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 $\mathcal{RE}$ 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 returns a sorted permutation of its input, and that a clique finding program correctly returns a clique.

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[1] 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.

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)(Nebel 1990). 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 $\mathcal{RE}$ set can be represented as a property), but insisting that our inference principles remain within polynomial-time.

Work on fast decision procedures in theorem proving is also closely related to this paper.(Nelson & Oppen 1980) Congruence closure, which decides the literal set satisfiability problem for the first-order theory of equality, is a polynomial-time procedure that has proven very useful in theorem proving(Owre, Rushby, & Shankar 1992). Congruence closure can be easily and naturally integrated with the inference techniques in this paper, and this work can be viewed as an extension of congruence closure (the extension addresses an undecidable problem, unlike congruence closure).

Since the problem we are addressing is undecidable, we cannot hope to demonstrate the adequacy of our

[1]We are using "stronger" here to refer to the partial order that relates two inference procedures when one always infers a strict superset of the theorems the other infers.

inference algorithm with a completeness theorem. Instead, we give examples of the kind of properties we can hope to infer with this algorithm. Unfortunately, but intrinsically, we do not have a clean characterization of the set of inferrable properties—the demand for such a characterization is one of the fetters we are throwing aside in search of stronger fast inference.

Nevertheless, it is important to have some means of evaluating our procedures. The examples contained herein are just a start—we envision and desire the development of a large corpus of such examples of programs and associated inferrable properties. Of course, for any such corpus there is a constant time algorithm that infers all the listed properties—what we really desire is a large corpus that can be handled by a single relatively simple polynomial-time algorithm. The algorithm we present below is intended only as a start in this direction—we believe more inference principles will need to be added to it to achieve the desired generality. Still, this algorithm gives interesting results on the examples exhibited, and was not designed with them specifically in mind. *Our primary intent is to convince the reader that an algorithm of this sort can make interesting progress in the undecidable domain of general purpose property inference, and do so within polynomial-time.*

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 `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:

$$e ::= x \quad | \ (\texttt{let}\ \ x\!:\!e\ \ e_1) \ | \ (f\ e_1 \cdots e_n) \\ | \ (\texttt{if}\ \ e : p^*\ \ e_1\ \ e_2) \ | \ {}'sym$$

where x can be any variable, f can be `cons`, `car`, `cdr`, or any n-ary program function-symbol, *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 `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.

```
(define (insert x:(a-number) l:(a-sorted-list))
  (if l:'nil
      (cons x l)
      (if x:(>= (car l))
          (cons x l)
          (cons (car l) (insert x (cdr l))))))

(define (sort l:(a-numlist))
  (if l:'nil
      l
      (insert (car l) (sort (cdr l)))))

(define (intersect l1:(a-list) l2:(a-list))
  (if l1:'nil
      'nil
      (if (car l1):(a-member-of l2)
          (cons (car l1)
                (intersect (cdr l1) l2))
          (intersect (cdr l1) l2))))

(define (largest-clique-in s:(a-list))
  (if s:'nil
      'nil
      (let n:(car s)
           c:(cons n (largest-clique-in
                      (intersect (neighbors n)
                                 (cdr s))))
           c':(largest-clique-in (cdr s)))
         (if (length c):(>= (length c'))
             c
             c'))))

(define (delete x:(a-thing)
                l:(a-list))
  (if l:'nil 'nil
      (if x:(car l)
          (cdr l)
          (cons (car l) (delete x (cdr l))))))
```

Figure 1: **Example program definitions.** We omit simple definitions which have no effect on the example inferences claimed later (`length`, `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 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))))

;; any list of numbers
(define (a-numlist)
  (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 `(every` p`))` is analogous to the ordinary application (f p) except that only output values that can be produced for *every* value p might return are kept. In other words, `(`f `(every` p`))` denotes the set of values that are f-related to every value of p. As a 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 & Givan 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 definition 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).

```
(define (a-permutation-of l:(a-list))
  (if l:'nil
      l
      (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**

Formally, property expressions are defined as:

$$
\begin{aligned}
p ::= x \;&\mid\; (\texttt{let}\ x\!:\!p\ p_1) \;\mid\; (\texttt{if}\ \ p_1\!:\!p_2\ \ p_3\ \ p_4)\\
&\mid\; (f\ p_1 \cdots p_n) \;\mid\; (f\ (\texttt{every}\ p))\\
&\mid\; (\texttt{one-of}\ s) \;\mid\; (\texttt{all-of}\ s)\\
&\mid\; (\texttt{not}\ p) \;\mid\; {}'\mathit{sym} \;\mid\; \perp \;\mid\; (\texttt{a-thing})
\end{aligned}
$$

where f can be `cons`, `car`, `cdr`, or any defined program or property function symbol, *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, $\perp$ denotes the empty set of values, and `(a-thing)` denotes the set of all possible values (quoted symbols or finite `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

[2] `one-of` is closely related to `amb` introduced by McCarthy in (McCarthy 1967)

[3] If you have trouble understanding the intended meaning of any of our property definitions, we recommend trying this evaluation strategy.

(`lambda` $x_1 : p_1, \cdots, x_n : p_n$ p) or (`fix` f $x_1 : p_1, \cdots, x_n : p_n$ p) where the body p must be deterministic (i.e. a program expression) if the symbol being defined is a program function-symbol[4]. The properties p_j *can* reference and depend on the variables $x_1, \ldots, x_{j-1}$. Note that we differentiate between program function-symbols and property function-symbols.

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

We note that the expression `(`f `(every` e`))` for *program expression* e has the same meaning as the expression `(`f e`)`. 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`)` will match the pattern `(`R `(every` s`))` with f instantiating R and e instantiating s.

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 *testable*.

An Inference Algorithm

Our inference algorithm accepts as input a new definition (of a program function or a property function) and a background library of *type theorems*, and produces as output some new type theorems about the newly defined symbol that are then added to the background library. By "type theorem" we mean a universally quantified formula of the form `forall` $x_1{:}p_1 \cdots x_n{:}p_n$. $p : p'$ where p will typically involve the new symbol and p' is viewed as a type or property being asserted about p. 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 `lambda` or `fix` expression e to the new function symbol g, with respect to background library $\mathcal{L}$ (in particular, e will appear as a subscript on our inference relation symbols $\vdash_e$ and $\vdash\!\circ_e$ to indicate their dependence on e).

Definition Analysis An initial inference stage generates very useful type theorems from a superficial 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 $\overset{\text{DA}}{\vdash}$ shown in figure 5 along with an example theorem.

```
forall l:(a-numlist)
  (sort l):(a-permutation-of l)

forall l:(a-numlist)
  (sort l):(a-sorted-list)

forall l:(a-list)
  (find-largest-clique l):(a-clique)
```

Figure 4: **Sample Output.** Some theorems generated automatically using the library in figure 9.

Recursive Descent Rules 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 $\vdash_e$ 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.

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

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 $\vdash$. It

[4] We must restrict recursive definitions to ensure that every `fix` expression accepted has a well-defined least fixed-point. This can be done for example with a simple syntactic restriction discussed in the longer version of this paper. In addition to this restriction, we require definitions of function symbols to be used in program expressions to be *syntactically terminating*. Checking termination is a deep problem itself, and can be addressed by methods similar to those in this paper(McAllester & Arkoudas 1996).

Start-DA

$$\frac{\begin{array}{l} e \text{ is (lambda } x_1 : p_1 \cdots x_n : p_n \ B) \\ \quad \text{or (fix } g \text{ (lambda } x_1 : p_1 \cdots x_n : p_n \ B)) \end{array}}{\begin{array}{l} \overset{\text{DA}}{\vdash}_e \ \text{Forall } x_1 : p_1 \cdots x_n : p_n \ B{:}(e \ x_1 \ldots x_n) \\ \overset{\text{DA}}{\vdash}_e \ \text{Forall } x_1 : p_1 \cdots x_n : p_n \ (e \ x_1 \ldots x_n){:}B \end{array}}$$

If-DA

$$\frac{\begin{array}{l} \overset{\text{DA}}{\vdash}_e \ \text{Forall } x_1 : p_1 \cdots x_n : p_n \ (\text{if } y_i : s \ B_1 \ B_2) * t \\ y_1 : t_1 \cdots y_n : t_n \text{ is a suitable reordering of } x_1 : p_1 \cdots x_n : p_n \end{array}}{\begin{array}{l} \overset{\text{DA}}{\vdash}_e \ \text{Forall } y_1 : t_1 \cdots y_i : (\text{all-of } t_i \ s) \cdots y_n : t_n \\ \qquad B_1 * t \\ \overset{\text{DA}}{\vdash}_e \ \text{Forall } y_1 : t_1 \cdots y_i : (\text{all-of } t_i \ (\text{not } s)) \cdots y_n : t_n \\ \qquad B_2 * t \end{array}}$$

Let-DA

$$\frac{\overset{\text{DA}}{\vdash}_e \ \text{Forall } x_1 : p_1 \cdots x_n : p_n \ (\text{let } x : s \ B){:}t}{\overset{\text{DA}}{\vdash}_e \ \text{Forall } x_1 : p_1 \cdots x_n : p_n \, x : s \ \ B{:}t}$$

One-of-DA

$$\frac{\overset{\text{DA}}{\vdash}_e \ \text{Forall } x_1 : p_1 \cdots x_n : p_n \ (\text{one-of } B_1 \ldots B_m){:}t}{\overset{\text{DA}}{\vdash}_e \ \text{Forall } x_1 : p_1 \cdots x_n : p_n \ B_i{:}t, \ i = 1 \ldots m}$$

Sample theorem: Forall l:(all-of (a-list) (not 'nil)) .
(a-member-of (cdr l)):(a-member-of l)

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 $r * s$ matches either $r{:}s$ or $s{:}r$, but must match the same way throughout the rule.

Analyze-If

$$\frac{\begin{array}{l} \Gamma = \Sigma \cup \mathit{THMS}_{\Sigma,e}(r) \\ \qquad \cup \ \mathit{THMS}_{\Sigma,e}(s) \\ \Gamma, \ r{:}s \qquad \vdash\!\circ_e \ \sigma(u1){:}t \\ \Gamma, \ \mathit{Neg}(r{:}s) \vdash\!\circ_e \ \sigma(u2){:}t \end{array}}{\Sigma \vdash\!\circ_e \sigma((\text{if } r{:}s \ u1 \ u2)){:}t}$$

Analyze-Let

$$\frac{\begin{array}{l} \Gamma = \Sigma \cup \mathit{THMS}_{\Sigma,e}(r) \\ \Gamma, \ x{:}r \ \vdash\!\circ_e \ \sigma(s){:}t \\ x \text{ not in } \Gamma \text{ or } t \end{array}}{\Sigma \ \vdash\!\circ_e \ \sigma((\text{let } x{:}r \ s)){:}t}$$

Analyze-Lambda

$$\frac{\begin{array}{l} \Gamma = \Sigma \cup \mathit{THMS}_{\Sigma,e}(r) \\ \Gamma, \ x : r \vdash\!\circ_e \sigma(B){:}t \\ x, x_1 \text{ not in } \Gamma, \ \ x_1 \text{ not in } B \\ e \text{ is } \texttt{(lambda } x : r \ \ B\texttt{)} \end{array}}{\Sigma \vdash\!\circ_e \begin{array}{l} \text{Forall } x_1 : r \\ \quad \sigma((e \ x_1)){:}[x_1/x] \, t \end{array}}$$

Basic-Analyze

$$\frac{\Sigma \ \vdash_e \ \ s : t}{\Sigma \vdash\!\circ_e s : t}$$

Analyze-Fix

$$\frac{\begin{array}{l} \Gamma = \Sigma \cup \mathit{THMS}_{\Sigma,e}(r) \\ \Gamma, \begin{array}{l} \text{Forall } x_1 : r \\ \quad \sigma(f \ x_1) : [x_1/x] \, I \end{array}, x : r \vdash\!\circ_e \sigma(B) : I \\ x \text{ and } x_1 \text{ not in } \Gamma, \ \ x_1 \text{ not in } B \\ e \text{ is } \texttt{(fix } f \ \ x : r \ \ B\texttt{)} \end{array}}{\Sigma \vdash\!\circ_e \begin{array}{l} \text{Forall } x_1{:}r \\ \quad \sigma((e \ x_1)) : [x_1/x] \, I \end{array}}$$

Figure 6: **Sequent Rules for** $\vdash\!\circ_e$. σ is discussed in the text. r, s, t, u, I, and B are any property expressions. The `fix` and `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 replaces by x_1. $\mathit{Neg}(r{:}s)$ is the formula r:(not s) if r is a program expression, and r:(a-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 $\Upsilon_0^r, \Upsilon_1^r \cdots$ where each Υ_i^r is a set of properties that is a subset of Υ_{i-1}^r. 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]:

$$\begin{aligned} \mathcal{T}_r(\Upsilon) &= \\ & \left\{ t \ \middle| \ \mathcal{L} \cup \left\{ \begin{array}{c} \texttt{Forall } x_1 : p_1 \ldots x_n : p_n \\ \sigma_r(g \ x_1 \ldots x_n) : \mathrm{Int}(\Upsilon) \end{array} \right\} \vdash\!\circ_e r : t \right\} \\ \Upsilon_0^r &= \mathcal{T}_r(\{\bot\}) \\ \Upsilon_{i+1}^r &= \mathcal{T}_r(\Upsilon_i^r) \cap \Upsilon_i^r \end{aligned}$$

where $\mathrm{Int}(\Upsilon)$ is the `all-of` expression representing the intersection of the members of Υ, and $\sigma_r(g \ x_1 \ldots x_n)$ is the result of replacing B by $(g \ x_1 \ldots x_n)$ in r. The design of the $\vdash_e$ relation given below ensures that the initial set Υ_0 is polynomial in size, thus ensuring that this process terminates in polynomially many iterations. The recursive descent structure of $\vdash\!\circ_e$ 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.

Start	$s \in e$	`Dom` $(\sigma(s))$
Subexp	`Dom` (r); $s \in r$	`Dom` (s)
Univ-Dom	forall x:s Φ	`Dom` (s)
Univ-Inst	forall x:s Φ; d:s, $d \in e$	$[d/x]\,\Phi$; `Dom` $([d/x]\,\Phi)$

Figure 7: **Domain Construction Inference Rules for $\vdash_e$.** d must be a *program expression*, r and s can be any property expressions. The notation $[d/x]\,s$ denotes s with each free occurrence of x replaced by d. We wrote $s \in r$ to mean s is a subexpression of r. The rule "Start" adds every monadic variant of an expression in e to the domain.

by introducing a new formula `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 `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 $\vdash_e$ are exhibited in figure 7. We denote the class of property expressions p for which `Dom` (p) is asserted by the symbol $\mathcal{A}$. The rules in figure 7 construct an $\mathcal{A}$ 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 `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 `Dom` () antecedents so that every property that appears in any conclusion is a monadic variant of a member of $\mathcal{A}$. This restriction ensures that the resulting $\vdash_e$ 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 $\vdash_e$ 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 *logarithm* of the library size.

Sym	c:d	d:c
Trans	r:s; s:t	r:t
Not-Sym	`Dom` (d); d:(not e)	e:(not d)
One-Of1	r:(one-of s t); r:(not s)	r:t
One-Of2	s:r, t:r; `Dom` ((one-of s t))	(one-of s t):r
Under-All-Of	`Dom` ((all-of s t)); r:s, r:t	r:(all-of s t)
Basic-One-Of	`Dom` ((one-of s t))	s:(one-of s t); t:(one-of s t)
Basic-All-Of	`Dom` ((all-of s t))	(all-of s t):s; (all-of s t):t
Selectors1	`Dom` ((cons c d))	c:(car (cons c d)); d:(cdr (cons c d))
Selectors2	`Dom` (r); r:(cons s t)	(car r):s; (cdr r):t
Always'	`Dom` (s)	s:s, $\bot$:s; s:(a-thing)
Strictness	`Dom` ((f s)); s : $\bot$	(f s) : $\bot$
Monotone1	s:t; r:(f s); `Dom` ((f t))	r:(f t)
Monotone2	s:t; r:(f (every t)); `Dom` (f (every s))	r:(f (every s))
Every-One-Of	r:(f (every s)); r:(f (every t)); u:(one-of s t); `Dom` (f (every u))	r:(f (every u))
Not-One-Of	r:(not (one-of s t))	r:(not s); r:(not t)

Figure 8: **Basic Inference Rules for $\vdash_e$.** c and d must be *program expressions.* r, s, t, and u can be any property expressions. f can be any function symbol, constructor or selector. We show rules for `one-of` and `all-of` for the two argument case and applications for one argument for readability.

```
forall l:(a-list) .
  l:(a-permutation-of l)

forall l:(a-list) .
  (a-permutation-of (a-permutation-of l)):
  (a-permutation-of l)

forall n:(a-thing) .
  n:(a-neighbor-of (every (neighbor-of n)))

forall x:(a-number) y:(all-of (a-number)
                              (not (>= x))) .
  x:(>= y)

forall l:(all-of (a-sorted-list)
                 (not 'nil))
       x:(>= (car l)) .
  x:(>= (every (a-member-of l)))
```

Figure 9: **Library Theorems.** The theorems needed from the user or the property library.

implied by either $\overset{\mathrm{DA}}{\vdash}_e$ or $\vdash\!\circ_e$, 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 inferrable properties.

References

Abramsky, S., and Hankin, C., eds. 1987. *Abstract Interpretation of Declarative Languages.* Ellis Horwood.

Brachman, R., and Schmolze, J. 1985. An overview of the kl-one knowledge representation system. *Computational Intelligence* 9(2):171–216.

McAllester, D., and Arkoudas, K. 1996. Walther recursion. In *13th International Conference on Automated Deduction.*

McAllester, D., and Givan, R. 1992. Natural language syntax and first order inference. *Artificial Intelligence* 56:1–20. ftp.ai.mit.edu:/pub/users/dam/aij1.ps.

McAllester, D., and Givan, R. 1993. Taxonomic syntax for first order inference. *JACM* 40(2):246–283. ftp.ai.mit.edu:/pub/users/dam/jacm1.ps.

McCarthy, J. 1967. A basis for a mathematical theory of computation. In Braffort, P., and Hirschberg, D., eds., *Computer Programing and Formal Systems.* North-Holland.

Milner, R. 1978. A theory of type polymorphism in programming. *JCSS* 17:348–375.

Nebel, B. 1990. Terminological reasoning is inherently intractable. *Artificial Intelligence* 43:235–249.

Nelson, G., and Oppen, D. 1980. Fast decision procedures based on congruence closure. *JACM* 27(2):356.

Owre, S.; Rushby, J.; and Shankar, N. 1992. Pvs: A prototype verification system. In *11th International Conference on Automated Deduction*, 748–752.