Obvious Properties of Computer Programs

Robert Givan

Department of Computer Science
Brown University, Box 1910, Providence, RI 02912
rlg@cs.brown.edu, http://www.cs.brown.edu/people/rlg/

Abstract

We explore the question of what properties of LISP programs can be made "obvious" to a computer system. We present a polynomial-time algorithm for inferring interesting properties of pure LISP programs. Building on previous work in knowledge representation for rapid inference, we present a language for representing properties of programs. We treat properties as generalized types, i.e., sets of program values. The property language is expressive enough to represent any RE set of LISP values as a property, and can naturally represent a wide variety of useful properties.

We then use a general technique to construct a polynomial-time property inference relation and use type-inference style program analysis to integrate this relation into an algorithm for inferring properties of programs. This algorithm is intended to work in the context of a library of background information, most of which is typically derived from previous runs of the algorithm. Due to the expressive representation system, no algorithm can infer every valid property—so instead of proving completeness we show our algorithm’s usefulness by giving examples of properties inferred. These examples include that insertion sort turns a value in the corresponding set of values. Taking this view, program properties are essentially just properties amenable to rapid inference. This representation derives from viewing properties as sets of program values—a program has a property if the program returns a value in the corresponding set of values. Taking this view, program properties are essentially just types in a rich type system, and we exploit this fact by drawing on traditional type inference techniques in constructing our inference algorithm.

Related previous work on polynomial-time inference has frequently achieved the polynomial-time bound largely by limiting the expressiveness of the language of inferrable properties (Brachman & Schmolze 1985). This approach is attractive because the resulting system is typically amenable to proofs of completeness theorems asserting that every valid representable property will be inferred. However, this approach limits the strength of the resulting procedure. Expressive language constructs not only facilitate the asking of hard questions (causing incompleteness), but also facilitate reasoning. There are in addition many questions that can be easily answered but can only be asked in an expressive language. In this paper we consider the effects of allowing a richly expressive language (any RE set can be represented as a property), but insisting that our inference principles remain within polynomial-time.

Introduction

Programmers quickly and easily see many properties of their programs that are invisible to program analysis systems. We find it obvious, for example, that concatenating two lists produces a list at least as long as either, or that mapping a function across a list produces a list of the same length. But these properties typically cannot even be represented by compilers. Theorem provers can represent such properties, but provide little counterpart for the quick reasoning of humans. In this paper, we explore the question “How strong can we make a fast inference procedure?”, where we formalize “fast” as “polynomial-time”, and we are especially interested in inferring properties that require expressive representation to state.

The answer to this question proves very sensitive to the representation system used in inference. We draw on our previous work with McAllester (1993; 1992) to select a representation for program properties amenable to rapid inference. This representation derives from viewing properties as sets of program values—a program has a property if the program returns a value in the corresponding set of values. Taking this view, program properties are essentially just types in a rich type system, and we exploit this fact by drawing on traditional type inference techniques in constructing our inference algorithm.
Program expressions are defined inductively as follows:

\[ e ::= x \mid (\text{let } x := e_1 \mid (f \ e_1 \cdots e_n) \mid (\text{if } e : p \ e_1 e_2) \mid \text{sym} \]

where \( x \) can be any variable, \( f \) can be \text{cons}, \text{car}, \text{cdr}, or any \( n \)-ary program function symbol, \text{sym} can be any symbol and \( p \) must be a testable property (see below). The intended meanings for the above program expressions should be clear from LISP, except that the \text{if} tests are new: \( e : p \) is true if the value \( e \) satisfies the property \( p \) (see below). Example program expressions appear in the definitions in figure 1.

\[
\begin{align*}
&\text{(define (insert x:(a-number) l:(a-sorted-list))} \\
&\quad (\text{if } l : \text{'nil}) \\
&\quad \quad (\text{cons } x l) \\
&\quad \quad (\text{if } x : (>=(\text{car } l)) \\
&\quad \quad \quad (\text{cons } x l) \\
&\quad \quad \quad (\text{cons } (\text{car } l) (\text{insert } x (\text{cdr } l)))) \\
&\text{(define (sort l:(a-numlist))} \\
&\quad (\text{if } l : \text{'nil}) \\
&\quad \quad 1 \\
&\quad \quad (\text{insert } (\text{car } l) (\text{sort } (\text{cdr } l)))) \\
&\text{(define (intersect l1:(a-list) l2:(a-list))} \\
&\quad (\text{if } l1 : \text{'nil}) \\
&\quad \quad '\text{nil} \\
&\quad \quad (\text{if } (\text{car } l1) : (\text{a-member-of } 12) \\
&\quad \quad \quad (\text{cons } (\text{car } l1) \\
&\quad \quad \quad \quad (\text{intersect } (\text{cdr } l1) 12)) \\
&\quad \quad \quad (\text{intersect } (\text{cdr } l1) 12)) \\
&\text{(define (largest-clique-in s:(a-list))} \\
&\quad (\text{if } s : \text{'nil}) \\
&\quad \quad '\text{nil} \\
&\quad \quad (\text{let } n : (\text{car } s) \\
&\quad \quad \quad c : (\text{cons } n (\text{largest-clique-in} \\
&\quad \quad \quad (\text{intersect } (\text{neighbors } n) \\
&\quad \quad \quad (\text{cdr } s)))) \\
&\quad \quad \quad c' : (\text{largest-clique-in } (\text{cdr } s)) \\
&\quad \quad \quad (\text{if } (\text{length } c) : (\geq (\text{length } c')) \\
&\quad \quad \quad c \quad c')) \\
&\text{(define (delete x:(a-thing) l:(a-list))} \\
&\quad (\text{if } l : \text{'nil}) \\
&\quad \quad '\text{nil} \\
&\quad \quad (\text{if } x : (\text{car } l)) \\
&\quad \quad (\text{cdr } l) \\
&\quad \quad (\text{cons } (\text{car } l) (\text{delete } x (\text{cdr } l))))
\end{align*}
\]

Figure 1: Example program definitions. We omit simple definitions which have no effect on the example inferences claimed later (\text{length}, \text{neighbors})

Property Expressions Our property language is derived from our previous work with McAllester(1992) on natural language syntax and its relationship to tractable inference. The representational features introduced in that work have never before been exploited in an automated reasoning system. Our property language is essentially the programming language extended by some new constructs that allow and facilitate the quantifier-free construction of sets of values. The most important of these constructs is nondeterminism. We use nondeterminism for its declarative value only—property expressions are not programs to be run. A property expression has in general many possible values via nondeterminism—the set of these values is the type (property) represented by the expression. Using recursion, property expressions can denote

Languages for Programs and Properties
We introduce the programming language and our representation for properties. We give as examples the program definitions of insertion sort and of a clique finding algorithm, and the property definitions of a permutation of a list, a sorted list, and a clique. In the next section we describe a polynomial-time algorithm that infers relationships between these programs and properties (e.g. that \text{sort} permutes its input).

Program Expressions The programming language we analyze is a typed, first-order, pure subset of LISP. It is “first-order” because, to ease the presentation, it does not include first class functions—user functions are introduced through definitions and used only by being applied—however, this work generalizes naturally to first class functions. The language is “typed” because it requires each variable to be given a user-provided type at its introduction; these types are given in an expressive property language and can capture much about the programmer’s intended use for the variable. Program expressions are defined inductively as follows:

\[ e ::= x \mid (\text{let } x := e_1 \mid (f \ e_1 \cdots e_n) \mid (\text{if } e : p \ e_1 e_2) \mid \text{sym} \]

where \( x \) can be any variable, \( f \) can be \text{cons}, \text{car}, \text{cdr}, or any \( n \)-ary program function symbol; \text{sym} can be any symbol and \( p \) must be a testable property (see below). The intended meanings for the above program expressions should be clear from LISP, except that the \text{if} tests are new: \( e : p \) is true if the value \( e \) satisfies the
infinite sets.

Nondeterminism is introduced by the one-of combinator\(^2\). The expression (one-of s t) where s and t
are property expressions denotes the nondeterministic choice between s and t. Alternatively, one-of can be viewed as the union operator on sets—(one-of s t) denotes the union of the types (properties) represented by
s and t. As an example, consider the property a-list in figure 2, which denotes the set of all finite LISP lists.

Nondeterminism further enhances the representational power of our property language when considered in conjunction with function application. Consider the expression (2* (a-number)). This expression

denotes the set of even numbers. This meaning can be derived by evaluating the expression nondeterministically\(^3\)—first, the argument to 2* returns an arbitrary natural number, then 2* deterministically doubles this number; the result can be any even number. A similar nondeterministic evaluation applies if we replace 2* with a nondeterministic “property function” such as greater-than (see figures 2 and 3 for examples).

```
(define (a-list)
  (one-of 'nil (cons (a-thing) (a-list))))

(define (a-member-of l:(a-list))
  (if l:'nil
      (one-of (car l) (a-member-of (cdr l))))
  ;; natural numbers (in unary)
  (define (a-number)
    (one-of 'nil (cons 'a (a-number)))))

(define (a-numlist)
  ;; any list of numbers
  (define (a-number)
    (one-of 'nil (cons (a-number) (a-numlist))))

;; the numbers >= x
(define (>= x:(a-number))
  (one-of x (cons 'a (>= x)))))
```

Figure 2: Example Property Definitions

A final representational feature of our property language allows the quantifier-free construction of properties
that would traditionally require quantifiers. Given a (deterministic or nondeterministic) function f of one argument, and a property expression p, the expression \(f\ (\text{every} \ p)\) is analogous to the ordinary application \(f\ p\) except that only output values that can be produced for every value of p might return are kept. In other words, \(f\ (\text{every} \ p)\) denotes the set of values that are \(f\)-related to every value of p. As a

\(\text{every}\) is closely related to \text{amb} introduced by
McCarthy in (McCarthy 1967).

\(\text{one-of}\) is analogous to \text{one-of}. In other words, \(\text{one-of}\) denotes the set of all possible values of \(p\) that are not possible values of \(\text{every}\ p\).

```
(define (a-permutation-of l:(a-list))
  (if l:'nil
      1
      (let x:(a-member-of l)
        (cons x (a-permutation-of
                (delete x l))))))

(define (a-sorted-list)
  (one-of 'nil
    (let l:(a-sorted-list)
      (cons (all-of (>= (every (a-member-of l)))
               (a-number))
      l))))

(define (neighbor-of n:(a-thing))
  (a-member-of (neighbors n)))

(define (a-clique)
  (one-of 'nil
    (let c:(a-clique)
      (cons (a-neighbor-of
             (every (a-member-of c)))
            c)))))
```

Figure 3: More Example Properties

more concrete example, the notion of a common factor of the numbers in a set s could be represented by the
property (factor-of (every (member-of s))). This construction was introduced in (McAllester 
& Girvan 1992), and is inspired by the English noun phrase construction “loves every man”. Its use in program
analysis is new and essential—it allows the quantifier-free construction of useful properties like “sorted list”
and facilitates the reasoning about these properties (other representations of these properties are more unwieldy,
making the resulting inference task more difficult).

Formally, property expressions are defined as:

\[ p := x \mid (\text{let } x:p \text{ } p) \mid (\text{if } p1:p2 \text{ } p3 \text{ } p4) \]

\[ (f \text{ } p1\ldots p4) \mid (f \text{ } \text{every} \ p) \]

\[ \text{all-of } s \mid (\text{not } p) \mid \text{sym} \mid \bot \mid (\text{a-thing}) \]

where \(f\) can be \text{cons}, \text{car}, \text{cdr}, or any defined program or property function symbol, \(\text{sym}\) any symbol, and
s is a finite set of property expressions. Note that every
program expression is also a property expression—when used as a property expression a program expression
denotes the singleton set containing the value of the program. This BNF includes some expressions not
yet mentioned: (all-of s) denotes those values that are possible values of every expression in s, (not p)
denotes those values that are not possible values of p, \(\bot\) denotes the empty set of values, and (a-thing) denotes the set of all possible values (quoted symbols or
finite \text{cons} expressions built up from them).

Programs We consider a sequence of function-symbol
definitions to be a program. A function-symbol
definition assigns to a new function-symbol either of
(\text{lambda } x_1: p_1, \ldots, x_n: p_n) \text{ or } (\text{fix } f \: x_1: p_1, \ldots, x_n: p_n) \text{ where the body } p \text{ must be deterministic (i.e., a program expression) if the symbol being defined is a program function-symbol}^4. \text{ The properties } p_j \text{ can reference and depend on the variables } x_1, \ldots, x_{j-1}. \text{ Note that we differentiate between program function-symbols and property function-symbols.}

\textbf{Semantics and Other Issues} \text{ We have a complete and detailed semantics for the above languages that will be presented in the full version of this paper.}

\text{We note that the expression } (f (\text{every } e)) \text{ for program expression } e \text{ has the same meaning as the expression } (f \: e). \text{ We will treat these two expressions as identical, even in the pattern matching used in applying the inference rules given later; so the expression } (f \: e) \text{ will match the pattern } (R (\text{every } s)) \text{ with } f \text{ instantiating } R \text{ and } e \text{ instantiating } s.

\text{Also, we have left unresolved above the issue of how to execute a program containing an if expression that tests a nondeterministic property. This peripheral issue is handled in detail in the full version of the paper; here we only comment that we require the user to provide a verified implementation of any nondeterministic property function used in an if test in a program function definition—this task is straightforward for the examples given in this paper. We refer to a property so implemented as \textit{testable}.}

\textbf{An Inference Algorithm} \text{ Our inference algorithm accepts as input a new definition (of a program function or a property function) and a background library of \textit{type theorems}, and produces as output some new type theorems about the newly defined symbol that are then added to the background library. By \textit{type theorem} we mean a universally quantified formula of the form }\text{forall } x_1: p_1, \ldots, x_n: p_n, p' \text{ where } p \text{ will typically involve the new symbol and } p' \text{ is viewed as a type or property being asserted about } p. \text{ Figure 4 shows example type theorems produced in analyzing the example programs shown earlier. Throughout this section we assume we are analyzing a definition assigning the \texttt{lambda} or \texttt{fix} expression } e \text{ to the new function symbol } g \text{, with respect to background library } \mathcal{L} \text{ (in particular, } e \text{ will appear as a subscript on our inference relation symbols } \vdash_{\mathcal{L}} \text{ and } \vdash^e_{\mathcal{L}} \text{ to indicate their dependence on } e).}

\textbf{Definition Analysis} \text{ An initial inference stage generates very useful \textit{type theorems} from a superficial}

\begin{align*}
\text{forall } l: & (a\text{-numlist}) \\
\text{(sort } l & ) : (a\text{-permutation-of } l) \\
\text{forall } l: & (a\text{-numlist}) \\
\text{(sort } l & ) : (a\text{-sorted-list}) \\
\text{forall } l: & (a\text{-list}) \\
\text{(find-largest-clique } l & ) : (a\text{-clique})
\end{align*}

\text{Figure 4: \textit{Sample Output}. Some theorems generated automatically using the library in figure 9.}

\text{analysis of the new definition. These theorems roughly correspond to the beta-reduction and beta-abstraction properties of the definition, processed in a form that makes them most accessible to the remainder of the algorithm—the processing breaks down the cases of top level if and one-of expressions, and handles rudimentary let instantiation. Definition analysis is formally the forward chaining inferential closure of the inference rules for } \vdash \text{ shown in figure 5 along with an example theorem.}

\textbf{Recursive Descent Rules} \text{ The main stage of inference proceeds by recursive descent into the new definition's body, accumulating type (property) information along the way. This recursive descent is primarily responsible for analyzing the let, if, lambda, and fix constructs in the definition. At each level of the descent, a basic inference engine (denoted } \vdash^e \text{ described in the next subsection) is used to generate relevant type theorems. The recursive descent phase can be viewed as an abstract interpretation(Abramsky & Hankin 1987)(Milner 1978) program analysis.}

\text{The inference rules for } \vdash^e \text{ in figure 6 formally describe the recursive descent analysis. These rules use some new notation. We use the notation } \sigma(p) \text{ to signify a pattern that matches } p, (\text{not } p), \text{ or any monadic function applied to } p, (f \: p). \text{ We call any expression that matches } \sigma(p) \text{ a \textit{monadic variant} of } p. \text{ We say that a formula } p' : t \text{ is about } p \text{ if } p' \text{ is a monadic variant of } p. \text{ Monadic function applications have special status here because many monadic functions act to destruct any input and return some part of it (e.g. } \texttt{car}, \texttt{member-of}, \text{ etc.). Finding properties of the part returned is generally useful in finding properties of the original whole. Lastly, we use the notation } \text{THMS}^e_{\Sigma_c}(s) \text{ (which we read \textquotedblleft theorems about } s \text{ provable from } \Sigma^e \text{) to abbreviate the set of all formulas about } s \text{ provable from } \Sigma \text{ using } \vdash^e. \text{ We require that when a pattern } \sigma(p) \text{ occurs more than once in a single rule, it must match } p \text{ in the same way each time; using the same monadic function, if any in each match.}

\text{The Analyze-If inference rule performs a simple case analysis, the Analyze-Lambda and Analyze-Let rules perform simple universal generalization, and the Basic-Analyze rule applies the basic inference relation } \vdash. \text{ It
Figure 5: **Rules for Definition Analysis.** In the rule If-DA, a reordering is suitable if it gives consequent theorems with no free variables—the rule does not fire for every suitable reordering but picks just one arbitrarily. The pattern \( r \cdot B \) matches either \( rs \) or \( xr \), but must match the same way throughout the rule.

Figure 6: **Sequent Rules for \( \vdash_e \), \( \sigma \) is discussed in the text.** \( r, s, t, u, l, \) and \( B \) are any property expressions. The \texttt{fix} and \texttt{lambda} rules are shown for one argument functions for readability. The notation \( [x_1/x]s \) stands for the expression \( s \) with every free occurrence of \( x \) replaced by \( x_1 \). \texttt{Neg}(\( r; s \)) is the formula \( r; (not \ s) \) if \( r \) is a program expression, and \( r; (a\text{-thing}) \) otherwise.

remains to specify how induction hypotheses for the rule Analyze-Fix are selected. Space allows only the following concise description: for each expression \( r \) which matches \( \sigma(B) \), we compute a sequence of hypotheses \( \overline{Y}_0, \overline{Y}_1, \ldots \) where each \( \overline{Y}_i \) is a set of properties that is a subset of \( \overline{Y}_{i-1} \). This sequence eventually reaches the desired fixed-point hypothesis to use in applying the rule Analyze-Fix with \( \sigma(B) = r \). The sequence is defined as follows\(^5\):

\[
\overline{T}_r(\overline{Y}) = \begin{cases} 
\{ t \mid r : t \} & \text{if } \overline{Y} = \emptyset \\
\overline{T}_r(\overline{Y}) \cap \overline{Y} & \text{if } \overline{Y} \neq \emptyset 
\end{cases}
\]

\( \overline{Y}_0 = \overline{T}_r(\emptyset) \)

\( \overline{Y}_{i+1} = \overline{T}_r(\overline{Y}_i) \cap \overline{Y}_i \)

where \( \overline{T}_r(\overline{Y}) \) is the all-of expression representing the intersection of the members of \( \overline{Y} \), and \( \sigma_r(g \cdot x_1 \ldots x_n) \) is the result of replacing \( B \) by \( (g \cdot x_1 \ldots x_n) \) in \( \overline{Y} \). The design of the \( \vdash_e \) relation given below ensures that the initial set \( \overline{Y}_0 \) is polynomial in size, thus ensuring that this process terminates in polynomially many iterations. The recursive descent structure of \( \vdash_e \) ensures that each iteration invokes \( \vdash_e \) polynomially often (in the size of \( e \)).

**Basic Inference** The basic inference relation \( \vdash_e \) is the heart of our inference system. This relation is designed by the following general methodology. We start

\(^5\) Alternatively, and more practically, the hypotheses for different values of \( r \) can be computed and used together, giving a somewhat stronger inference relation. We present the simpler form here for ease in presentation.
by introducing a new formula \( \text{Dom}(p) \) that is used only as a flag for the inference process (these formulas have no semantic content). The intended meaning for this flag is that the property \( p \) is of interest to the reasoning process if and only if \( \text{Dom}(p) \) has been asserted. We then write a set of domain construction inference rules that ensure that this flag is asserted about an appropriately broad class of properties (usually beginning with those properties appearing directly in the problem), taking care to limit this class to polynomial size.

The domain construction rules for \( \gamma \), are exhibited in figure 7. We denote the class of property expressions \( p \) for which \( \text{Dom}(p) \) is asserted by the symbol \( A \). The rules in figure 7 construct an \( A \) which is polynomial in the size of the definition being analyzed plus the size of the background library\(^6\), given an assumed upper bound on the depth of quantification in the library (i.e. the number of variables in a \texttt{forall} construct is bounded).

After designing the domain construction rules, we write a separate set of inference rules aimed at capturing the semantics of the language constructs. We write these rules with little or no concern for the computational complexity of computing their closure. Finally, we restrict these rules by adding \( \text{Dom}() \) antecedents so that every property that appears in any conclusion is a monadic variant of a member of \( A \). This restriction ensures that the resulting \( \gamma \), relation can be computed in a forward-chaining manner in polynomial time in the size of the definition being analyzed plus the background library. We show the domain-restricted versions of the semantics capturing rules for \( \gamma \), in figure 8.

**Inference Algorithm Summary** For each definition encountered, the inference algorithm adds to the background library of type theorems those theorems

---

\(^6\)The type structure provided by properties makes it possible to design the algorithm to avoid using parts of the library involving types that do not hold of the expression being analyzed. For this reason we expect the complexity in practice to be polynomial in the \textit{logarithm} of the library size.
implied by either $\vdash_e$ or $\vdash_e^\star$, each of which is polynomial time computable in the size of the new definition body $e$ plus the size of the background library. Each occurrence of the $e$ in each theorem is replaced by the new symbol being defined before adding to the library.

**Background Library Needed For Examples**

The insertion sort and clique finding examples given above rely on the presence in the background library of a few theorems that this algorithm does not find on its own. These theorems are shown in figure 9. These are not theorems about insertion sort or clique finding, but rather theorems about the properties involved. The need for these theorems reflects that this algorithm needs a deeper understanding of properties such as a-permutation-of than that it attains by reading the definition in order to infer that property for some programs. Most theorems needed are automatically inferred—one such theorem was shown in figure 5. Also, the library theorems in our examples concern only the definitions in the type library, not the programs being analyzed—all the theorems needed about the target programs are automatically inferred. Nevertheless, the library theorems needed point up opportunities to strengthen the algorithm we have described—by analyzing the proofs of these theorems to determine which aspects fail to be discovered automatically we may discover new inference principles which can usefully be added to a polynomial time inference procedure.

**Conclusions**

We have presented a novel language for defining arbitrary properties of computer programs. The representational features of this property language were selected to enlarge the set of polynomial-time checkable property-program relationships. We have presented, in as much detail as space allows, an inference procedure which can infer interesting properties of simple computer programs in the context of a library of background knowledge, and guarantees completion in polynomial time in the size of its input.

We do not claim that the algorithm we have presented is distinguished among similar algorithms. We intend this algorithm as an example of what may be accomplished in this area. We desire the construction of stronger similar algorithms, together with a large corpus of example program-property theorems on which to test such algorithms. These algorithms can be seen as roughly analogous to the human notion of “obvious consequence”—what consequences can be inferred quickly? The human “obviousness engine” works with a very expressive language, is naturally incomplete, returns its answers quickly, and has no apparent clean characterization of the set of conclusions it finds. We propose the study of the machine counterpart to this human notion wherein we require rapid termination and refuse to limit expressiveness simply to get a clean characterization of the inferable properties.

**References**


