather different approaches to how computation is modeled. The *proof normalization* approach views the state of a computation as a proof term and the process of computing as normalization (via β-reduction or cut-elimination). Functional programming can be explained using proof-normalization as its theoretical basis [23]. The *proof search* approach views the state of a computation as a sequent (a structured collection of formulas) and the process of computing as the search for a proof of a sequent: the changes that take place in sequents capture the dynamics of computation. Proof search has been used to provide a theoretical foundation for logic programming [33] and to justify the design of new logic programming languages [31].

The divisions proposed above are informal and suggestive: such a classification is helpful in pointing out different sets of concerns represented by these two broad approaches (reduction, confluence, etc, versus unification, backtracking search, etc). Of course, a real advance in computation logic might allow us merge or reorganize this classification.

This classification can help to categorize the various proof systems that have been used to reason about computation systems. For example, the computation-as-model approach usually implies that one divides the problem of reasoning about computation into two steps. In the first step, one *implements mathematics* via some set-theory or higher-order logic (for example, HOL [14], Isabelle/ZF [46], PVS [44]). In the second step, one reduces program correctness problems to mathematics. Thus, data structures, states, stacks, heaps, invariants, etc, all are represented as various kinds of mathematical objects. One then reasons directly on these objects using standard mathematical techniques (induction, primitive recursion, fixed points, well-founded orders, etc).

Researchers who specify computation using the proof-normalization approach usually first implement mathematics, but this time, in a constructive mathematics, using, for example, Martin-Löf type theory [23] and higher-order intuitionistic logic or dependent type theory (for example, Coq [9] and NuPRL [8]).

This paper describes another possibility for the construction of a prover that takes its inspiration from the proof search approach to the specification of computation.

2 The proof search paradigm

As one builds cut-free proofs of sequents (in the sense of, say, Gentzen [12]), sequents change and this change, reading proofs bottom-up, is used to capture the dynamics of computation that one intends to model. The cut-rule and the cut-elimination process do not have roles to play during this simulation of computation: instead, they can play important roles in reasoning about specified computations.

While proof search can be seen as a means of giving a broad foundation to logic programming, there are a number of aspects of proof search (as computation) that have not been embraced by the general logic programming community. For example, proof search relies primarily on proof theory for new designs and analysis tools, instead of model theory as is more customarily used by logic programming researchers. Proof search generally stresses logically correct deduction even if non-logical techniques, such as dropping all occur-checks during unification and using the `!` pruning or “cut” operator of Prolog, can yield more effective executions. Also, proof search design and theory also focuses on the meaning of logical connectives and quantifiers for expressiveness and for new designs. Such a focus is in contrast to, say, constraint logic programming [21].

As we highlight in the rest of this section, the proof search paradigm allows for a relatively straightforward treatments of such “intensional” aspects of computation as binders, binder mobility, and resource management.

2.1 Encoding symbolic expression via λ-tree syntax.

Most human authored and readable symbolic expressions start life as strings: such linearly organized data are full of semantically unimportant information such as white space, infix/postfix operators, and parentheses. Before processing such *concrete syntax*, one removes much of this concrete nonsense by parsing the data into a more abstract representation we call here *parse trees* (often also called *abstract syntax*).

Most parsing technology unfortunately leaves the names of bound variables in the resulting parse trees. Although binders are, of course, important aspects of the meaning of computational objects, the name of variables used to encode binders are another form of concrete nonsense. Since dealing with bindings in syntax is a well known problem, various techniques are available to help make this concrete and semantically meaningless aspect of syntax more abstract. One approach to bindings in syntax uses deBruijn numerals [5]. While such an encoding has proved its value within implementations, deBruijn numerals seem too explicit and complicated to support declarative reasoning of syntax. Other approaches involve the direct use of named variables and a version of set theory to accommodate names and renaming [11].

The *higher-order abstract syntax* (*hoas*) [47] approach to encoding syntax proposes that bindings in data should be mapped to the bindings found in whatever programming or specification language one is using. Within functional programming, term-level binders are then encoded as functional objects. While some interesting specifications have resulted [10, 18], this approach has numerous semantic problems. For example, while expressions with bindings are still intended to be finite and syntactic objects, the corresponding functions yields values that are usually infinite in extension. Also, there are usually many more constructors for function spaces than simply the λ-abstraction within a functional programming setting: for example, recursive function definition.

In contrast, the proof search approach to the specification of computation allows for a different approach to hoas. In that setting, λ-terms modulo various weak subsets of λ-conversion can be used to directly encode expressions. Here, α-conversion abstracts away from the names of bindings, β_0-conversion allows for *binder mobility* [28, 30], and β-conversion allows for object-level substitution. We shall use the term λ-*tree syntax* [29] to denote the proof search approach to hoas. While there is a long history of using λ-tree syntax in specifying computation, starting with Huet and Lang [20] and Miller and Nadathur [32], few computer systems actually support it directly: the λProlog [40] programming language and the Isabelle [43] and Twelf [48] specification languages are the best known exceptions.

Using meta-level application to encode object-level applications is standard practice: for example, one uses meta-level application to apply, say, *cons*, to two arguments: (*cons* X L). The λ-tree syntax approach is simply a dualizing of this practice that uses meta-level abstraction to encode object-level binders.

2.2 Encoding computational processes as formula or as terms

It seems that there are two choices one can make when encoding "active" components of a computation into proof search. Usually, these active objects, such as computation threads, automata states, etc, which we collectively call here as simply "processes", are encoded as terms. In this *process-as-term* approach, predicates are then used to state relationships between computational items. For example, we have statements such as "M has value V", "in context Γ, M has type σ", "P makes an A transition and becomes Q", etc. Given such atomic formulas, one then encodes operational semantics as compound formulas within either an intuitionistic or a classical logic. For an example of encoding the π-calculus using this process-as-term approach, see [35, 54] and Section 6.

With the availability of linear logic and other sub-structural logics, it seems sensible to consider another style of encoding where processes are encoded directly as formulas. In the *process-as-formula* approach to encoding, formulas no longer denote truth values: instead they denote "resources" which can change over time. In such a setting, the combinators of a processes calculus are mapped to logical connectives and the environment of a computation thread (including, for example, memory and other threads) are modeled via a logical context (within a sequent, for example). In principle, this approach requires fewer non-logical constants than are used with the process-as-term approach. There is a large literature of specifying programming language features in this manner using linear logic [31].

While encodings using the process-as-formula approach can often capture the notion of process execution or of reachability, they fail to directly support rich notions of program or process equivalences, such as bisimulation or observational equivalence. To capture these equivalences, the process-as-term approach has provided more successes.

3 Operational semantics as logic specification

A common style of operational semantics specification is presented as inference rules involving relations. We illustrate how such semantic specifications can be mapped into logical specifications.

For example, some of the rules for specifying CCS [37] are given by the following inference rules.

$$\frac{P \xrightarrow{A} R}{P + Q \xrightarrow{A} R} \qquad \frac{Q \xrightarrow{A} R}{P + Q \xrightarrow{A} R} \qquad \frac{P \xrightarrow{A} P'}{P|Q \xrightarrow{A} P'|Q} \qquad \frac{Q \xrightarrow{A} Q'}{P|Q \xrightarrow{A} P|Q'}$$

By viewing $+$ and $|$ as constructors of processes and $\cdot \xrightarrow{\cdot} \cdot$ as a predicate of three arguments, it is easy to write these inference rules as the following first-order Horn clauses.

$$\forall P \forall Q \forall A \forall R[P \xrightarrow{A} R \supset P + Q \xrightarrow{A} R]$$

$$\forall P \forall Q \forall A \forall R[Q \xrightarrow{A} R \supset P + Q \xrightarrow{A} R]$$

$$\forall P \forall A \forall P' \forall Q[P \xrightarrow{A} P' \supset P|Q \xrightarrow{A} P'|Q]$$

$$\forall P \forall A \forall Q' \forall Q[Q \xrightarrow{A} Q' \supset P|Q \xrightarrow{A} P|Q']$$

For a slightly more challenging specification of operational semantics, we consider a specification of call-by-name evaluation, which involves bindings and substitution (call-by-value evaluation can also be used here just as easily). Let the type tm denote the syntactic category of untyped λ-terms and let the two constructors abs of type $(tm \to tm) \to tm$ and app of type $tm \to tm \to tm$ denote abstraction and application within the untyped λ-calculus, respectively. This encoding places α-equivalence classes of untyped λ-terms in one-to-one correspondence with $\beta\eta$-equivalence classes of terms of type tm. To specify call-by-name evaluation, we use an infix binary predicate $\Downarrow$ to denote evaluation between two arguments of type tm. Call-by-name evaluation can be specified by the following two inference rules.

$$\frac{}{(abs\ \lambda x.S) \Downarrow (abs\ \lambda x.S)} \qquad \frac{M \Downarrow (abs\ \lambda x.S) \quad (S[x/N]) \Downarrow V}{(app\ M\ N) \Downarrow V}$$

To translate these inference rules into logic, one needs to explain carefully the proper treatment of binding (here, λx) and the definition of term-level substitution (here, $S[x/N]$). As is often observed, these details are complex and there are a number of different solutions. Furthermore, dealing with all those details does not help in understanding the essential *semantics* of such a specification rule. Fortunately, we can simply invoke the λ-tree approach to syntax to address these problems. In particular, we assume that our logic contains variables of higher-order type (in particular, of type $tm \to tm$) and that it contains an equality of simply types that includes $\beta\eta$-conversion. In this way, we can simply reuse the careful specification done by, say, Church in [6], of how λ-abstraction and logic interact. Given this motivation, we can now choose to write the above specification as simply the following (higher-order) Horn clauses [41]:

$$\forall R.[(abs\ R) \Downarrow (abs\ R)]$$

$$\forall M \forall N \forall V \forall R.[M \Downarrow (abs\ R) \wedge (R\ N) \Downarrow V \supset (app\ M\ N) \Downarrow V]$$

Here, R has type $tm \to tm$ and corresponds to the expression $\lambda x.S$ and the substitution $S[x/N]$ is replaced by the expression $(R\ N)$.

Various forms of static analysis, such a typing, can be specified using inference rules as well. Consider, for example, the specification of simple typing for the untyped λ-calculus. To specify simple typing for the untyped λ-calculus, we introduce the logic-level type ty to denote the syntactic category of simple type expressions and use the constructors *gnd* of type *ty* (denoting a ground, primitive type) and *arr* of type $ty \to ty \to ty$ (denoting the function type constructor). The usual rule for simple typing is given as follows:

$$\frac{\Gamma \vdash M\colon (arr\ U\ T) \qquad \Gamma \vdash N\colon U}{\Gamma \vdash (app\ M\ N)\colon T} \qquad \frac{\Gamma, x\colon T \vdash S\colon U}{\Gamma \vdash (abs\ \lambda x.S)\colon (arr\ T\ U)}(\dagger)$$

The second inference rule has the proviso (†): x must be a new variable; that is, it is not free in T, U, nor in any of the pairs in Γ. To encode these inference rules into logic, we first pick a binary predicate *typeof*, whose arguments are of type *tm* and *ty*, respectively, to denote the colon relation above. Then the following formulas provide an elegant encoding of these typing inference rules.

$$\forall M \forall N \forall T \forall U [typeof\ M\ (arr\ U\ T) \wedge typeof\ N\ U \supset typeof\ (app\ M\ N)\ T]$$

$$\forall R \forall T \forall U [\forall x.[typeof\ x\ T \supset typeof\ (R\, x)\ U] \supset typeof\ (abs\ R)\ (arr\ T\ U)]$$

Notice that these formulas are no longer Horn clauses. The use of λ-tree syntax allows for dispensing with any explicit reference to bindings. The use of the implication in the body of clauses means that the explicit context Γ is being managed implicitly by logic. The term-level binding in λx can be seen as "moving" to the formula-level binding $\forall x$. During proof search, this formula-level binding will be replaced with an eigenvariable: thus, this formula-level binding will move to a proof-level binding. Such *binder mobility* gives λ-tree syntax one of its strength: a specification does not need to specify details about how binders are encode, instead, binders only need to be moved from term-level to formula-level to proof-level bindings. Details of binders need to be addressed only by implementors of the logic.

4 What good is a logic specification anyway?

People working in programming language specification and implementation have a history of using declarative tools. For example, both lexical analyzers and parsers are often generated by special tools (e.g., lex and yacc) that work from such declarative specifications as regular expressions and context-free grammars. Similarly, operational semantics has been turned into interpreters via logic programming engines [4] and denotational semantics have been used to generate compilers [45].

Given a history of interest in declarative techniques to specify programming language systems, it seems natural to now focus on the question: why should anyone care that we have written an operational semantic specification or a typing relation declaratively? What benefits should arise from using λ-tree syntax, from using intuitionistic logic or linear logic?

One benefit arises from the fact that logic is a difficult discipline to follow: the efforts of the specifier to hammer a specification into a declarative setting that lacks, for example, side-conditions, can often lead to new ways of thinking about what one is specifying. Such rarefied and declarative settings can also allow broad results to be inferred from specifications: for example, the fact that bisimulation is a congruence can be established for process calculi (see, for example, [15, 56]) or for functional programming languages [19] by checking syntactic conditions on the declarative specification of operational semantics.

Another benefit is that an implementation of logic might provide a uniform means to animate a wide range of logic specifications.

The benefit that concerns us here, however, is that a logic specification should facilitate the inferring of formal properties. While this might sound obvious, designing a "meta-logic" for reasoning about logic specifications requires some work. We motivate via some examples one particular meta-logic.

5 Example: A subject-reduction theorem

Consider again the specification of evaluation and typing given in Section 3. The following theorem is usually called the *type preservation* or the *subject-reduction* theorem. The informal proof of this theorem below is taken from [24].

Theorem 1 *If P evaluates to V and P has type T then V has type T.*

Proof. We write $\vdash B$ to mean that there is a *uniform proof* of B, where uniform proofs are certain kinds of cut-free proofs that have been used to formalize the notion of goal-directed proof search [33]. Restricting to such uniform proofs in this setting does not result in a loss of completeness. We proceed by induction on the structure of a uniform proof of $P \Downarrow V$ that for all T, if $\vdash \mathit{typeof}\ P\ T$ then $\vdash \mathit{typeof}\ V\ T$. Since $P \Downarrow V$ is atomic, its proof must end by backchaining on one of the formulas encoding evaluation. If the backchaining is on the $\Downarrow$ formula for *abs*, then P and V are both equal to $\mathit{abs}\ R$, for some R, and the consequent is immediate. If $P \Downarrow V$ is proved using the $\Downarrow$ formula for *app*, then P is of the form $\mathit{app}\ M\ N$ and for some R, there are shorter proofs of $M \Downarrow (\mathit{abs}\ R)$ and $(R\,N) \Downarrow V$. Since $\vdash \mathit{typeof}\ (\mathit{app}\ M\ N)\ T$, this typing relation must have been proved using backchaining and, hence, there is a U such that $\vdash \mathit{typeof}\ M\ (\mathit{arr}\ U\ T)$ and $\vdash \mathit{typeof}\ N\ U$. Using the inductive hypothesis, we have $\vdash \mathit{typeof}\ (\mathit{abs}\ R)\ (\mathit{arr}\ U\ T)$. This atomic formula must have been proved by backchaining on the *typeof* formula for *abs*, and, hence, $\vdash \forall x.[\mathit{typeof}\, x\ U \supset \mathit{typeof}\,(R\,x)\ T]$. Since our meta-language is (intuitionistic) logic, we can instantiate this quantifier with N and use cut and cut-elimination to conclude that $\vdash \mathit{typeof}\,(R\,N)\ T$. (This last step is essentially a "substitution lemma" which comes for free given cut elimination and our use of λ-tree syntax.) Using the inductive hypothesis a second time yields $\vdash \mathit{typeof}\, V\ T$.

This proof is clear and natural and we would like our meta-logic to support similarly structured proofs. This example suggests that the following features would be valuable in the meta-logic.

1. *Two distinct logics.* In the above informal proof, there are clearly two distinct logics being used. One logic is written with logical syntax and describes

some relations, e.g. typability and evaluation. The second logic is written with English text: atomic formulas of that logic are (provability) judgments about the object-logic. This use of two distinct logics – one for the specifying operational semantics and typing and one for the meta-logic – is an important aspect of our approach to reasoning about computation.

2. *Structural induction* over object-level sequent calculus proofs was used. Obviously induction over other structures (e.g., expressions, formulas) and co-induction play important roles in meta-level reasoning about computation.
3. The *instantiation of meta-level eigenvariables* was used in this proof. In particular, the variable P was instantiated in one part of the proof to $(abs\ R)$ and in another part of the proof to $(app\ M\ N)$. Notice that such instantiation of eigenvariables within a proof does not happen in proof search in conventional sequent calculi.
4. The *inversion of assumed judgment* was used in the above proof a few times, leading, for example, from the assumption $\vdash typeof\ (abs\ R)\ (arr\ U\ T)$ to the assumption $\vdash \forall x[typeof\ x\ U \supset typeof\ (R\,x)\ T]$. The specification of *typeof* allows the implication to go in the other direction, but given the structure of the specification of *typeof*, this direction can also be justified at the meta-level.

The system Twelf is capable of proving such type preservation properties along rather similar lines, except that an explicit meta-logic with an explicit induction rule is replaced by a meta-level tool that checks properties such as coverage and termination [51].

In the example above, bindings in the object-logic and object-language played a small role: they were treated only by instantiation. In the next section, we consider the π-calculus since it provides a more challenging problem for dealing with bindings in syntax and in computations.

6 Example: A π-calculus specification

To encode the syntax of the π-calculus, let the types p, n, and a denote the syntactic category of processes, names, and actions, respectively. A signature for the π-calculus can thus be listed as

$$0 : p, \qquad out : n \to n \to p \to p, \qquad in : n \to (n \to p) \to p,$$
$$+,\ | : p \to p \to p, \quad match : n \to n \to p \to p, \quad \nu : (n \to p) \to p.$$

For example, the expression $x(y).P$, where x is a name and y is a binding with scope P, can be encoded using a constructor *in* as the expression $(in\ x\ (\lambda y.P'))$. Similarly, the restriction operator $\nu x.P$ can be encoded as $\nu(\lambda x.P)$.

We next introduce three constructors for actions: τ denotes the silent action and the down arrow $\downarrow$ and up arrow $\uparrow$ encode input and output actions, resp: in particular, the expression $(\downarrow xy)$ denotes an input action on channel x of value y. Notice that the two expressions, $\lambda y.\uparrow xy$ and $\uparrow x$, denoting *abstracted actions*, are equal up to η-conversion and can be used interchangeably.

To specifying the operational semantics of the π-calculus, we use the horizontal arrow $\longrightarrow$ to relate a process with an action and a continuation (a process), and the "harpoon" $\longrightharpoonup$ to relate a process with an *abstracted* action and an *abstracted* continuation (of types $n \rightarrow a$ and $n \rightarrow p$, resp.).

The following three rules (named (CLOSE), (RES), (OPEN)) are part of the specification of one-step transitions for the π-calculus: the full specification using λ-tree syntax can be found in, for example, [34, 36].

$$\frac{P \xrightharpoonup{\downarrow X} M \qquad Q \xrightharpoonup{\uparrow X} N}{P \mid Q \xrightarrow{\tau} \nu y.(My \mid Ny)} \qquad \frac{\forall n(Nn \xrightarrow{A} Mn)}{\nu n.Nn \xrightarrow{A} \nu n.Mn} \qquad \frac{\forall y(Ny \xrightarrow{\uparrow Xy} My)}{\nu y.Ny \xrightharpoonup{\lambda y.\uparrow Xy} \lambda y.My}$$

The (CLOSE) rule describes how a bound input and bound output action can yield a τ step with a ν-restricted continuation. The (RES) rule illustrates how λ-tree syntax and appropriate quantification can remove the need for side conditions: since substitution in *logic* does not allow for the capture of bound variables, all instances of the premise of this rule have a horizontal arrow in which the action label does not contain the universally quantified variable free. Thus, the usual side condition for this rule is treated declaratively. There is a direct translation of such inference rules into, say, λProlog [40], in such a way that one can directly animate the operational semantics of the π-calculus.

7 Example: Bisimulation for the π-calculus

There seems to be something questionable about the use of the universal quantifier in the premises of the operational semantics for the π-calculus above. For example, the (RES) rule says that if $Nn \xrightarrow{A} Mn$ is provable for all instances of the free variable n then the transition $\nu n.Nn \xrightarrow{A} \nu n.Mn$ is justified. This does not seem to be a completely correct sense of what is implied by the original specification rule of the π-calculus. A more correct sense of the rule should be something like: if $Nn \xrightarrow{A} Mn$ is provable for some *new* name n, then the above conclusion is justified. In a proof search setting involving only positive inference about computation (for example, judgments involving only *may* behavior of a process), such a quantifier appears only positively and is instantiated with a new (proof-level bound) variable called an *eigenvariable*. In this setting, the notion of *new* name is supported well by the universal quantifier. If, however, negative information is being inferred, as is possible with judgments involving *must* behaviors, then the universal quantifier is instantiated with any number of existing names. This seems like the wrong meaning for this rule.

To illustrate this example more concretely, note that for any name x, the process $\nu y.[x = y]\bar{x}z$ is bisimilar to 0: that is, this process can make no transitions. This fact also seems to follow from the nature of bindings: the scope of the bindings for x and for y are such that any instance of x can never equal y (a simple consequence of that fact that sound substitutions avoid variable capture).

Now, proving this bisimulation fact should be equivalent to proving

$$\forall x \forall A \forall P' \neg(\nu y.[x = y]\bar{x}z \xrightarrow{A} P')$$

Using the above operational semantics, this should be equivalent to proving

$$\forall x \forall A \forall P'' \neg \forall y([x = y]\bar{x}z \xrightarrow{A} P'') \quad \text{and} \quad \forall x \forall A \forall P'' \exists y \neg([x = y]\bar{x}z \xrightarrow{A} P'')$$

Now it seems that standard proof theory techniques will not achieve a proof: somehow we need to have additional information that for every name there exists another name that is distinct from it. Adopting such an axiom is often done in many settings, but this seems to go against the usual spirit of sequent calculus (a system usually containing *no* axioms) and against the idea that proof theory is, in fact, an ideal setting to deal with notions of bindings and scope directly.

To come up with a proof theoretic approach to address this problem with using the $\forall$-quantifier in operational semantics, Miller and Tiu [35, 36] introduced the ∇-quantifier: the informal reading of $\nabla x.Bx$, in both positive and negative settings, is equivalent to Bx for a new name x. To support this new quantification, sequent calculus is extended to support a notion of "generic judgments" so that "newness" remains a (proof-level) binding and can be seen as being hypothetical. That is, the truth condition of $\nabla x.Bx$ roughly reduces to the conditional "if a new name c is created then Bc." Notice that no assumption about whether or not the domain of quantification is non-empty is made (this detail makes ∇ behave differently from the Gabbay-Pitts "newness quantifier" [11]). If one is interested only in establishing one-step transitions (and not their negation), then it is possible to use ∇ and $\forall$ in the premises of the operational semantics for the π-calculus interchangeably.

Using ∇-quantification instead of $\forall$-quantification in the premise of the (RES) rule does, in fact, allow proving the formula

$$\forall x \forall A \forall P' \neg(\nu y.[x = y]\bar{x}z \xrightarrow{A} P'),$$

since this now reduces to $\forall x \forall A \forall P'' \nabla y \neg([x = y]\bar{x}z \xrightarrow{A} P'')$. If one follows the proof theory for ∇ carefully [36] this negation is provable because the expressions $\lambda y.x$ and $\lambda y.y$ do not unify (for free variable x). Notice that the binding of y is maintained all the way to the level of unification where, in this case, it ensures the correct failure to find an appropriate instance for x.

Using the ∇-quantifier, it is now easy and natural to specify bisimulation for the π-calculus with the equivalence displayed in Figure 1. Notice the elegant parallelism between using $\forall$ to quantify bisimulation of abstracted continuations for bound inputs and using ∇ to quantify bisimulation of abstracted continuations for bound outputs. As is shown in [54], proving this formula in intuitionistic logic yields *open bisimulation* [49]. Without an inference rule for co-induction, this equivalence is only correct for the *finite* π-calculus (a subset of the π-calculus that does not contain the replication operator ! nor recursive definition of processes). If we add the excluded-middle assumption $\forall w \forall z(w = z \vee w \neq z)$ (which

$$\begin{aligned}
\mathit{bisim}\ P\ Q \equiv\ & \forall A \forall P' [P \xrightarrow{A} P' \Rightarrow \exists Q'.Q \xrightarrow{A} Q' \wedge \mathit{bisim}\ P'\ Q'] \wedge \\
& \forall A \forall Q' [Q \xrightarrow{A} Q' \Rightarrow \exists P'.P \xrightarrow{A} P' \wedge \mathit{bisim}\ Q'\ P'] \wedge \\
& \forall X \forall P' [P \xrightarrow{\downarrow X} P' \Rightarrow \exists Q'.Q \xrightarrow{\downarrow X} Q' \wedge \forall w.\mathit{bisim}\ (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q' [Q \xrightarrow{\downarrow X} Q' \Rightarrow \exists P'.P \xrightarrow{\downarrow X} P' \wedge \forall w.\mathit{bisim}\ (Q'w)\ (P'w)] \wedge \\
& \forall X \forall P' [P \xrightarrow{\uparrow X} P' \Rightarrow \exists Q'.Q \xrightarrow{\uparrow X} Q' \wedge \nabla w.\mathit{bisim}\ (P'w)\ (Q'w)] \wedge \\
& \forall X \forall Q' [Q \xrightarrow{\uparrow X} Q' \Rightarrow \exists P'.P \xrightarrow{\uparrow X} P' \wedge \nabla w.\mathit{bisim}\ (Q'w)\ (P'w)]
\end{aligned}$$

Fig. 1. A specification of bisimulation for the π-calculus.

is trivially true in a classical meta-theory), the resulting specification can be used to specify *late bisimulation* [38] (see [54] for the precise statements regarding this specification). A logic programming-style implementation of proof search provides an immediate *symbolic bisimulation* checker (in the sense of [17, 3]) for the finite π-calculus [54, 53].

8 The LINC meta-logic

The three examples above allow us to now motivate the design of a meta-logic that can be used to state properties of object-level provability and, hence, to reason about operational semantics (via their encodings into object-logics). Our meta-logic is called LINC, an acronym coined by Tiu [52] for "lambda, induction, nabla, co-induction." This logic contains the following three key features.

First, LINC is based on the predicative and intuitionistic fragment of Church Simple Theory of Types [6] (restricted to the axioms $1-6$). Provability for this logic can be described as being essentially Gentzen's LJ sequent calculus [12] to which is added simple typing for all variables and constants, quantification at all types (excluding the type of predicates), and an inference rule that allows $\beta\eta$-conversion on any formula in a sequent. This logic provides support for λ-tree syntax. Considering a classical logic extension of LINC is also of some interest, as is an extension allowing for quantification at predicate type.

Second, LINC incorporates the proof-theoretical notion of *definition*, a simple and elegant device for extending logic with the if-and-only-if closure of a logic specification (similar to the *closed-world assumption* [7]). This notion of definition was developed by Hallnäs and Schroeder-Heister [16, 50] and, independently, by Girard [13]. This feature of definitions allows for the "inversion of assumed judgments" mentioned at the end of Section 5 and for the ability to capture not just *may* behavior but also *must* behavior. In particular, definitions are central to the treatment of bisimulation mentioned in Section 7 (see also [27]) and for doing model checking directly with operational semantics (see, for example, [53, 55]). It also allows for certain failures of proof search to be turned into successful proofs of negations. Definitions are also a natural place to incorporate inductive

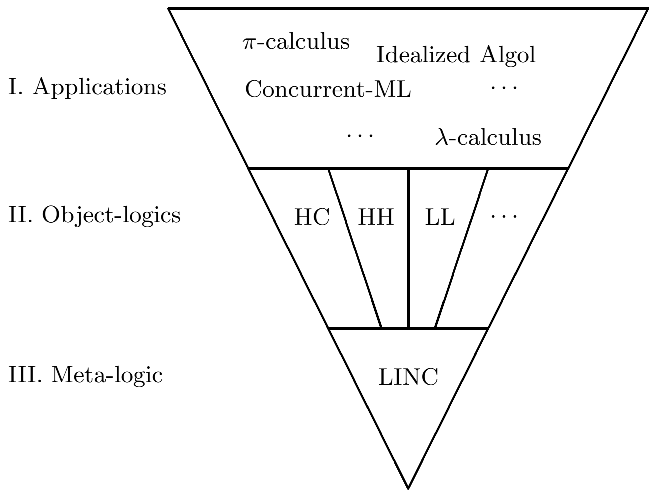

Fig. 2. A three level architecture.

and co-inductive inference rules: for full details, see paper by McDowell, Miller, Momigliano, and Tiu [25, 26, 35, 39, 52].

Third, LINC contains the ∇ quantifier, which, as we just illustrated, allows for more natural and direct reasoning about syntactic encodings based on λ-tree syntax.

As Tiu has shown in [52], under restrictions of appropriately "stratified" definitions (a restriction which rules out, for example, a predicate being defined as its own negation), the LINC logic satisfies cut-elimination. The logic $FO\lambda^{\Delta\mathbb{N}}$ of [25, 26] is a subset of LINC, corresponding roughly to the fragment that results from deleting the ∇-quantifier, removing co-induction, and limiting induction to natural number induction.

The principal use of the ∇-quantifier is helping with the treatment of bindings in λ-tree syntax encodings. In fact, we know of no use of ∇ in specifications that involve only, say, first-order terms. It is also the case that ∇ is interchangeable with $\forall$ when definitions are "positive": that is, when they contain no occurrences of implications and negations. In such Horn clause-like definitions, one can interchange these two quantifiers in the body of definitions without affecting the atomic formulas that are provable [36].

9 Formal reasoning about logic specifications

Figure 2 presents an architecture for organizing the various symbolic systems involved with the specification of and reasoning about computation systems. The top level contains the *many* applications about which we hope to provide formal proofs. Possible applications should include programming languages, specifications languages, security protocols, type systems, etc.

The middle layer contains a *few* object-level logics, such as Horn clauses (HC), hereditary Harrop formulas (HH) [33], and linear logic (LL). These logics all have well understood meta-theories and their operational semantics is given by proof search following the normal forms dictated by uniform proofs and backchaining [33] or focused proofs [1]. In fact, all of these logics can be seen as modularly sitting inside one single logic, namely, linear logic.

The bottom layer consists of the single logic LINC, where object-level provability must be encoded (as an abstracted logic programming interpreter) and important results about object-level provability (including cut-elimination) must be proved. Also, object-level logic specifications used to capture aspects of an application must also be encoded into the meta-level. Since the meta-logic and object-logic share the same application and abstraction, terms used to encode application-level objects (for example, a π-calculus expression) are the same at both levels.

To illustrate these three-levels, consider the proof of the subject-reduction theorem in Section 5. The application level contains two classes of linguistic items: untyped λ-terms (constructed using *abs* and *app*) and simple type expressions (constructed using *gnd* and *arr*). The formulas in Section 5 that specify evaluation and typing are object-level (hereditary Harrop) formulas. At the meta-level, such formulas are simply terms, where object-level predicates, such as $\Downarrow$ and *typeof*, are now binary constructor in the meta-logic and where object-level logic connectives and quantifiers are also meta-level term constructors (requiring meta-level λ-abstraction to encode quantifiers). Formulas at the LINC (meta-logic) level must now encode the notion of provability for the object-level logic as well as any other judgments that are specific to the application being considering (such as, say, bisimulation). For example, provability of hereditary Harrop formulas can be defined in LINC via a predicate, say, $seq\ \Gamma\ B$ to describe when the object-level formula B is provable from the list of object-level formulas in Γ and the object-level formulas describing evaluation and typing (see [26, Section 4.3] for specifics on how this can be done). The meta-level formula that one wishes to prove within LINC is then

$$\forall P \forall V[seq\ nil\ (P \Downarrow V) \supset \forall T[seq\ nil\ (typeof\ P\ T) \supset seq\ nil\ (typeof\ V\ T)]]$$

This and many similar theorems are proved in [26, 36].

For another example, consider again the π-calculus examples given above. In Section 6, operational semantics was given using Horn clauses that allowed $\forall$ in their bodies. When we moved to Section 7, we needed to make sure that these $\forall$-quantifiers were replaced by ∇-quantification. This transition is now easily explained: when specifying at the LINC level an interpreter for object-level Horn clauses, that interpreter will naturally translate the object-level conjunction and existential quantifier to the meta-level conjunction and existential quantifier. It will, however, need to translate the object-level universal quantifier to the meta-level ∇-quantifier (for full details, see [36, Section 6]).

10 Future work and conclusions

We have described how logic can be used to specify operational semantics: the logics used for this purpose are essentially logic programming languages based in either classical, intuitionistic, or linear logic. These logics generally use higher-type quantification in order to support λ-tree syntactic representation. Logic is also used to reason about specifications made in this first logic. This second logic is thus a *meta-logic* for reasoning about provability in those *object-logics.* A particular meta-logic, LINC, is based on intuitionistic logic and incorporates the ∇-quantifier and principles of induction and co-induction.

Armed with the meta-logic LINC, with several interesting examples of using it to reason about computation, and with several years of experience with implementing proof search systems involving the unification of λ-term, it is now time to build prototype theorem provers for LINC and develop larger examples. Already, we can use λProlog [40] via its Teyjus implementation [42] to animate specifications given in a number of object-logics. A simple model checking-style generalization of (part of) λProlog has also been implemented and used to verify various simple properties of, say, the π-calculus [55, 53].

One of the goals of the Parsifal project at INRIA is to use this two level logic approach to reason formally about operational semantics, say, in the context of the POPLmark challenge [2]. We also hope to use this framework to reason about specification logics themselves: for example, to prove soundness of logics used to annotate programming languages for extended static checking, such as the ESC/Java2 object logic [22]. Consistency of two simpler object-logics have been proved in [26] by showing showing formally in (a subset of) LINC that cut-elimination holds for them.

Acknowledgments I am grateful to the anonymous reviewers for their helpful comments on an earlier draft of this paper. This work has been supported in part by the INRIA "Equipes Associées" Slimmer, ACI grants GEOCAL and Rossignol, and the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This paper reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein.

References

1. J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.
2. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference,* LNCS, pages 50–65. Springer-Verlag, 2005.
3. M. Boreale and R. D. Nicola. A symbolic semantics for the π-calculus. *Information and Computation*, 126(1):34–52, April 1996.

4. P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of SIGSOFT'88: Third Annual Symposium on Software Development Environments (SDE3)*, Boston, 1988.
5. N. Bruijn. Lambda calculus notation with namefree formulas involving symbols that represent reference transforming mappings. *Indag. Math.*, 40(3):348–356, 1979.
6. A. Church. A formulation of the simple theory of types. *J.of Symbolic Logic*, 5:56–68, 1940.
7. K. L. Clark. Negation as failure. In J. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
8. R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.
9. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.
10. J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, April 1995.
11. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
12. G. Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.
13. J.-Y. Girard. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu, February 1992.
14. M. Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, July 1985.
15. J. F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100:202–260, 1992.
16. L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. II. Programs as definitions. *J. of Logic and Computation*, 1(5):635–660, October 1991.
17. M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, Feb. 1995.
18. M. Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Symp. on Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, 1999.
19. D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.
20. G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
21. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on the Principles of Programming Languages*, 1987.
22. J. R. Kiniry, P. Chalin, and C. Hurlin. Integrating static checking and interactive verification: Supporting multiple theories and provers in verification. In *VSTTE'05, Proceedings of Verified Software: Theories, Tools, Experiements*, Zurich, Switzerland, October 2005.
23. P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland.
24. R. McDowell and D. Miller. A logic for reasoning with higher-order abstract syntax. In G. Winskel, editor, *12th Symp. on Logic in Computer Science*, pages 434–445, Warsaw, Poland, July 1997. IEEE Computer Society Press.

25. R. McDowell and D. Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.
26. R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.
27. R. McDowell, D. Miller, and C. Palamidessi. Encoding transition systems in sequent calculus. *Theoretical Computer Science*, 294(3):411–437, 2003.
28. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.
29. D. Miller. Abstract syntax for variable binders: An overview. In J. Lloyd and et. al., editors, *Computational Logic - CL 2000*, number 1861 in LNAI, pages 239–253. Springer, 2000.
30. D. Miller. Bindings, mobility of bindings, and the ∇-quantifier. In J. Marcinkowski and A. Tarlecki, editors, *18th International Workshop CSL 2004*, volume 3210 of *LNCS*, page 24, 2004.
31. D. Miller. Overview of linear logic programming. In T. Ehrhard, J.-Y. Girard, P. Ruet, and P. Scott, editors, *Linear Logic in Computer Science*, volume 316 of *London Mathematical Society Lecture Note*, pages 119 – 150. Cambridge University Press, 2004.
32. D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In S. Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.
33. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
34. D. Miller and C. Palamidessi. Foundational aspects of syntax. *ACM Computing Surveys*, 31, September 1999.
35. D. Miller and A. Tiu. A proof theory for generic judgments: An extended abstract. In *18th Symp. on Logic in Computer Science*, pages 118–127. IEEE, June 2003.
36. D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, Oct. 2005.
37. R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.
38. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part II. *Information and Computation*, pages 41–77, 1992.
39. A. Momigliano and A. Tiu. Induction and co-induction in sequent calculus. In M. C. Stefano Berardi and F. Damiani, editors, *Post-proceedings of TYPES 2003*, number 3085 in LNCS, pages 293 – 308, January 2003.
40. G. Nadathur and D. Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.
41. G. Nadathur and D. Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990.
42. G. Nadathur and D. J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.
43. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer, 2002. LNCS Tutorial 2283.
44. S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.
45. L. Paulson. Compiler Generation from Denotational Semantics. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge Univ. Press, 1984.

46. L. C. Paulson and K. Grąbczewski. Mechanizing set theory: Cardinal arithmetic and the axiom of choice. *J. of Automated Deduction*, 17(3):291–323, Dec. 1996.
47. F. Pfenning and C. 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.
48. F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conference on Automated Deduction*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer.
49. D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33(1):69–97, 1996.
50. P. Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, IEEE, June 1993.
51. C. Schürmann and F. Pfenning. A coverage checking algorithm for LF. In *Theorem Proving in Higher Order Logics: 16th International Conference, TPHOLs 2003*, volume 2758 of *LNCS*, pages 120–135. Springer-Verlag, 2003.
52. A. Tiu. *A Logical Framework for Reasoning about Logical Specifications.* PhD thesis, Pennsylvania State University, May 2004.
53. A. Tiu. Model checking for π-calculus using proof search. In M. Abadi and L. de Alfaro, editors, *CONCUR*, volume 3653 of *LNCS*, pages 36–50. Springer, 2005.
54. A. Tiu and D. Miller. A proof search specification of the π-calculus. In *3rd Workshop on the Foundations of Global Ubiquitous Computing*, volume 138 of *ENTCS*, pages 79–101, Sept. 2004.
55. A. Tiu, G. Nadathur, and D. Miller. Mixing finite success and finite failure in an automated prover. In *Proceedings of ESHOL'05: Empirically Successful Automated Reasoning in Higher-Order Logics*, pages 79 – 98, December 2005.
56. A. Ziegler, D. Miller, and C. Palamidessi. A congruence format for name-passing calculi. In *Proceedings of SOS 2005: Structural Operational Semantics*, Electronic Notes in Theoretical Computer Science, Lisbon, Portugal, July 2005. Elsevier Science B.V.