Inferring Program Specifications in Polynomial-Time

Robert Givan

Massachusetts Institute of Technology,
NE43-430, Cambridge, MA 02139, USA
rlg@ai.mit.edu, http://www.ai.mit.edu/people/rlg/rlg.html

Abstract. We consider the problem of automatically inferring properties of programs. Our approach is to explore the application of familiar type inference principles to a "type system" sufficiently expressive that the typing problem is effectively the checking of program specifications. We use familiar syntax-directed type inference rules to give a polynomial-time procedure for inferring type theorems in this rich type system. We discuss examples of simple functional programs and the specification information this procedure automatically infers. The enriched notion of type allows the definition of any recursively enumerable set as a type, and includes argument-dependent output types for functions. The inference procedure is capable for example of automatically inferring that an insertion sort program always returns a sorted permutation of its input.

Keywords: Functional Programming, Type Inference, Verification, Induction

1 Introduction

Many researchers have studied type inference systems for functional programming languages[10, 6, 11, 2]. The typical goal of such research is to allow the programmer to omit type declarations without losing the benefits they provide. The types inferred by such systems are typically similar to the primitive types of a typed programming language with typings for functions added (so that $\alpha \rightarrow \beta$ is a type whenever α and β are). Many such type inference systems can be described by sets of locally-acting syntax-directed type inference rules.

More recently, effective type inference systems have been given for more expressive type systems, e.g., allowing conditional types[1]. The stated motivation for such increased expressiveness is to be able to infer types for more programs to ensure type safety. These systems, like most type inference systems, typically have poor worst-case complexity while retaining practical effectiveness.

We believe that there is a continuum between checking type safety and verifying program correctness. As the language of the types inferred becomes more expressive, the type inferences can more precisely characterize the outputs of the programs being analyzed. Rather than inspire our type system from the types present in programming languages, we suggest we draw inspiration from the types present in programmers' analysis of their own programs. Not only do programmers use a very expressive type language (natural language), but we observe that they are effective at quickly analyzing their own programs to draw expressive typing conclusions. For example, a programmer writing an insertion sort program can typically quickly and easily verify that his program returns a sorted permutation of its input—we view this as the typing conclusion that the output of "sort(l)" has the "types" "a sorted list" and "a permutation of l".

We take this human capability, along with the above-stated trend in type inference systems, as evidence that there must exist fast and effective "type inference" algorithms for very rich type systems. We define in this paper a generalization of the traditional notion of "type" to a much more expressive notion of "specification", or "spec", and then give a type inference style algorithm for inferring specifications for functional program expressions. Our algorithm runs in polynomial-time, and is capable of automatically inferring specifications such as the fact that insertion sort returns a sorted permutation of its input.

Note that the specifications "a permutation of the input" and "a sorted list" differ from types in traditional type systems in at least two ways. The first specification depends on the actual input to the function (not just the input's type). Such types are known as *dependent types*[13, 4], and our system depends critically on including such types in our "specification" language. Second, the set of "sorted lists" is not definable by a simple grammar, and so is not a *regular* type. [14, 12, 3, 15] Our specification language allows any $\mathcal{RE}$ set to be defined as a program specification. Note that this property allows one program, possibly very inefficient but simple to understand, to serve as a correctness specification for another program, more efficient but harder to understand.

We envision an interactive programming system in which programmers write programs that include information about the specifications the programs are intended to meet, using an expressive specification language. As the program is written, the system checks that it is well-typed in the sense that no function is applied to arguments that don't provably meet the declared argument specifications for the function. Ideally, the system would be able to infer specifications for expressions quickly and with human-level competence. Where necessary, the programmer would switch to a theorem proving mode and prove lemmas necessary to aid the verification of the well-typedness.

Note that such a system would not require programmers to prove any more than they desired about the program. By providing more, or less, specification information to the system, the programmer can control where on the continuum from checking run-time type safety to verifying program specifications the programming process falls. By adding more specification information, the programmer can be sure that not only is "plus" receiving only numerical inputs, but that "merge" is in fact passed two sorted lists, for example. By adding even more, it may become verifiable that "mergesort" correctly sorts its input.

Short of achieving human-level specification inference, we believe that the simplicity of the inference rules defining our algorithm makes it possible for a programmer to develop the ability to predict what expressions the system will be able to compute specifications for, and where and how it will need help. This property may make an interactive environment based on this system acceptable to some programmers in spite of below human-level specification inference.

The remainder of this paper is structured as follows: first, we present several examples of simple programs and their automatically computed specifications; second, we present the formal syntax and semantics of our programming and specification languages; third, we present our inference algorithm, and then revisit the examples to demonstrate how it works.

2 Some Examples of Quickly Verifiable Specifications

We begin with an informal discussion of our programming and specification languages, and examples of simple programs and their automatically computed specifications. Later sections will contain a more formal treatment.

2.1 Example Programs

The programming language we will use is a simplified, typed first-order variant of LISP. We call it *first-order* because it does not include first-class functions; rather, user functions are introduced only through definitions (possibly recursive) and used only by being applied to arguments.[1] We call it *typed* because every variable is given at its introduction a user-provided *specification* (sometimes abbreviated *spec*). These specifications function much like types in a simply typed programming language, except that they range over our specification language, which is much more expressive than any familiar type system. Because the specification language is so expressive, we don't expect providing specifications for variables to be a significant burden on programmers, though it will still carry some of the advantages of simply typed languages.

The programming language includes constructor and selector symbols (e.g. `cons, car, cdr`) and has the intended semantics that each program expression denotes some term in the Herbrand closure of the constructor symbols.[2]

Unlike LISP, our language syntax has a distinguished formula category, with formulas of the form $e{:}s$ meaning "e meets the spec s". We will discuss the computation implied by such formulas later. We now discuss computing specs for three example programs. The user-provided definitions of the specification functions (e.g. `(a-number)`) in these examples are shown and explained below.

Our first example program recursively defines `+` on numbers represented in unary as lists of the symbol `'a`. This program defines `+` to be a function that operates on two arguments. Each of the arguments is declared to meet the spec `(a-number)`. Our system automatically determines that `(+ x y)` is always greater than or equal to `x` and `y` (i.e., meets the specs `(≥ x)` and `(≥ y)`).

Our second example program defines insertion sort on a list of numbers, using functions `insert` and `sort`. Our system automatically finds that `(insert x l)`

[1] We omit first-class functions only for simplicity here. We believe this work extends naturally to higher-order languages.

[2] We require recursive definitions to be syntactically terminating.

returns a sorted permutation of `(cons x l)`, i.e., meets the specs `(a-permutation-of (cons x l))` and `(a-sorted-list)` and that `(sort l)` always returns a sorted permutation of the list `l`.[3]

Our third and last example program is a first-order version of LISP's `mapcar`. Here, we map a fixed function `f` across a list of numbers. Our system automatically infers the spec `(samelength-as l)` from reading the definition of `map-f`.

```
(define (+ (x (a-number))
           (y (a-number)))
  (if x:'nil
      y
      (cons 'a (+ (cdr x) y))))

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

```
(define (map-f (l (a-numlist)))
  (if l:'nil
      l
      (cons (f (car l))
            (map-f (cdr l)))))

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

Fig. 1. Example program definitions.

2.2 Example Specifications

The specification language is less familiar. This language is essentially our programming language extended by a nondeterministic `either` combinator.[9, 8] Expressions in the specification language can take on more than one possible value. The set of possible values of a specification expression can be viewed as the type defined by that expression.

The `either` combinator applied to two expressions yields an expression that can nondeterministically take on any of the values of either of the two arguments. For example, the expression `(either 'a 'b)` can take on either of two the values `'a` or `'b`, and is our way of representing the type {`'a`, `'b`}. Note that with recursion, a single nondeterministic expression have an infinity of values.

We add two other new combinators to the language to take the intersection or the set complement of the possible values of their arguments (`both` and `not`,

[3] As we will exhibit below, the system has no built-in knowledge of permutations. `(a-permutation-of l)` is a spec defined by the user for arbitrary list `l`. Once the user (or a specification library) provides that definition and proves two simple and natural theorems about it, the system can infer that `sort` has the desired spec.

respectively). Finally, we add a universal spec (a-thing) that nondeterministically returns any value at all, and an empty spec $\perp$ that returns no values.

Note that specs, like programs, can contain variables. Moreover, a specification variable can be bound by a program, in which case it refers to the object that the program variable is eventually instantiated with. This means that specs can represent dependent types, i.e., types that depend on an argument to the function being defined. This added expressiveness is an important element of our system, and the examples described in this section centrally involve dependent types.

Figure 2 exhibits the definitions for all the specification functions used in the examples above. Consider, e.g., the definition shown of the function `a-number`. Given this definition, `(a-number)` denotes the set of all flat lists of `'a` symbols.

```
;; all lists with samelength as l
(define (samelength-as (l (a-list)))
  (if l:'nil
      l
      (cons (a-thing)
            (samelength-as (cdr l)))))

(define (a-list-member-of (l (a-list)))
  (if l:'nil
      bottom
      (either (car l)
         (a-list-member-of (cdr l)))))

;; sorted lists starting with x
(define (an-slist-from (x (a-number)))
  (either 'nil
     (cons x (an-slist-from (>= x)))))

(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))))))

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

(define (a-number)
  (either 'nil
        (cons 'a (a-number))))

(define (a-list)
  (either 'nil
          (cons (a-thing)
                (a-list))))

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

;; a sorted list of numbers
(define (a-sorted-list)
  (an-slist-from (a-number)))

;; the numbers >= x
(define (>= (x (a-number)))
  (either x
          (cons 'a (>= x))))

;; the numbers > x
(define (> (x (a-number)))
  (both (>= x) (not x)))
```

Fig. 2. Example Specification Definitions

Finally, we say that a program expression e *satisfies* a spec s, written e:s, if the value denoted by e is one of the possible values taken on by s. Abusing notation, we also say that a spec t *satisfies* another spec s, written t:s, if every value of t can be taken by s (analogous to the standard notion of *subtype*).

As a simple example, consider the spec expression for "non-zero number", `(cons 'a (a-number))`. Every value of this expression is a value of `(a-number)`, so the expression satisfies the spec `(a-number)`.

In the definition of `insert` above we used the formula `x:(> (car l))` in an `if` test. However, `>` is a specification function, defined by a nondeterministic program. As in this case, our nondeterministic expressions often have infinitely many values, and so cannot be executed. To use the `>` function in an `if` test we must require `>` to have associated with it a means of computing membership in the resulting specification, given particular arguments. This *implementation attachment* can be written in our programming language, and proven to compute the desired result with a theorem prover. These steps are straightforward for `>`.

We choose to write our `if` tests in this manner because it makes the extraction of relevant type information from the test as straightforward as possible for the inference mechanism. In the next section we will discuss the general restriction we need to place on formulas appearing in programs to ensure that they are computable. The use of attachment can always be avoided with no loss in clarity or program effectiveness, all that is lost is those specification inferences that depend on the type information in the `if` test in question.

3 A Programming Language with Specifications

Program Expressions . The program expressions are a first-order typed LISP with constructor and selector functions, recursive definitions, `let`, and `if`:

$$e ::= x \mid (\texttt{let}\ x : e\ e_1) \mid (f\ e_1 \cdots e_n) \mid (\texttt{if}\ e : s^*\ e_1\ e_2)$$

where f can be n-ary constructor, selector or n-ary program function-symbol, and s^* must be *testable* (see below). We often write a quoted symbol as an abbreviation for the application of a 0-ary constructor.[4]

Specification Expressions . Specification expressions are formed from the same grammar extended with `either`, `both`, `not`, $\perp$, and `a-thing`:

$$\begin{aligned} s ::= x\ &\mid (\texttt{let}\ x : s\ s_1) \mid (f\ s_1 \cdots s_n) \mid (\texttt{if}\ s_1 : s_2\ s_3\ s_4) \\ &\mid (\texttt{either}\ s_1\ s_2) \mid (\texttt{both}\ s_1\ s_2) \mid (\texttt{not}\ s) \mid \perp \mid (\texttt{a-thing}) \end{aligned}$$

where f can now be any n-ary constructor or selector, or n-ary program *or specification* function-symbol that is defined or being defined. Note that every program expression is also a specification expression.[5]

[4] A full language would also include boolean operations in the formulas in `if` tests. No extra difficulties are presented by this extension.

[5] We include `both` and `not` for convenience—they can also be taken to abbreviate appropriate expressions using `let` and `if`, recognized by the inference process.

Programs . We consider a sequence of function-symbols definitions to be a program. A function-symbol definition assigns to a new function-symbol either of (`lambda` $x_1 : s_1, \cdots, x_n : s_n\ \ s$) or (`fix` $f\ \ x_1 : s_1, \cdots, x_n : s_n\ \ s$) where the body s must be deterministic (i.e. a program expression) if the symbol being defined is a program function-symbol. The specs s_j *can* reference and depend on the variables $x_1, \ldots, x_{j-1}$. Note that we differentiate between defined program function-symbols and defined specification function-symbols.

We must restrict recursive definitions to ensure that every `fix` expression accepted has a well-defined least fixed point. For this language it suffices to prohibit recursive calls in positions that are not *syntactically monotone*—we exclude recursive calls inside the test of an `if`, inside an odd number of `not` expressions, or inside the type specifications of the parameters of the definition. 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[7], but here we simply require that there be some argument to the function whose Herbrand size is reduced in each recursive call.

Semantics . Our semantic domain is the Herbrand closure over the constructor functions, with an error (ϵ) element adjoined. When a function is applied to objects outside its domain (as indicated by the specs on its formal parameters) it returns ϵ. Each program expression denotes either a Herbrand term or the error element. Each specification expression denotes a subset of the domain.

Assigning meanings to the various expressions compositionally is routine; we discuss only the unusual cases in specification meaning. Because a specification denotes a *set* of objects, the meaning of function application may not be obvious: to apply a function f to sets $\alpha_1, \ldots, \alpha_n$, choose objects $x_1, \ldots, x_n$ from the α_i respectively, and compute $f(x_1, \ldots, x_n)$. The function application will denote the set of values that can be obtained in this manner. Viewed as a nondeterministic computation, we first nondeterministically compute arguments for the function, and then apply the function, with many possible results. The `s1:s2` test of an `if` expression is true exactly when `s1` denotes a subset of `s2`'s denotation. `(a-thing)` denotes the set of all domain objects, and $\perp$ denotes the empty set. `Either`, `both`, and `not` are computed with union, intersection, and set complement relative to the domain, respectively. Any time a program expression is used as a specification, it's denotation is the set containing the object it denotes.

Note that in both specifications and program expressions, `fix` and `lambda` expressions have only one meaning (not nondeterministically many): in each case it is a relation over the domain, a functional relation for program expressions.

Implementation Attachments We require the user to attach verified program expressions to any specification function used in a way requiring it to be computed (in this language, in the test of an `if`). We also place sufficient restrictions on the use of specifications in programs to ensure that the programs can be run.

A spec s^* is *testable* if it is either a program expression or the application of a *computable* specification function to program expressions.[6] A specification

[6] The application of a specification function to a non-program expression is not directly

function is computable if it has been given a proven program implementation.

As an example, consider the `if` test `x:(> (car l))` in `insert`. Our language forces us to write this uncomputable test rather than the more familiar `(> x (car l))`. However, to avoid this restriction we can just write the program for the predicate form of > returning `'true` or `'false`, and then use `'true: (> x (car l))`. Doing this will lose the advantages our system gains from extracting type information about the formal parameter x from the `if` test.

To retain these advantages, our user must take the same predicate definition (call it `>-imp`), shown in Fig. 3 and prove the attachment theorems shown. Our system recognizes theorems of this form and will then allow the implemented specification function to appear in program expressions, as in `insert`.

```
(define (>-imp (x (a-number))
               (y (a-number)))
  (if x:'nil
      'false
      (if y:'nil
          'true
           (>-imp (cdr x) (cdr y)))))
```

```
forall y:(a-number) x:(> y)
  (>-imp x y):'true

forall y:(a-number) x:(not (> y))
  (>-imp x y):'false
```

Fig. 3. The implementation of the function > and the associated attachment theorems.

4 Program Analysis: Inferring Specifications

For each new user definition our system extracts type lemmas which are then used in the analysis of future definitions. Our algorithm thus operates in the context of a library of previously derived knowledge. This library is just a set of known universally quantified specification formulas `forall` $x_1{:}s_1 \cdots x_n{:}s_n$. $s{:}t$. We call such formulas *type theorems*. The general problem that the algorithm in this section attacks we call the *specification inference problem*. Given a library $\mathcal{L}$ of type theorems (about already processed definitions) and a new definition assigning some `lambda` or `fix` expression e to some new function symbol g, analyze e to generate new type theorems about g to add to the library $\mathcal{L}$. We present the algorithm and then discuss its application to the examples above.

4.1 An Inference Algorithm

There are three parts to our central analysis algorithm. First, a forward-chaining inference closure intended as a notion of "obvious consequence" ($\vdash_e$); second, a syntax-directed, type-inference inspired inference relation ($\vdash\!\!\circ_e$) that manages the application of $\vdash_e$; and third, a preprocessing stage which prepares the new definition for analysis by $\vdash\!\!\circ_e$. Our solution to the specification inference problem is to add to $\mathcal{L}$ those theorems inferred when $\vdash\!\!\circ_e$ is applied to the definitions generated by preprocessing.

computable even if all the functions involved have attachments.

The Forward-Chaining Inference Relation $\vdash_e$ We now define a polynomial-time computable inference relation $\vdash_e$, where e is the `lambda` or `fix` expression being analyzed. Given a premise set Σ of formulas (e.g. $s\!:\!t$), we say that $\Sigma \vdash_e s\!:\!t$ for specs s and t whenever $s\!:\!t$ is in the closure over Σ of the inference rules given below. Note that in addition to reasoning about specification formulas, the inference rules draw (and use) conclusions of the form Dom (s) for specification expression s. These *domain analysis* conclusions have no intended semantic meaning and are used by the algorithm to limit the scope of the reasoning to remain within polynomial time. We will prove that there are at most polynomially many conclusions Dom (s) inferred. The intended intuition is that the inference process reasons only about expressions in this polynomial-sized "domain". $\vdash_e$ is defined by the inference rules given in Fig. 4.

Sym
$$\frac{\texttt{p:q}}{\texttt{q:p}}$$

Trans
$$\frac{\texttt{r:s} \quad \texttt{s:t}}{\texttt{r:t}}$$

Not-Sym
$$\frac{\texttt{Dom (d)} \quad \texttt{d:(not e)}}{\texttt{e:(not d)}}$$

Either1
$$\frac{\texttt{r:(either s t)} \quad \texttt{r:(not s)}}{\texttt{r:t}}$$

Either2
$$\frac{\texttt{s:r, t:r} \quad \texttt{Dom (either s t)}}{\texttt{(either s t):r}}$$

Under-Both
$$\frac{\texttt{Dom (both s t)} \quad \texttt{r:s, r:t}}{\texttt{r:(both s t)}}$$

Basic-Either
$$\frac{\texttt{Dom (either s t)}}{\texttt{s:(either s t)} \quad \texttt{t:(either s t)}}$$

Basic-Both
$$\frac{\texttt{Dom (both s t)}}{\texttt{(both s t):s} \quad \texttt{(both s t):t}}$$

Always
$$\frac{\texttt{Dom (s)}}{\texttt{s:s}, \ \bot\texttt{:s} \quad \texttt{s:(a-thing)}}$$

Selectors1
$$\frac{\texttt{Dom (cons p q)}}{\texttt{p:(car (cons p q))} \quad \texttt{q:(cdr (cons p q))}}$$

Selectors2
$$\frac{\texttt{Dom (r)} \quad \texttt{r:(cons s t)}}{\texttt{(car r):s} \quad \texttt{(cdr r):t}}$$

Strictness
$$\frac{\texttt{Dom (f s1...sn)} \quad \texttt{si:}\bot}{\texttt{(f s1...sn):}\bot}$$

Monotonicity
$$\frac{\texttt{s1:t1...sn:tn} \quad \texttt{d:(f s1 ...sn)} \quad \texttt{Dom (f t1...tn)}}{\texttt{d:(f t1 ...tn)}}$$

Constructors
$$\frac{\texttt{Dom (c1 s1...sn)} \quad \texttt{Dom (c2 t1...tm)} \quad \texttt{c1} \neq \texttt{c2}}{\texttt{(c1 s1...sn) : (not (c2 t1...tm))}}$$

Dom-Always
$$\frac{}{\texttt{Dom (a-thing)} \quad \texttt{Dom (}\bot\texttt{), Dom (e)}}$$

Dom-Subexp
$$\frac{\texttt{Dom (}r\texttt{)} \quad s \textit{ a subexp of } r}{\texttt{Dom (}s\texttt{)}}$$

Univ-Dom
$$\frac{\texttt{forall } x\texttt{:}s \ \Phi}{\texttt{Dom (}s\texttt{)}}$$

Univ-Inst
$$\frac{\texttt{forall } x\texttt{:}s \ \Phi \quad p\texttt{:}s,\ p \textit{ appears in } e}{[p/x]\,\Phi \quad \texttt{Dom (}[p/x]\,\Phi\texttt{)}}$$

Fig. 4. Basic Inference Rules for $\vdash_e$**.** p and q must be *program expressions.* r, s and t can be any specification expressions. f can be any function symbol, constructor or selector. c is any constructor. The selector rules are shown for cons/car/cdr. In the rule Univ-Inst, the notation $[r/x]\,s$ denotes s with each free occurrence of x replaced by r.

Let Σ be a premise set of type theorems. Let $\mathcal{A}$ be the set of all expressions s such that Dom (s) is inferred by forward-chaining the above rules from Σ. We observe the following two complexity bounds:

1. $\mathcal{A}$ has at most polynomially many members in the size of Σ, and
2. The forward-chaining can be computed in polynomial-time in the size of $\mathcal{A}$.

The first bound follows from the observation (provable by induction on the length of derivation) that every spec in $\mathcal{A}$ is either $\perp$, `(a-thing)`, a subexpression of e, or a subexpression of a universal formula `forall` $x_1 : s_1 \ldots x_n : s_n\ \Phi$ in Σ with its variables replaced by subexpressions of e. The last case forces us to limit the quantification depth of formulas in Σ to some constant—then there are only polynomially many instances of universal formulas in Σ on subexpressions of e.[7]

To see the second bound, observe that by induction on the length of derivation every spec in any new conclusion is of the form: `s`, `(not s)`, `(car s)`, or `(cdr s)` for some s in $\mathcal{A}$. There are only polynomially many such conclusions and for any not yet closed premise set we can find a new conclusion in polynomial time.

We wish to point out that, although there are a large number of rules given above, they are clearly not designed for the specific examples we've exhibited. Each rule is a natural and simple local rule capturing a small piece of the meaning of one language construct. The important thing about these rules is that they capture a large polynomial time fragment of the quantifier-free inference problem. For any new language features, we can always capture some polynomial-time portion of the possible new inferences in similar forward-chaining rules. The examples serve to demonstrate the power this kind of simple rule set can wield.

The Syntax-Directed Inference Relation $\vdash\!\!\circ_e$ We now use the $\vdash_e$ relation just defined to define a stronger $\vdash\!\!\circ_e$ relation that handles `let`, `if`, `lambda`, and `fix` by adding the sequent inference rules shown below. These rules are roughly analogous to typical type inference rules: they are syntax directed, so that typing of any expression can be done in a linear number of $\vdash_e$ closures. The Analyze-Fix, Analyze-Lambda, and Beta-Abstract-e rules are shown for one-argument expressions, but the analogous rules for arbitrary arity are intended. We use the expression $THMS_{\Sigma,e}(s)$ to abbreviate the set of all specification formulas of the form $s{:}t$ provable from Σ using $\vdash\!\!\circ_e$.

The analyze-if rule does a simple case analysis on the `if` test. Because our $\vdash_e$ inference rules reason only about positive specification formulas, we negate formulas with the meta-function *Neg*, which takes as input a formula `s:t` and returns `s:(not t)` if `s` is a program expression, and `s:(a-thing)` otherwise. Analyze-let is implicitly doing universal generalization when r is not a program expression. Analyze-Nondet-App provides rudimentary reasoning about nondeterministic applications. The Analyze-Lambda, Analyze-Fix, and Beta-Abstract-e rules are needed only at the top level of function symbol definitions.

It remains to specify how induction hypotheses for the rule Analyze-Fix are selected. Space allows only the following concise description: we compute a sequence of hypotheses $\Upsilon_0, \Upsilon_1 \cdots$ where each Υ_i is a set of specifications which is a subset of Υ_{i-1}. This sequence eventually reaches the desired fixed point

[7] For lemmas that were derived by the system, the bound on quantification depth can derive from bounds on the arity of functions and the depth of `Let` nesting within analyzed definitions.

Analyze-If

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

Analyze-Let

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

Analyze-Nondet-App

$$\frac{\begin{array}{l}\Sigma \vdash\!\circ_e\ \mathtt{(let\ x1{:}s1\ldots xn{:}sn\ (f\ x1\ldots xn)){:}}t\\ \text{No } x_i \text{ in } \Sigma \text{ or } t\text{, some } s_i \text{ non-program}\end{array}}{\Sigma \vdash\!\circ_e\ \mathtt{(f\ s1\ldots sn){:}}t}$$

Beta-Abstract-e

$$\frac{e \text{ is } (\text{lambda } x{:}s\ B)}{\Sigma \vdash\!\circ_e\ \text{forall } x : s\ B{:}(e\ x)}$$

Analyze-Lambda

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

Analyze-Fix

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

Fig. 5. Sequent Rules for $\vdash\!\circ_e$. *Neg* is discussed in the text. r, s, t, u, I, and B are any specification expressions.

hypothesis. The sequence is defined as follows:

$$\mathcal{T}(\Upsilon) = \left\{ t \;\middle|\; \mathcal{L} \cup \left\{ \begin{array}{c} \texttt{Forall}\ x_1 : s_1 \ldots x_n : s_n \\ (g\ x_1 \ldots x_n) : \mathcal{B}(\Upsilon) \end{array} \right\} \vdash\!\circ_e B : t \right\}$$

$$\Upsilon_0 = \mathcal{T}(\{\bot\})$$
$$\Upsilon_{i+1} = \mathcal{T}(\Upsilon_i) \cap \Upsilon_i$$

where $\mathcal{B}(\Upsilon)$ is the `both` expression intersecting all the members of Υ

Definition Preprocessing Suppose we are presented with a new definition to analyze, defining the symbol f to be a function with arguments $x_1 \ldots x_n$ of types $s_1 \ldots s_n$ and (possibly recursive) body B. Rather than simply apply the $\vdash\!\circ$ relation directly to f, we begin by *factoring* the definition of f into a set of definitions $f_1, \ldots, f_n$ such that f is semantically the same function as one that nondeterministically picks one of the f_i and applies it. Our purposes in factoring are twofold: it enables the easy statement of some useful theorems; and it supports some limited case analysis in reasoning about applications of the function.

We first factor the definition of f into the set of definitions $f, f_1, \ldots, f_k$ where $f_1, \ldots, f_k$ are all definitions that can be produced from f by the rewrite rules in figure 6 but cannot be further rewritten (if two definitions differ only in the ordering of their formal parameters, we include only one of them in the f_i). We note that the size of $f_1, \ldots, f_k$ is linear in the size of f.

Either-Def	Let-Def
`(define (f...) (either B1 B2))`	`(define (f x1:s1...xn:sn) (let x:s B))`
`(define (f...) Bi), i=1 or 2`	`(define (f x1:s1...xn:sn x:s) B)`

If-Def

```
(define (f x1:s1...xn:sn) (if xi:s B1 B2))
y1:t1...yn:tn is a suitable reordering of the xi:si
----------------------------------------------------
(define (f y1:t1...yi:(both s ti) yn:tn) B1)
(define (f y1:t1...yi:(both (not s) ti) yn:tn) B2)
```

Fig. 6. Rewrite Rules for Definition Factoring. In the If-Def rule, a reordering is suitable if it yields an output definition with no free variables.

To try and convey some intuition about the f_i defined by this rewrite process, consider an f_i with arguments $y_1 \ldots y_m$ of types $t_1 \ldots t_m$ and body B_i. The y_j are made up of the (possibly reordered) original formal parameters of f (the x_j) and the `let` variables in scope around B_i in B. The t_j restrict the y_j to exactly those values that those variables can take in B_i during an evaluation of f. Thus, $(f_i\ t_1 \ldots t_m)$ can take on exactly the values that B_i contributes to f. We call this the *output type of* f_i.

Now consider an application of f, $(f s'_1 \cdots s'_n)$. We seek to characterize the contribution of B_i to the values of this application. We will write $t_{x_j,i}$ for the type spec of the variable x_j in the parameter list of f_i. We claim that $[(\texttt{both}\ t_{x_1,i}\ s'_1)/t_{x_1,i}] \cdots [(\texttt{both}\ t_{x_n,i}\ s'_n)/t_{x_n,i}]\,(f_i\ t_1 \ldots t_m)$ is the characterization we seek. We call this expression the *output type of* f_i *when restricted to the* s'_j. The `either` expression unioning the output types of the f_i restricted to the s'_j is equivalent to $(f s'_1 \cdots s'_n)$. We called this union expression the *factored application of* f *to the* s'_j.

Finally, we eliminate f entirely from the bodies of the f_i by replacing each application of f with its factored form, to get new definitions $f'_1, \ldots, f'_k$, and redefine f to be the union (`either`) of the f'_i. We use the $\vdash\!\circ$ relation from the previous section to analyze the mutually recursive f'_i, and then f, and add the resulting theorems to our library $\mathcal{L}$.

The combination of definition factoring with the $\vdash\!\circ$-rule Beta-Abstract-e yields many useful theorems which we call *factoring theorems*. Several examples of these are shown in Fig. 7. Factoring also enhances the conditional analysis done by the recursive descent algorithm.

4.2 Inferring the Specifications in Our Examples

We now return to our example programs and discuss their analysis

The theorems shown in Fig. 7 are among those generated automatically when reading the specification definitions shown in Sect. 2. These and others like them are used in calculating the specifications cited for the example programs.

```
(1) forall l:(both (a-list) (not 'nil))
      (cons (a-thing) (samelength-as (cdr l))) : (samelength-as l)

(2) forall l:(both (a-list) (not 'nil))
           z:(not (car l))
      (cons (car l) (delete z (cdr l))) : (delete z l)

(3) forall l:(both (a-list) (not 'nil))
           z:(a-list-member-of l)
      (cons z (a-permutation-of (delete z l))) : (a-permutation-of l)

(4) forall l:(both (a-list) (not 'nil)) (cdr l):(delete (car l) l)
(5) forall l:(both (a-list) (not 'nil)) (car l):(a-list-member-of l)
(6) forall x:(a-number) (either x (cons 'a (>= x))) : (>= x)
(7) forall x:(a-number) (> x):(both (>= x) (not x))
(8) (a-numlist):(a-list)
(9) (a-numlist):(either 'nil (cons (a-number) (a-numlist)))
(A) (either 'nil (cons (a-number) (a-list))):(a-numlist)
```

Fig. 7. Some automatically generated theorems used in the examples. All but (8) and (9) are generated by factoring analysis of specification definitions. (8) and (9) are generated by the inferential closure analysis of `a-numlist`.

Each example requires the presence of some additional simple and natural theorems, shown in Fig. 8. We intend either that the user have proven these theorems using a theorem prover or that he is using a specification library containing the definitions and theorems. Each theorem captures a basic property of the definitions, rather than a property targeted to any of our examples.[8] Most of the theorems can be proven automatically by a simple inductive theorem prover.

To conclude our discussion of these examples, we show some of the critical inference steps involved in drawing one of the tougher conclusions. To determine that `(insert x l)` has the specification `(a-permutation-of (cons x l))`, the system must first choose that specification as an inductive hypothesis. This happens because it is a specification of the base case `(cons x l)`, by the theorem that any list is a permutation of itself—and a simple inference chain that demonstrates that `(cons x l):(a-list)`. Once we have `(a-permutation-of (cons x l))` as an inductive hypothesis, the analysis of the recursive case of the `if` body goes as shown in Fig. 9. Similar chains of reasoning are involved in automatically drawing the other specification conclusions cited above.

[8] An exception to this is the second lemma on the right, which mitigates a weakness in our reasoning about nondeterminism. Stronger polynomial-time reasoning about non-determinism is possible and is addressed in the full paper[5].

```
forall l:(a-list)           forall l:(both (a-sorted-list) (not 'nil))
  l:(samelength-as l)         l:(an-slist-from (car l))

forall l:(a-list)           forall n:(a-number)
  l:(a-permutation-of l)          l:(both (an-slist-from (>= n))
                                          (not 'nil))
forall x:(a-number)           (car l):(>= n)
       y:(not (> x))
  x:(>= y)                  forall l:(a-list)
                              (a-permutation-of (a-permutation-of l)):
forall n:(a-number)           (a-permutation-of l)
  (>= (>= n)):(>= n)
forall n:(a-number) (cons 'a (>= n)):(>= (cons 'a n))
```

Fig. 8. The theorems needed from the user or the specification library.

```
(cons x (cdr lst))
   by theorem (4) above is under  (cons x (delete (car lst) lst))
   by selectors rule is under     (cons (car (cons x lst))
                                        (delete (car lst)
                                                (cdr (cons x lst))))
   by theorem (2) is under        (delete (car lst) (cons x lst))  (*)

(cons (car lst) (insert x (cdr lst)))
   by ind hyp is under   (cons (car lst)
                               (a-permutation-of (cons x (cdr lst))))
   by (*) is under       (cons (car lst)
                               (a-permutation-of
                                (delete (car lst) (cons x lst))))
   by theorem (3) is     (a-permutation-of (cons x lst))  as desired.
```

Fig. 9. The main inference chain involved in analyzing `insert`. First, `(cons x (cdr lst))` is analyzed to get the result labelled (*). This result is used to analyze the recursive branch of `insert`. The inductive hypothesis puts `(insert x (cdr lst))` under `(a-permutation-of (cons x (cdr lst)))`. Not every inference rule used is cited.

5 Conclusion

We presented a polynomial-time type inference inspired algorithm for inferring properties, viewed as types, of functional programs. We also gave an expressive language for defining new types to extend the "type system" of the algorithm. Several extensions to this work are addressed in the full version[5].

An important remaining task is developing a better understanding of what properties this algorithm and related algorithms can infer. This algorithm is capable for example of checking the correctness of merge-sort much like it checks insertion sort. In contrast, difficulties arise in a naive attempt to check quicksort. It remains an open problem to cleanly characterize the set of checkable properties.

There are many natural specification theorems that this algorithm will not infer. Exploration of such examples will suggest many ways of enriching the set of checkable properties while remaining within polynomial time. One of the difficulties in working in an area where completeness is out of reach lies in knowing when you've done enough, or whether there is such a point.

An interesting area for future research is the question of how to identify for the programmer the place where a proof is failing. When a forward-chaining process fails to generate a desired formula, it is not clear where to place the blame. Explaining failure may require backward chaining from the goal.

6 Acknowledgment

I would like to thank David McAllester for his significant role in creating the paradigm for this research.

References

1. A. Aiken and E. Wimmers. Soft typing with conditional types. In *ACM Symposium on Principles of Programming Languges*, pages 163–173, 1994.
2. Alexander Aiken and Edward Wimmers. Type inference with set constraints. Research Report 8956, IBM, 1992.
3. J. A. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10:19–35, 1980.
4. R. L. Constable et. al. *Implementing Mathematics with the Nuprl Development system.* Prentice-Hall, 1986.
5. Robert Givan. *Automatically Inferring Properties of Computer Programs.* PhD thesis, Massachusetts Institute of Technology, 1996. http://www.ai.mit.edu-/people/rlg/papers/thesis.ps.
6. P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of ACM Conference on Principles of Programming Languages*, 1991.
7. D. McAllester and K. Arkoudas. Walther recursion. Submitted to CADE-13, available at http://www.ai.mit.edu/people/dam/termination.html, 1996.
8. D. McAllester and R. Givan. Taxonomic syntax for first order inference. *JACM*, 40(2):246–283, April 1993. internet file ftp.ai.mit.edu:/pub/users/dam/jacm1.ps.
9. J. McCarthy. A basis for a math. theory of computation. P. Braffort & D. Hirschberg, eds., *Computer Programing & Formal Systems.* North-Holland, 1967.
10. Robin Milner. Type polymorphism in programming. *JCSS*, 17:348–375, 1978.
11. John C. Mitchell. A type inference appproach to reduction properties and semantics of polymorphic expressions. In *Proceedings 1986 ACM Symposium on Lisp and Functional Programming*, pages 308–319, 1986.
12. P. Mishra and U. S. Reddy. Declaration-free type checking. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 7–21. ACM, 1985.
13. John C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation.* Springer-Verlag, 1974.
14. J.W. Thatcher. Tree automata: an informal survey. In A. V. Aho, editor, *Currents in Theory of Computation*, pages 143–172. Prentice-Hall, 1973.
15. W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B, Formal Methods and Semantics,* pp. 133-164. MIT Press, 1990.

This article was processed using the LaTeX macro package with LLNCS style