Automatically Inferring Properties of Computer Programs

by

Robert Lawrence Givan Jr.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

Author .....
Department of Electrical Engineering and Computer Science
May 3, 1996

Certified by .....
David Allen McAllester
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Automatically Inferring Properties of Computer Programs

by

Robert Lawrence Givan Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on May 3, 1996, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

This thesis presents two independent pieces of research.

First, 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 present such a type system, and use familiar syntax-directed type inference rules to give a polynomial-time procedure for inferring type theorems in this 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.

We present both first-order and higher-order versions of our sample programming language and inference algorithms for both languages. We believe most of the interesting inferential challenges are already present in the first-order case.

The second piece of research we present addresses the satisfiability of sets of formulas in a particular set constraints language. We consider set expressions built up from set constants by union, set complement, and taking the image of a set expression under a function or relation. Previous work in this area has neglected the "Tarskian" case where the functions and relations are allowed to take on arbitrary meanings, rather than only a standard Herbrand meaning. We prove that the satisfiability of a finite set of subset formulas between these "Tarskian" set expressions is in nondeterministic doubly exponential time. Our proof is by reduction to a new Diophantine inequation solvability problem, which we show to be in nondeterministic exponential time, but conjecture to be in $\mathcal{NP}$.

Thesis Supervisor: David Allen McAllester
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

As is customary, I would like to acknowledge my overwhelming indebtedness to the assistance provided to me by my thesis supervisor; in this case, David McAllester.

Having done this, I'd like to go above and beyond the customary, and further acknowledge David for sticking with me through the nine years I spent on the road to this thesis. He has set for me a stunning example of what can be accomplished by hard work and a brilliant mind. And he has maintained throughout a good humor and patience that made him an excellent teacher and mentor. Thank you, David.

I also want to acknowledge the assistance of my thesis readers John Guttag and Albert Meyer, each of whom met with me many times in an effort to help me write the best possible thesis.

John Guttag and Patrick Winston were responsible for magically coming up with the financial support that made my final thesis writing binge possible. Along the way, I also enjoyed generous financial support from the National Science Foundation and the Fannie and John Hertz Graduate Fellowship Program. Thanks also to Tomás Lozano Peréz for generously providing me his personal laptop on which much of my thesis was written.

My girlfriend Lindsay Haugland sacrificed my presence (for better or for worse) for most of 5 months while I was coming down the stretch, and provided the loving support needed for me to sustain my effort in writing. She spent many an hour listening to the woes of a stressed author.

Distinguished in their role in both personal support and technical assistance were my special friends Morris Dworkin and Sola Grantham. I could not have accomplished half of what I have without their help.

Finally, I must thank many of my peers, colleagues and fellow graduate students for everything from late night conversations to assistance in practice talks to reading drafts and making comments. This group includes Gary Borchardt, Jon Doyle, Mike Jones, Norm Margolis, Olin Shivers, Gerry Sussman, Peter Szolovits, and Ken Yip among many others.

Chapter 1

Introduction

This thesis presents two largely unrelated pieces of research. First, we present a natural and expressive type language along with a new polynomial-time type inference system that infers types in this language. Second, we show the decidability and bound the complexity of a previously unstudied problem in the area of set constraints. We discuss the first piece of work now, and will further introduce the set constraints work later in the thesis.

Many researchers have studied type inference systems for functional programming languages[34, 24, 35, 5]. 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[2]. 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 base our type system on the types present in programming languages, 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 (these algorithms are of course incomplete). We define in this thesis 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 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*[38, 14], 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.[37, 10, 41] 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 "merge-sort" correctly sorts its input.

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 thesis is organized as follows:

1. We discuss previous work on inference technology and inferring properties of programs.

2. We present a first-order version of our programming language and inference algorithm, starting with several examples of simple first-order programs and their automatically computed specifications and moving to a formal presentation of the language and algorithm.

3. We present a higher-order version of the same language, its syntax and semantics, and enhancements to the inference algorithm for dealing with the higher order features.

4. We present a further enhancement to the algorithm we call "automatic existential instantiation", separated from the rest because of its subtlety.

5. We introduce a new problem in the area of set constraints, resolve its decidability, and bound its complexity.

Chapter 2

Previous Type Inference Technology

2.0.1 Inference Engine Technology

Researchers have tried a variety of techniques to deal with the general intractability of automated inference. One approach has been to limit the expressiveness of the underlying logic to a tractable language. By limiting the expressive power available to the user, a system can force the user to represent his inference problem so that it can be solved quickly—of course, this may be difficult or impossible for the user to do. This approach can be very useful for solving problems which fall within the expressive power of the language. As an example of this approach, we consider concept languages ([9], [36], [39], [15]). A concept language is a language of expressions intended to denote sets. The primitive atomic formula of a concept language is the subset, or subsumption relation. By restricting the concept formation operators in the language, the subsumption question for the language (that is, the question of whether a given subsumption formula follows from some others) can be made tractable. Many concept languages have fast complete inference procedures for recognizing valid taxonomic consequences.

The formulas of our programming language can be viewed as a concept language, but the concept formation operators of our language are too expressive to have a

fast complete inference mechanism. Instead, we provide fast incomplete automated inference, and rely on user interaction with a complete proof system for completeness.

Another approach to the intractability problem is to allow the inference procedure to run arbitrarily long, or even forever, for some problems, while allowing the user to interrupt the procedure. The procedure is only of use in those cases where it terminates quickly enough, and in practice this is similar (though perhaps less predictable) to having a fast but incomplete decision procedure such as that we provide.

One system which takes this approach is the NQTHM inference system ([8]), based on a logic of pure LISP with recursive definitions and recursive DEFSTRUCT (the "shell" principle). The inference engine is a term rewriting system with much heuristic knowledge about LISP and induction built in. Inference is not guaranteed to terminate, and termination may depend on the order or the exact statement of the available lemmas. Because of the logic's close connection to LISP and the built-in heuristic knowledge about LISP, NQTHM is ideally suited for the verification of properties of LISP programs. However, it has also been used very successfully to verify abstract mathematical theorems, with some extra effort and awkwardness needed to represent the mathematics in LISP. Many of NQTHM's heuristics could be added to our system for reasoning about the subset of our language which corresponds to the NQTHM logic. Also, adding a Kleene closure relational operator to our reasoning mechanisms, as described in [30] could make it possible to derive in a general-purpose forward-chaining manner many of the properties of recursive definitions which are computed heuristically by NQTHM.

An entire family of inference systems are the descendants of the LCF system ([19], [42], [18]). These systems provide a complete set of simple inference rules, and essentially require that the user provide a full formal proof of his theorem. To assist him in this task, the user is provided with a "tactic" language—essentially, a language for writing programs that generate proofs. By building up a library of useful (and often domain-specific) tactics, the user can avoid the arduous detail of full formal proofs. It is possible to build a forward chaining notion of obviousness like that in our inference system into an LCF-based system as a primitive inference

rule (or built-in tactic)([13]), however, most LCF systems have been less ambitious than our system in providing aggressive automatic inference—relying instead upon user written tactics and proofs.

Many systems give the user some procedural control over the inference process which can help it solve a problem faster. For example, term rewriting based inference systems ([8], [42]) allow the user to control which lemmas are used in which directions as rewrite rules. By carefully choosing these rewrite rules, the user can dramatically affect the effectiveness of the inference procedure. Similarly, the order of the clauses and the choice of resolution strategy employed can be used to control the effectiveness of a resolution based inference system. Because the user has this procedural control, these systems also provide a means of analyzing the inference process to understand "where it went wrong." This helps the user determine what procedural fixes need to be made. It is unfortunate that such procedural assistance to the inference engine appears to be a practical necessity—the most generally useful inference systems today are written in this style. Our inference system provides a notion of obvious consequence which takes no procedural information from the user—our hope is to demonstrate that this can be made practical.

2.0.2 Type Checking and Type Evaluation in Other Automated Inference Systems

Many automated inference systems perform some form of type checking on the expressions in their input. Most systems differ from our system in that the underlying syntax does not collapse the term/type distinction. Therefore, many systems have a completely separate type language, often providing a means for defining new types using predicates. Type checking in these systems can still be viewed as a means of organizing general purpose inference, since a type obligation can be converted into a predicate to be verified by whatever means are available. However, previous type checking systems have tended to emphasize ensuring the well-formedness of the expressions being checked rather than using the type checking as a framework for

organizing powerful general purpose inference procedures like the forward-chaining procedures we present here. Our type inference system returns a potentially rich set of types for the expression being checked, as well as ensuring its well-formedness.

A number of automated inference systems have been written based on the ML type system [33]. This type system consists of primitive types, type variables, and function spaces between types. Any type expression containing type variables represents the intersection of all the types that could be obtained by replacing the type variables by type expressions without variables. The use of type variables allows the ML system to represent polymorphic functions such as the identity function: the ML type of the identity function would be the function space between a particular type variable and itself. This type system is much less expressive than our type system —for example, it cannot represent a type whose instances are sorted lists. This lack of expressiveness can be mitigated by allowing the user to define new primitive types using arbitrary predicate expressions, but this method is relatively awkward and has not to our knowledge been used to organize general-purpose inference around types as intended in our system.

In ML-based systems, unlike our system, each syntactic expression has a single correct type expression (possibly containing type variables). This type expression is automatically computed by the ML system by using pre-existing knowledge about the types of the constants already defined, along with unification, to pick the appropriate instances of any polymorphic functions involved as well as the appropriate types for the undeclared variables. This approach allows the ML user to have the benefits of a strongly typed language without having to specify types for his variables in most cases. Our system, by contrast, requires every variable to have a declared type—this is necessary because in our richer type system, there is no single distinguished type for each of the defined constants. Because our type system is very expressive, it is generally not difficult to identify an appropriate, but not limiting, type for each variable in a program.

Other automated inference systems have implemented algorithms with some similarity to our algorithm for analyzing recursive definitions. The Boyer-Moore theorem

proving system [8] uses a very limited type system in which every semantic value must be a member of one of a finite collection of types. When type-checking a new definition, the system computes a *type set* for the defining expression: a subset of the finite collection of types such that under any interpretation of the variables in the defining expression, the expression must take on a value within a type in the subset. Given a recursive definition, the Boyer-Moore system computes a fixed-point type set (one such that the definition can be seen to have that type set by assuming that recursive applications of the function have that type set) by repeatedly type-checking the definition with ever larger type sets assumed for the recursive calls. Because there is a maximal finite type set which is always a fixed point, this process always terminates with a fixed point, and in practice often finds a useful fixed point type set.

The recursive definition type inference for our language described below must deal with the difficulty that our type system is significantly richer than that of the Boyer-Moore prover. The primary means we use to deal with this richness is by relying on the forward-chaining inference technology to produce useful and finite answers quickly. This technology essentially limits the set of types being considered to the finite set of types which have been mentioned in the proof, enriched in various limited ways (primarily by instantiation of known theorems), ensuring termination. So our type inference for recursive definitions differs from that in Boyer-Moore in that it allows a much wider variety of types in the finite set of types being considered.

Both our type inference algorithm for recursive definition, and that just described for Boyer-Moore can be viewed as instances of a general abstract interpretation algorithm for evaluating recursive definitions described in [1]. This general algorithm captures the idea of dealing with recursion by computing a fixed-point which can be proven by induction.

Chapter 3

First Order Specification Inference

3.1 Some Examples of Quickly Verifiable Specifications

We begin with an informal discussion of our programming and specification languages, and then discuss some examples of simple programs and the specifications our algorithm can automatically compute for them. Later sections will contain a formal syntax and semantics of our programming language and the specification language, as well as a complete description of the algorithm. Readers who have trouble understanding these examples informally are encouraged to return to them after scanning the later sections.

3.1.1 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

[1] We omit first-class functions only for simplicity here. We show how to extend this work naturally to higher-order languages in Chapter 4.

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 contains constructor symbols (e.g. `cons`) and corresponding selectors (`car` and `cdr`), and has the intended semantics that each program expression denotes some term in the Herbrand closure over the constructor symbols (or bottom).

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.

Throughout the remainder of this section we exhibit computed specs for three example programs. Our purpose here is to demonstrate some of the usefulness of our algorithm and specification language—later in this chapter we will discuss *how* these specs are computed. The user-provided definitions of the specification functions (e.g. `(a-number)`) in these examples are shown and explained just 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 both `x` and `y` (i.e., meets the specs `(`$\geq$ `x) and (`$\geq$ `y)`).

```
(define (+ (x (a-number))
           (y (a-number)))
  (if x:'nil
      y
      (cons 'a (+ (cdr x) 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)` returns a permutation of `(cons x l)`, i.e., meets the spec `(a-permutation-of (cons x l))` and that `(sort l)` always returns a permutation of the list `l`.[2]

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

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

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

[2] 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.

3.1.2 Example Specifications

The specification language is less familiar. This language is essentially our programming language extended by a nondeterministic `either` combinator.[32, 28] Expressions in the specification language can take on more than one possible value. This nondeterminism may make some readers uncomfortable: we are simply using it to provide expressions that define new types. In particular, the set of possible values of a specification expression can be viewed as the type defined by that expression, and we call it the set denoted by the specification.

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 can 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`, respectively). Finally, we add a universal spec `(a-thing)` that nondeterministically returns any value at all, and an empty spec $\perp$ that has 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 dealing with it efficiently marks an important contribution of this work. The examples described in this section centrally involve dependent types.

Below we exhibit 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)))))
```

The specification `(samelength-as l)` is a dependent type, depending on the value of the variable `l`. Given a list `l`, the specification `(samelength-as l)` nondeterministically denotes any list of the same length as `l`.

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

The specification `(a-list-member-of l)` nondeterministically denotes any member of the argument list `l`.

```
(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))))))
```

The specification `(a-permuation-of l)` nondeterministically denotes an arbitrary permutation of the input list `l`. The specification language is powerful enough to allow a concise and natural definition of this concept.

```
(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))))

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

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

These remaining definitions are self-explanatory.

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. By abuse of notation, we can also say that a spec t *satisfies* another spec s, written t:s, if every value taken by 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.

We return to these examples after formally defining our language and inference mechanisms.

3.2 A Programming Language with Specifications

We give here a more formal treatment of our demonstration language.

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.[3]

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

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

where f can now be any n-ary constructor or selector, or n-ary program *or specification* function-symbol. Note that every program expression is also a specification expression.[4]

Programs . We consider a sequence of function-symbols definitions to be a program. A function-symbol definition assigns to a new function-symbol either (`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. This distinction is important in our inference system as it is often important to know syntactically that an expression denotes exactly one value, as is the case for pure program expressions.

We must restrict recursive definitions to ensure that every `fix` expression accepted

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

[4] 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.

has a well-defined least fixed point. For this language it suffices to prohibit recursive calls in positions that are not *syntactically monotone*—this excludes 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*[5]. Checking termination syntactically is a deep problem itself[26], but here we settle for simply requiring that there be some argument to the function whose Herbrand size is reduced (by selector application) in all the recursive calls (this restriction is the variant of primitive recursion[?] appropriate for recursion on the structure of Herbrand terms).

We also define some convenient abbreviations. We use (`both` s_1 s_2) to abbreviate (`not (either (not` s_1`) (not` s_2`)))`, and (`a-thing`) to abbreviate (`not` $\perp$). We frequently want to select the values of an expression which satisfy some formula, and so define the abbreviation (`some-such-that` x s Φ) to stand for (`let` x`:`s (`if` Φ `x` $\perp$)). We can then use this abbreviation to define `forall` $x:s$ Φ to abbreviate s : (`some-such-that` x s Φ).

Semantics . Our semantic domain is the Herbrand closure over the constructor functions, with an error (ϵ) element adjoined (note that the closure is taken *before* the error element is adjoined). When a function is applied to objects outside its domain (as indicated by the specs on its formal parameters) it returns ϵ. Each specification expression denotes a subset of the domain. If the specification expression is a program expression the subset will contain only one element, and one can think of the program expression as denoting that element. We call sets containing only one domain element *program values*.

Assigning meanings to the various expressions compositionally is routine; we discuss the unusual cases in the meaning of specifications before presenting the full semantics formally. Because a specification denotes a *set* of objects, the meaning of function application may not be obvious: we apply the function pointwise; i.e., to ap-

[5] We will remove this restriction in the higher-order version of the language.

ply 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 to them, with many possible results. The `s1:s2` test of an `if` expression is true exactly when `s1` denotes a subset of `s2`'s denotation. $\perp$ denotes the empty set. `Either` and `not` are computed with union and set complement relative to the domain, respectively.

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.

The meaning $\mathcal{M}_\rho(e)$ of each language expression e within the Herbrand domain $\mathcal{M}$ is determined relative to a variable assignment ρ that maps variables to members of $\mathcal{M}$ and function symbols to their defining relations over $\mathcal{M}$, always mapping constructor functions to their standard interpretation over the Herbrand domain. Note that although function symbols have meanings that are relations over the Herbrand domain the language is not higher order because these relations are always immediately applied, never passed around as first-class objects). $\mathcal{M}_\rho(e)$ is defined as follows:

- $\mathcal{M}_\rho(x) = \{\rho(x)\}$ for variable or function symbol x,
- $\mathcal{M}_\rho(\texttt{let } x\texttt{:}s\ s_1) = \bigcup_{d \in \mathcal{M}_\rho(s)} \mathcal{M}_{\rho[x:=d]}(s_1)$
- $\mathcal{M}_\rho(\texttt{(apply } f\ s_1 \ldots s_n\texttt{)}) = \left\{ x \;\middle|\; \begin{array}{c} \exists s_1 \in \mathcal{M}_\rho(s_1) \cdots s_n \in \mathcal{M}_\rho(s_n) \\ \langle s_1, \cdots, s_n, x \rangle \in \mathcal{M}_\rho(f) \end{array} \right\}$
- $\mathcal{M}_\rho(\texttt{(if } s_1 : s_2\ s_3\ s_4\texttt{)}) = \left\{ \begin{array}{ll} \mathcal{M}_\rho(s_3) & \text{when } \mathcal{M}_\rho(s_1) \subseteq \mathcal{M}_\rho(s_2) \\ \mathcal{M}_\rho(s_4) & \text{otherwise,} \end{array} \right\}$
- $\mathcal{M}_\rho(\texttt{(either } s_1\ s_2\texttt{)}) = \mathcal{M}_\rho(s_1) \cup \mathcal{M}_\rho(s_2)$
- $\mathcal{M}_\rho(\texttt{(not } s\texttt{)}) = \mathcal{M} - \mathcal{M}_\rho(s)$
- $\mathcal{M}_\rho(\perp) = \{\}$

- $\mathcal{M}_\rho\,(\texttt{(lambda}\ x_1\texttt{:}s_1 \ldots x_n\texttt{:}s_n\ B\texttt{)}) = \{\langle d_1, \ldots, d_n, \mathcal{M}_\rho\,([d_i/x_i]\,B)\rangle \,|\, d_i \in \mathcal{M}\}$
  note that the s_i do not affect the meaning.

- $\mathcal{M}_\rho\,(\texttt{(fix}\ f\ x_1\texttt{:}s_1 \ldots x_n\texttt{:}s_n\ B\texttt{)}) = \bigcup_i \mathrm{Next}^i\,(\texttt{(lambda}\ x_1\texttt{:}s_1 \ldots x_n\texttt{:}s_n\ \bot\texttt{)})$
  where $\mathrm{Next}\,(d) = \mathcal{M}_{\rho[f:=d]}\,(\texttt{(lambda}\ x_1\texttt{:}s_1 \ldots x_n\texttt{:}s_n\ B\texttt{)})$

- $\mathcal{M}_\rho\,(c) = \{\,\langle d_1, \cdots, d_n, c(d_1, \cdots, d_n)\rangle \,|\, \text{each } d_i \in \mathcal{M} - \{\epsilon\}\,\}$
  $\cup\ \{\,\langle d_1, \cdots, d_n, \epsilon\rangle \,|\, \text{each } d_i \in \mathcal{M} \text{ and some } d_i \text{ is } \epsilon\,\}$
  for constructor function symbol c of arity n,

- $\mathcal{M}_\rho\,(s_j) = \{\,\langle c(d_1, \ldots, d_n), d_j\rangle \,|\, d_i \in \mathcal{M} - \{\epsilon\}\,\}$
  $\cup\ \{\,\langle d, \epsilon\rangle \,|\, d \in \mathcal{M} \wedge \neg\exists\, d_1 \cdots d_n.d = c(d_1, \ldots, d_n)\,\}$
  where s_j is the j'th selector for n-ary constructor c.

In the above definition $\rho[x := d]$ is the variable assignment ρ modified just at x to give the value d, $[s/x]\,e$ stands for the replacement of all free occurrences of x in e by s, and $\texttt{Next}^i\,(d)$ denotes i applications of `Next` to d.

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 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.

[6] The application of a specification function to a non-program expression is not directly computable even if all the functions involved have attachments.

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

The programmer can avoid attachment with no loss in effectiveness or clarity. Only the type inferences that follow from the `if` test in question are lost.

```
(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
```

3.3 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 specification formulas $s : t$. Most useful formulas from the library will have the more restricted form `forall` $x_1{:}s_1 \cdots x_n{:}s_n$. $s{:}t$—where s, t, and the s_i contain no occurrences of `if` or `let` (remember that `forall` is an abbreviation for a specification formula). We call such formulas *type theorems*.

All the facts concluded automatically by our algorithm will be type theorems. The forward-chaining reasoning process that uses facts from the library does not reason

about `let` or `if` expressions in theorems, so facts from the library that are not type theorems will often be of limited value.

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 $\mathcal{L}$.

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, another forward chaining analysis ($\overset{\text{DA}}{\vdash}$) which generates useful special cases of the beta-reduction/abstraction facts for the new definition; and third, a syntax-directed, type-inference inspired inference relation ($\vdash\!\!\circ_e$) that manages the application of the two forward-chaining closures. Our solution to the specification inference problem is to add to $\mathcal{L}$ those formulas Φ such that $\mathcal{L} \vdash\!\!\circ\ \Phi$, except that we replace each occurrence of e in Φ with the newly defined symbol g. We now consider each of the three parts of the algorithm in turn.

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".

In the following rules, p and q are meta-variables standing for any program expressions; r, s and t are meta-variables standing for any specification expressions; f

can be any function symbol, constructor or selector; c is any constructor; the selector rules are shown for `cons`, `car`, and `cdr`.

The following rules define $\vdash_e$:

Sym

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

Trans

$$\frac{r{:}s \qquad s{:}t}{r{:}t}$$

Either1

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

Either2

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

Not-Sym

$$\frac{\texttt{Dom}\ (r) \qquad r{:}\texttt{(not } s\texttt{)}}{s{:}\texttt{(not } r\texttt{)}}$$

Under-Both

$$\frac{\texttt{Dom}\ \texttt{((both } s\ t\texttt{))} \qquad r{:}s,\ r{:}t}{r{:}\texttt{(both } s\ t\texttt{)}}$$

Basic-Either

$$\frac{\texttt{Dom}\ \texttt{((either } r\ s\texttt{))}}{r{:}\texttt{(either } r\ s\texttt{)} \qquad s{:}\texttt{(either } r\ s\texttt{)}}$$

Basic-Both

$$\frac{\texttt{Dom}\ \texttt{((both } r\ s\texttt{))}}{\texttt{(both } r\ s\texttt{)}{:}r \qquad \texttt{(both } r\ s\texttt{)}{:}s}$$

Strictness

$$\frac{\texttt{Dom}\ \texttt{((}f\ s_1 \ldots s_n\texttt{))} \qquad \text{Some } s_i{:}\bot}{\texttt{(}f\ s_1 \ldots s_n\texttt{)}{:}\bot}$$

Always

$$\frac{\texttt{Dom}\ (r)}{r{:}r,\ \bot{:}r \qquad r{:}\texttt{(a-thing)}}$$

Selectors1

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

Selectors2

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

Monotonicity

$$\frac{s_1{:}t_1 \ldots s_n{:}t_n \qquad r{:}\texttt{(}f\ s_1\ \ldots s_n\texttt{)} \qquad \texttt{Dom}\ \texttt{((}f\ t_1 \ldots t_n\texttt{))}}{r{:}\texttt{(}f\ t_1\ \ldots t_n\texttt{)}}$$

Constructors

$$\frac{\texttt{Dom}\ \texttt{((}c_1\ s_1 \ldots s_n\texttt{))} \qquad \texttt{Dom}\ \texttt{((}c2\ t_1 \ldots t_m\texttt{))} \qquad c_1 \neq c_2}{\texttt{(}c_1\ s_1 \ldots s_n\texttt{)}\ :\ \texttt{(not (}c_2\ t_1 \ldots t_m\texttt{))}}$$

Dom-Always

`Dom` (`(a-thing)`)

`Dom` ($\bot$)

Dom-Start

s appears in e

`Dom` (s)

Univ-Dom

forall x:s Φ

`Dom` (s)

Univ-Inst

forall x:s Φ

p:s, where p appears in e

$[p/x]\,\Phi$

`Dom` ($[p/x]\,\Phi$)

Dom-Subexp

`Dom` (r)

s a subexpression of r

`Dom` (s)

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. 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 entire forward-chaining can be computed in polynomial-time in the eventual 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 $\bot$, `(a-thing)`, a subexpression of e, or some subexpression of a universal formula `forall` $x_1 : s_1 \ldots x_n : s_n$ Φ in Σ with its variables replaced by subexpressions of e. Only the last case poses a challenge, forcing us to limit the quantification depth of the formulas in Σ to some constant. Specifically, we require that no formula in Σ has a depth of `let` nesting greater than some fixed constant (remember that `forall` is an abbreviation for a `let` expression). Given this restriction 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 $\mathcal{A}$. But there are only polynomially many possible conclusions over such specs, and for any partially closed premise set we can find a new consequence in polynomial time.

Theorem 3.1 *For any premise set Σ with bounded* `let` *nesting, and function symbol definition e, we can compute the forward-chaining closure of Σ under the rules defining $\vdash_e$ 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.

Definitional Theorems Inferred by $\overset{\text{DA}}{\vdash}_s$ We define here a second forward chaining inference relation $\overset{\text{DA}}{\vdash}_s$ which generates some simple definitional axioms from each `lambda` or `fix` expression s encountered. $\overset{\text{DA}}{\vdash}$ is defined by the following inference rules.

Start-DA

s is `(lambda` $x_1{:}s_1 \cdots x_n{:}s_n$ B`)`
or `(fix` g `(lambda` $x_1{:}s_1 \cdots x_n{:}s_n$ B`))`

$\overset{\text{DA}}{\vdash}_s$ `Forall` $x_1{:}s_1 \cdots x_n{:}s_n$ B`:(`s $x_1 \ldots x_n$`)`

[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.

Either-DA

$$\frac{\overset{DA}{\vdash_s}\ \texttt{Forall}\ x_1{:}s_1 \cdots x_n{:}s_n\ \texttt{(either}\ B_1\ B_2\texttt{)}{:}t}{\overset{DA}{\vdash_s}\ \texttt{Forall}\ x_1{:}s_1 \cdots x_n{:}s_n\ B_i{:}t\texttt{,}\ i = 1\ \texttt{or}\ 2}$$

Let-DA

$$\frac{\overset{DA}{\vdash_s}\ \texttt{Forall}\ x_1{:}s_1 \cdots x_n{:}s_n\ \texttt{(let}\ x{:}s\ B\texttt{)}{:}t}{\overset{DA}{\vdash_s}\ \texttt{Forall}\ x_1{:}s_1 \cdots x_n{:}s_n x{:}s\ B{:}t}$$

If-DA

$$\frac{\begin{array}{l}\overset{DA}{\vdash_s}\ \texttt{Forall}\ x_1{:}s_1 \cdots x_n{:}s_n\ \texttt{(if}\ y_i : s\ B_1\ B_2\texttt{)}{:}t \\ y_1{:}t_1 \cdots y_n{:}t_n \text{ is a suitable reordering of } x_1{:}s_1 \cdots x_n{:}s_n\end{array}}{\begin{array}{l}\overset{DA}{\vdash_s}\ \texttt{Forall}\ y_1{:}t_1 \cdots y_i{:}\texttt{(both}\ t_i\ s\texttt{)} \cdots y_n{:}t_n \\ \qquad B_1{:}t \\ \overset{DA}{\vdash_s}\ \texttt{Forall}\ y_1{:}t_1 \cdots y_i{:}\texttt{(both}\ t_i\ \texttt{(not}\ s\texttt{))} \cdots y_n{:}t_n \\ \qquad B_2{:}t\end{array}}$$

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. These rules are intended to be used in a forward-chaining manner, triggered by Start-DA whenever a `fix` or `lambda` expression is encountered during the recursive descent described in the next subsection.

The Syntax-Directed Inference Relation $\vdash\!\circ_e$ We now use the $\vdash_e$ and $\overset{DA}{\vdash}$ relations 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. In these rules, r, s, t, u, u_1 and u_2 are meta-variables matching any specification expressions. *Neg* is

discussed in the text below. The Analyze-Fix and Analyze-Lambda rules are shown for one-argument expressions, but the analogous rules for arbitrary arity are intended. We use the expression $THMS_\Sigma\,(s)$ to abbreviate the set of all specification formulas "about s" provable from Σ using $\vdash_\circ$. A specification formula Φ is "about s" if Φ is of the form $s : t$ for some t, or if s is a `lambda` or `fix` expression and Φ is `forall` $x_1 : u_1 \ldots x_n : u_n$ `(`$s\ x_1 \ldots x_n$`)` $: t$ for some specifications t and $u_1 \cdots u_n$, and variables $x_1 \cdots x_n$ not appearing in s, where n is the arity of s.

Analyze-If

$$\frac{\begin{array}{l}\Gamma = \Sigma \cup THMS_\Sigma\,(r) \cup THMS_\Sigma\,(s) \\ \Gamma,\ r : s \quad \vdash_{e}\ u_1 : t \\ \Gamma,\ Neg\,(r{:}s) \quad \vdash_{e}\ u_2 : t\end{array}}{\Sigma \vdash_{e} \texttt{(if}\ r : s\ u_1\ u_2\texttt{)} : t}$$

Analyze-Lambda

$$\frac{\begin{array}{l}\Gamma = \Sigma \cup THMS_\Sigma\,(r) \\ \Gamma,\ x : r \vdash_{e} B : t \\ x \text{ and } x_1 \text{ not in } \Gamma,\ x_1 \text{ not in } B\end{array}}{\Sigma \vdash_{e} \begin{array}{l}\texttt{Forall}\ x_1 : r \\ \quad \texttt{((lambda}\ x : r\ B\texttt{)}\ x_1\texttt{)} : [x_1/x]\,t\end{array}}$$

Analyze-Let

$$\frac{\begin{array}{l}\Gamma = \Sigma \cup THMS_\Sigma\,(r) \\ \Gamma,\ x : r \vdash_{e} s : t \\ x \text{ not in } \Sigma \text{ or } t\end{array}}{\Sigma \vdash_{e} \texttt{(let}\ x : r\ s\texttt{)} : t}$$

Analyze-Fix

$$\frac{\begin{array}{l}\Gamma = \Sigma \cup THMS_\Sigma\,(r) \\ \Gamma,\ \begin{array}{l}\texttt{Forall}\ x_1 : r \\ \quad (f\ x_1) : [x_1/x]\,I\end{array}\ ,\ x : r \vdash_{e} B : I \\ x \text{ and } x_1 \text{ not in } \Gamma,\ x_1 \text{ not in } B\end{array}}{\Sigma \vdash_{e} \begin{array}{l}\texttt{Forall}\ x_1 : r \\ \quad \texttt{((fix}\ f\ x : r\ B\texttt{)}\ x_1\texttt{)} : [x_1/x]\,I\end{array}}$$

Analyze-Apply

$$\frac{\begin{array}{l}\Gamma = \Sigma \cup THMS_\Sigma\,(r) \cup THMS_\Sigma\,(s) \\ \Gamma \vdash_{e} \texttt{(apply}\ r\ s\texttt{)} : t\end{array}}{\Sigma \vdash_{e} \texttt{(apply}\ r\ s\texttt{)} : t}$$

Analyze-DA

$$\frac{\begin{array}{l}\overset{\text{DA}}{\vdash}_{s}\ \Phi \\ s \text{ is a } \texttt{lambda} \text{ or } \texttt{fix} \text{ expression in } e\end{array}}{\vdash_{e} \Phi}$$

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-Apply is the only place where the forward-chaining $\vdash_e$ relation is used. The Analyze-Lambda, Analyze-Fix, and Analyze-DA rules are needed only at the top level of function symbol definitions (because that is the only place `lambda` and `fix` occur—this restriction will be relaxed in the higher-order version of the language). We describe just below how induction hypotheses for the Analyze-Fix are selected.

We can now formally define the use of $\vdash^{\circ}_e$ to generate theorems about a new definition. Say we wish new theorems about the function symbol g, defined as the `lambda` or `fix` expression e. If e is a `lambda` expression `(lambda` $x_1{:}s_1 \ldots x_n{:}s_n\ B)$, we simply return the set of theorems provable by $\vdash^{\circ}_e$ of the form `Forall` $x_1{:}s_1 \ldots x_n{:}s_n\ (e\ x_1 \ldots x_n){:}t$. Note that t can contain the formal parameter variables x_i, giving us the important ability to discover dependent types. For recursive definitions $e =$ `(fix` $g\ x_1{:}s_1 \ldots x_n{:}s_n\ B)$, we prove type specs about e by induction. To facilitate describing this process we define the operation $\mathcal{T}$ on sets of specifications Σ as follows:

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

where $\mathcal{B}(\Sigma)$ is the `both` expression representing the intersection of all the members of Σ (remember that $\mathcal{L}$ is the library of previously derived theorems known to the system before encountering the new definition). Intuitively, $\mathcal{T}(\Sigma)$ is the set of types that can be proven for the body B under the inductive assumption that recursive calls satisfy all the specs in Σ (note that some of these specs may depend on the formal parameters x_i).

In order to apply the Analyze-Fix rule, we need to find a self-supporting induction hypothesis—that is, a set of specifications Σ such that $\mathcal{T}(\Sigma) = \Sigma$. We find this fixed-point of $\mathcal{T}$ by computing a decreasing sequence of candidate sets $\Sigma_0, \Sigma_1, \ldots$, until we reach a fixed-point or an empty set, as follows:

$\Sigma_0 = \mathcal{T}(\{\bot\})$

$\Sigma_{i+1} = \mathcal{T}(\Sigma_i) \cap \Sigma_i$

Once we find a fixed-point Σ_{fix} of $\mathcal{T}$ this way, we return the set of theorems

$$\left\{ \begin{array}{c} \texttt{Forall } x_1 : s_1 \ldots x_n : s_n \\ (g\ x_1 \ldots x_n) : t \end{array} \middle| \; t \in \Sigma_{\text{fix}} \right\}$$

Intuitively, we have started by analyzing the "base case" of the definition by seeing what types we can prove for B under the assumption that recursive calls diverge. We then take those types together as an inductive hypothesis about recursive calls and see what subset of them can be proven about B. We repeat this step, each time shrinking the inductive hypothesis until it reaches a fixed point. The resulting set of types, found in polynomial time in the size of e and $\mathcal{L}$[8], is the largest *self-justifying* set of types for g in $\mathcal{T}(\bot)$. Note that this approach to recursive definitions can be easily and naturally extended to sets of mutually recursive definitions.

3.3.1 Inferring the Specifications in Our Examples

We now return to our example programs from Sect. 3.1, and explore the steps involved in computing the claimed specifications for the programs.

The theorems shown below are among those generated automatically when reading the specification definitions shown in Sect. 3.1. These and others like them are used in calculating the specifications cited for the example programs. The first six theorems are examples of theorems produced by the $\overset{\text{DA}}{\vdash}$ inference relation

```
(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))
```

[8] In practice, the runtime should be polynomial in the size of the *relevant* part of the library, that is those lemmas whose type restrictions apply to the expressions in the definition.

```
      (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) forall l:(a-numlist) l:(a-list)
(9) (a-numlist):(either 'nil (cons (a-number) (a-list)))
(A) (either 'nil (cons (a-number) (a-list))):(a-numlist)
```

Each example also requires the presence of some additional simple and natural theorems. 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 the theorems. Each theorem captures a basic property of the definitions, rather than a property targeted to any of our examples.[9] Many of the theorems can be proven by stating them to a simple inductive theorem prover. The theorems needed are:

```
forall l:(a-list)
  l:(samelength-as l)

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

forall x:(a-number)
       y:(not (> x))
  x:(>= y)
```

[9] 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 Chapter 5.

```
forall n:(a-number)
  (>= (>= n)):(>= n)
forall n:(a-number) (cons 'a (>= n)):(>= (cons 'a n))
forall l:(a-list)
  (a-permutation-of (a-permutation-of l)):
  (a-permutation-of l)
```

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 states that any list is a permutation of itself—and a simple inference chain 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 below. Similar chains of reasoning are involved in automatically drawing the other specification conclusions cited above.

```
(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.

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.

Chapter 4

Higher Order Specification Inference

We consider here extensions to the first-order language presented in Chap. 3 to a general purpose higher-order language. We then extend the inference mechanisms presented to handle the new language features. We believe that the most difficult inference problems are present already in the first-order case. The techniques presented here are straightforward extensions of the first-order specification inference techniques—this chapter is provided not only to exhibit the extension, but also as an opportunity to present a higher-order specification language based on nondeterminism. The specification language given here uses nondeterminism to achieve the expressive power of Zermelo-Fraenkel set theory and allows particularly elegant expression of a wide range of mathematical concepts.

4.1 A Higher-Order Programming Language

Program Expressions We add new constructs to our first-order programming language to allow first-class function values. First, we allow `lambda` expressions of zero or one argument as program expressions—we call `lambda` expressions of no arguments *thunk* expressions, and we intend functions of more than one argument to be represented in a Curried manner. Second, we allow first-class recursive functions

by including as program expressions `fix` expressions. The new program expression BNF is given below.

$$e ::= x \mid (\texttt{apply}\ e) \mid (\texttt{apply}\ f\ e) \mid (c\ e_1 \cdots e_n) \mid (\texttt{if}\ \phi\ e_1\ e_2) \mid$$
$$\mid (\texttt{fix}\ g\ e^*) \mid (\texttt{lambda}\ ()\ e) \mid (\texttt{lambda}\ x{:}s\ e)$$

where f can be either any program expression or any selector function symbol, c is an n-ary constructor or selector symbol, g and x are variables, s is a specification expression (defined below), and ϕ is a testable formula (defined below). We often write a quoted symbol as an abbreviation for the application of a 0-ary constructor symbol, and we often omit the constructor `apply`.

We restrict the body e^* of `fix` expressions to be a lambda expression of k arguments with a body $B[g]$ (when un-Curried) such that every occurrence of g in B is syntactically monotone and continuous as an operator of k arguments. This property is defined below in the context of the specification language.

Specification Expressions To complement the higher-order features added to the programming language, we add related features to the specification language. As before, we allow all program features in specifications. A `lambda` specification is always deterministic (i.e., has just one possible value), however, its body may be nondeterministic, in which case it denotes a nondeterministic function (i.e., a relation). In addition, we add operator and thunk types corresponding to partial, nondeterministic versions of the traditional arrow types. We also add a new construct `a-domain-member-of` which nondeterministically returns a member of the domain of its argument—this construct yields no value on non-operator arguments. Figure 4-1 gives the new BNF for specification expressions.

We must restrict recursive definitions to ensure that every `fix` expression accepted has a well-defined least fixed point. It suffices to prohibit recursive calls in positions that are not *syntactically monotone and continuous*. We explain the semantic motivation behind this restriction below in the section on the semantics for our higher-order language. Here, we just state the syntactic restriction. First, we require the body

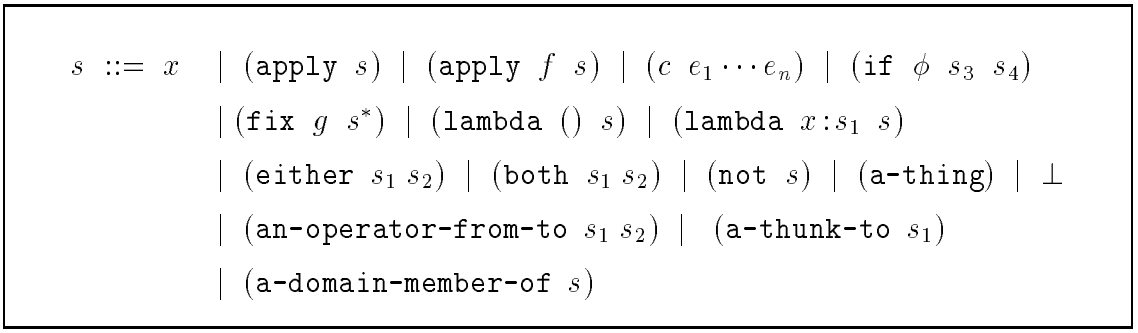

Figure 4-1: **Specifications BNF.** ϕ is a formula (defined below), x and g are variables, s^* is syntactically monotone and continuous, and f can be any specification expression or selector function symbol, and c is any n-ary constructor.

of every `fix` expression to be a `lambda` expression. An expression `(fix g (lambda x1`:$e_1 \ldots$`xn`:e_n C`[g]))` is syntactically monotone and continuous if g does not occur in any e_i, and every occurrence of g in C satisfies the following restrictions:

- g occurs only in applications to n argument expressions,
- g does not occur inside any of the following constructs:
  - any `fix` expression,
  - any `lambda` expression, or
  - any formula.

Formulas We also extend the simple first-order language by adding formula constructs to increase the effectiveness of the inference rules we give later (`there-exists` and `at-most-one`) [28]. We also add boolean combinations of specification formulas for convenience (`if` expressions can already be used to express any desired combination). The formula constructs `there-exists` and `at-most-one` assert that a specification is non-empty, or deterministic, respectively. The addition of `small` is necessitated by the richer semantics necessitated by the addition of operators and thunks as well as our intent to capture the expressiveness of ZF. The intended semantics of `small` is discussed in section 4.2. The new BNF for formulas is given

below:

$$\phi ::= s_1 : s_2 \mid (\texttt{boolean-not}\ \phi_1) \mid (\texttt{and}\ \phi_1\ \phi_2) \mid (\texttt{small}\ s)$$
$$\mid (\texttt{there-exists}\ s) \mid (\texttt{at-most-one}\ s)$$

An *atomic formula* is a formula with no occurrences of `and` or `not`. A *literal* is an atomic formula or its negation.

As in our first-order language, we must ensure that formulas appearing in program expressions (as opposed to specification expressions) can be evaluated deterministically. In Sect. 3 we defined what it meant for a first-order specification to be *testable*. Here we extend that definition to the formulas of our higher-order language. The following formulas are testable:

- the formula $s{:}t$ where both s and t are program expressions, or
- the formula $e : (f\ e_1 \cdots e_n)$ for program expressions $e, e_1 \cdots e_n$, and implemented function symbol f— A function symbol is implemented if it has been given a proven implementation attachment as described in section 3.2, or
- the formula is e: `(some-such-that` x `(a-thing)` Φ`)` where Φ is testable, or
- Any boolean combination of testable formulas.

Abbreviations There are several useful abbreviations that we define to extend the above language. Actual implementations may profitably choose to directly implement and reason about these constructs rather than translate them into the above primitives.

We observe that there is a natural isomorphism between nondeterministic thunks and sets (mapping each thunk to the set containing exactly its values when applied). We can thus use thunks to represent sets, which we do by using `(the-set-of-all` s`)` to abbreviate the expression `(lambda () ` s`)` and `(a-member-of` s`)` to abbreviate `(apply s)`. Likewise, we will write the formula `(a-subset-of` s`)` for `(a-thunk-to` s`)`.

Now that we have added `lambda` to the language, we can define `let` as an abbreviation by taking (`let` x:s s_1) to be mean (`apply (lambda` x:s s_1) s_1).

We also find it convenient to abbreviate (`and (there-exists` s) (`at-most-one` s)) as (`singleton` s), and (`and` s:t t:s) as (`=` s t).

We will also use boolean combinations other than `and` and `boolean-not`, assuming the natural translations into `and` and `boolean-not`.

When applying an implementation attachment function, e.g. $<$, it will be convenient to abbreviate the testable formula e:(`some-such-that` x (`a-thing`) (`<` x p):`'true`) as (`apply-imp <` e p).

Finally, we keep the abbreviations `some-such-that` and `forall` that were defined in the first-order chapter, expanding the occurrences of `let` into `lambda` as just described.

Examples As examples of the use of the higher order features of the language, we show higher-order versions of the definitions of three functions analyzed earlier: `insert`, `sort`, and `mapcar`. These definitions rely on some basic definitions that we also show:

```
(define (a-set)
  (both (a-subset-of (the-set-of-all (a-thing)))
        (a-thing)))

(define (an-operator) (an-operator-from-to (a-set) (a-set)))

(define (mapcar f:(an-operator) l:(a-list))
  (if l:'nil
      l
      (cons (f (car l)) (mapcar f (cdr l)))))

(define (a-transitive-operator)
  (some-such-that r (an-operator)
    (forall x:(a-domain-member-of r)
```

```
       (r (r x)):(r x))))

(define (a-total-operator)
  (some-such-that r (an-operator)
    (forall x:(a-domain-member-of r)
            y:(a-domain-member-of r)
       (or x:(r y) y:(r x)))))

(define (an-irreflexive-operator)
  (some-such-that r (an-operator)
    (forall x:(a-domain-member-of r)
       x:(not (r x)))))

(define (an-ordering)
  (both (a-transitive-operator)
        (a-total-operator)
        (an-irreflexive-operator)))

(define (the-domain-of (r (an-operator)))
  (the-set-of-all (a-domain-member-of r)))

(define (the-range-of (r (an-operator)))
  (the-set-of-all (r (a-domain-member-of r))))

(define (an-implementation-of (r (an-operator)))
  (some-such-that r' (an-operator-from-to (the-domain-of r)
                       (an-operator-from-to (the-range-of r)
                         (the-set-of-all (either 'true 'false))))
    (forall x:(the-domain-of r)
       (= (r x)
          (some-such-that y (the-range-of r)
            (r' x y):'true)))))
```

```
(define (insert (> (an-implementation-of (an-ordering)))
                (x (a-domain-member-of >))
                (lst (a-list)))
  (if lst:'nil
      (cons x lst)
      (if (apply-imp > (car lst) x)
          (cons x lst)
          (cons (car lst) (insert > x (cdr lst))))))

(define (sort >:(an-implementation-of (an-ordering))
              lst:(a-list-of (a-domain-member-of >)))
  (if lst:'nil
      lst
      (insert > (car lst) (sort > (cdr lst)))))
```

4.2 Semantics

The addition of higher-order constructs to the language considerably complicates our semantic treatment. It will no longer do to use a simple Herbrand universe, as our domain must also contain functions. As noted above, zero-argument functions can be thought of as representing sets—this means our domain must contain an object which is the denotation of `(the-set-of-all (a-thing))`, which should surely not be a possible value of `(a-thing)` itself. This problem is exactly analogous to the need for a set/class distinction in set theory. The semantics we give here is similar to that given by McAllester for the Ontic representation language[31]. Our treatment varies from the more standard complete partial order semantics—we call our semantics *set theoretic*.

To avoid a contradiction then, we must allow `(a-thing)` to denote only some part of our domain. We wish this part of the domain to contain the universe of everything

we normally talk about, and thus it must be itself a model of ZF set theory. We call this part of the domain the *predicative* universe. If we were satisfied to omit the constructor `a-thing` from our language, the predicative universe would suffice as a model for our language (cons cells and functions can be encoded as sets in the model domain). Unfortunately, *impredicative* expressions (those using `a-thing` or earlier impredicative definitions) are very useful and convenient, as in the following definition of `a-list-of`, a function which constructs a list of members drawn from a particular set `s`:

```
(define (a-list-of s:(a-set))
  (either '()
          (cons (a-member-of s) (a-list-of s))))
```

The impredicative construct here is `a-set`. Our predicative universe will not contain any value for `(the-set-of-all (a-set))`. Defining `(a-set)` to be "every set" (i.e., every predicative set) requires the use of `a-thing`:

```
(define (the-universe)
   (the-set-of-all (a-thing)))
(define (a-set)
   (both (a-subset-of (the-universe))
         (a-thing)))
```

Without the impredicatively defined `(a-set)` it is unclear how we could state that `a-list-of` can be applied to "any set." [1]

In order to provide a semantics for the impredicative `a-thing`, we take a model of our language to be a model of set theory which contains the predicative universe (another model of set theory) as a domain element. The larger model will be able to give meaning to all the impredicative expressions in our language without creating any paradox.

[1] note: `a-list-of` is itself impredicative because it relies on `a-thing` in its definition. However, applications of `a-list-of` to predicative arguments can be syntactically recognized as predicative, since the impredicative part of `a-list-of` is limited to it's input domain.

More formally, we define a *standard model of ZF set theory* to be a family of sets $\mathcal{M}$ which satisfies the following properties:

- $\mathcal{M}$ is closed under powerset—the powerset of any member is a member.
- $\mathcal{M}$ is closed under member-of—any member of a member is a member.
- The union of any *small* subset of $\mathcal{M}$ is a member of $\mathcal{M}$—a subset is small if it is countable, or if it's cardinality is no larger than some member of $\mathcal{M}$.

From Gödel's Incompleteness result we know that the existence of standard models of ZF cannot be established using methods within ZF. Nevertheless, this existence is effectively assumed by the working mathematician and is a valid theorem in our inference system (implicitly, because the inference rules can derive `(there-exists (the-set-of-all (a-thing)))`, and that `(the-set-of-all (a-thing))` models all the axioms of ZF).

We can now take a model $\mathcal{M}$ of our language to be any standard model of ZF which contains as a distinguished member $\mathcal{M}_{\text{Pred}}$ another standard model of ZF. Since $\mathcal{M}$ is a universe of sets, whereas our language constructs denote thunks, operators, and constructor values, we will need a way to encode thunks, etc. as sets. We can use any encoding that uses each set at most once—i.e. we can recover the thunk or operator or constructor application from the set representing it unambiguously. We assume we have such an encoding (perhaps built from tagged tuples using the standard means of representing numbers and tuples as sets), and refer to it as follows:

- Given a set $t \in \mathcal{M}$, `(encoded-thunk` t`)` is the set encoding the thunk that can return any value in t. We sometimes write `encoded-set` for `encoded-thunk`.
- Given an relation $r \subseteq \mathcal{M} \times \mathcal{M}$ and domain set $d \in \mathcal{M}$, `(encoded-relation` r d`)` is the set encoding r restricted to domain d.
- Given a constructor function symbol c of n arguments, along with sets $a_1 \cdots a_n$ (representing arguments for c), `(encoded-`c`-app` $a_1 \ldots a_n$`)` represents the output of c on $a_1 \cdots a_n$.

Using this encoding, we now give the formal semantics of each of our language constructs. A program expression will denote a singleton set in $\mathcal{M}$ (containing the value returned by the program). A specification expression will denote an arbitrary set in $\mathcal{M}$ (the set of possible values of the specification)—if the specification is a program expression, the denotation will be a singleton or empty set. A formula will denote either **true** or **false**. For any variable interpretation ρ mapping language variables to elements of a model $\mathcal{M}$, we define the meaning $\mathcal{M}_\rho(e)$ of any language expression e recursively on the structure of e as follows[2]:

- $\mathcal{M}_\rho\,(s_1 : s_2)) = \begin{cases} \textbf{true} & \text{when } \mathcal{M}_\rho\,(s_1) \subseteq \mathcal{M}_\rho\,(s_2), \\ \textbf{false} & \text{otherwise} \end{cases}$

- $\mathcal{M}_\rho\,(\texttt{(boolean-not } \phi\texttt{)}) = \begin{cases} \textbf{true} & \text{when } \mathcal{M}_\rho\,(\phi) = \textbf{false}, \\ \textbf{false} & \text{otherwise} \end{cases}$

- $\mathcal{M}_\rho\,(\texttt{(and } \phi_1\ \phi_2\texttt{)}) = \begin{cases} \textbf{true} & \text{when } \mathcal{M}_\rho\,(\phi_1) = \textbf{true} \text{ and } \mathcal{M}_\rho\,(\phi_2) = \textbf{true}, \\ \textbf{false} & \text{otherwise} \end{cases}$

- $\mathcal{M}_\rho\,(\texttt{(small } s\texttt{)}) = \begin{cases} \textbf{true} & \text{when } \mathcal{M}_\rho(s) \in \mathcal{M}_{\text{Pred}}, \\ \textbf{false} & \text{otherwise} \end{cases}$

- $\mathcal{M}_\rho\,(\texttt{(there-exists } s\texttt{)}) = \begin{cases} \textbf{true} & \text{when } \mathcal{M}_\rho(s) \text{ is non-empty}, \\ \textbf{false} & \text{otherwise} \end{cases}$

[2]Note that we are overloading $\mathcal{M}$, as it is both a set (the domain of the model) and a function (the valuation function giving meaning to syntax). Which meaning is intended is always clear from the context.

- $\mathcal{M}_\rho\,(\texttt{(at-most-one}\ s\texttt{)}) = \left\{ \begin{array}{ll} \textbf{true} & \text{when } \mathcal{M}_\rho(s) \text{ is empty or singleton,} \\ \textbf{false} & \text{otherwise} \end{array} \right.$

- $\mathcal{M}_\rho\,(x) = \{\rho(x)\}$, for any variable x,

- $\mathcal{M}_\rho\,(\texttt{(apply}\ t\texttt{)}) = \left\{ x \;\middle|\; \begin{array}{l} \exists t' \in \mathcal{M}_\rho\,(t)\ .\ \exists s \in \mathcal{M} \\ \quad t' = \texttt{(encoded-thunk}\ s\texttt{)} \ \wedge\ x \in s \end{array} \right\}$

- $\mathcal{M}_\rho\,(\texttt{(apply}\ f\ s\texttt{)}) = \left\{ x \;\middle|\; \begin{array}{l} \exists f' \in \mathcal{M}_\rho\,(f)\ .\ \exists r \in \mathcal{M} \times \mathcal{M}\ .\ \exists d \in \mathcal{M} \\ \quad f' = \texttt{(encoded-relation}\ r\ d\texttt{)} \ \wedge \\ \quad \exists s' \in \mathcal{M}_\rho\,(s) \cap d\ .\ \langle s', x\rangle \in r \end{array} \right\}$

- $\mathcal{M}_\rho\,(\texttt{(}c\ s_1 \ldots s_n\texttt{)}) = \left\{ x \;\middle|\; \begin{array}{l} \exists s'_1 \in \mathcal{M}_\rho\,(s_1) \cdots s'_n \in \mathcal{M}_\rho\,(s_n) \\ \quad x = \texttt{(encoded-}c\texttt{-app}\ s'_1 \ldots s'_n\texttt{)} \end{array} \right\}$

  for any constructor function symbol c,

- $\mathcal{M}_\rho\,(\texttt{(}c_i\ s\texttt{)}) = \left\{ x \;\middle|\; \begin{array}{l} \exists s'_1 \in \mathcal{M} \cdots s'_n \in \mathcal{M} \\ \quad s'_i \in \mathcal{M}_\rho\,(s) \ \wedge\ x = \texttt{(encoded-}c\texttt{-app}\ s'_1 \ldots s'_n\texttt{)} \end{array} \right\}$

  where c_i is the i'th selector for constructor c,

- $\mathcal{M}_\rho\,(\texttt{(if}\ \phi\ s_1\ s_2\texttt{)}) = \left\{ \begin{array}{ll} \mathcal{M}_\rho\,(s_1) & \text{when } \mathcal{M}_\rho\,(\phi) = \textbf{true} \\ \mathcal{M}_\rho\,(s_2) & \text{otherwise,} \end{array} \right\}$

- $\mathcal{M}_\rho\,(\texttt{(fix}\ g\ s\,[g]\texttt{)}) = \text{Next}\,(g_\infty)$

$$\text{where} \quad g_\infty = \bigcup_i \text{Next}^i(\{\})$$
$$\text{Next}\,(d) = \mathcal{M}_{\rho'(d)}\,(s\,[\texttt{(apply g)}])$$
$$\rho'(d) = \rho\,[\,g := \texttt{(encoded-thunk}\ d\texttt{)}\,]$$

Intuitively, $\text{Next}\,(d)$ *contains the value of* s *with* g *replaced by choice from* d*.*

- $\mathcal{M}_\rho\,(\texttt{(lambda ()}\ s\texttt{)}) = \{\texttt{(encoded-thunk}\ \mathcal{M}_\rho\,(s)\texttt{)}\}$

- $\mathcal{M}_\rho\,(\texttt{(lambda}\ x\texttt{:}s_1\ s\texttt{)}) = \{\texttt{(encoded-operator}\ r\ d\texttt{)}\}$
  $$\text{where} \quad d = \mathcal{M}_\rho\,(s_1)$$
  $$r = \{\,\langle y, z\rangle \mid y \in d \ \wedge\ z \in \mathcal{M}_{\rho[x:=y]}\,(s)\,\}$$

- $\mathcal{M}_\rho\,(\texttt{(either}\ s_1\ s_2\texttt{)}) = \mathcal{M}_\rho\,(s_1) \cup \mathcal{M}_\rho\,(s_2)$

- $\mathcal{M}_\rho\,(\texttt{(both}\ s_1\ s_2\texttt{)}) = \mathcal{M}_\rho\,(s_1) \cap \mathcal{M}_\rho\,(s_2)$

- $\mathcal{M}_\rho\,(\texttt{(not}\ s\texttt{)}) = \mathcal{M}_\rho\,(\texttt{(a-thing)}) - \mathcal{M}_\rho\,(s)$

- $\mathcal{M}_\rho\,(\texttt{(a-thing)}) = \left\{ s \in \mathcal{M}_{\text{Pred}} \;\middle|\; \begin{array}{l} s \text{ is an } \texttt{encoded-thunk}, \texttt{encoded-oper-} \\ \texttt{ator}, \text{ or } \texttt{encoded-}c\texttt{-app} \text{ for some } c \end{array} \right\}$

- $\mathcal{M}_\rho\,(\bot) = \{\}$

- $\mathcal{M}_\rho\,(\texttt{(an-operator-from-to}\ s_1\ s_2\texttt{)}) =$

$$\left\{ \texttt{(encoded-relation } r \ \ d\texttt{)} \;\middle|\; \begin{array}{l} \exists d_1 \in \mathcal{M} \ \ \exists d_2 \in \mathcal{M} \\ \quad r \subseteq d_1 \times d_2 \ \ \wedge \\ \quad \texttt{(encoded-set } d_1\texttt{)} \in \mathcal{M}_\rho\,(s_1) \ \ \wedge \\ \quad \texttt{(encoded-set } d_2\texttt{)} \in \mathcal{M}_\rho\,(s_2) \end{array} \right\}$$

- $\mathcal{M}_\rho\,(\texttt{(a-thunk-to } s\texttt{)}) =$

$$\left\{ \texttt{(encoded-thunk } t\texttt{)} \;\middle|\; \begin{array}{l} \exists d \in \mathcal{M} \\ \quad t \subseteq d \ \ \wedge \\ \quad \texttt{(encoded-set } d\texttt{)} \in \mathcal{M}_\rho\,(s) \end{array} \right\}$$

- $\mathcal{M}_\rho\,(\texttt{(a-domain-member-of } s\texttt{)}) =$

$$\left\{ x \;\middle|\; \begin{array}{l} \exists r \in \mathcal{M} \times \mathcal{M} \ \ \exists d \in \mathcal{M} \\ \quad x \in d \ \ \wedge \ \ \texttt{(encoded-relation } r \ \ d\texttt{)} \in \mathcal{M}_\rho\,(s) \end{array} \right\}$$

Given a particular model $\mathcal{M}$ and variable interpretation ρ, we can view the body $C[\mathsf{g}]$ of a `fix` expression as a *context* mapping meanings of g to subsets of $\mathcal{M}$. In order to discuss the semantic motivation for our syntactic restrictions on `fix` expressions, we introduce the notation $C_{\mathcal{M},\rho}$ for this function, so that $C_{\mathcal{M},\rho}(f) = \mathcal{M}_{\rho[\mathsf{g}:=f]}(C[\mathsf{g}])$ for any suitable domain element f.

In order to ensure that our recursive expressions have well-defined least fixed points, we make the restriction that the the corresponding context function $C_{\mathcal{M},\rho}$ is monotone and continuous. Considering relations as sets of ordered pairs, we say that $C_{\mathcal{M},\rho}$ is *monotone* if whenever $f \subseteq g$ we have $C_{\mathcal{M},\rho}(f) \subseteq C_{\mathcal{M},\rho}(g)$. $C_{\mathcal{M},\rho}$ is *continuous* if for any infinite sequence of relations $f_1 \subseteq f_2 \subseteq f_3 \cdots$, we have that $C_{\mathcal{M},\rho}(\cup_i f_i)$ is the same as $\cup_i C_{\mathcal{M},\rho}(f_i)$. The syntactic restrictions placed above in Sect. 4.1 are sufficient to ensure that the body of any accepted `fix` expression satisfies both of these semantic restrictions.

Monotonicity and continuity of $C_{\mathcal{M},\rho}$ together ensure that a least fixed point for C

can be constructed by iterating $C_{\mathcal{M},\rho}$ on the empty relation $f_\perp$ countably many times. We write $C^i_{\mathcal{M},\rho}(f)$ for the application of $C_{\mathcal{M},\rho}$ to f iterated i times. Monotonicity ensures that for any i, $C^i_{\mathcal{M},\rho}(f_\perp) \subseteq C^{i+1}_{\mathcal{M},\rho}(f_\perp)$. This then ensures that we can use continuity to show that $\cup_i C^i_{\mathcal{M},\rho}(f_\perp)$ is a fixed-point of C, as follows. Call the desired fixed-point expression C_∞. Continuity implies that $C_{\mathcal{M},\rho}(C_\infty) = \cup_i C_{\mathcal{M},\rho}(C^i_{\mathcal{M},\rho}(f_\perp))$ but the right hand side is just C_∞ again, as desired.

4.3 Higher-Order Specification Inference

The specification inference mechanisms for our first-order language, defined earlier in Sect. 3.3, generalize naturally to the higher order language just presented. Rather than present the inference system again here, we present only the extensions needed to the above system.

Elimination of Compound Formulas The first-order analysis presented earlier in Sect. 3.3 handled only atomic formulas in the test of an `if` expression. Atomic formulas expose the type information involved in the test more readily than compound formulas. While a practical system might choose to use a general mechanism such as Boolean constraint propagation [**?**] to extract type information from a compound formula, here we simply eliminate compound formulas entirely. Each `if` expression with a top-level $\neg$ expression in its test can be rewritten as an equivalent `if` expression with a smaller test without the $\neg$ by switching the branches. Similarly, each `if` expression with and top-level `and` expression in its test can be rewritten to nested smaller `if` expressions with smaller tests without the `and` by testing each combination of the truth values of the subexpressions of the `and`. This second rewrite does increase the length of the overall expression—in order to keep a linear bound on this process we must bound the complexity of the tests in `if` expressions. We assume that a rewrite like the one described here is performed before any definition is considered by the system, so we reason here only about expressions containing only atomic formulas.

Definition Analysis Revisited In analyzing our first-order language, the only place where `lambda` and `fix` constructs were encountered was at the top level of a definition. The first-order analysis algorithm provides a means for generating a set of theorems about top-level `lambda` or `fix` expressions—this same method can be applied to any internal `lambda` or `fix` expression encountered during recursive descent. Therefore, the first-order algorithm can be applied directly to higher order expressions just as it is. When a `lambda` or `fix` construct is encountered during recursive descent, a process identical to the top-level first-order definition analysis is performed, resulting in a set of new type theorems about the `lambda` or `fix` expression—these type theorems may be used later in the recursive descent analysis of the surrounding expressions.

However, due to the richer semantic domain we have chosen for our higher-order language, there are several ways we can strengthen the first-order algorithm. We will add some new sequent rules to the syntax-directed $\vdash_\circ$ inference relation that was defined in the first-order chapter. Also, we will add many new inference rules to the forward-chaining $\vdash_e$ inference relation, replacing some of the rules that were already present.

Extensions to $\vdash_\circ$ We first observe that the Analyze-Fix and Analyze-Lambda rules can be used internally during recursive descent just as they were at the top level for first-order definitions. We intend these rules to be used in the same manner that `lambda` and `fix` expressions were analyzed by the $\vdash_\circ$ relation in the first-order case—in particular, the selection of induction hypotheses for the Analyze-Fix rule is done in exactly that same manner. We present now some extensions to $\vdash_\circ$ for the higher-order language.

First, we give a higher-order version of the first-order `let` rule, since `let` is no longer explicitly in the language as it can be naturally represented by a `lambda` application.

Analyze-Let

$\Gamma = \Sigma \cup \mathit{THMS}_\Sigma\,(s)$

$\Gamma,\ x{:}s \vdash B{:}t$

x not in Γ or t

$\Sigma \vdash_{\circ_e}$ `((lambda` $x{:}s\ B)\ s)$$:t$

We also need rules that reason about whether a given `lambda` or `fix` expression is predicative. These rules essentially determine syntactically whether an expression relies on `(a-thing)` or not. Whenever a `lambda` or `fix` expression is encountered during recursive descent we attempt to prove that it is `small` using one of these rules.

Lambda-Smallness

$\Gamma = \Sigma \cup \mathit{THMS}_\Sigma\,(t)$

$\Gamma \vdash$ `(small` t`)`

$\Gamma,$ `(small` x`)` $\vdash$ `(small` B`)`

x not free in Γ

$\Sigma \vdash$ `(small (lambda` $x{:}t\ B$`))`

Fix-Smallness

$\Gamma,$ `(small` g`)` $\vdash$ `(small` B`)`

g not free in Γ

$\Sigma \vdash$ `(small (fix` $g\ B$`))`

Many useful expressions are not themselves small, but return small types (that is, a small set of possible values) whenever they are applied to small types (e.g. `a-list-of` as defined above). Recognizing this property greatly enlarges the set of expressions that are quickly recognized as predicative. To this end, we introduce a new expression `(small-after` $s\ n$`)` where s is a specification expression and n is a natural number. These expressions are not in the external language, but are used by the inference process to record the corresponding properties—that s applied to n small classes yields a small class. We will see below how the forward-chaining inference rules draw the

appropriate `small` conclusions using `small-after` facts. Here we give sequent rules for deriving `small-after` facts whenever `lambda` or `fix` expressions are encountered.

Lambda-Small-After

$\Gamma = \Sigma \cup \mathit{THMS}_{\Sigma}\,(t)$

Γ, `(small` x`)` $\vdash$ `(small-after` B n`)`

x not free in Γ

Σ $\vdash$ `(small-after (lambda` x`:`t B`)` $n+1$`)`

Fix-Small-After

Σ, `(small-after` g n`)` $\vdash$ `(small-after` B n`)`

g not free in Σ

Σ $\vdash$ `(small-after (fix` g B`)` n`)`

We also add similar new constructs and sequent rules (along with forward-chaining inference rules later) for determining when a `lambda` or `fix` construct will return a value (`there-exists-after`) or return at most one value (`at-most-one-after`). These rules are important in strengthening the reasoning about when higher order expressions are total or functional, respectively.

Lambda-At-Most-One-After

$\Gamma = \Sigma \cup \mathit{THMS}_{\Sigma}\,(t)$

Γ, `(at-most-one` x`)` $\vdash$ `(at-most-one-after` B n`)`

x not free in Γ

Σ $\vdash$ `(at-most-one-after (lambda` x`:`t B`)` $n+1$`)`

Fix-At-Most-One-After

Σ, `(at-most-one-after` g n`)` $\vdash$ `(at-most-one-after` B n`)`

g not free in Σ

$\Sigma \vdash$ `(at-most-one-after (fix` g B`)` n`)`

Lambda-There-Exists-After

$\Gamma = \Sigma \cup \mathit{THMS}_{\Sigma}(t)$

Γ, `(there-exists` x`)` $\vdash$ `(there-exists-after` B n`)`

x not free in Γ

$\Sigma \vdash$ `(there-exists-after (lambda` x`:`t B`)` $n+1$`)`

Fix-There-Exists-After

Σ, $\vdash$ `(there-exists-after` B n`)`

g not free in Σ

$\Sigma \vdash$ `(there-exists-after (fix` g B`)` n`)`

Obvious Consequences Revisited The only remaining changes needed to the specification inference algorithm lie in the underlying forward chaining horn-clause inference rules. These rules, as before, perform simple, locally-acting inferences—each rule is based on the semantics of one language construct or occasionally of the interaction of two language constructs. We need a number of new rules to deal with the new language constructs in the higher order language.

All the rules in use for the first-order inference system are also used for the higher-order system, with the following exceptions:

- The rule that places every expression under `(a-thing)` is not used.

- The rules with variables (p,q) restricted to program expressions are superseded by rules below. These were `Symmetry`, `Selectors1`, and `Univ-Inst`.

We introduce the new rules added in groups organized by the construct being reasoned about. First, we present the new rules which reason about `lambda` expressions. Note that our central reasoning about `lambda` expressions is accomplished by the recursive descent rules above. In addition we have the following simple rules:

Lambda-Term

`Dom` (`(lambda` x`:`s B`)`)

`(singleton (lambda` x`:`s B`))`

App-Existence

`(there-exists (apply` f s`))`

`(singleton` s`)`

---

s`:(a-domain-member-of` f`)`

Lambda-Domain

`Dom` (`(lambda` x`:`s B`)`)

`(= (a-domain-member-of (lambda` x`:`s B`))` s`)`

We also add the following inference rules for reasoning about thunk (`lambda ()`) expressions:

Singleton-Thunk

`Dom` (`(lambda () ` t`)`)

`(singleton (lambda () ` t`))`

Apply-Thunk

`Dom` (`(lambda () ` t`)`)

`(= (apply (lambda () ` t`)) ` t`)`

Equate-Thunks

(= (apply (lambda () s_1)) (apply (lambda () s_2)))

(= (lambda () s_1) (lambda () s_2))

Next we add the following rules for thunk and operator spaces:

Op-Domain

f : (an-operator-from-to s t), Dom (f)

(a-domain-member-of f) : (a-member-of s)

Op-Range

f : (an-operator-from-to s t), Dom (f)

(apply f (a-domain-member-of f)) : (a-member-of t)

Thunk-Range

f : (a-thunk-to s)

(apply f) : (a-member-of s)

Infer-Thunkspace

(singleton s)

(apply (lambda () t)) : (a-member-of s)

Dom ((a-thunk-to s)), Dom (t)

(lambda () t) : (a-thunk-to s)

We also add the following rules which reason about `(a-thing)` and smallness:

Things-are-Small

s : `(a-thing)`, `(singleton` s`)`

`(small` s`)`

Small-Is-Thing

`(small` s`)`

---

s : `(a-thing)`

Monadic-Smallness

`(small` s`)`, `Dom` $((c\ s))$

c any monadic term constructor except `not`

`(small` $(c\ s)$`)`

Small-If

`(small` s_1`)`, `(small` s_2`)`

`Dom` $(($`if` $\Phi\ s_1\ s_2))$

`(if` $\Phi\ s_1\ s_2$`)`

Binary-Smallness

`(small` s`)`, `(small` t`)`

`Dom` $((c\ s\ t))$

c any binary term constructor exc. `lambda`, `fix`

`(small` $(c\ s\ t)$`)`

Small-After-Zero1

`(small-after` $f\ 0$`)`

`(small` f`)`

Small-After-Zero2

`(small` f`)`

`(small-after` $f\ 0$`)`

Small-After-Dec

(small-after f $n+1$)

(small t)

Dom $((f\ t))$

(small-after $(f\ t)$ n)

The following rules deal with pure formulas—i.e., the only terms involved are rule meta-variables.

Infer-And

Φ, Ψ

Dom ((and Φ Ψ))

(and Φ Ψ)

And1

(and Φ Ψ)

Φ

And2

(and Φ Ψ)

Ψ

Exists-Up

(there-exists s)

$s:t$

(there-exists t)

At-Most-One-Down

(at-most-one t)

$s:t$

(at-most-one s)

Small-Down

(small t)

$s:t$

(small s)

At-Most-One-After-Zero1

(at-most-one-after f 0)

(at-most-one f)

At-Most-One-After-Zero2

(at-most-one f)

(at-most-one-after f 0)

At-Most-One-After-Dec

(at-most-one-after f $n+1$)

(at-most-one t)

Dom $((f\ t))$

(at-most-one-after $(f\ t)$ n)

There-Exists-After-Dec

(there-exists-after f $n+1$)

(there-exists t), Dom $((f\ t))$

t:(a-domain-member-of f)

(there-exists-after $(f\ t)$ n)

There-Exists-After-Zero1

(there-exists-after f 0)

(there-exists f)

There-Exists-After-Zero2

(there-exists f)

(there-exists-after f 0)

Symmetry

(there-exists s)

(at-most-one t)

s:t

---

t:s

Variables

x is a variable

(singleton x)

In each of the next two rules, c can be any selector or constructor function symbol, or any of a-domain-member-of, apply, an-operator-from-to, or a-thunk-to. n is taken to be the arity of c.

Monotonicity

(is u_1 v_1)...(is u_n v_n)

(is $(c\ u_1 \ldots u_n)$ $(c\ v_1 \ldots v_n)$)

Arg-Existence

(there-exists $(c\ u_1 \ldots u_n)$)

(there-exists u_i)

The following rules deal with expressions constructed with a constructor. We show

these rules for the constructor `cons` and its selectors `car` and `cdr`, but we intend them to act on any constructor function present in the target language.

Cons-Is-A-Fun

`(singleton` s t`)`

`(singleton (cons` s t`))`

Sel-Fun

`(singleton (cons` s t`))`

`(singleton` s`)`

`(singleton` t`)`

Selectors1a

`(there-exists` t`)`

`Dom ((cons` s t`))`

---

s`:(car (cons` s t`))`

Selectors1b

`(there-exists` s`)`

`Dom ((cons` s t`))`

---

t`:(cdr (cons` s t`))`

We also add the following rules for `if` expressions:

If-True

`Dom ((if` Φ s t`))`

Φ

`(=` `(if` Φ s t`)` s`)`

If-False

`Dom ((if` Φ s t`))`

`(boolean-not` Φ`)`

`(=` `(if` Φ s t`)` t`)`

Finally, we add the following rules for instantiation of universal formulas and construction of the inference domain:

Range-Exp

`(lambda` x`:`s B`)` appears in e

`Dom (((lambda` x`:`s B`) (a-domain-member-of (lambda` x`:`s B`))))`

Univ-Inst

`forall` $x{:}t$ Φ

$s{:}t$, where s appears in e

`(singleton` s`)`

`[`s`/`x`]` Φ

`Dom` (`[`s`/`x`]` Φ)

We now argue that the higher order version of $\vdash$ can still be computed in polynomial time. The argument is very similar to the first-order case. Let Σ be a premise set. 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 to complexity bounds:

1. $\mathcal{A}$ has at most polynomially many members in the size of Σ, and
2. The entire forward-chaining can be computed in polynomial-time in the eventual 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 $\bot$, `(a-thing)`, a subexpression of e, (`f` `(a-domain-member-of` `f`)) for some f appearing in e, or some subexpression of a universal formula `forall` $x_1 : s_1 \ldots x_n : s_n$ Φ in Σ with its variables replaced by subexpressions of e. Only the last case poses a challenge, forcing us to limit the quantification depth of the formulas in Σ to some constant. Specifically, we require that no formula in Σ has a depth of `lambda` nesting greater than some fixed constant. Given this restriction there are only polynomially many instances of universal formulas in Σ on subexpressions of e.[3]

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`)`, `(cdr` s`)`, (`s` `(a-domain-member-of` s`))`, `(lambda ()` s`)`, or `(a-member-of` s`)` for some s in $\mathcal{A}$.

[3]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 `lambda` nesting within analyzed definitions.

But there are only polynomially many possible conclusions over such specs, and for any partially closed premise set we can find a new consequence in polynomial time.

Theorem 4.1 *For any premise set* Σ *with bounded* `lambda` *nesting, and function symbol definition* e*, we can compute the forward-chaining closure of* Σ *under inference rules defining the higher-order* $\vdash_e$ *in polynomial time.*

Example Inferences The inference algorithm presented here, together with the enhancements discussed in the next chapter for reasoning about specification definitions, infers specifications for the `sort` and `mapcar` examples given above that are directly analogous to those inferred for the first order version. In particular, we automatically infer that (`sort` $>$ l) returns (`a-permutation-of` l), and that (`mapcar` f l) returns (`samelength-as` l).

We will discuss the lemmas and inference steps involved in these inferences after presenting the enhancements in the next chapter.

Chapter 5

Reasoning About Nondeterminism

The algorithm presented so far is designed primarily for analyzing deterministic program expressions, although as we've seen it also applies usefully to nondeterministic specification expressions. In this chapter we consider an extension to the algorithm specifically designed for reasoning about nondeterministic expressions. This extension increases the power of the algorithm in reasoning about specification expressions—this increase in power can even help in the analysis of deterministic program expressions because specification expressions can occur as argument types and within the formulas of deterministic expressions. We present in this chapter an enhancement to our inference algorithms which automatically creates deterministic witnesses that enhance the analysis of nondeterministic expressions. We first present an extension to our forward-chaining notion of "obvious consequence" ($\vdash_e$), and then show how to integrate that extension most usefully into our syntax directed relation $\vdash^{\circ}_e$. Throughout this section we assume that we are analyzing expressions in the higher order version of our language.

5.1 Adding Existential Instantiation to $\vdash_e$

The expression to be analyzed (and its subexpressions) may have zero, one, or many values. Many useful inference principles, particularly universal instantiation, can operate only on terms with exactly one value (they are not sound otherwise). Because

we want the power of these inference principles to be used in our evaluation, we need to ensure that we are reasoning about expressions with exactly one possible value whenever possible. In particular, whenever we know that a certain type of object exists, we would like to have a witness object of that type to reason about.

The desired witness creation is a form of existential instantiation. In our language, we can represent existential facts without explicit quantifiers. We discuss here means of providing automatic existential instantiation for two forms of quantifier free existential information. First, if the formula `(there-exists` s`)` is known for some specification expression s, it is implied that there exists some value for new variable x_s such that $x_s : s$. Second, if the formula $x : (R\ s)$ is known, it is implied that there exist some values for new variables $y_{s,x}$ and $y_{R,x}$ such that $y_{s,x} : s$, $y_{R,x} : R$, and $x : (y_{R,x}\ y_{s,x})$.

By using these two forms of existential instantiation as new inference principles we can expand the number of singleton expressions available to the reasoning process. Ideally, these principles would be applied whenever the prerequisite existential information was present, possibly by the addition of new inference rules like the following to the forward-chaining $\vdash_e$ closure:

Ex-Inst-1

`(there-exists` s`),` with s not a program exp.

x_s is a new variable

$x_s : s$, `Dom` (x_s)

App-Ex-Inst-1

x:`(apply` R s`)`, with `(apply` R s`)` not a program exp.

$y_{s,x}$ is a new variable

$y_{R,x}$ is a new variable

$y_{s,x}$:s, $y_{R,x}$:`R` x:`(apply` $y_{R,x}$ $y_{s,x}$`)`

`Dom` `((apply` $y_{R,x}$ $y_{s,x}$`))`

Univ-Gen

x_s:t

x_s does not occur in t

---

s:t

There are several points to explain about these rules:

- An expression s is "not a program exp." if it is a specification expression that is not a program expression. This restriction is made because program expressions can never have more than one value and so do not need to be instantiated in this manner. Without this restriction the rules will try to create witnesses under the witnesses it creates, and so on.

- Also, the variable subscripts in these rules play two significant roles.

    1. Only those variables introduced by the rule Ex-Inst can be generalized by Univ-Gen—these variables are indicated in the rules by having only one subscript.

    2. The soundness of the universal generalization rule Univ-Gen is ensured by the antecedent requiring that x_s does not occur in t—it is essential that the spec t not depend on the choice of x_s. The relevant dependencies are kept track of by the appearance of x_s in the subscripts of those variable that depend on x_s (all of which are created by the rule App-Ex-Inst).

- Finally, each rule that requires a "new variable" can be fired infinitely many ways that differ only in the choice of the new variable. We do not allow any rule to fire twice where the only difference is in the choice of the new variable(s). This restriction has no effect on the inference relation on expressions that do not involve the new variables introduced during the inference process.[1]

Unfortunately, the addition of these rules to the forward-chaining inference relation produces a procedure not guaranteed to terminate because no limit is placed on the possible domain growth. For instance, if it is known that s:(`R` s) for some non-variable specification s such that (`there-exists` s) is also known, then each new variable x created under s will cause the creation of yet another new constant $x_{s,x}$ again under s, ad infinitum.

This difficulty can be solved by restricting the above inference rules as follows. We introduce a new expression `Inst` (s, x) for specification s and variable x which, like `Dom` (s) earlier, has no intended semantics and is used by the inference process to limit the extent of the forward chaining. Intuitively, we can think of `Inst` (s, x) as standing for "existentially instantiate s for instance x." We then modify the first two rules above and add one new rule, as follows:

Ex-Inst

(`there-exists` s), with s not a program exp.

x_s is a new variable

`Dom` (x_s), `Inst` (s, x_s)

[1] This can be proven by considering a transformation on proofs which converts an unrestricted proof into a proof satisfying the restriction by renaming all the "extra" new variables to the original new variable.

App-Ex-Inst

x:(apply R s), with (apply R s) not a program exp.

Inst ((apply R s), x)

$y_{s,x}$ is a new variable

$y_{R,x}$ is a new variable

Inst (R, $y_{R,x}$), Inst (s, $y_{s,x}$), x:(apply $y_{R,x}$ $y_{s,x}$)

Dom ((apply $y_{R,x}$ $y_{s,x}$))

Inst

Inst (s, x)

---

x:s

The intended intuition is that these rules do a recursive descent into each expression in the inference domain, doing existential instantiation. This descent can be done in linear time in the size of the domain. The analysis is complicated by the fact that the inference domain is grown by the instantiation process. This problem can be handled by noticing that the expressions added to the domain are all pure (deterministic) program expressions, and so cause no further existential instantiation.

We can now show easily that the resulting inferential closure is polynomial in the input size (we are closing under the higher-order $\vdash_e$ inference relation from chapter 4 with the above three inference rules added). The key observation is again that the set $\mathcal{A}$ of specifications s such that Dom (s) is inferred is polynomial in the problem size. In the following observations we use $\mathcal{A}_{\text{ho}}$ to denote the set of specifications s such that Dom (s) is concluded by one of the old inference rules (any rule other than the three just added). To see that $\mathcal{A}$ is polynomial in size, we observe:

- $\mathcal{A}_{\text{ho}}$ is polynomial in size, using the same argument given to show the inference domain in chapter 4 is polynomial in size.

- Rule Ex-Inst can only fire with s bound to a member of $\mathcal{A}$, because every member of $\mathcal{A} - \mathcal{A}_{\text{ho}}$ is a deterministic program expression.
- Polynomially many `Inst` facts are derived:
  - For each fact derived of the form `Inst` (t, x), the subscripts in x identify a unique path from a member s of $\mathcal{A}_{\text{ho}}$ to a subexpression of s which is t. By "path" here we mean a sequence $s_1 \cdots s_n$ such that $s_1 = s$, $s_n = t$, and each s_{i+1} is an immediate subexpression of s_i. No two `Inst` facts correspond to the same path (this can be proven by a simple induction on path length).
  - There are only polynomially many such paths.
- There are linearly many new domain members for each new `Inst` fact.

Given then the conclusion that $\mathcal{A}$ is polynomial in the size of the input problem, the same arguments given in Chapter 4 allow us to conclude that the inferential closure can be computed in polynomial time.

We now observe that we can add rules similar to App-Ex-Inst for language constructors other than `Apply`, where there is similar existential information available. We add the following rules, and claim that the polynomial time bound still holds by essentially the same argument as before.

Op-Ex-Inst

x:`(an-operator-from-to` s t`)`

`(an-operator-from-to` R s`)` not a program exp.

`Inst` (`(an-operator-from-to` s t`)`, x)

$y_{s,x}$ is a new variable

$y_{t,x}$ is a new variable

`Inst` $(s, y_{s,x})$, `Inst` $(t, y_{t,x})$, x:`(an-operator-from-to` $y_{s,x}$ $y_{t,x}$`)`

`Dom` (`(an-operator-from-to` $y_{s,x}$ $y_{t,x}$`)`)

Unary-Ex-Inst

x:(`a-thunk-to` s), with (u R) not a program exp.

u is one of: `a-thunk-to`, `apply`, or `a-domain-member-of`

`Inst` ((`a-thunk-to` s), x)

$y_{s,x}$ is a new variable

`Inst` (s, $y_{s,x}$), x:(`a-thunk-to` $y_{s,x}$)

`Dom` ((`a-thunk-to` $y_{s,x}$))

Cons-Ex-Inst

x:(`cons` s t), with (`cons` R s) not a program exp.

`Inst` ((`cons` s t), x)

$y_{s,x}$ is a new variable

$y_{t,x}$ is a new variable

`Inst` (s, $y_{s,x}$), `Inst` (t, $y_{t,x}$), x:(`cons` $y_{s,x}$ $y_{t,x}$)

`Dom` ((`cons` $y_{s,x}$ $y_{t,x}$))

We still need one last embellishment to our forward-chaining rule set in order to achieve our original purpose in creating explicit existential witnesses, which was to provide inference principles such as universal instantiation the appropriate targets to act upon. Our universal instantiation rule from Chapter 4 was restricted to acting on subexpressions of the input expression e. We must relax this restriction to allow the rule to instantiate upon our existential witness variables.

This relaxation must be done carefully to avoid reverting to a nonterminating procedure. In particular, we will ensure that only existential witness variables derived from the original input expression are used in instantiation—not witness variables derived from theorem instances created by previous instantiation. We do this by keeping track of which existential instantiations stem directly from the original input expression and only allowing universal instantiation on these instance variables.

In fact, our subscripts already keep track of this information: we will say that an instance variable *derives from* an expression if that expression is at the root of the subexpression path represented in its subscript. So the variable x_{s,y_t} derives from t—note that s must be a subexpression of t for this variable to have been created as an instance variable. We now replace the Univ-Inst rule from Chapter 4 with the following rule:

Univ-Inst

`forall` $x{:}t$ Φ

$s{:}t$

s derives from or is a subexp. of e

`(singleton` s`)`

`[`s`/`x`]` Φ

`Dom (``[`$s$`/`$x$`]` $\Phi$`)`

As argued above, there can be at most polynomially many instance variables which derive from e. Given a bound on the depth of quantification (as before) we can then once again infer that the number of domain members added by this rule is polynomial in the problem size. Given this bound, the other parts of our polynomial-time argument go through as before.

Theorem 5.1 *For any premise set Σ with bounded* `lambda` *nesting, and function symbol definition e, we can compute the forward-chaining closure of Σ under inference rules defining the higher-order $\vdash_e$ along with those added in this section in polynomial time.*

5.2 Integrating Existential Instantiation With $\vdash_e$

The enhanced version of $\vdash_e$ just described strengthens the overall inference algorithm without requiring any changes to the sequent based definition of the top level inference relation $\vdash\!\!\circ_e$. However, there are a couple opportunities to integrate the existential

instantiation with the recursive descent analysis to further strengthen the inference procedure. In particular, the rules for `lambda` are implicitly doing an existential instantiation when they choose a particular x to represent the formal parameter in analyzing the body.

Consider a `lambda` expression (`lambda` $x{:}s\ B$). Our central sequent rule for analyzing this expression reduces the analysis to finding types for B under the assumption that $x{:}s$. But this assumption is implicitly assuming that there exist values of s—and this assumption may justify other existential instantiations. Intuitively, we can let the forward-chaining inference algorithm do these other instantiations by informing it that indeed x is an existential witness for s. We do this by replacing the rule Analyze-Lambda with the following rule:

Analyze-Lambda

$\Gamma = \Sigma \cup \mathit{THMS}_{\Sigma}(r)$

$\Gamma,\ \texttt{Inst}\ (r,\ x) \vdash^{\circ}_{e} B{:}t$

x and x_1 not in Γ, x_1 not in B

$\Sigma \vdash^{\circ}_{e}$ `Forall` $x_1{:}r$ (`(lambda` $x{:}r\ B$`)` x_1`)`$:t$

The only change is the replacement of the sequent antecedent $x{:}r$ by the antecedent `Inst` $(r,\ x)$ in the first rule antecedent. We make the same modification to the rules Lambda-Small, Lambda-Small-After, and Lambda-At-Most-One-After. These changes do not affect the complexity of the syntax directed application of the inference relation.

There is one more change we can make to the syntax directed inference relation which will enhance the value of our automatic existential instantiation. We note that most goal sequents in our backward chaining application of the inference relation have consequents of the form $s{:}t$ for specification expressions s and t. When attempting to draw such a conclusion, we can assume without loss of generality that there exist values of s. If no such values exist, then every specification t holds of s anyway. This

argument justifies the addition of the following sequent rule:

Assume-Existence

$$\frac{\Sigma,\ \texttt{(there-exists}\ s\texttt{)} \vdash^{\circ}_{e} s:t}{\Sigma \vdash^{\circ}_{e} s:t}$$

We must state how this rule will be applied within the syntax-directed directed inference relation, since it itself is not syntax directed. We assume that this rule is applied immediately whenever an appropriate goal sequent is encountered which does not already have the explicit existence assumption in its antecedent. This does not change the complexity of the inference procedure.

Example Inferences We now discuss the analyses of the example higher-order programs for `mapcar` and `sort` shown in chapter 4. As mentioned in chapter 4, the enhanced higher-order algorithm can infer the specification `(samelength-as` l`)` for `(mapcar` f l`)` and the specification `(a-permutation-of` l`)` for `(sort` $>$ l`)`. We discuss here some of the key inference steps involved and point out what extra lemmas are needed from the user in support of the inference process. The inference chains involved are very similar to the first-order case.

In the `mapcar` example, the key difference from the first-order case comes in realizing that the expression `(f (car lst))` is in fact `(a-thing)`. This inference was trivial in the first-order case, but requires a bit of inference now. The first key inference involved comes in the analysis of the definition of `(an-operator)`. We restate that definition here for reference:

```
(define (an-operator) (an-operator-from-to (a-set) (a-set)))
```

The inference needed is that `(an-operator)` meets the specification `(a-thing)`. Intuitively, this property rests upon the fact that each value of `(an-operator)` is an operator from a particular value of `(a-set)` to another such value. But `(a-thing)` is closed under powerset, and any particular value of `(a-set)` is in `(a-thing)`, so any such operator is in fact in `(a-thing)`, as our forward chaining smallness rules will

infer.

Notice that this reasoning relied upon picking a specific value of `(an-operator)`, and then corresponding specific values of `(a-set)` for its domain and range. These are exactly the existential instantiations that occur as a result of the inference rules added in this chapter, and as a result our inference system does in fact conclude that `(an-operator)` meets the specification `(a-thing)`.

Once `(an-operator)` is known to be `(a-thing)`, the analysis of `mapcar` is not problematic. Our forward-chaining inference will determine that the function variable f is therefore also `(a-thing)`, and that as a result the expression `(f (car lst))` is `(a-thing)` as well (using the various smallness inference rules). The rest of the analysis of `mapcar` goes as it did in the first-order case.

The issues involved in the analysis of `sort` and `insert` are much the same as those just explained. The only complications come from the fact that we have abstracted over the specification function `<` in writing the higher-order versions. Abstracting over a specification function is tricky because when we call `sort` we need to be passing it a program (not a specification) or the call itself will not be a program expression. This means we need to pass in the implementation attachment function for `<`.

The following two lemmas need to be present in order to deal with this extra complexity. We can assume the first lemma is provided by the system (along with the definition of `(an-operator)`) as a way of easing exactly this problem (which comes up anytime you abstract over a specification). The second lemma will have to be proven by the writer of the "orderings" library which is being used (e.g. the definition of `an-ordering`, etc.).

```
forall >':an-operator
       > :an-implementation-of >'
       x :(a-thing)
       y :(a-thing)
  (and (implies (apply-imp > x y)  x:(>' y))
       (implies boolean-not((apply-imp > x y))
                x:(not (>' y))))
```

```
forall >:an-ordering x:(a-domain-member-of >) (> x):(not x)
```

The first lemma will be applied for instance in the analysis of `insert`. Automatic existential instantiation triggered by the newest version of the Analyze-Lambda rule will instantiate the argument type `(an-implementation-of (an-ordering))` to give a particular ordering `>'` such that `>` is `(an-implementation of >')`. Our universal instantiation mechanism will then be able to instantiate the first lemma so that the appropriate information will be concluded when the `if` test `(apply-imp > (car lst) x)` is assumed true or false.

Chapter 6

Tarskian Set Constraints

6.1 Introduction

The reasoning required to infer properties of expressions in computer programs can often be cast as reasoning about containment relationships between simple set expressions. In a set expression, we have set constants representing unknown sets, and various operations for combining set expressions into compound set expressions (e.g. union). Which combining operations we allow will of course critically affect the complexity of the reasoning needed to understand the expressions. We will discuss here only set expression languages containing at least the combinators union and set complement (relative to the model universe). We call a finite set of containment (subset) assertions between set expressions a *set constraint*. We will be concerned with deciding the consistency of set constraints—that is, whether or not there is any way of interpreting the set constants and function symbols appearing in the constraint so that all the containment assertions are simultaneously satisfied.

Recent interest in set constraints has focused almost exclusively on set constraints which allow Herbrand function applications. These set expressions are intended to be interpreted over the Herbrand universe built from the function symbols of the language. Relative to this universe, each function symbol has a fixed standard Herbrand interpretation, mirroring the construction of the universe. Given set expressions $S_1 \cdots S_n$ (each of which denotes a subset of the Herbrand universe), the application

expression $f(S_1, \ldots, S_n)$ denotes the set of all Herbrand domain elements which are $f(s_1, \ldots, s_n)$ for some domain elements $s_1 \cdots s_n$ in the denotations of $S_1 \cdots S_n$. As an example, the expression $cons(X, Y)$ would denote the set of all the domain elements that are formed by applying *cons* to elements chosen from the meanings of X and Y.

Herbrand set constraints problems arise naturally in the analysis of functional programs involving constructors.[2, 21, 20, 16] Several researchers have found complexity bounds for various Herbrand set constraint languages. Basic Herbrand set constraints (union, set complement, and Herbrand application) have been shown to be nondeterministic exponential time complete.[3, 6] Later research has shown that adding projection functions (constructor function inverses, such as *car*) along with negative set constraints (e.g. $E_1 \not\subseteq E_2$) does not change this complexity. [4, 11, 12]

Amidst all the attention to the complexity of Herbrand set constraints satisfiability questions, surprisingly little notice has been taken of similar questions for an older form of set expressions which we will call Tarskian set expressions. Function symbols in these set expressions have no standard interpretation. Instead, we allow each model to choose its own interpretation for the function symbols as functions over the model domain. Meanings of applications are computed as before, once the function symbol meaning is assigned. So, for example, $Append(X, Y)$ will denote the set of all values obtained by applying the meaning of *Append* (which will vary with the model) to elements chosen from X and Y.

Tarskian set expressions were investigated as early as 1951 in work by Tarski, who did not consider computational complexity issues.[23, 22] In the set expressions Tarski analyzed, function symbols could denote arbitrary relations, not just functional relations. Such function symbols can be viewed as nondeterministic functions—when applied to a single tuple of domain elements they can generate many outputs. Applications of such function symbols are still defined in much the same way as before: the expression $R(E_1, \ldots, E_n)$ denotes the set of all values that can be output by applying the meaning of R "nondeterministically" to a tuple chosen from the meanings of the E_i.

In this work we discuss the complexity of checking consistency of set constraints

built up from set constants, union, complement, and Tarskian application of function symbols which may or may not be syntactically deterministic (i.e., we have two classes of function symbols, one of which is syntactically deterministic). Note that we allow function symbols to have arity zero, in which case they are essentially constant symbols denoting single domain elements. The presence of such constants greatly complicates the analysis. We know of few related complexity results on Tarskian constraints. In earlier work we have shown that satisfiability of such Tarskian constraints with no occurrences of union or complement was decidable in cubic time under the assumption that hash table operations take constant time.[28] In subsequent work (published earlier), we showed that union and intersection could be added to the language while keeping satisfiability in cubic time.[17]

In the work presented here, we show the Tarskian set constraints problem with union, complement, and both deterministic and nondeterministic application (including zero arity) to be in nondeterministic doubly exponential time. The proof is by reduction to the solvability of a new special form of Diophantine inequation set, which we show to be in nondeterministic exponential time, but conjecture to be in $\mathcal{NP}$. Proving this conjecture would shave an exponential off of our doubly exponential complexity bound. Also, in joint work with McAllester, Kozen, and Witty, we show several related complexity results for other variations of Tarskian set constraints languages.[29]

6.2 Tarskian Set Constraints

A Tarskian set constraint language is defined by giving 3 sets: a set of set constants, a set of function symbols, and a set of relation symbols, with each function and relation symbol associated with a specific finite arity, possibly zero. We define a *set expression* over such a language recursively to be either a set constant, the complement of a set expression, the union of two set expressions, or an n-ary function or relation symbol applied to n set expressions (where n may be 0).

Definition: A *set expression* is either

- a set constant,
- the set complement $\neg C$ of a set expression C,
- a union $(C_1 \cup C_2)$ of two set expressions C_1 and C_2,
- or an application $R(C_1, \cdots, C_n)$ of an n-ary function or relation symbol R to n set expressions $C_1, \cdots, C_n$.

A *set constraint* is a finite set of subset formulas $C_1 \subseteq C_2$ between set expressions C_1 and C_2.[1] We write $C_1 = C_2$ as an abbreviation for the two formulas $C_1 \subseteq C_2$ and $C_2 \subseteq C_1$. A model $\mathcal{M}$ of a set constraint language $\mathcal{L}$ gives a countable domain set D and an interpretation for each of the symbols of $\mathcal{L}$: each set constant is interpreted as a subset of D, each n-ary function symbol as a function from D^n to D, and each n-ary relation symbol as subset of D^{n+1}. Models interpret set expressions as subsets of their domains in the obvious manner, formalized as follows:

Definition: Let $\mathcal{M}$ be a model with semantic domain D. For any set expression C we define the semantic interpretation of C, written $\mathcal{M}\ [\![C]\!]$, to be a subset of D determined by the following conditions.

- If C is a set constant α then $\mathcal{M}\ [\![\alpha]\!]$ is the subset of D assigned to α by $\mathcal{M}$.
- If C_1 and C_2 are class expressions, then $\mathcal{M}\ [\![(C_1 \cup C_2)]\!]$ is the union of $\mathcal{M}\ [\![C_1]\!]$ and $\mathcal{M}\ [\![C_2]\!]$.
- If R is an n-ary function or relation symbol, and $C_1, \cdots, C_n$ are class expressions, then $\mathcal{M}\ [\![R(C_1, \cdots, C_n)]\!]$ is the set of all y such that there exist elements $x_1 \ldots, x_n$ in $\mathcal{M}\ [\![C_1]\!], \cdots, \mathcal{M}\ [\![C_n]\!]$ respectively such that $\langle\, x_1, \cdots, x_n, y \,\rangle$ is an element of the function or relation that $\mathcal{M}$ assigns to R. Essentially, this is the image of $\mathcal{M}\ [\![C_1]\!] \times \cdots \times \mathcal{M}\ [\![C_n]\!]$ under $\mathcal{M}\ [\![R]\!]$.

[1] Negative subset formulas $C_1 \not\subseteq C_2$ can be expressed by writing $f() \subseteq (C_1 \cap \neg C_2)$ where f is a new 0-ary function symbol.

For any subset formula $C_1 \subseteq C_2$ we define the semantic interpretation of $C_1 \subseteq C_2$, denoted $\mathcal{M}\ [\![C_1 \subseteq C_2]\!]$, to be **T** if $\mathcal{M}\ [\![C_1]\!]$ is a subset of $\mathcal{M}\ [\![C_2]\!]$, and **F** otherwise.

We say that a set constraint is *consistent* if there is a model which interprets each subset formula in the constraint as true.

6.3 Decidability

In this section, we prove that the problem of determining the consistency of a set constraint is decidable in non-deterministic doubly exponential time. We do this by reducing the problem to the problem of finding a solution to an (exponentially larger) set of Diophantine inequalities of the special form defined below.

Definition: A set of Diophantine inequalities

$$\{\, p_i(x_1, \ldots, x_n) \leq q_i(x_1, \ldots, x_n) \mid 1 \leq i \leq m \,\}$$

between polynomials p_i and q_i over non-negative integer variables x_i is *semi-linear* if every p_i is linear, and every q_i is either linear or is the product of variables.

We conjecture that the satisfiability problem for semi-linear Diophantine inequalities is in $\mathcal{NP}$; however, we have only been able to prove the following result.

Theorem 6.1 (Semi-Linear Decidability Theorem) *The problem of determining the satisfiability of a semi-linear set of Diophantine inequalities is solvable in non-deterministic exponential time.*

Proof: Consider a semi-linear set of m Diophantine inequalities over n variables where the largest constant which appears has b bits. Each inequality is either linear or non-linear—we divide the problem into linear and non-linear sub-problems. The linear sub-problem can be converted into an equisatisfiable set of linear Diophantine

equations $Ax = B$ by introducing new "slack" variables. Call a variable x_i *bounded* in $Ax = B$ if there is a non-negative integer k such that no *rational* solution to $Ax = B$ has a value for x_i greater than k. An analysis of the maximum possible upper bound that can be placed by a system of linear constraints shows that any bounded variable can take on integer values describable with at most $O(bn \log n)$ bits[7]. Using linear programming (over the rationals) we can determine in polynomial time which variables occur in nonzero solutions of Ax=0 and hence which variables are bounded.[2][25] Our non-deterministic procedure can now guess the values of the bounded variables. We can then replace each bounded variable by the guessed value, simplifying the linear and non-linear subproblems. While substituting in the guesses, some of the non-linear constraints become linear and must be added to the resulting linear sub-problem, yielding new linear and non-linear subproblems in fewer variables.

We can repeat this process until all the variables in the resulting linear problem are unbounded. Let $A'x = B'$ be the resulting linear problem. We check using integer programming whether $A'x = B'$ is satisfiable—i.e., whether $A'x = B'$ has a non-negative integral solution. If it does not, we fail and try different guesses. Otherwise, we show below that the original system was satisfiable, so the algorithm can terminate saying so.

The above procedure terminates in nondeterministic exponential time. We have guessed values for at most n variables, in at most n stages, where the values at stage k are represented with at most $b(cn \log n)^{k+1}$ for some constant c. We have thus guessed $O(2^n)$ many bits, and every number appearing in the sequence of linear subproblems explored is represented in at most $O(2^n)$ many bits. We have used a polynomial-time linear programming algorithm k at each stage to check for boundedness, and a nondeterministic polynomial time integer programming algorithm at the end to check consistency, all of which keeps us within nondeterministic exponential time.

Finally, we show that the algorithm's answer is correct. It is clear that if there is no sequence of guesses that give a consistent linear subproblem with all remaining

[2]Note that a variable x_i is bounded if and only if there is no solution to $Ax = 0$ over the non-negative rationals such that $x_i \neq 0$.

variables unbounded, then there can be no solution to the original problem—so when the algorithm fails there is in fact no solution. We now suppose that there is such a sequence of guesses, and show that there is a solution to the original problem. It is a fact of linear programming that since all the variables left are unbounded, there must be a non-negative rational solution α to $A'x = 0$ such that all components of α are non-zero.[40] We can assume without loss of generality that α is integral because any non-integral α can be made integral by multiplying by an appropriate constant. Take any non-negative integral solution β to $A'x = B'$ (we just showed that there is one) and the vector $\beta + c\alpha$ is a solution to the final linear problem for any c. For sufficiently large c this vector also solves the final non-linear problem, because the non-linear expressions must eventually grow faster than the linear expressions. But then $\beta + c\alpha$, augmented by the guessed values for the bounded variables, is a solution to the original semi-linear problem. Q. E. D. (Semi-Linear Decidability)

We use this result to complete the decidability proof for Tarskian set constraints as follows. Let Σ be a set constraint. When we refer to the *size* of Σ, written $|\Sigma|$, we will mean the number of distinct subexpressions appearing in Σ. We will assume for complexity analysis some fixed finite bound on the arity of function and relation symbols—higher arity functions and relations can be represented by using pairing functions. We non-deterministically reduce the problem of determining whether there is a model of Σ to the satisfiability of a semi-linear set of Diophantine inequations over a exponentially many variables relative to the size of Σ. Our reduction takes non-deterministic exponential time, and there will be a model of Σ just in case there is some execution of the reduction that yields a satisfiable semi-linear inequation set of exponential size. Since we have shown that satisfiability for such inequation sets is in non-deterministic exponential time, this places set constraint consistency in non-deterministic doubly exponential time.

The variables in our inequation set will represent the cardinalities of various subsets of the domain of a potential model of Σ. We will show that there is a solution to the inequation set if and only if there is a model of Σ whose domain gives the various described subsets the cardinalities assigned by the solution. We need Dio-

phantine variables for enough different subsets of the domain to capture every aspect of a model that bears on the truth of Σ. In what follows we will describe the Diophantine variables as though we had a particular model in mind that the variables are describing. As we introduce the variables, we will allow them to take on either any non-negative integral value or the value $*$ to represent the fact that the corresponding set is infinite. To complete the reduction we will at the end guess which variables actually take on the value $*$ and eliminate the inequations involving those variables to get an inequation set involving only non-negative integer variables.

First, we partition the domain of the model using the meanings of the subexpressions appearing in Σ. We call any subset σ of these subexpressions a Σ-*type*, and will say that a domain element of a model has Σ-type σ in that model if σ is exactly those subexpressions of Σ which contain the domain element in that model. It should be clear that every domain element has exactly one Σ-type. We will call a Σ-type σ *inhabited* in a model if there is a domain element d of type σ, and will say that d inhabits σ.

To facilitate the descriptions below we define the following abbreviation. We say that a relation or function symbol R *can map* the Σ-types $\sigma_1, \ldots, \sigma_n$ to the Σ-type τ whenever σ contains every subexpression $R(E'_1, \ldots, E'_n)$ of Σ such that each E'_i in the corresponding σ_i. This condition ensures that the members of τ do not trivially prohibit R from mapping domain elements of types $\sigma_1 \cdots \sigma_n$ to a domain element of type τ. $R(\sigma_1, \ldots, \sigma_n) \rightsquigarrow \tau$.

The first step in our reduction is to guess which Σ-types are inhabited—but in order for there to be a model of Σ consistent with our guesses, they must satisfy the following conditions. We say that a set I of Σ-types is *locally consistent* if:

1. **(Upward Closure Constraint)** Each Σ-type in I is upward closed under the subset statements in Σ; that is, if a Σ-type is in I and contains a set expression E then it also contains every set expression F such that $E \subseteq F$ is in Σ.

2. **(Completeness Constraint)** For each negation subexpression $\neg E$ in Σ, each type in I contains exactly one of the expressions E and $\neg E$.

3. **(Disjunction Resolution)** Every type in I contains a disjunctive set expression $(E_1 \cup E_2)$ *if and only if* it contains at least one of E_1 and E_2.

4. **(Predecessor Existence Constraint)** For each Σ-type τ in I containing a relation or function application $R(E_1, \ldots, E_n)$, there must be Σ-types $\sigma_1, \ldots, \sigma_n$ also in I containing $E_1, \ldots, E_n$, respectively, which R can map to τ.

5. **(Function Totality Constraint)**. For any function symbol f of arity n, any expression $f(E_1, \ldots, E_n)$ in Σ, and any Σ-types $\sigma_1, \ldots, \sigma_n$ in I containing $E_1, \ldots, E_n$ respectively, there must be a Σ-type σ in I that f can map $\sigma_1, \ldots, \sigma_n$ to.

By guessing a locally consistent set I of inhabited types, we have made a number of guesses exponential in the size of Σ—for each subset of Σ we have guessed whether or not it is inhabited. The existence of a locally consistent set of inhabited Σ-types is enough to ensure that Σ is satisfiable and thus has a model, whenever Σ does not contain any function symbols of arity zero (a model construction very similar to that in the proof below easily establishes this fact—the resulting model will have countably many domain objects of each inhabited type). However, the presence of constant functions in the language forces us to consider the cardinalities of various subsets of the domain. Consider for example the set constraint $\{ c_1 \subseteq \neg c_2,\ (c_1 \cup c_2) \subseteq f(c_3) \}$ where c_1, c_2, and c_3 are zero arity function applications. This constraint has a locally consistent set of inhabited types (the three types $\{c_3,\ \neg c_2\}$, $\{c_1,\ \neg c_2\ (c_1 \cup c_2),\ f(c_3)\}$, and $\{c_2,\ (c_1 \cup c_2),\ f(c_3)\}$) but is not satisfiable because $f(c_3)$ can contain only one element but must contain the distinct meanings of c_1 and c_2.

To recognize such inconsistencies we must reason about the *cardinalities* of the subsets of the domain inhabiting each Σ-type. We introduce a nonnegative integer variable z_σ for each inhabited Σ-type σ, representing the cardinality of the set of inhabitants of σ. We must also reason about the cardinalities of the images of the

functions applied to the Σ-types—the cardinality of the n-ary function f applied to types $\sigma_1, \ldots, \sigma_n$ cannot be larger than the product of the cardinalities z_{σ_1} thru z_{σ_n}.

To this end, we define the *range expressions for* f, for each n-ary function symbol f, to be the expressions of the form $f(\sigma_1, \ldots, \sigma_n)$, where $\sigma_1, \ldots, \sigma_n$ are any n inhabited Σ-types. We will say that a domain element d of a model is in a range expression $f(\sigma_1, \ldots, \sigma_n)$ in that model if there are some domain elements $d_1, \ldots, d_n$ of Σ-types $\sigma_1, \ldots, \sigma_n$ respectively such that d is in $f(d_1, \ldots, d_n)$. We will say that $f(\sigma_1, \ldots, \sigma_n)$ is *covered by* an application expression $f(E_1, \ldots, E_n)$ whenever each E_i is contained in the corresponding σ_i. We will say that a range expression is inhabited in a model when there is some domain element in the range expression in that model—note that a range expression is inhabited exactly if all of its argument types are inhabited. Finally, we say that a range expression $f(\sigma_1, \ldots, \sigma_n)$ can map to a type σ if f can map $\sigma_1, \ldots, \sigma_n$ to σ.

We introduce an additional nonnegative integer variable $z_{f(\sigma_1, \ldots, \sigma_n)}$ for each inhabited range expression $f(\sigma_1, \ldots, \sigma_n)$ representing the cardinality of the set of domain elements inhabiting the range expression.

Simply guessing cardinalities for the variables z_σ and $z_{f(\sigma_1, \ldots, \sigma_n)}$ will still not enable us to check the consistency of Σ. To see the remaining problem, consider the constraint $\{ c_1 \subseteq \neg c_2,\ f((c_3 \cup c_4)) = (c_1 \cup c_2) \}$ for zero arity function applications $c_1 \cdots c_4$. This constraint is trivially consistent, as f can map c_3 to c_1 and c_4 to c_2, for instance. However, if we enlarge the constraint by adding to it the two similar formulas $f((c_4 \cup c_5)) = (c_1 \cup c_2)$ and $f((c_3 \cup c_5)) = (c_1 \cup c_2)$, we no longer have a consistent constraint, even though the local cardinality conditions appear acceptable. The problem is that no mapping for f is consistent with all three constraints on f—f must map two of the three constants c_3, c_4, and c_5 to either c_1 or c_2 by the pigeonhole principle.

In order to detect this type of inconsistency, we must reason explicitly about the types of the predecessors implied by our cardinality guesses. In particular, consider a Σ-type σ which contains an application expression $f(E)$. For each domain element d inhabiting σ, there must be some predecessor domain element d' in $\mathcal{M}\ [\![E]\!]$ such that

$f(d') = d$. But what is the Σ-type of d? There may be many Σ-types containing E that f can map to σ—we must guess which one d' will inhabit. In general, for each domain member inhabiting σ and each application expression in σ, we must guess a range expression which is covered by the application expression and can map to σ. In the inconsistent constraint just given, it is not possible to make such predecessor type guesses for both c_1 and c_2 in such a way that the natural local cardinality constraints (elucidated below) are satisfied (some one of c_3, c_4, or c_5 will end up being asked to be a f-predecessor to both c_1 and c_2).

To formalize this reasoning, we introduce the notion of a "predecessor accounting. A *predecessor accounting* Δ for a Σ-type σ is a mapping from the application expressions in σ to range expressions covered by them that can map to σ. We say that Δ *charges* a range expression if that range expression is actually in the range of Δ. For any domain element d of type σ, we say that a given accounting Δ *accounts for* d if d inhabits all of the range expressions in the range of Δ. In a given model, many predecessor accountings can account for given domain element, but there must always be at least one that does. Intuitively, we will choose one primary accounting for each domain element and then introduce cardinality variables to count the domain elements each accounting is primary for. We can then write constraints to ensure that the range expressions charged in the accounting can be large enough to contain all the domain elements whose primary predecessor accountings charge them.

To this end we introduce a new nonnegative integer variable $x_{\sigma,\Delta}$ for each Σ-type σ and predecessor accounting Δ for σ, to represent the number of domain elements of the model of type σ whose primary accounting is Δ.

We are now ready to state Diophantine constraints on the new integer variables in an attempt to ensure the existence of a satisfying model. We define the *Tarskian consistency constraints* for a set constraint Σ and a locally consistent set of Σ-types I (those chosen as inhabited) to be the following constraints on the variables z_σ and ${}^z f(\sigma_1, \ldots, \sigma_n)$, and $x_{\sigma,\Delta}$ for $\sigma, \sigma_1, \ldots, \sigma_n$ all in I, and Δ a predecessor accounting for σ:

1. **(Habitation Constraints)** For every type σ in I, z_σ must be greater than or equal to one.

2. **(Account Limit Constraints)** For each inhabited range expression $f(\sigma_1, \ldots, \sigma_n)$, the value of $z_{f(\sigma_1, \ldots, \sigma_n)}$ must be as large as the sum of all $x_{\sigma,\Delta}$ for which Δ charges $f(\sigma_1, \ldots, \sigma_n)$.

3. **(Account Cover Constraints)** For every type σ in I, the sum over all predecessor accountings Δ for σ of the variables $x_{\sigma,\Delta}$ must be equal to z_σ.

4. **(Predecessor Constraint).** For any n-ary function symbol f and types $\tau_1, \ldots, \tau_n$, $z_{f(\tau_1, \ldots, \tau_n)}$ must not exceed the product of the cardinalities of the τ_i, or 1 if $n = 0$.

We note now that the Tarskian consistency constraints for a constraint Σ and locally consistent set of Σ-types I involve a number of variables at most exponential in the number of subexpressions of Σ. This can be seen by observing the following facts:

1. There are at most exponentially many subsets σ (and thus variables z_σ) of the set of all subexpressions of Σ.

2. For any particular Σ-type σ there clearly are at most linearly many members of σ.

3. Given our fixed bound on the arity of function symbols there are at most exponentially many range expressions (and thus variables $z_{f(\sigma_1, \ldots, \sigma_n)}$).

4. There are exponentially many functions from a linear sized domain to an exponential sized range (and thus exponentially many variables $z_{\sigma,\Delta}$).

To conclude our non-deterministic reduction we guess which of the variables we have introduced has the infinite value $*$. We require that this guess be consistent with the constraints just given—i.e., if there is an infinite variable on the small side of an inequality there must be one on the large side. Given this restriction on the guess, we

drop any constraints which mention infinite variables as trivially true, leaving us with a semi-linear set of Diophantine inequalities over a number of variables exponential in the size of Σ. Our earlier theorem shows that we can find a solution to these inequalities if one exists in nondeterministic exponential time (relative to the number of variables, which is itself exponential in the size of Σ).

We now show that there is a model of Σ if and only if there is a locally consistent set of Σ-types I such that the Tarskian consistency constraints for Σ and I are solvable.

Theorem 6.2 (Model Existence) *A set constraint Σ has a model if and only if there exist a locally consistent set of Σ-types I such that the Tarskian consistency constraints for Σ and I are satisfiable over the nonnegative integers plus the infinite value $*$.*

Proof: We first suppose that $\mathcal{M}$ is a model of Σ. The first step is to choose which Σ-types are inhabited. Let I be exactly those Σ-types which are inhabited in $\mathcal{M}$. Using the fact that this choice of I derives from a model, it is easy to check that the upward closure, completeness, disjunction resolution, predecessor existence, and function totality constraints upon our choice are satisfied, so that I is locally consistent.

We now give a solution to the inequation set, with the variable values ranging over the non-negative integers and the infinite value $*$. This solution includes a choice of which variables are infinite that is consistent with the constraints, and a finite solution to the constraints not mentioning infinite variables.

First, we assign each variable z_σ the number of domain elements of $\mathcal{M}$ that have Σ-type σ, or $*$ if there are infinitely many such elements. Then, for each function symbol f and each domain element d of $\mathcal{M}$, let σ be the Σ-type of d, and pick one predecessor accounting Δ of σ which accounts for d in $\mathcal{M}$ and call it the chosen predecessor accounting of d. Assign each variable $x_{\sigma,\Delta}$ to be the number of domain elements whose chosen predecessor accounting was Δ, or $*$ if there are infinitely many such elements.

These valuations for the variables give a solution to the constraints listed above, concluding this direction of the proof.

For the more difficult direction of the proof, suppose that we have guessed a locally consistent set I of types as habitable and solved the resulting set of Tarskian consistency constraints over the nonnegative integers plus $*$. We must build a model $\mathcal{M}$ of Σ.

The domain of our model will be the union over all inhabited types σ of sets $\{ \sigma^i \mid i \leq z_\sigma \}$ where for each type σ and each index i we assume that σ^i is a distinct object, and we treat the value $*$ of the variables that were guessed to be infinite as the first infinite cardinal ω.

We define $\mathcal{M}$ on each set constant P to be the set of all domain elements σ^i whose base type σ contains P. We define $\mathcal{M}$ on each relation symbol R of arity n to be the relation that maps each n-tuple $\langle (\sigma_1)^{(i_1)}, \ldots, (\sigma_n)^{(i_n)} \rangle$ to every domain element τ^j such that R can map $\sigma_1, \ldots, \sigma_n$ to τ.

Finally, we define $\mathcal{M}$ on the function symbols. For each type inhabited σ and each function symbol f, the variables $x_{\sigma,\Delta}$ for all Δ must sum to z_σ—therefore, we can choose a partition of the domain elements of base type σ into subsets $\sigma_[\Delta]$ of size $x_{\sigma,\Delta}$, respectively.

To define $\mathcal{M}$ on the n-ary function symbol f, consider an n-tuple of inhabited Σ-types $\langle \sigma_1, \ldots, \sigma_n \rangle$. We define f simultaneously on all tuples of domain elements whose base types are $\sigma_1, \ldots, \sigma_n$, respectively. First, construct an enumeration of such tuples, which we will call the *domain* enumeration. Next, construct a *range* enumeration of the domain elements that f is required to cover when applied to such tuples: that is, enumerate all domain elements that are members of any set $\sigma[\Delta]$ such that $f(\sigma_1, \ldots, \sigma_n)$ is charged by Δ.

If this range enumeration is empty, we need to find some element to map to: first, if there is any tuple of expressions $E_1, \ldots, E_n$, members of $\sigma_1, \ldots, \sigma_n$ respectively, such that $f(E_1, \ldots, E_n)$ is in Σ, then the function totality constraint guarantees us that there is some inhabited type τ that f can map $\sigma_1, \ldots, \sigma_n$ to, so we can take the range enumeration to consist of the single element τ^1 for any such τ; otherwise let

the range enumeration consist of any single domain element.

The predecessor and account limit constraints ensure us that the domain enumeration is at least as long as the range enumeration. Define f to map each tuple from the domain enumeration to the corresponding element of the range enumeration. If the range enumeration is shorter than the domain enumeration, simply extend it by repeating the last element until it is the same length.

This definition of f ensures that every member of a set $\sigma[\Delta]$ does in fact inhabit every range expression in Δ, which will in the proof below ensure that each such member in habits all the applications expressions in σ.

In order to prove that $\mathcal{M}$ is a model of Σ, we prove first by induction on the structure of the subexpressions of Σ that every domain element σ^i of $\mathcal{M}$ has Σ-type σ in $\mathcal{M}$. Once we have proven this, the upward closure constraint on our choice of inhabited types ensures that every subset formula in Σ is satisfied by $\mathcal{M}$.

Lemma: Every domain element σ^i of $\mathcal{M}$ has Σ-type σ in $\mathcal{M}$.

Proof: We prove by induction on the structure of the subexpressions E of Σ that the meaning of E in $\mathcal{M}$ is the set of domain elements of $\mathcal{M}$ whose base type contains E.

The lemma is true for set constants by the definition of $\mathcal{M}$.

Suppose E is the negation of an expression E_1 for which the lemma holds. We show that the lemma holds for E. Our induction hypothesis tells us that $\mathcal{M}$ $[\![E_1]\!]$ is the set of domain elements whose base Σ-types contain E_1—then of course $\mathcal{M}$ $[\![E]\!]$ is the complement of this set, i.e., the set of domain elements whose base types do not contain E_1. The completeness constraint on the set of inhabited types ensures that every inhabited type contains exactly one of E_1 and $\neg E_1$—so the set of domain elements which do not contain E_1 is exactly the set of domain elements whose base types contain $\neg E_1$, as desired.

Now suppose E is the union of two expressions E_1 and E_2 for which the lemma holds, and show that the lemma holds for E. Our induction hypothesis tells us that $\mathcal{M}$ $[\![E_1]\!]$ (and respectively, $\mathcal{M}$ $[\![E_2]\!]$) is just the set

of all domain members whose base type contains E_1 (respectively, E_2). So $\mathcal{M}\ ((E_1 \cup E_2))$ is the set of all domain members whose base type contains either E_1 or E_2. The upward closure and disjunction resolution constraints ensure that this is exactly the set of all domain members whose base type contains $(E_1 \cup E_2)$, as desired.

Now suppose E is the application of the n-ary relation symbol R to expressions $E_1, \ldots, E_n$ for which the lemma holds. We first show the forward direction—that every domain element whose base type contains $R(E_1, \ldots, E_n)$ is also in $\mathcal{M}\ [\![R(E_1, \ldots, E_n)]\!]$. Consider a domain element σ^k where σ that contains $R(E_1, \ldots, E_n)$. σ must be among the Σ-types we selected as inhabited. Then, by the predecessor existence constraint, we must also have selected as inhabited Σ-types some $\sigma_1, \ldots, \sigma_n$ containing the $E_1, \ldots, E_n$, respectively, that R can map to σ. But then by definition $\mathcal{M}\ [\![R]\!]$ maps $\langle (\sigma_1)^1, \ldots, (\sigma_n)^1 \rangle$ to d. But by our induction hypothesis, each element ${\sigma_i}^1$ must be in the corresponding $\mathcal{M}\ [\![E_i]\!]$, so σ^k must be in $\mathcal{M}\ [\![R(E_1, \ldots, E_n)]\!]$, as desired.

For the reverse direction, suppose that a domain element σ^k is in $\mathcal{M}\ [\![R(E_1, \ldots, E_n)]\!]$, and show that $R(E_1, \ldots, E_n)$ is in σ. Because σ^k is in $\mathcal{M}\ [\![R(E_1, \ldots, E_n)]\!]$, there must be domain elements $(\sigma_1)^{(i_1)}, \ldots, (\sigma_n)^{(i_n)}$ in $\mathcal{M}\ [\![E_1]\!], \ldots, \mathcal{M}\ [\![E_n]\!]$ respectively, such that R can map $\sigma_1, \ldots, \sigma_n$ to σ. But, by our induction hypothesis, since each $(\sigma_j)^{(i_j)}$ is in $\mathcal{M}\ [\![E_j]\!]$ we have that each σ_j contains E_j. Then, by the meaning of "can map to σ", we have that σ contains $R(E_1, \ldots, E_n)$, as desired.

The final case to consider is when E is the application $f(E_1, \ldots, E_n)$ of an n-ary function symbol f to n expressions $E_1, \ldots, E_n$ for which the lemma holds. Again, we first consider the forward direction—we suppose σ^k is a domain element in $\mathcal{M}\ [\![f(E_1, \ldots, E_n)]\!]$ and show that σ contains $f(E_1, \ldots, E_n)$. Since σ^k is in $\mathcal{M}\ [\![f(E_1, \ldots, E_n)]\!]$, it must be the image under f of some tuple of domain elements $(\sigma_1)^{(i_1)}, \ldots, (\sigma_n)^{(i_n)}$ which are members, respectively, of $\mathcal{M}\ (E_1), \ldots, \mathcal{M}\ (E_n)$. Our induction hypothesis

then implies that for each j, E_j is a member of σ_j. Then by our definition of $\mathcal{M}$, when f was defined for tuples from $\sigma_1, \ldots, \sigma_n$, σ^k must have been in the range enumeration. There are two ways that this can happen. First, σ^k could be an element of $\sigma[\Delta]$ for some Δ containing $f(\sigma_1, \ldots, \sigma_n)$. In this case, f can map $\sigma_1, \ldots, \sigma_n$ to σ (since Δ is a predecessor accounting for σ). But then, since each E_i is in the corresponding σ_i, $f(E_1, \ldots, E_n)$ must be in σ as desired. Second, if all $\sigma[\Delta]$ for Δ containing $f(E_1, \ldots, E_n)$ are empty, σ^k must be τ^1 for some τ that f can map $\sigma_1, \ldots, \sigma_n$ to—this once again implies that $f(E_1, \ldots, E_n)$ is in σ (which is τ) as desired.

It remains to show the reverse direction: we take an arbitrary domain element σ^k such that σ contains $f(E_1, \ldots, E_n)$ and show that this σ^k is a member of $\mathcal{M}\ [\![f(E_1, \ldots, E_n)]\!]$. σ must be among the Σ-types in I, therefore z_σ must be non-zero by the habitation constraint. The account cover constraint then ensures us that there is some predecessor accounting Δ such that $x_{\sigma,\Delta}$ is non-zero, and thus that $\sigma[\Delta]$ is non-empty. Since σ contains $f(E_1, \ldots, E_n)$, $\Delta(f(E_1, \ldots, E_n))$ must be a range expression $f(\sigma_1, \ldots, \sigma_n)$ that is covered by $f(E_1, \ldots, E_n)$ and can map to σ—i.e., such that $E_1, \ldots, E_n$ are members of $\sigma_1, \ldots, \sigma_n$ respectively, and f can map $\sigma_1, \ldots, \sigma_n$ to σ. But then, by our inductive hypothesis about the E_i, every tuple of domain elements of respective base types $\sigma_1, \ldots, \sigma_n$ must belong, respectively, to $\mathcal{M}\ [\![E_1]\!], \ldots, \mathcal{M}\ [\![E_n]\!]$. This is the entire domain enumeration for f on $\sigma_1, \ldots, \sigma_n$, so the entire range enumeration for f on $\sigma_1, \ldots, \sigma_n$ must be contained in $\mathcal{M}\ [\![f(E_1, \ldots, E_n)]\!]$. But this range enumeration must include σ^k since it is a member of $\sigma[\Delta]$ and Δ charges $f(\sigma_1, \ldots, \sigma_n)$. Q. E. D. (Model Existence Lemma)

This concludes the proof of the our model existence theorem, and with it the proof of our main result that Tarskian set constraints are in non-deterministic doubly exponential time:

Theorem 6.3 (Tarskian Set Constraints Decidability Theorem)
The problem of deciding the consistency of a Tarskian set constraint Σ is in non-deterministic doubly exponential time in the number of subexpressions of Σ.

Chapter 7

Conclusion

In this thesis we have argued against the practice of limiting the expressiveness of type systems in programming languages. We believe that complete type inference for an inexpressive type system is less valuable than the achievable incomplete type inference for an expressive type system. We have presented a particular expressive type system, and a particular incomplete inference algorithm for inferring types for programs in this expressive system. We argue the usefulness of our algorithm by giving examples of programs that it can find interesting types for—types that cannot even be expressed in traditional type systems of limited expressiveness.

The type language we presented is based on the use of nondeterminism to represent sets of values. This use of nondeterminism, inspired by the work of David McAllester on the Ontic verification system, allows the natural use of the programming language to represent types, and makes it straightforward to collapse the syntactic distinction between term and type. This collapse allows the same inference mechanisms to reason both about program definitions and type definitions. Reasoning directly about types is inherently more powerful than reasoning only about single objects themselves, leading to more effective inference.[27, 28] In chapter 5 we discussed some techniques for enhancing the reasoning mechanism to increase its effectiveness in reasoning about nondeterministic expressions.

The algorithm presented is effective in the sense that it runs in polynomial time. It is similar to traditional type inference algorithms in that it is based on a syntax-

directed set of typing rules. These rules do a recursive descent into the expression being analyzed. We apply a powerful forward-chaining inference engine at each level of the recursive descent. The overall algorithm can be viewed as an abstract interpretation algorithm with a particularly rich class of information being generated at each level of the interpretation. Our algorithm discovers such type information as that a simple insertion sort program always returns a permutation of its input.

A skeptical interpreter of this work might object that the algorithm may be targeted explicitly to the handful of examples given, since no wider survey of its applicability has been done yet. We believe that the simplicity of the underlying inference rules constitutes argues against this interpretation. Each inference rule captures a simple local property of the language syntax rather than any more complex problem specific fact. Each of the examples given uses the vast majority of the inference rules.

The inference rules given were selected from a much larger set of candidate inference rules—the selections were to some degree motivated by the target examples. However, even adding the entire larger set of inference rules would leave the algorithm in polynomial-time. It is likely that consideration of a very wide class of examples would make clear that a somewhat larger set of (polynomial-time) inference rules is needed. We believe that this enlargement upon consideration of new examples will quiesce, converging upon a widely useful polynomial-time inference relation much like that presented here.

Another objection from the skeptics might be that the polynomial-time complexity of the inference algorithm is small comfort when the polynomial is worse than quadratic, and the problems being considered may be quite large. In particular, we wish to be able to consider target programs in the context of a large library of knowledge, much of which has nothing to do with the target programs. In answer to this objection, we point out that the polynomial-time bound is a worst case bound, and that in particular that worst case assumes that all of the knowledge library is being instantiated for every target program expression. In practice, we expect that our organization of instantiation around the types of the quantified variables will limit such instantiation—only theorems about relevant types will be instantiated on a program

expression. Informally, this type restriction appears to result in an "average" case complexity that is polynomial in the logarithm of the library size.

Yet another objection that must be considered stems from the observation that the examples appear to depend on the exact choice of representation for the specifications computed. For example, if we had defined `a-permutation-of` using bijections between list-members, it is not clear that the algorithm would have computed the same results in analyzing `insert`. This objection is critical—we believe that how knowledge is represented has an immediate strong effect on what can be quickly inferred from it. It is then no surprise that our algorithm reflect just such a dependence. Users wishing to write less constructive definitions of key properties may well have to take the time to prove them equivalent to the more useful constructive definitions. While we do not give any algorithm for recognizing which forms of definition will prove most useful, it is apparent that specification definitions that are constructed as nondeterministic programs for computing values of the specification being defined. We wrote the specification definitions used in this thesis without conscious attention to making them useful for the algorithm.

Our inference algorithm also provides a polynomial-time automatic induction procedure. We believe that many useful inductive theorems that seem "obvious" to human programmers can be discovered by automated induction procedures that guarantee quick termination. While this work gives an example of such a procedure, we believe that significant improvement can be made in the power of the automated induction provided here by incorporating induction into the forward-chaining inference procedure.[30]

Although we believe that the most significant inference complexity in type inference is present in typing first-order languages, we have also shown how to extend our programming language and inference algorithm to a higher order language, and argued that our techniques generalize to that case. From our vantage point, the most interesting part of the higher order presentation lies in the set-theory based semantics and the ability of the language to naturally capture all of representation and inference principles of Zermello-Fraenkel set theory.

We also presented a separate piece of research in the area of set constraints. We focused on a previously neglected type of set constraint involving function symbols without standard interpretations (uninterpreted function symbols, or Tarskian function symbols). We showed that consistency for a simple set constraint language involving these function symbols is decidable in nondeterministic doubly exponential time.[1] Along the way we defined a new type of Diophantine solvability problem which we called solvability of semi-linear Diophantine inequations. We showed this problem to be solvable in nondeterministic exponential time, but conjecture it to be in $\mathcal{NP}$.

Although we made no attempt to present an efficient practical algorithm for checking the consistency of Tarskian set constraints, we believe that such an algorithm exists. Previous related set constraints work (e.g. on Herbrand set constraints[2]) has found practical algorithms in cases where the worst case complexity was similarly unattractive. We believe that an effective Tarskian set constraints based algorithm could be usefully integrated into a modern type inference system.

Finally, to conclude we mention several directions for future research in this area (including some already mentioned):

- Find and implement an effective consistency checking algorithm for Tarskian set constraints and incorporate it into a type inference system.
- Show the solvability problem for semi-linear Diophantine inequations to be in $\mathcal{NP}$ (and get a tighter complexity bound on Tarskian set constraints).
- Incorporate inductive inference principles into a polynomial-time decidable forward chaining inference system like the one presented above for type inference.
- Incorporate expressive natural language features into the specification language presented here to increase the effectiveness of the underlying inference procedures.[27]

[1] Aiken, Wimmers, and Lakshman[2] have given a decision procedure which is useful in practice for the related Herbrand set constraints satisfiability problem, which has a similarly bad worst case complexity.

Bibliography

[1] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Limited, 1987.

[2] A. Aiken, E. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *ACM Symposium on Principles of Programming Languges*, pages 163–173, 1994.

[3] Alexander Aiken, Dexter Kozen, Moshe Vardi, and Edward Wimmers. The complexity of set constraints. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Proc. 1993 Conf. Computer Science Logic (CSL'93)*, volume 832 of *Lect. Notes in Comput. Sci.*, pages 1–17. Eur. Assoc. Comput. Sci. Logic, Springer, September 1993.

[4] Alexander Aiken, Dexter Kozen, and Edward Wimmers. Decidability of systems of set constraints with negative constraints. *Infor. and Comput.*, 1994. To appear. Also Cornell University Tech. Report 93-1362, June, 1993.

[5] Alexander Aiken and Edward Wimmers. Type inference with set constraints. Research Report 8956, IBM, 1992.

[6] Leo Bachmair, Herald Ganzinger, and Uwe Waldmann. Set constraints are the monadic class. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 75–83, Montreal, Canada, 19–23 June 1993. IEEE Computer Society Press.

[7] I. Borosh and L. B. Treybig. Bounds on positive integral solutions of linear diophantine equations. *Proceedings of the American Mathematical Society*, 55:299–304, 1976.

[8] Robert S. Boyer and J Struther Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.

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

[10] J. A. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10:19–35, 1980.

[11] W. Charatonik and L. Pacholski. Negative set constraints with equality. In *Proc. 9th Symp. Logic in Computer Science*, pages 128–136. IEEE, July 1994.

[12] W. Charatonik and L. Pacholski. Set constraints with projections are in nexptime. In *Proceedings of the 35th Foundations of Computer Science*, pages 642–653. IEEE, November 1994.

[13] Wilfred Chen. Tactic-based theorem proving and knowledge-based forward chaining: an experiment with nuprl and ontic. In *CADE-11*, pages 552–566. Springer-Verlag, June 1992.

[14] R. L. Constable et. al. *Implementing Mathematics with the Nuprl Development system*. Prentice-Hall, 1986.

[15] Francesco Donini, Maurizio Lenzerini, Daniele Nardi, and Werner Nutt. The complexity of concept languages. In *Proceedings of KR91*, pages 151–162. Morgan Kaufmann Publishers, 1991.

[16] Thom Frühwirth, Ehud Shapiro, Moshe Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 75–83. IEEE Computer Society Press, 1991.

[17] Robert Givan and David McAllester. New results on local inference relations. In *Principles of Knolwedge Representation and Reasoning: Proceedings of the Third International Conference*, pages 403–412. Morgan Kaufman Press, October 1992. internet file ftp.ai.mit.edu:/pub/users/dam/kr92.ps.

[18] M. Gordon and T. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[19] Michael Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Springer-Verlag, 1979. Volume 78 of Lecture Notes in Computer Science.

[20] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51. IEEE Computer Society Press, 1990.

[21] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *ACM Symposium on Principles of Programming Languges*, pages 197–209. Association for Computing Machinery, 1990.

[22] B. Jònsson and A. Tarski. Boolean algebras with operators. part i. *Amer. J. Math*, 73:891–939, 1951.

[23] B. Jònsson and A. Tarski. Boolean algebras with operators. part ii. *Amer. J. Math*, 74:127–167, 1952.

[24] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of ACM Conference on Principles of Programming Languages*, 1991.

[25] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[26] D. McAllester and K. Arkoudas. Walther recursion. Submitted to CADE-13, available at http://www.ai.mit.edu/people/dam/termination.html, 1996.

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

[28] 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.

[29] D. McAllester, R. Givan, D. Kozen, and C. Witty. Tarskian set constraints. In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, Rutgers, NJ, USA, 1996. IEEE Computer Society Press.

[30] David McAllester. Some observations on cognitive judgements. In *AAAI-91*, pages 910,915. Morgan Kaufmann Publishers, July 1991. internet file ftp.ai.mit.edu:/pub/users/dam/aaai91a.ps.

[31] David McAllester. On the declarative value of nondeterminism. Internet File ftp.ai.mit.edu:/pub/users/dam/ontic-spec.ps, 1993.

[32] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programing and Formal Systems*. North-Holland, 1967.

[33] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[34] Robin Milner. Type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[35] 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.

[36] Bernhard Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43:235–249, 1990.

[37] M. O. Rabin. Decidability of second order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[38] John C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation*. Springer-Verlag, 1974.

[39] M. Schmidt-Schaub and G. Smalka. Attributive concept descriptions with complements. *Artificial Intelligence*, 47:1–26, 1991.

[40] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley, 1986.

[41] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B, Formal Methods and Semantics,* pp. 133-164. MIT Press, 1990.

[42] Javier Thayer William Farmer, Joshua Guttman. Imps: An interactive mathematical proof system. In *CADE-10*, pages 653–654. Springer-Verlag, 1990.