Observations about using logic as a specification language

Dale Miller *

200 S. 33rd Street, Computer Science Department
University of Pennsylvania, Philadelphia, PA 19104–6389 USA
dale@saul.cis.upenn.edu
Phone: +1-215-898-1593, Fax: +1-215-898-0587

Abstract

This extended abstract contains some non-technical observations about the roles that logic can play in the specification of computational systems. In particular, computation-as-deduction, meta-programming, and higher-order abstract syntax are briefly discussed.

1 Two approaches to specifications

In the specification of computational systems, logics are generally used in one of two approaches. In one approach, computations are mathematical structures, containing such items as nodes, transitions, and state, and 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. For example, next-time modal operators are used to describe the possible evolution of state; expressions in the Hennessey-Milner are evaluated against the transitions made by a process; and Hoare logic uses formulas to express pre- and post-conditions on a computation's state. We shall refer to this approach to using logic as *computation-as-model*. In such approaches, the fact that some identifier x has value 5 is represented as, say a pair $\langle x, 5\rangle$, within some larger mathematical structure, and logic is used to express propositions *about* such pairs: for example, $x > 3 \wedge x < 10$.

A second approach uses logical deduction to model computation. In this approach the fact that the identifier x has value 5 can be encoded as the proposition

*The work reported here has been funded in part by ONR N00014-93-1-1324, NSF CCR-91-02753, and NSF CCR-92-09224.

"x has value 5." Changes in state can then be modeled by changes in propositions within a derivation. Of course, changing state may require that a proposition no longer holds while a proposition that did not hold (such as "x has value 6") may hold in a new state. It is a common observation that such changes are naturally supported by linear logic and that deduction (in particular, backchaining in the sense of logic programming) can encode the evolution of a computation. As a result, it is possible to see the state of a computation as a logical formula and transitions between states as steps in the construction of a proof. We shall refer to this approach to using logic as *computation-as-deduction.*

There are many ways to contrast these two approaches to specification using logic. For example, consider their different approaches to the "frame problem." Assume that we are given a computation state described as a model, say M_1, in which it is encoded that the identifier x is bound to value 5. If we want to increment the value of x, we may need to characterize all those models M_2 in which x has value 6 and *nothing else has changed.* Specifying the precise formal meaning of this last clause is difficult computationally and conceptually. On the other hand, when derivations are used to represent computations directly, the frame problem is not solved but simply avoided: for example, backchaining over the clause

$$x \text{ has value } n \multimap x \text{ has value } n+1$$

might simply change the representation of state in the required fashion.

In the first approach to specification, there is a great deal of richness available for modeling computation, since, in principle, such disciplines as set theory, category theory, functional analysis, algebras, *etc.*, can be employed. This approach has had, of course, a great deal of success within the theory of computation.

In contrast, the second approach seems thin and feeble: the syntax of logical formulas and proofs contains only the most simple structures for representing computational state. What this approach lacks in expressiveness, however, is ameliorated by the fact that it is more intimately connected to computation. Deductions, for example, seldom make reference to infinity (something commonly done in the other approach) and steps within the construction of proofs are generally simple and effective computations. Recent developments in proof theory and logic programming have also provided us with logics that are surprisingly flexible and rich in their expressiveness. In particular, linear logic [6] provides flexible ways to model state, state transitions, and some simple concurrency primitives, and higher-order quantification over typed λ-terms provides for flexible notions of abstraction and encodings of object-level languages. Also, since specifications are written using logical formulas, specifications can be subjected to rich forms of analysis and transformations.

To design logics (or presentations of logics) for use in the computation-as-deduction setting, it has proved useful to provide a direct and natural operational interpretation of logical connective. To this end, the formalization of *goal-directed search* using *uniform proofs* [14, 16] associates a fixed, "search semantics" to logical connectives. When restricting to uniform proofs does not cause a loss of completeness, logical connectives can be interpreted as fixed search primitives. In this way, specifier can write declarative specifications that map directly to descriptions of computations. This analysis of goal-directed proof search has lead to the design of the logic programming languages λProlog, Lolli, LO, and Forum. Some simple examples with using these languages for specifications can be found in [1, 10, 14]. The recent thesis [2] provides two modest-sized Forum specifications: one being the operational semantics of a functional programming language containing references, exceptions, and continuation passing, and the other being a specification of a pipe-lined, RISC processor.

> *Observation 1.* Logic can be used to make statements about computation by encoding states and transitions directly using formulas and proof. This use of logic fits naturally in a logic programming setting where backchaining can denote state transition. Both linear logic and higher-order quantification can add greatly to the expressiveness of this paradigm.

2 An example

The following specification of reversing a list and the proof of its symmetry illustrates how the expressiveness of higher-order linear logic can provide for natural specifications and convenient forms of reasoning.

```
reverse L K :- pi rv\(
  pi X\(pi M\(pi N\(rv (X::M) N :- rv M (X::N)))) => rv nil K -:
   rv L nil).
```

Here we use a variant of λProlog syntax: in particular, lists are constructed from the infix `::` and `nil`; `pi X\` denotes universal quantification of the variable `X`; `=>` denotes intuitionistic implication; and, `-:` and `:-` denote linear implication and its converse. This one example combines some elements of both linear logic and higher-order quantification.

To illustrate this specification, consider proving the query

```
?- reverse (a::b::c::nil) Q.
```

Backchaining on the definition of reverse above yields a goal universally quantified by `pi rv\`. Proving such a goal can be done by instantiating that quantifier with a new constant, say `rev`, and proving the result, namely, the goal

```
  pi X\(pi M\(pi N\(rev (X::M) N :- rev M (X::N)))) => rev nil Q -:
   rev (a::b::c::nil) nil).
```

Thus, an attempt will be made to prove the goal `(rev (a::b::c::nil) nil)` from the two clauses

```
pi X\(pi M\(pi N\(rev (X::M) N :- rev M (X::N)))).
rev nil Q.
```

(Note that the variable `Q` in the last clause is free and not implicitly universally quantified.) Given the use of intuitionistic and linear implications, the first of these clauses can be used any number of times while the second must be used once (natural characterizations of inductive and initial cases for this example). Backchaining now leads to the following progression of goals:

```
rev (a::b::c::nil)  nil.
rev (b::c::nil) (a::nil).
rev (c::nil) (b::a::nil).
rev  nil  (c::b::a::nil).
```

and the last goal will be proved by backchaining against the initial clause and binding `Q` with `(c::b::a::nil)`.

It is clear from this specification of `reverse` that it is a symmetric relation: the informal proof simply notes that if the table of `rev` goals above is flipped horizontally and vertically, the result is the core of a computation of the symmetric version of reverse. Given the expressiveness of this logic, the formal proof of this fact directly incorporates this main idea.

Proposition. Let `l` and `k` be two lists and let $\mathcal{P}$ be a collection of clauses in which the only clause that contains an occurrence of `reverse` in its head is the one displayed above. If the goal `(reverse l k)` is provable from $\mathcal{P}$ then the goal `(reverse k l)` is provable from $\mathcal{P}$.

Proof. Assume that the goal `(reverse l k)` is provable from $\mathcal{P}$. Given the restriction on occurrences of `reverse` in $\mathcal{P}$, this goal is provable if and only if it is proved by backchaining with the above clause for `reverse`. Thus, the goal

```
pi rv\(
  pi X\(pi M\(pi N\(rv (X::M) N :- rv M (X::N)))) =>
  rv nil k -: rv l nil)
```

is provable from $\mathcal{P}$. Since this universally quantified formula is provable, any instance of it is provable. Let `rev` be a new constant not free in $\mathcal{P}$ of the same type as the variable `rv`. The formula that results from instantiating this quantified goal with the λ-term `x\y\(not (rev y x))` (where `\` is the infix symbol for λ-abstraction and `not` is the logical negation, often written in linear logic using the superscript $\perp$). The resulting formula,

```
pi X\(pi M\(pi N\(not (rev N (X::M)) :- not (rev (X::N) M)))) =>
not (rev k nil) -: not (rev nil l),
```

is thus provable from $\mathcal{P}$. This formula is logically equivalent to the following formula (linear implications and their contrapositives are equivalent in linear logic).

```
    pi X\(pi M\(pi N\(rev (X::N) M :- rev N (X::M)))) =>
    rev nil l -: rev k nil
```

Since this code is provable and since the constant `rev` is not free in $\mathcal{P}$, we can universally generalize over it; that is, the following formula is also provable.

```
pi rev\(
    pi X\(pi M\(pi N\(rev (X::N) M :- rev N (X::M)))) =>
     rev nil l -: rev k nil)
```

From this goal and the definition of `reverse` (and α-conversion) we can prove `(reverse k l)`. Hence, `reverse` is symmetric. ∎

This proof should be considered elementary since it involves only simple linear logic identities and facts. Notice that there is no direct use of induction. The two symmetries mentioned above in the informal proof are captured in the higher-order substitution `x\y\(not (rev y x))`: the switching of the order of bound variables captures the vertical flip and linear logic negation (via contrapositives) captures the the horizontal flip.

3 Meta-programming and meta-logic

An exciting area of specification is that of specifying the meaning and behavior of programs and programming languages. In such cases, the code of a programming language must be represented and manipulated, and it is valuable to introduce the terms *meta-language* to denote the specification language and *object-language* to denote the language being specified.

Given the existence of two languages, it is natural to investigate the relationship that they may have to one another. That is, how can the meaning of object-language expressions be related to the meaning of meta-level expressions. One of the major accomplishments in mathematical logic in the first part of this century was achieved by K. Gödel by probing this kind of reflection, in this case, encoding meta-level formulas and proofs at the the object-level [7].

Although much of the work on meta-level programming in logic programming has also been focused on reflection, this focus is rather narrow and limiting: there are many other ways to judge the success of a meta-programming language apart from its ability to handle reflection. While a given meta-programming language might not be successful at providing novel encodings of itself, it might provide valuable and flexible encodings of other programming languages. For example, the π-calculus provides a revealing encoding of evaluation in the λ-calculus [17], evaluation in object-oriented programming [28], and interpretation of Prolog programs [12]. Even the semantic theory of the π-calculus can be fruitfully exploited to probe the semantics of encoded object-languages [27]. While it has been useful as a meta-language, it does not seem that the π-calculus would yield an interesting encoding of itself.

Similarly, λProlog has been successful in providing powerful and flexible specifications of functional programming languages [8, 21] and natural deduction proof systems [5]. Forum has similarly been used to specify sequent calculi and various features of programming languages [2, 14]. It is not clear, however, that λProlog or Forum would be particularly good for representing their own operational semantics.

> *Observation 2.* A meta-programming language does not need to capture its own semantics to be useful. More importantly, it should be able to capture the semantics of a large variety of languages and the resulting encoding should be direct enough that the semantics of the meta-language can provide semantically meaningful information about the encoded object-language.

A particularly important aspect of meta-programming is the choice of encodings for object-level expressions. Gödel used natural numbers and the prime factorization theorem to encode syntactic values: an encoding that does not yield a transparent nor declarative approach to object-level syntax. Because variables in logic programming range over expressions, representing object-level syntax can be a particularly simple, at least for certain expressions of the object language. For example, the meaning of a type in logic programming, particularly types as they are used in λProlog, is a set of *expressions* of a given type. In contrast, types in functional programming (say, in SML) generally denote sets of *values*. While the distinction between expressions and values can be cumbersome at times in logic programming (`2 + 3` is different than `5`), it can be useful in meta-programming. This is particularly true when dealing with expressions of functional type. For example, the type `int -> int` in functional programming denotes functions from integers to integers: checking equality between two such functions is not possible, in general. In logic programming, particularly in λProlog, this same type contains the code of expressions (not functions) of that type: thus it is possible to represent the syntax of higher-order operations in the meta-programming language and meaningfully compare and compute on these codes. More generally, meta-level types are most naturally used to represent object-level syntactic categories. When using such an encoding of object-level languages, meta-level unification and meta-level variables can be used naturally to probe the structure of object-level syntax.

> *Observation 3.* Since types and variables in logic programming range over expressions, the problem of naming object-level expressions is often easy to achieve and the resulting specifications are natural and declarative.

4 Higher-order abstract syntax

In the last observation, we used the phrase "often easy to achieve." In fact, if object-level expressions contain bound variables, it is a common observation that representing such variables using only first-order expressions is problematic since notions of bound variable names, equality up to α-conversion, substitution, *etc.*, are not addressed naturally by the structure of first-order terms. From a logic programming point-of-view this is particularly embarrassing since all of these notions are part of the meta-theory of quantification logic: since these issues exist in logic generally, it seems natural to expect a logical treatment of them for object-languages that are encoded into logic. Fortunately, the notion of *higher-order abstract syntax* is capable of declaratively dealing with these aspects of object-level syntax.

Higher-order abstract syntax involves two concepts. First, λ-terms and their equational theory should be used uniformly to represent syntax containing bound variables. Already in [3], Church was doing this to encode the universal and existential quantifiers and the definite description operator. Following this approach, instantiation of quantifiers, for example, can be specified using β-reduction.

The second concept behind higher-order abstract syntax is that operations for composing and decomposing syntax must respect at least α-conversion of terms. This appears to have first been done by Huet and Lang in [11]: they discussed the advantages of representing object-level syntax using simply typed λ-terms and manipulating such terms using matching modulo the equational rules for λ-conversion. Their approach, however, was rather weak since it only used matching (not unification more generally). That restrictions made it impossible to express all but the simplest operations on syntax. Their approach was extended by Miller and Nadathur in [15] by moving to a logic programming setting that contained $\beta\eta$-unification of simply typed λ-terms. In that paper the central ideas and advantages behind higher-order abstract syntax are discussed. In the context of theorem proving, Paulson also independently proposed similar ideas [20].

In [23] Pfenning and Elliot extended the observations in [15] by producing examples where the meta-language that incorporated λ-abstractions contained not just simple types but also product types. In that paper they coined the expression "higher-order abstract syntax." At about this time, Harper, Honsell, and Plotkin in [9] proposed representing logics in a dependent typed λ-calculus. While they did not deal with the computational treatment of syntax directly, that treatment was addressed later by considering the unification of dependent typed λ-expressions by Elliott [4] and Pym [25].

The treatment of higher-order abstract syntax in the above mentioned papers had a couple of unfortunate aspects. First, those treatments involved unification with respect to the full $\beta\eta$-theory of the λ-calculus, and this general theory is computational expensive. In [11], only second-order matching was used, an operation that is NP-complete; later papers used full, undecidable unification. Second, various different type systems were used with higher-order abstract syntax, namely simple types, product types, and dependent types. However, if abstract syntax is essentially about a treatment of bound variables in syntax, it should have a presentation that is independent from typing.

The introduction of L_λ in [13] provided solutions to both of these problems.

First, L_λ provides a setting where the unification of λ-terms is decidable and has most general unifiers: it was shown by Qian [26] that L_λ-unification can be done in linear time and space (as with first-order unification). Nipkow showed that the exponential unification algorithm presented in [13] can be effectively used within theorem provers [19]. Second, it was also shown in [13] that L_λ-unification can be described for *untyped* λ-terms: that is, typing may impose additional constraints on unification but L_λ-unification can be defined without types. Thus, it is possible then to define L_λ-like unification for various typed calculi [22].

> *Observation 4.* L_λ appears to be one of the weakest settings in which higher-order abstract syntax can be supported. The main features of L_λ can be merged with various logical systems (say, λProlog and Forum), with various type systems (say, simple types and dependent types) [21], and with equational reasoning systems [18, 24].

While existing implementations of λProlog, Isabelle, Elf, and NuPRL all make use of results about L_λ, there is currently no direct implementation of L_λ. It should be a small and flexible meta-logic specification language.

References

[1] J.M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[2] Jawahar Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD thesis, University of Pennsylvania, February 1995.

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

[4] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, volume 355, pages 121–136. Springer-Verlag LNCS, April 1989.

[5] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[6] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[7] Kurt Gödel. On formally undecidable propositions of the principia mathematica and related systems. I. In *Martin Davis, The Undecidable.* Raven Press, 1965.

[8] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.

[9] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, June 1987.

[10] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[11] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[12] Benjamin Li. A π-calculus specification of Prolog. In *Proc. ESOP 1994*, 1994.

[13] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[14] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science*, pages 272–281, Paris, July 1994.

[15] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[16] 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.

[17] Robin Milner. Functions as processes. 17th Int. Coll. Automata, Languages and Programming Warwick University, UK, LNCS 443, pp. 167–180, Springer Verlag July 1990.

[18] Tobias Nipkow. Higher-order critical pairs. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 342–349. IEEE, July 1991.

[19] Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 64–74. IEEE, June 1993.

[20] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[21] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–321, Monterey, CA, June 1989.

[22] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 74–85. IEEE, July 1991.

[23] 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.

[24] Christian Prehofer. *Solving Higher-Order Equations: From Logic to Programming*. PhD thesis, Technische Universität München, 1995.

[25] David Pym. *Proofs, Search and Computation in General Logic*. PhD thesis, LFCS, University of Edinburgh, 1990.

[26] Zhenyu Qian. Linear unification of higher-order patterns. In J.-P. Jouannaud, editor, *Proc. 1993 Coll. Trees in Algebra and Programming*. Springer Verlag LNCS, 1993.

[27] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, May 1994.

[28] David Walker. π-calculus semantics of object-oriented programming languages. LFCS Report Series ECS-LFCS-90-122, University of Edinburgh, October 1990.