The Nimble Type Inferencer for Common Lisp-84

HENRY G. BAKER

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663

We describe a framework and an algorithm for doing *type inference* analysis on programs written in full Common Lisp-84 (Common Lisp without the CLOS object-oriented extensions). The objective of type inference is to determine tight lattice upper bounds on the range of runtime data types for Common Lisp program variables and temporaries. Depending upon the lattice used, type inference can also provide range analysis information for numeric variables. This lattice upper bound information can be used by an optimizing compiler to choose more restrictive, and hence more efficient, representations for these program variables. Our analysis also produces tighter *control flow* information, which can be used to eliminate redundant tests which result in *dead code*. The overall goal of type inference is to mechanically extract from Common Lisp programs the same degree of representation information that is usually provided by the programmer in traditional strongly-typed languages. In this way, we can provide some classes of Common Lisp programs execution time efficiency expected only for more strongly-typed compiled languages.

The Nimble type inference system follows the traditional lattice/algebraic *data flow* techniques [Kaplan80], rather than the logical/theorem-proving *unification* techniques of ML [Milner78]. It can handle *polymorphic variables and functions* in a natural way, and provides for "case-based" analysis that is quite similar to that used intuitively by programmers. Additionally, this inference system can deduce the termination of some simple loops, thus providing surprisingly tight upper lattice bounds for many loop variables.

By using a higher resolution lattice, more precise typing of primitive functions, polymorphic types and case analysis, the Nimble type inference algorithm can often produce sharper bounds than unification-based type inference techniques. At the present time, however, our treatment of higher-order data structures and functions is not as elegant as that of the unification techniques.

Categories and Subject Descriptors:

General Terms: compilers, data types, lattices, Boolean algebras, dataflow, static analysis, polymorphism.

Additional Key Words and Phrases: Common Lisp, ML, type inference, interpreted language, compiled language.

---

1. INTRODUCTION

High-level programming languages can be grouped into two camps—the *compiled* languages such as Fortran, Algol, Pascal, Ada and C—and the *interpreted* languages such as Lisp, APL, and Smalltalk. The compiled languages put great emphasis on high levels of compile-time type checking and execution speed, while the interpreted languages put great emphasis on run-time type flexibility and speed of program development. As might have been expected, each camp has worked hard to add the advantages of the other camp to its own inherent advantages. Interpreted languages have developed sophisticated compilers for higher execution speed and safety, while compiled languages have developed interpreters and incremental compilers for faster program development and more sophisticated typing systems for more flexible programs and data structures.

Type inferencing is a technique by which the interpreted languages move towards the goal of achieving the safety and execution speed of traditional compiled languages, while preserving the flexibility of runtime data typing. Type inferencing is the mechanical extraction of "type declarations" from a program that has not provided this information, which is required of the programmer by traditional compiled languages. An optimizing compiler can then use this more precise type information to generate more efficient code.

This paper describes a type inferencer we have developed to test some of the algorithmic limits of the whole notion of type inferencing. We have chosen to test our approach using Common Lisp, as it is a standardized, dynamically-typed language which already contains the infrastructure—a type description language and optional type declarations—that is needed to support type inference. These type declarations are then used by the type inferencer to demonstrate its typing of a program. To date, we have focussed on extracting the best type information possible using algorithmic methods, while giving less emphasis to computational efficiency of these methods.

This paper is structured to provide both the background and approach of the type inference system being developed:
- Section 2 describes the nature of "type inference"
- Section 3 describes the goals of the Nimble type inference system
- Section 4 shows the capabilities of our algorithm through a number of examples
- Section 5 describes the Kaplan-Ullman type inference algorithm, from which ours evolved
- Section 6 describes the Nimble type inference algorithm
- Section 7 shows an example of the Nimble type inference algorithm in operation
- Section 8 briefly analyzes the complexity of the Nimble algorithm
- Section 9 discusses previous work on type inference, both for other languages and Lisp
- Section 10 concludes.

2. TYPE INFERENCE

To begin, we need to construct a theoretical framework and background for performing "type inference" on Common Lisp programs. We call the process of automatically determining the datatype of objects "type *inference*" rather than "type *checking*", since Common Lisp already has a well-developed dynamic type system in which types are computed and checked at run-time. However, to improve the efficiency of Lisp programs which have been compiled for execution on modern RISC architectures without type-checking hardware, more information is required at compile time. We need to identify during compilation those variables which will not utilize the full range of Lisp runtime datatypes, and give them specialized representations which can be dealt with more efficiently. This process is called *representation analysis*. Representation analysis has quite different goals from the problem of checking type safety at compile time, which we will call *compile time type checking*. However, these two problems are intimately related, and maximal run-time speed is achieved when the results of both kinds of analyses are available.

There has been some confusion about the difference between *type checking* for the purposes of *compiling* traditional languages, and *type checking* for the purposes of ensuring a program's *correctness*. While the redundancy which results from incorporating type declarations enhances the possibility of detecting semantic errors at compile time, this redundancy is not the best kind for that purpose. The goal of strong typing declarations in compiled languages is the efficient mapping of the program onto hardware datatypes, yet hardware datatypes may carry little semantic meaning for the programmer. For detecting semantic errors, the most powerful kinds of redundancy are provided by *abstract data types* and constructs such as *assert*. Abstract data types model the semantic intent of the programmer with respect to individual variable values, so that global properties of these individual values (e.g., *evenness* or *primeness* of an integer value) are maintained. The *assert* construct allows for the specification of complex *relationships among* several variables. However, since we are interested in improving run-time efficiency, we will assume that the program is already semantically correct, and will therefore concern ourselves only with the determination of tight lattice bounds on the values of variables.

Performing type inference requires proving many small theorems about programs, and therefore runs the risk of being confused with the more difficult task of theorem-proving for the purpose of proving programs correct relative to some external criteria. While some of the techniques may be similar to both tasks, the goals are completely different. For example, it is considered acceptable and routine for correctness provers to interact with a programmer when reasoning about the correctness of his code, but these interactions are not appropriate in a type inferencer. The whole point of type inference is to prove these small theorems and insert type declarations *mechanically*, since the *raison d'etre* of typeless languages is to eliminate unnecessary redundancies—e.g., declarations—from programs which clutter up the code and reduce productivity in program development and maintenance. Thus, if a type inferencer cannot routinely prove a certain class of theorems without human interaction, then its use as a productivity-enhancing tool will be severely limited.

We have chosen to perform static type inference on programs in the dynamically-typed Common Lisp programming language [CLtL84]. Common Lisp-84 has a reasonably complex type system. This system involves 42 simple type specifiers, 4 standard type operators, and 21 type specialization forms. Common Lisp has the usual primitive integer and floating point types, characters, the traditional *atoms* and *conses* of Lisp, vectors, arrays, and strings. In addition, Common Lisp has a host of non-traditional datatypes, such as *hash tables*, *readtables*, and other special purpose types. The datatype system can also be extended through user-defined *structures*, which are analogous to *structures* in C and to *records* in Pascal and Ada.

Common Lisp functions, unlike functions in traditional compiled languages, are inherently *polymorphic*. A function may accept arguments of *any* datatype, and perform operations on them without restriction. Of course, many built-in functions will generate runtime errors if not given arguments of the proper types. The Lisp function +, for example, will complain if it is not given numbers to add. However, the programmer is under no obligation to restrict his own functions in such a simple manner. For example, he is free to define his own + function in which numeric arguments are summed in the normal fashion, but non-numeric arguments are coerced into their "print length" before summing.

Traditional compiled languages such as Algol, Pascal, C and Ada cannot offer this level of flexibility, because they *must* determine a unique datatype for every value at compile time. This is because their built-in operations cannot accept more than one kind of datatype representation. While this level of specificity regarding datatypes allows for great efficiency in the datatypes and operations that they *do* support, these languages often cripple the programmer by forcing him to program in styles which may not be the most natural for the problem. These styles of programming may sometimes be *less efficient* than a style based on dynamic data types.

There has been some significant progress in incorporating more complex type specification and type checking into compiled languages. The most developed (in the sense of actual implementation) of these languages is *ML* [Milner78, Harper86]. ML allows for parameterized datatype specifications by means of *type variables*, which can stand for "any actual type". These variables are automatically instantiated during the type checking process, allowing for a much more elegant solution to the problem of polymorphic functions such as *length* (*length* produces the length of any list, regardless of the type of its elements). In Pascal, a different *length* procedure would be required for each type of list element. In Ada, it is possible to state a generic *length* function, but this generic function is not a true function, but only a template from which a true function must be instantiated before it can be called [AdaLRM].

The ML compiler can handle the typing problem of *length* by introducing a type variable (denoted by `'`<type-variable-name>) which stand for *all* actual types, so that arguments to *length* can be specified more generically as `list('A)` instead of `list(integer)` or `list(character)`. The ML compiler itself then determines the appropriate actual type to substitute for the type variable in every instance where *length* is used. However, while ML allows for a parameterized type specification, <u>ML still requires that the datatype of every *instance* of the variable or procedure be resolved to a single actual type</u>. This is because ML is still a traditional strongly-typed language, without the dynamic run-time datatypes necessary to handle full polymorphism. Thus, through parameterized type specification, ML extends the power of traditional "compiled" languages in the direction of type polymorphism without giving up the level of efficiency on standard hardware expected of these languages.

The Nimble type inferencer, on the other hand, approaches the same goal of high efficiency on "standard" hardware, but from the opposite direction: instead of restricting the programming language itself to consist of only those constructs which can be efficiently compiled, the Nimble type inferencer identifies those *usages* of variables and functions within Common Lisp that can be efficiently compiled, and utilizes the traditional Lisp runtime datatype system as a fall-back strategy. Thus, the ML and Nimble type inferencing algorithms can be considered analogous in the sense that they are both looking for constructs that can be efficiently compiled. They differ, however, on what is done when no such constructs are found: ML considers this case an error, while the Nimble type inferencer accepts the program, but its subsequent execution will involve a higher level of run-time type checking than otherwise.

3. GOALS OF THE NIMBLE TYPE INFERENCE SYSTEM

The goals of the Nimble type inferencing algorithm are as follows:

- produce the most highly constrained set of datatypes possible for each variable *definition*, to allow for the efficient representation of the variable
- produce the most highly constrained set of datatypes possible for each variable *use*, to allow for the elimination of *redundant datatype checking* during execution

More succinctly, Nimble would like to:

- utilize hardware datatypes whenever possible
- eliminate as much redundant run-time type-checking as possible

These two goals are intimately related, and extremely important when compiling Common Lisp for a modern RISC architecture.

Utilizing hardware datatypes is important because some hardware datatypes are extremely efficient. For example, on the Intel i860 [Intel89], 32-bit 2's complement integers, 32-bit IEEE floating point numbers and 64-bit IEEE floating point numbers are all extremely efficient and extremely fast. However, in order to compile a Common Lisp program using hardware instructions for these datatypes, we must know that the variables and temporaries accessed by these instructions can take on *only* the appropriate types at run-time. In other words, we must be able to *prove* that any run-time type checks for those variables will *always* show the variables and values to be of the correct type; i.e., <u>we must prove that the run-time type checks are redundant</u>. Thus, the analysis necessary to eliminate redundant type checks is a prerequisite for the utilization of specialized representations.

What do we mean by redundant runtime type-checks? Consider the following Common Lisp recursive program for the factorial function:

```
(defun fact (n)
  (if (zerop n) 1
      (* n (fact (1- n)))))
```

During the execution of `(fact 5)`, we first must execute `(zerop 5)`, which first checks whether `5` is a number, and if so, whether it is zero. If `5` is not a number, then an error is signalled. Having determined that `5` is not zero, we then compute `(1- 5)`, which in turn checks again whether `5` is a number, and what type of number it is, so that it can perform the appropriate decrement operation, whether for integers, rationals, floating-point numbers, or complex numbers. However, since we have already checked `5` for numberhood, we have shown that the type check during the first part of the function `1-` is redundant. We then pass the result `5-1=4` back to `fact` for a recursive computation. However, we must once again check `4` for numberhood as part of the call to `zerop` in the recursive call, so there is a redundant type check within every recursive call at the entry to `zerop`.

A similar kind of redundant type checking occurs during the unwinding of the recursion. At the bottom of the recursion, `fact` returns `1`, which is a number, as well as an integer. Yet `*` will check again whether this result is a number, and what kind of number it is, before performing its computation. By induction, we know that `fact` will always return a number, if it returns at all, since `1` is a number, and `*` always returns a number. Hence the check for numberhood on entry to `*` inside `fact` is always redundant.

In modern RISC architectures type checks are expensive, because they involve conditional branching which is extremely expensive on a pipelined architecture due to the "pipeline turbulence" it causes. Pipeline turbulence is the inefficiency that results from idle time slots within the pipeline as a result of the conditional branch. *Error checks* of the sort described above can be implemented on pipelined architectures as conditional branches whose branch is almost never taken. The *type dispatch* operation, however, used to find the appropriate type of instruction to execute (integer, rational, floating-point, complex) almost always branches, but to different places at different times, and therefore almost always causes pipeline turbulence.

While there exist architectures and compilers whereby this sort of pipeline turbulence can be minimized (e.g., the *Trace* technology used in the Multiflow machine [Ellis86]), it still extracts a significant performance penalty. For example, on many RISC architectures, 32-bit integer operations are 4-6 times faster than conditional branches and hence to a first order approximation, the type checks take up *all* of the execution time!

4. THE CAPABILITIES OF THE NIMBLE TYPE INFERENCE SYSTEM

In this section, we show some of the capabilities of the Nimble algorithm to automatically infer tight lattice upper bounds on the datatypes of variables.

The most trivial of type inference algorithms should be capable of inferring the types of *constants*, whether they be constant objects, or constant built-in functions. Thus, even in the absence of additional information, the Nimble type inferencer (henceforth called "NTI") can *automatically* infer the following types in our simple recursive factorial program (automatically inferred information is shown in **boldface** type):

```
(defun fact (n)
  (declare (type number n))
  (if (zerop n) (the integer 1)
      (the number
           (* n (the number
                     (fact (the number
                                (1- n))))))))
```

Given the additional information that fact is only called from elsewhere with *integer* arguments, NTI can *automatically* infer the following more restrictive types:

```
(defun fact (n)
  (declare (type integer n))
  (if (zerop n) (the integer 1)
      (the integer
           (* n (the integer
                     (fact (the integer
                                (1- n))))))))
```

If fact is only called from elsewhere with *non-negative* integer arguments, NTI can *automatically* infer the even more restrictive types:

```
(defun fact (n)
  (declare (type nonnegative-integer n))
  (if (zerop n) (the positive-integer 1)
      (the positive-integer
           (* n (the positive-integer
                     (fact (the non-negative-integer
                                (1- n))))))))
```


This last example deserves some comment. The ability of NTI to predict that `n` will never become negative, given that `n` started as a positive integer, is a reasonably deep inference. Since this inference then allows NTI to conclude that the result of fact will always be *positive* shows a significant level of subtlety.

If `fact` is only called from elsewhere with "small" non-negative integers (i.e., non-negative *fixnums*, in Common Lisp parlance), NTI can *automatically* infer the following types:

```
(defun fact (n)
  (declare (type nonnegative-fixnum n))
  (if (zerop n) (the positive-fixnum 1)
      (the positive-integer
           (* n (the positive-integer
                     (fact (the non-negative-fixnum
                                (1- n)))))))))
```

This ability of NTI to automatically prove "fixnum-hood" for all recursive calls to fact is quite significant, because it allows a compiler to represent `n` as a 32-bit 2's complement integer, and utilize a hardware "decrement" operation to implement `(1- n)`.

4.1 INCORPORATING PROGRAMMER-SUPPLIED INFORMATION

In the factorial example above, NTI could determine that the *argument* to fact would always be a nonnegative fixnum, and could show that the *result* would be a positive integer, but no bound could be placed on the size of this integer. *We* know, of course, that the factorial function grows very rapidly, and that arguments greater than 12 will produce results greater than 2^{32}. However, the NTI algorithm will usually be incapable of inferring such deep mathematical results, and in these cases it will be necessary for the programmer to supply additional information to aid NTI in its analysis if he wants to obtain the most efficient code. The programmer can supply this information in several different ways.

The NTI algorithm relies on all data provided by the programmer, both from declarations and from actual use, so that the following programs all produce the same code (except for the wording of any error message):

```
(defun test1 (x y)
  (declare (integer x) (type (integer 0 *) y))         ; Info as declaration
  (expt x y))

(defun test2 (x y)
  (assert (integerp x))                              ; Info as assertions
  (assert (typep y '(integer 0 *)))
  (expt x y))

(defun test3 (x y)
  (if (and (integerp x)                              ; Info as conditional
           (integerp y)
           (not (minusp y)))
      (expt x y)
      (error "Bad types for ~S, ~S in test3" x y)))

(defun test4 (x y)
  (etypecase x                                       ; Info as typecase
    (integer
      (etypecase y
        ((integer 0 *) (expt y x))))))

(defun test5 (x y)
  (assert (or (zerop y) (plusp y)))                  ; Info as assertion
  (the integer (expt x y)))                          ; and declaration
```

(The last example is not necessarily true in all Common Lisps, but true in most, as it assumes that only rational arguments can produce rational results from the two-argument exponential function.)

4.2 GENERATING DECLARATIONS WITH THE NIMBLE TYPE INFERENCER

Common Lisp already has a syntactically well-specified mechanism for the programmer to declare his intentions regarding the range of values (and hopefully the representations) of variables and temporaries—the *declaration*. Declarations allow the programmer to specify to the compiler that more specialized and efficient representations can be used to speed up his program. In fact, the output of the Nimble type inferencer after its analysis of an input program is a copy of the input program with additional declarations inserted.

Unfortunately, the *meaning* of declarations is not well-specified in Common Lisp-84. While the Common Lisp standard [CLtL84,p.153] states that "declarations are completely optional and *correct* declarations do not affect the meaning of a correct program" (emphasis supplied), the meaning of "correct declaration" is not made clear, and the standard allows that "an implementation is not required to detect such errors" (where "such error" presumably means the violation of a declaration, i.e., "error" ≡ "incorrect declaration").

The Nimble type inferencer takes a slightly different point of view. NTI considers any programmer-supplied declarations to be a material part of the program. These declarations are treated as *constraining* the possible values which a variable or a temporary can assume, in much the same way that the "type" of an array in Common Lisp constrains the possible values which can be stored into the array. NTI enforces this "constraint" semantics of declarations by inserting dynamic type checks of its own when it cannot prove that the constraint will always be satisfied.

Since our declarations *do* constrain the values of variables and temporaries, they *can* change the meaning of a program, in the sense that a program with declarations can exhibit a smaller range of correct behavior than the same program with the declarations elided. Therefore, declarations can become a source of bugs, as well as a way to find bugs (assuming that a compiler or run-time system complains about declarations which actually constrain a program's meaning). The fact that declarations may actually introduce bugs into previously correct programs means that the traditional advice to "only add declarations to well-debugged programs" becomes circular and nonsensical!

However, if we advocate that programmers always decorate their programs with declarations, we risk losing something far more important than the correctness of previously debugged programs. We risk losing the advantages of polymorphism entirely. Lisp has traditionally been considered a *polymorphic* language, where functions can accept arguments of different types at different times, and produce different types as results. For example, consider a straight-forward implementation of the *arcsinh* function:

```
(defun asinh (z)
  (log (+ z (sqrt (1+ (* z z))))))
```

This function should work properly regardless of the argument type. `asinh` is *polymorphic*, and must return a floating-point result of the same format when given a floating-point argument, and must return a complex floating-point result of the same format when given a complex floating-point argument. This polymorphism can be implemented in one of two ways: *true polymorphism*, where the function can actually handle all of the possible types, and *overloading*, where any instance of the function works on only one type, and the particular instance is chosen at compile time. Overloading is more analogous to macro-expansion, except that the choice of which macro-expansion to use is dependent upon the type of the argument (and possibly upon the type of the expected result). Overloading is usually more efficient than true polymorphism, because no type dispatching need be performed at run-time.

By leaving the declarations *out* of a function definition, and instead inferring them when the argument and result types are already known, we can have reasonably generic function definitions. These generic functions can be stored in a library and compiled with the properly inferred types on demand. The alternative is precompiling a whole host of versions of the same function with differently declared types, and choosing the proper one at compile time through overloading techniques. The overloading technique is strongly advocated in the Ada language, but is only moderately successful due to the restrictions of that language, and due to the poor interaction of overloading with other Ada language features, such as separate compilation [AdaLRM]. Due to these problems, the default alternative supplied by most Common Lisp implementations is to endure the systematic inefficiency of truly polymorphic library routines.

The Nimble approach, on the other hand, allows for the above definition of `asinh` to act as more like an Ada *generic function* [AdaLRM], where each instance—whether in-line or out-of-line—will be compiled using only the functionality required by that instance. Thus, if a programmer only calls `asinh` with single-precision floating-point arguments in a particular program, then the `asinh` function compiled for that program will be specialized to work with only single-precision floating-point arguments and results. This ability to automatically specialize the standard Common Lisp libraries to the actual usage of a program is unique to the Nimble approach.

If one actually *wants* a truly polymorphic function which is also efficient—e.g., for a high-performance compiled library function—then one should utilize the following programming technique when using the Nimble type inferencer:

```
(defun asinh (z)
 (typecase z
  (single-float                    (log (+ z (sqrt (1+ (* z z))))))
  (short-float                     (log (+ z (sqrt (1+ (* z z))))))
  (double-float                    (log (+ z (sqrt (1+ (* z z))))))
  (long-float                      (log (+ z (sqrt (1+ (* z z))))))
  ((complex single-float)          (log (+ z (sqrt (1+ (* z z))))))
  ((complex short-float)           (log (+ z (sqrt (1+ (* z z))))))
  ((complex double-float)          (log (+ z (sqrt (1+ (* z z))))))
  ((complex long-float)            (log (+ z (sqrt (1+ (* z z))))))
  (fixnum                          (log (+ z (sqrt (1+ (* z z))))))
  (integer                         (log (+ z (sqrt (1+ (* z z))))))
  (rational                        (log (+ z (sqrt (1+ (* z z))))))
  ((complex rational)              (log (+ z (sqrt (1+ (* z z))))))
  (t  (log (+ z (sqrt (1+ (* z z))))))))
```

This version of `asinh` is more efficient than the previous one when compiled for all numeric types, because while the expressions within each arm are identical, they are compiled differently. Within each arm, `z` is inferred as having a different type, and hence different versions of `log`, `+`, `sqrt`, etc., are compiled within that arm. In the first arm, for example, `+`, `1+`, and `*` can be open-coded with single hardware instructions, and for some processors, `log` and `sqrt` have single hardware instructions, as well. (If the programmer also specifies in his calling program that this version of `asinh` should be expanded inline, then the Nimble algorithm will in most cases be able to eliminate the type check on the entry to the body of `asinh`, as well as the dead code from all the unused arms.)

The Nimble type inferencing algorithm allows the programmer to use standard *case-based programming* in addition to declarations to tell the compiler what to do. Consider, for example, the `abs` function:

```
(defun abs (z)
  (cond ((complexp z)
         (sqrt (+ (sqr (realpart z)) (sqr (imagpart z)))))
        ((minusp z) (- z))
        (t z)))
```

NTI is able to infer that within the first `cond` clause `z` is complex, hence numeric, hence `realpart` and `imagpart` are defined and cannot fail. Furthermore, those functions produce real results, so `sqr` produces non-negative real results, hence `sqrt` always produces a real result. Within the second `cond` clause, NTI can infer that `z` is a negative real, because `minusp` cannot be applied to other than real numbers, and it is true only for negative reals. NTI therefore concludes that the second clause produces a positive real result. It may be surprising, but NTI also infers that `z` is a non-negative real in the third `cond` clause even though it apparently could be any Lisp type other than complex and negative numbers. However, since `minusp` is not defined for non-numeric types, control will never reach the final clause unless `z` is a number, and since `minusp` cannot be applied to complex numbers and will return true for negative numbers, `z` must therefore be a non-negative real in the third clause. Putting all the clauses together, NTI concludes that the result from `abs` must always be a non-negative real number.

It is also instructive to see whether NTI can infer that the square function `sqr` always produces a non-negative result for non-complex arguments.

```
(defun sqr (z)
  (* z z))
```

Given this simple definition for `sqr`, NTI can only conclude that `z` is numeric, and cannot say anything about the sign of `z` when `z` is real. However, consider the following definition for the `sqr` function, which is actually more efficient for complex arguments:

```
(defun sqr (z)
  (cond ((complexp z)
         (let ((r (realpart z)) (i (imagpart z)))
           (complex (- (sqr r) (sqr i)) (* 2 r i))))
        ((minusp z) (* z z))
        (t (* z z))))
```


While the second and third clauses in `sqr` do not help in improving the efficiency of `sqr`, they do allow NTI to infer that `sqr` applied to real arguments produces a non-negative real result. This is because NTI can easily infer that negative*negative is positive and non-negative*non-negative is non-negative, and can put these facts together to infer that the result of `sqr` must always be non-negative for real arguments. (Note that while NTI required the programmer to split out the negative real and non-negative real cases in order to extract the most information, an optimizing "back-end" could later merge the two clauses back together, so that no efficiency is necessarily lost.)

The Nimble type inference algorithm is not only "case-based", but also "state-based". This means that NTI can properly type the following program:

```
(defun test ()
  (let ((x 0))
    (let ((y x))
      (setq x 0.0)
      (values x y))))
```

For this program, NTI will provide declarations along the lines of the following code:

```
(defun test ()
   (let ((x 0)) (declare (type (or fixnum single-float) x))
     (let ((y x)) (declare (fixnum y))
       (setq x (the single-float 0.0))
       (values (the single-float x) (the fixnum y)))))
```

The Nimble type inferencer can also handle functions with side-effects, without open-coding them:

```
(defun test-swap ()
  (let ((x 0) (y 0.0))
    (flet
      ((swap (&aux z) (setq z x x y y z)))
      (print (list x y))
      (swap)
      (values x y))))
```

NTI would annotate this example in a manner similar to that shown below:

```
(defun test-swap ()
  (let ((x 0) (y 0.0))
        (declare (type (or fixnum single-float) x y))
    (flet
       ((swap (&aux z) (declare (fixnum z))
          (setq z x x y y z)))
       (print (list (the fixnum x) (the single-float y)))
      (swap)
       (values (the single-float x) (the fixnum y)))))
```

As this example shows, the Nimble type inferencer is able to keep track of the different states of the variables `x` and `y` at different places in the program.

5. KAPLAN-ULLMAN TYPE INFERENCE

Since the Nimble type inference algorithm is derived from the Kaplan-Ullman type inference algorithm [Kaplan80], we will discuss the Kaplan-Ullman approach in detail. It should first be pointed out that the Kaplan-Ullman approach to type inference uses the same type of lattice techniques as does traditional *data flow* analysis in compilers [Aho86].

The Kaplan-Ullman algorithm utilizes a simple state-based model of computation. Within this model, a program consists of a finite number of variables, and a finite directed graph with nodes, each of which describes a simple computational step. The simple steps consist of assignment statements of the form z:=f(a,b,c,...), where a,b,c,...,z are program variables, and f(,,,...) is a predefined function. Constants are introduced as functions of zero arguments, and the flow of control is non-deterministic. The Kaplan-Ullman type inference algorithm seeks to find an assignment of variable names to datatypes in the datatype lattice which is *consistent* with the needs of the program, and is also *minimal*, in that any smaller assignment of datatypes is no longer consistent.

In the classical non-deterministic manner, the flow of control is terminated only by branches of the computation that *fail* in the sense that there are *no* legitimate values for variables. In this setting, *predicates* are modelled by partial functions whose results are ignored. One particularly valuable partial function of this kind is "assert(p)" which is defined only for the argument "true".

Even though the control flow of the Kaplan-Ullman model is non-deterministic, it can accurately model deterministic programs through the following mapping. An "if-then-else" statement can be modelled as follows:

```
(if (p x y) (setq z (f x y))
            (setq z (g x y)))
```

can be modelled as:

```
(alt (progn (assert      (p x y))                (setq z (f x y)))
     (progn (assert (not (p x y)))               (setq z (g x y))))
```

In this case, the construct `(alt A B)` means "perform either `A` or `B`", with the choice made non-deterministically, and following either arm describes a legitimate computation. In the specific usage above, only one (or none) of the arms will actually succeed, since it cannot happen that both *p* and *not p* are true at the same time. Therefore, while the `alt` construct is powerful enough to introduce true non-determinism, we will only need it under the controlled conditions given above.

The Kaplan-Ullman model introduces a *state* for each node in the graph which describes the best approximation to the actual datatypes of all of the program variables at that point in the program. A *state* can be thought of as a simple vector of components—one for each program variable—whose values are elements of the datatype lattice needed by the algorithm. We will actually use *two* states for each program point, since we will want to describe the state just *before* as well as the state just *after* the execution of a particular node.

The Kaplan-Ullman model then proceeds to model the program statements z:=f(a,b,c,...) in terms of their effect on these program states. Since we are utilizing the datatype lattice, rather than actual run-time values, we will be inferring the *datatype* z from knowledge of the *datatypes* a,b,c,... The Kaplan-Ullman model utilizes a *"t-function"*, which answers such questions in the most conservative way. Thus, if we have an approximation to the datatypes a,b,c,..., the t-function of f(,,,...) will allow us to produce an approximation to the datatype of z. Perhaps as importantly, the t-function of f(,,,...) will also allow us to approximate a,b,c,... given the datatype of z.

There is nothing paradoxical about these "t-functions". For example, if we know that the result of `(+ x y)` is an integer, then it must be the case that `x`,`y` were both rationals (given the standard policy of floating-point *contagion* in Common Lisp). While this information doesn't allow us to pin down the types of `x`,`y` more specifically to be integers (e.g., 3/4 + 1/4 = 1), we can usually get additional information regarding one or both of `x` and `y`, and thus pin down the types of all three quantities `x`,`y`, and `x+y` more exactly.

As an aside, it should be noted that this class of inferences is analogous to the *Bayesian analysis* of Markov processes [???], where the states of the program are identified with Markov states, and the transition functions are associated with transition probabilities. However, the Kaplan-Ullman model produces information about the range of *possibilities*, rather than *probabilities*. It may be possible to extend type inference analysis to perform probability analysis, given some method of producing *a priori* probabilities about frequency of execution. Such a probability analysis would enable a compiler to generate better code for the most probable cases.

Kaplan and Ullman call the propagation of information in the normal direction of computation (i.e., from arguments to function values) *forward inferencing*, while the propagation of information in the retrograde direction (i.e., from function values to argument values) *backward inferencing*. Thus, for every statement, we can compute a forward inference function which carries the "before-state" into the "after-state", as well as a backward inference function which carries the "after-state" into the "before-state". If we assign an index to every statement in the program, we can then construct a vector of states for the entire program, which then becomes a representation of the *flowstate* of the program.

This flowstate has little in common with the state of a program running on actual data. In the case of an executing program, the "state" of the program would be a program counter, holding the index of the currently executing program statement, and an environment vector, holding the current values of all of the program variables. Our data type "flowstate", on the other hand, consists of a vector of vectors whose elements are in the datatype lattice. The outer vector is indexed by program statements; the inner vectors are indexed by program variables. The reason for requiring a different state for each program statement is that the approximation to the datatype of a program variable at one particular program statement could be different from the datatype assumed at another program statement. This is especially true when *variable assignment* is involved, since the datatype of a program variable can then be different from one program statement to another.

Once we have represented the flowstate of an entire program in the above manner, we can now describe the forward and backward inferencing operations in terms of matrix multiplication operations. To do this, we construct a *diagonal* square matrix F of order `n`, where `n` is the number of program statements. The element F_{ii} is a function from states to states which describes the transition of the program statement `i` in going from a before-state to an after-state. The off-diagonal elements are constant functions yielding the bottom state, which assign to all program variables the bottom datatype.

We also construct a square matrix C of order n in which the element C_{ij} is a function from states to states which describes the connectivity of the program statements. C_{ij} is the identity function if program statement j immediately follows program statement i, and it is the constant function producing the bottom state otherwise. We will use a form of matrix multiplication in which the usual role of "+" is taken by the lattice *join* operation (here extended componentwise to vectors of elements), and the usual role of "*" is taken by functional composition.

Given a program flowstate S, which is a vector of states indexed by program statements, we can compute the effect of a single execution of each program statement by means of the matrix-vector product F•S. This product takes the before-flowstate S into an after-flowstate F•S. Similarly, the matrix-vector product C•S takes the after-flowstate S into a before-flowstate C•S. Thus, we can utilize the composition F•C•S to take an after-flowstate to another after-flowstate.

Kaplan and Ullman show that all of these operations are *monotonic* in the datatype lattice, and since this lattice must obey the finite chain condition, the following sequence has a limit for every S:

S, F•C•S, F•C•F•C•S, F•C•F•C•F•C•S, ...

In particular, the limit exists when S is the (vector of) bottom element(s) of the datatype lattice. This limit is one consistent datatype labeling of the program, although it may not be a *minimal* such labeling (in the datatype lattice ordering).

The previous limit can actually be taken with a finite computation. This computation will converge in a finite amount of time due to the finite chain condition of the original datatype lattice, and the finite numbers of program variables and program statements. The operation of the program can intuitively be described as assuming each program variable takes on the value of the bottom lattice element, and then propagating this assumption throughout the entire program. Whenever a statement of the form z:=C (wherein z is assigned a constant), the resulting after-state is forced to include the possibility of the datatype of the constant, and therefore this state can no longer assign z the bottom datatype. If a later assignment of the form x:=f(z) occurs, then the forward inferencing function will force x to include the datatype f(C) (as determined by the "t-function" for f), and so x will no longer be assigned the bottom datatype, either, if C is in the domain of f. In this fashion, the datatypes of all program variables at all program points are approximated from below until an assignment is reached that is consistent for all of the program statements. Within this computation, the assignment for a particular variable at a particular program point is always adjusted monotonically *upwards* from the bottom element of the lattice. (This datatype propagation process is analogous to the solution of a partial differential equation by relaxation, where the initial boundary conditions are propagated into the interior of the region until a consistent solution is obtained.)

It is obvious that this solution is consistent, but upon closer examination, we find that this solution is not necessarily minimal. The non-minimal solution arises because forward inferencing utilizes information in the same order that it occurs during normal program flow, and a more stringent constraint on the datatype of a variable may occur later in the program than the computation producing a value for the variable. Consider, for example, the sequence

x:=sqrt(y); if x>0 then A else B

In this sequence, x is constrained to be a "number" by virtue of being the result of a square root operation, but x is not necessarily a "non-complex number" until it is compared with 0 in the following step. Since complex numbers cannot be compared using the ">" operator, x must have been real, which means that y must have been non-negative. Thus, in order to compute a minimal datatype assignment for this sequence, a type inferencer must utilize "backward inferencing" in addition to "forward inferencing".

The Kaplan-Ullman algorithm therefore extends the matrix inferencing process to include backwards inferencing. To implement this, we define a backwards inferencing matrix B, which is diagonal and takes after-states into before-states. We now require the dual of the connection matrix C, where the dual matrix takes before-states into after states; this dual connection matrix is simply the *transpose* C^T of the forward connection matrix C. We then construct the infinite sequence

$$S,\ C^T\bullet B\bullet S,\ C^T\bullet B\bullet\ C^T\bullet B\bullet S,\ C^T\bullet B\bullet\ C^T\bullet B\bullet\ C^T\bullet B\bullet S,\ \ldots$$

This sequence is monotonic, and must also converge to a limit by the finite chain condition. This limit is a consistent approximation to the legal datatypes of the program variables, but is usually different from the limit of the forward inferencing chain. We could perform both of these sequence computations separately, followed by a lattice *meet* operation on the results to get a better approximation than either the forward or the backward inferencing computations could produce separately. The result of this complex computation, however, would still not be minimal.

To obtain a better result, Kaplan and Ullman utilize a technique for incorporating an *upper bound* for each state into the approximation process such that no intermediate state is allowed to violate the upper bound. This improved process can best be described as a small program:

S := bottom; **repeat** {oldS := S; S := (F•C•S) *meet* U} **until** S=oldS;

Not allowing the upper bound U to be violated at any point results in a tighter lower bound, but it is still not the tightest possible.

The final Kaplan-Ullman algorithm is a nested pair of loops. The inner loop is similar to the one above, in which the upper bound remains constant, while the lower bound runs from bottom up to convergence. The result of this inner process is a consistent legal datatype assignment, and no actual computation can stray outside its boundaries without producing an error. Thus, at the completion of a pass through this inner loop, we can take this lower bound as the new upper bound. Thus, the outer loop runs the inner loop using the lattice *top* element as an upper bound, then takes the resulting lower bound as the new upper bound, and begins the next pass of the inner loop. This process continues until the upper bound no longer changes, or equivalently, the greatest lower bound equals the least upper bound. In order to use all of the available information, Kaplan and Ullman suggest alternating forwards inferencing and backwards inferencing with each iteration of the outer loop.

/* Complete Kaplan-Ullman Type Inference Algorithm. */
/* U is upper bound, L is lower bound. */
U := top;
repeat { oldU := U;
L := bottom; **repeat** {oldL := L; L := (F•C•S) *meet* U} **until** L=oldL;
U := L;
L := bottom; **repeat** {oldL := L; L := (C^T•B•S) *meet* U} **until** L=oldL;
U := L; } **until** U=oldU;

Kaplan and Ullman prove that within a certain mathematical framework, their algorithm is optimal. They propose that their algorithm produces the *minimal* datatype assignment within the space of all datatype assignments which can be computed via any technique which utilizes a finite procedure in a lattice having the finite chain condition.

However, the Kaplan-Ullman inferencing algorithm does not produce the minimal assignment outside the stated framework. Consider the statement z:=x*x in a datatype lattice where negative and non-negative numbers are different datatypes. Let us further assume that there are alternative paths before this statement that assign to x both negative and non-negative numbers. Since the t-function for this statement must allow for all possibilities, it claims that {+,-}*{+,-}=>{+,-}; in other words, real multiplication can produce both positive and negative results, given arguments which are both positive and negative. Nevertheless, *we* know that x*x is always non-negative, no matter what the arguments are, so long as they are real. In this instance, the t-function for multiplication is not producing the sharpest possible information. This is because it does not notice that the two arguments to "*" are not only *dependent*, but identical.

A smarter t-function which detected duplicated argument variables would give better information in this instance, but this smarter t-function would be easily defeated by a simple transformation into the computationally equivalent sequence [y:=x; z:=x*y]. The problem arises from the approximation Kaplan and Ullman make in the representation of the states of the program variables. The approximation provides that only *rectangular* regions in the Cartesian product of datatypes can be represented as the states of the program variables; in other words, only a variable's own datatype can be represented, not the relationship between its datatype and that of another variable. As a result, the information produced by statements like z:=assert(x>y) is ignored, because there is no way to accurately represent the non-rectangular region "x>y" in our space of program variable states.

In the Nimble type inference algorithm, we make this same approximation in our representation of states as did Kaplan and Ullman, and for the same reason: it would take an extraordinarily complex representation and a very demanding calculation in order to keep track of all such relationships. As an indication of the complexity of this task, the "first-order theory of real addition with order" can result in a multiply-exponential time complexity for its decision procedure [Ferrante75].

Curiously, the statement [if x<0 then z:=x*x else z:=x*x] allows the Kaplan-Ullman algorithm to conclude that z is always positive! This result obtains because the programmer has forced the algorithm to perform case analysis which produces the intended result.

Several rules must be observed in order for a Kaplan-Ullman algorithm to work at all. Since the high-water-marks (greatest lower bounds) achieved during any lower-bound propagation will become the upper bounds for the next iteration, it is essential that we have found the *true* high-water-mark. This means that the lower-bound propagation must continue until there is *no* change, no matter how small. It is also important to see the *entire* program, since even a small piece of the program could introduce some new behavior, which would violate some high-water-mark achieved on some earlier cycle in the algorithm.

Since we are performing limit steps, the datatype lattice must obey the finite chain condition (i.e. it must be a *discrete* lattice [MacLane67]), else the algorithm may not converge. If the lattice is not discrete, then the limit steps may run forever. For example, the lattice described in our implementation of Common Lisp's `subtypep` predicate [Baker92] does *not* obey the finite chain condition because its representation allows individual integers to be represented, and the lattice of finite ranges of integers does not obey the finite chain condition.

6. THE NIMBLE TYPE INFERENCE ALGORITHM

The Nimble type inference (NTI) algorithm produces essentially the same information as would a "straight-forward" implementation of the Kaplan-Ullman algorithm, but it does so using a completely different representation and a completely different organization of its tasks. As a result, it is also much more efficient than a straight-forward Kaplan-Ullman implementation.

The NTI algorithm represents program states in a way which is much more tuned to the requirements of an expression-oriented language than a statement-oriented language. The NTI algorithm also carries explicit lower and upper bounds for each state. These lower and upper bound computations are performed simultaneously. While this technique does not improve the worst-case speed of the Kaplan-Ullman algorithm, it dramatically improves the performance in the common cases.

Consider the example: z:=f(x,y). The "before-state" for this statement has both a lower bound and an upper bound, between which any legal datatype assignment must lie. During forward inferencing, we can utilize the lower bounds of x and y to compute a (new) lower bound for z, and we can utilize the upper bounds of x and y to compute a (new) upper bound for z. Of course, these new bounds are still subject to previously determined lower and upper bounds for z. For this case, we have the following forward propagation rule for z:=f(x,y):

upper(z) = t-function(f,0,upper(x),upper(y)) meet upper(z)
lower(z) = t-function(f,0,lower(x),lower(y)) meet upper(z)

If we have a program point where two states split, as in the beginning of an `alt` (alternative) statement, then we propagate the lower and upper bounds differently from the above case. In this case, we have the following forward propagation rule for when A splits into B and C:

upper(B) = upper(A) meet upper(B)
upper(C) = upper(A) meet upper(C)
lower(B) = lower(A) meet upper(B)
lower(C) = lower(A) meet upper(C)

If we have a program point where two states join ("merge"), as in the end of an `alt` expression, then we propagate the lower and upper bounds using the forward propagation rule for B and C merging into A:

upper(A) = (upper(B) join upper(C)) meet upper(A)
lower(A) = (lower(B) join lower(C)) meet upper(A)

Backward propagation rules for the three cases above follow a similar pattern.

Backward propagation rule for z:=f(x,y):

upper(x) = t-function(f,1,upper(x),upper(y),upper(z)) meet upper(x)
upper(y) = t-function(f,2,upper(x),upper(y),upper(z)) meet upper(y)
upper(z) = upper(zbefore)

Backward propagation rule for split of A into B and C:

upper(A) = (upper(B) join upper(C)) meet upper(A)
lower(A) = (lower(B) join lower(C)) meet upper(A)

Backward propagation rule for join of B,C into A:

upper(B) = upper(A) meet upper(B)
upper(C) = upper(A) meet upper(C)
lower(B) = lower(A) meet upper(B)
lower(C) = lower(A) meet upper(C)

The proof of termination for our algorithm is somewhat less obvious than for the Kaplan-Ullman algorithm, since we modify both lower and upper bounds simultaneously. However, upper bounds still always decrease monotonically, as we always "meet" any new value with the old one. But the lower bounds do not necessarily monotonically increase, since a new upper bound can decrease a previous lower bound.

Nevertheless, our algorithm does converge. The proof goes as follows. The upper bound computations are completely independent of the lower bound computations; they monotonically decrease, and due to the finite chain condition on the datatype lattice, they still converge to a limit in a finite number of steps. Once the upper bounds stabilize, the lower bounds can, from that point on, only monotonically increase. The situation with stable upper bounds is identical to the situation in the original Kaplan-Ullman proof. Since the finite chain condition also applies to the lower bound computation, that case also converges, in a finite number of steps, to an assignment which is less than or equal to the stabilized upper bound.

Carrying both lower and upper bounds simultaneously does *not* relieve us of the necessity of performing backwards inferencing. Our previous example of

x:=sqrt(y);if x>0 then A else B

does not allow us to carry back the information about the realness of x into the previous statement, whence we can conclude that x is actually non-negative, and hence y is non-negative. However, the simultaneous computation of upper and lower bounds does improve the speed of convergence.

The question arises as to why an iterative limit step is required at all; i.e., why can't we produce the best solution with only a single pass through the program? The answer is that certain programs combined with certain datatype lattices require iteration. Consider the following simple program:

```
(labels
  ((foo (x)
     (if (zerop x) 0
         (1+ (foo (1- x))))))
  (foo 10000))
```

`foo` is an expensive identity function for non-negative integers which goes into an infinite recursion for any other numbers. Let us assume a datatype lattice having different datatypes for the five ranges: zero, positive integers less than 100, positive integers 100 or greater, negative integers greater than -100, and negative integers -100 or less. Using this datatype lattice, the first forward inferencing iteration of our algorithm will assign `x` the type $\{i|i\geq 100\}$, and the union of the types $\{i|0<i<100\}$and $\{i|i\geq 100\}$ (i.e., $\{i|i>0\}$) for the result of `(1- x)`. The second iteration will assign `x` the type $\{i|i>0\}$ (from the union of $\{i|i\geq 100\}$ and $\{i|i>0\}$), and assign to `(1- x)` the union of $\{0\}$ and $\{i|i>0\}$ (i.e., $\{i|i\geq 0\}$). The third iteration will assign `x` the type $\{i|i\geq 0\}$, but the conditional expression will separate the cases `x`=0 and `x`>0, so that `(1- x)` is again assigned the type $\{i|i\geq 0\}$.

Simultaneously with the above iteration to compute the datatype of `x`, the algorithm is also approximating the result of `(foo x)`. This result starts out $\{\}$, but then becomes $\{0\}$ as a result of the first arm of the conditional. The second iteration assigns the result the union of $\{0\}$ and $\{i|0<i<100\}$ (i.e., $\{i|0\leq i<100\}$). The third and subsequent iterations assign the result $\{i|i\geq 0\}$.

Thus, the type inference algorithm applied to this program infers the constraints `x`≥0 and `(foo x)`≥0. Note that the algorithm cannot conclude that `(foo x)`=`x` or even that `(foo 10000)`>100, but it is still remarkable that the algorithm *does* conclude that `x` does *not* go negative, and that `(foo x)` also does not go negative!

From the above example, it should be obvious that iterations are necessary in the type inference algorithm in order to properly handle the limit operation on looping and recursive constructs. These constructs could cause NTI to work its way up from the bottom of the datatype lattice to the top with only one step per iteration. While this looping is quite expensive computationally, it is very important, as it allows sharp bounds information to be inferred regarding loop and recursion index variables.

We thus are led to the simple data type looping/recursion type inference principle:

Loops and recursion in the input program force iteration in the type inference algorithm.

On the other hand, with a properly designed integer datatype lattice, this type inference algorithm can conclude that certain simple loops 1) terminate; and 2) stay within the bounds of an implementation-defined short integer type (e.g., $-128\leq x<128$, or $-2^{30}\leq x<2^{30}$).

We will also show that no fixed number (i.e., independent of the size of the program text) of limit steps (the inner loops in the Kaplan-Ullman type inference program given above) is sufficient to guarantee that we have produced the minimal datatype assignment within the Kaplan-Ullman inferencing framework.

Consider the following program:

```
a:=read(); b:=read(); c:=read(); d:=read(); e:=read();
w:=read(); x:=read(); y:=read(); z:=read();
assert a*w>0; v:=0; assert b*x>0; v:=1; assert c*y>0; v:=2; assert d*z>0; v:=3;
assert b*w>0; v:=4; assert c*x>0; v:=5; assert d*y>0; v:=6; assert e*z>0; v:=7;
assert e>0;
```

For this example, we will assume that the datatype lattice for numbers simply distinguishes between positive, negative and zero numbers. The inclusion of the dummy assignments to the variable v in the program forces the creation of multiple states during inferencing, which drastically slows down the propagation of information.

During the first (forward) inferencing pass, all variables are set to take on any values, since they are assigned the results of "read" operations, which can produce any type of value. However, we quickly learn that all of these values must be numbers, because they are used in multiplication operations, which can accept only numeric datatypes. Only at the very end of the forward inferencing pass do we learn that one of the variables—e—is a *positive* real number.

During the second (backward) inferencing pass, the knowledge that e is real and positive allows us to conclude that z is also real and positive, hence d is also real and positive. During the third (forward) inferencing pass, the knowledge that d is real and positive allows us to conclude that y is real and positive. During the fourth (backward) inferencing pass, the knowledge that y is real and positive allows us to conclude that c is real and positive. During the fifth (forward) inferencing pass, we conclude that x is a positive real. During the sixth (backward) inferencing pass, we conclude that b is a positive real. During the seventh (forward) inferencing pass, we conclude that w is a positive real. During the eighth (backward) inferencing pass, we conclude that a is a positive real. Thus, we have been forced into *eight* separate limit passes in order to finally converge on the minimal datatype assignment, which concludes that *all* of the variables must have been positive real numbers.

While this example was specially contrived to defeat type inferencing algorithms based on the Kaplan-Ullman algorithm, it can be extended in the obvious way to force the inferencing algorithm to perform *any* number of steps. Thus, through our counter-examples, we have shown that the Kaplan-Ullman inferencing algorithm must consist of a doubly nested loop, where the inner loop raises the lower bounds and the outer loop lowers the upper bounds by alternate forward and backward inferencing passes, with no *a priori* limits on the number of either the inner or the outer loop iterations.

An open question is whether we can change the representation of the states in the Nimble type inferencing algorithm to more quickly converge even on contrived examples like the one above.

6.1 THE DISCRETE DATATYPE LATTICE OF COMMON LISP "SINGLE-VALUES"

The Nimble type inferencing algorithm utilizes a Boolean algebra for its datatype lattice, both because it is more precise, and because it can be efficiently implemented by means of bit-vectors. These bit-vectors encode the possibilities of the different datatypes of a value as a union of "atomic" datatypes, where "atomic" here means an *atom* in a Boolean algebra, and not a Common Lisp *atom*. Thus, a bit-vector could encode the union {integer,single-float,character}, meaning that the corresponding variable could take on as a value at run-time either an integer, a single-precision floating point number, or a single character (a highly improbable situation!). Unlike the situation in strongly-typed languages such as C or Ada, there is no *standard* lattice required by Common Lisp. We could utilize a lattice which distinguished between Lisp *symbols* which were *keywords* and Lisp *symbols* which were not *keywords*, or we could lump the two together as simply Lisp *symbols*. Similarly, we are free to distinguish between negative and non-negative integers—essentially treating them as having different datatypes. A lattice allowing finer distinctions, and the resultant greater resolution for distinguishing datatypes, requires longer bit-vectors to implement. The only requirement is that the lattice satisfy the finite chain condition. This is easily achieved if the total number of bits in all the bit-vectors is bounded by an *a priori* bound. One implication of this requirement is that we cannot distinguish every single integer, but must group them into a finite set of equivalence classes.

We can normally utilize relatively short bit-vectors in the range of 32-64 bits. This is true since the majority of important Common Lisp distinctions can be made with less than 32 bits [Baker92], and increasing the resolution further can dramatically increase the space and time required to manipulate the Kaplan-Ullman t-functions (this is discussed in a later section). We have found that the datatype resolution required for proper *analysis* greatly exceeds the number of datatypes needed for efficient *representation*. For example, even though small positive and negative integers are both represented by the same hardware datatype, it is useful to propagate finer distinctions during analysis, since the final lattice upper bounds achieved can be significantly smaller. In the case where it is desired to achieve high performance on numeric programs, we have found the following numeric distinctions to be useful:

Integer classes:

$\{i \mid i < -2^{31}\}$

$\{-2^{31}\}$; we need this class to handle the asymmetry of 2's complement arithmetic

$\{i \mid -2^{31} < i < -1\}$

$\{-1\}$

$\{0\}$

$\{1\}$

$\{i \mid 1 < i < 2^{31}\}$

$\{2^{31}\}$; ditto.

$\{i \mid 2^{31} < i\}$

Floating point number classes (type,range):

type:

short-float
single-float
double-float
long-float

range:

$\{x \mid -\infty < x < -1.0\}$

$\{-1.0\}$

$\{x \mid -1.0 < x < -0.0\}$

$\{0.0,-0.0\}$; use of IEEE standard can force $\{0.0\}$,$\{-0.0\}$ distinction.

$\{x \mid 0.0 < x < 1.0\}$

$\{1.0\}$

$\{x \mid 1.0 < x < \infty\}$

(Note that these distinctions look remarkably similar to those used by those artificial intelligence researchers in "qualitative reasoning" [IJCAI-89]; perhaps they, too, should investigate Kaplan-Ullman inferencing.)

We do *not* currently attempt to track the contents of the higher-order datatypes of Common Lisp, as is routinely done in ML [Milner78]. In other words, we utilize the single class `cons` to stand for all cons cells, regardless of their contents. While tracking the contents of Lisp lists would be extremely valuable, the resolution required is far too great to be handled by our current methods (however, see [Baker90b], in which we extend ML-style unificational type inference to also perform storage use inferencing on these higher-order datatypes). Similarly, we do *not* currently attempt to track the types of higher order functions of Common Lisp, but lump them all together as `function`. Once again, this would be extremely valuable, as the experience of ML has shown, but again the resolution required is computationally prohibitive.

Note that this lumping of functional objects together as one class does not prevent us from keeping track of the arguments and results of user functions, but only of functional values ("funargs" or "closures"). This is possible, because unlike Scheme [Scheme], Common Lisp keeps functions mostly separate from data objects, and hence is amenable to more a classical compiler treatment of typed functions.

6.2 THE LATTICE OF COMMON LISP "MULTIPLE-VALUES"

Common Lisp, unlike other dialects of Lisp, has a curious notion of *multiple values*. These multiple values are not lists or vectors, and hence are not first-class objects. They can, however, be returned from functions or accepted as function arguments under certain carefully controlled conditions. These multiple values cause problems for program analysis; while they are rarely used, they *could* be used, and thus they must be everywhere allowed for.

Unlike our rather casual treatment of the other higher order datatypes in Common Lisp, we *must* model multiple values carefully. If we do not, we would not be able to infer anything about the results of any function call, and Lisp programs consist mainly of function calls. Multiple values can be more easily modeled than the other higher order data types due to two properties: they are *functional* objects, in the sense that their size and components cannot be altered via side-effects once they have been constructed; and they are not first class, in the sense that no references to them can be created or compared via `eq`.

Part of the problem of representing multiple values stems from the fact that one can construct functions which can return a different number of multiple values at different times—a multiple value "polymorphism". Furthermore, some primitive Lisp functions—such as `eval` or `apply`—can return any number of multiple values. Further complication arises from the fact that multiple values are coerced, in many circumstances, into a single value; the first one, if it exists, or `nil`, if it doesn't. Yet one can also write Lisp code which can tell how many values were returned by a function call, and what these values were.

The Nimble type inferencer represents multiple values using a record structure of 3 components. The first component is an integer interpreted as a bit-vector which indicates the set of possible numbers of values which are being represented. Thus, a multiple-value which represents a normal single value uses the integer $2 = 2^1$, meaning that the only possible number of values is 1. "Zero values" is represented by the integer $1 = 2^0$, and the multiple-value returned by the Common Lisp `floor` function is represented by the integer $4 = 2^2$, meaning that exactly 2 values are returned. If a function sometimes returns one value and sometimes returns two values, then the number of values is represented by $6 = 2^1+2^2$. The number 0 then represents "*no* possible multiple-values"—*not* zero values—i.e., the function never returns! Finally, the resulting types for functions like `eval` and `apply` are represented by -1 (= $\sum^{\infty} 2^i$ in 2's complement notation!). With this encoding for the number of components in a multiple value, it is easy to perform lattice *meet* and *join*—they are simply `logand` and `logior` of the representation numbers. (The finite chain condition ("fcc") holds so long as the number of multiple-values is *a priori* bounded; in pathological cases, it is necessary to limit the resolution of the number of values to "0,1,2,...,31,≥32", for example, in order to guarantee fcc.)

The second and third components of the multiple-value record structure are the "finite part" and the "infinite part" of the multiple-value representation. The finite part is a simple vector whose values are elements of the single-value datatype lattice—typically bit-vectors. The infinite part is a single element of the single-value datatype lattice. The interpretation of these two elements is similar to the digits and sign of a 2's complement number; any component (bit) whose index is less than the length of the finite part can be found in the finite part, and any component (bit) whose index is greater than the length of the finite part has the value of the infinite part. The reason for this structure is the fact that all multiple-values in Common Lisp have only finite lengths, but since our lattice is also a Boolean algebra, we must be capable of producing a complement, as well. But complements of finite sequences are sequences whose infinite parts are all the same—hence still representable in our scheme.

We can now describe *meet* and *join* operations on our multiple-value datatype lattice. The number-of-values integers are combined using logand or logior, respectively. Then the finite part and infinite part of the result are computed. Before performing either a meet or a join, we must first extend the finite part of the shorter operand to the length of the longer operand by extending it with copies of its infinite part. We then perform an element-wise meet or join utilizing the single-value meet or join operation from the single-value lattice. Finally, we canonicalize by collapsing the finite part of the result as far as possible; any elements at the end of the finite part vector which are equal to the infinite part are simply ignored and the vector is shortened. Thus, the implemention of these meet and join operations is similar to the implementation of addition and subtraction of multiple-precision 2's-complement integers.

Some examples of this encoding are given below. The reason for the `nil` in the infinite part of most of the multiple-values below is that Common Lisp specifies that access to unsupplied values from a multiple-value returns `nil`.

```
   (values)                         <1,<>,{nil}>
   (values 0)                       <2,<{0}>,{nil}>
   (floor 3 2)                      <4,<{1},{0}>,{nil}>
   (if …                            <6,<{0,1},{1,nil}>,{nil}>
       (values 0)
      (values 1 1))
(eval x)                            <-1,<>,t>
```

(Note that our representation of multiple-values is ideal as a lattice representation for any datatype consisting of finite-length sequences—e.g., character strings. The union of the strings "hi", represented by <4,<{h},{i}>,{}>, and "there", represented by <32,<{t},{h},{e},{r},{e}>,{}>, is represented by <36,<{h,t},{i,h},{e},{r},{e}>,{}>.)

6.3. THE LATTICE OF COMMON LISP "STATES"

Once the preliminary transformations described at the end of this section have been performed on a Common Lisp program, only lexically-scoped variables and global variables remain. Due to their sheer number, we have currently decided not to independently track the contents of true global variables, which in Common Lisp are the so-called "value cells" of symbols, and hence components of the "symbol" record structure. These have presently been lumped together into the single class which tracks the union of all symbol value cells. While this approximation loses valuable information about the use of global variables, the approximation must be used in all existing Lisp implementations, because these variables can be changed by any evaluated function, or by the user himself, and therefore must be capable of holding any Lisp datatype.

More interesting is our treatment of lexically-scoped variables and temporaries. The Nimble type inference algorithm collapses a Lisp program into a Fortran-like program by ignoring the recursive aspect. In other words, in our approximation, the various stack frames of a recursive function are modeled by a single stack frame which is a kind of union of all of the stack frames from the recursion. Thus, each program state within a function can be represented by its "alist environment"—i.e., a vector which associates a "single-value" lattice element which each lexically visible variable. Note that lexical variables not directly visible from within the function are not present in any of the program states within the function, since any changes to any visible variable can only be an effect of the function itself. The representation of temporaries is even easier, since a temporary can have only one "producer" and (at most) one "consumer", and these are tied (after collapsing the stack) directly to the program text. Therefore, these temporaries need not participate in the program states directly, but only as a result of being saved away (via `let`), or assigned to a program variable (via `setq`).

Thus, a program *state* is an environment which consists of only the named lexical variables visible at that point in the program; temporaries and global variables do not appear. Furthermore, this environment is a simple linked Lisp list of "single-value" lattice elements which is indexed by the integer which is the lexical depth of the corresponding variable in the current lexical environment. This list, and all of its elements, are *functional*, in that we perform no side-effects on them. This means that we are allowed to freely share the *tails* of these lists, which we do as much as possible. Since each program state is almost the same as its predecessors and its successors, and since most of the changes occur at the top of this list (the inner-most lexical scope), only minor modifications need to be made in order to produce the next state from a previous state.

This massive sharing of state tails saves space, but more importantly, it saves time. This is because during the processing of statements to produce the next state, only a small amount of processing need be performed. When using tail-sharing, this processing is typically O(`1`) instead of O(`n`), where `n` is the number of lexical variables visible at this point. Similarly, when performing the lattice *meet* and *join* operations on states, we usually only process the top few items, because the rest of the tails are identical. Thus, by using a functional representation of states and massive tail-sharing, we can represent full state information during our inferencing for only a little more than the storage and time used for a single state which is global to the whole program (like ML [Milner78]).

6.4 THE EFFICIENT IMPLEMENTATION OF KAPLAN-ULLMAN "T-FUNCTIONS"

The success of the Kaplan-Ullman type inference algorithm depends critically on getting sharp type information from the primitive functions of the language. In other words, given a function and a set of bounds on its arguments, we must produce the sharpest possible bounds for the result of the function. In addition to forward and backward inferencing information, we would also like to get "side-ways" type inference information; this information is extracted from the interaction of constraints on the various arguments of multiple-argument functions. The symmetry between arguments and results leads to a symmetrical representation for the t-function information. This representation is in the form of a mathematical *relation*, which is simply a subset of a Cartesian product of domains. In particular, if f:AxB->C, then f can be represented by a relation, i.e., a subset of the Cartesian product AxBxC. The datatype domain induces an equivalence relation on A, B, and C, in such a way that the t-function for f becomes the function f':A'xB'->C', where A', B', and C' are the *quotient sets* of A', B', and C' *induced* by the equivalence relation. While the full relation for f may be infinite in size, if A', B', and C' are all finite, then the Cartesian product A'xB'xC' is also finite, and hence f' can be represented by a finite relation on A'xB'xC'.

If A', B', and C' are all small, then f' can be represented by a list of legal triples, or alternatively a 3-dimensional bit matrix in which bit ijk is on if and only if k∈ f'(i,j). Neither of these representations is particularly small or computationally efficient if the number of arguments to (or returned values from) a function is very large. Nevertheless, if the tightest possible information is required, then a smaller representation may not exist unless the function can somehow be factored into a composition of simpler functions.

Standard binary operations like "+" will require a bit matrix of size n^3 to be represented, where `n` is the number of "representatives" (see [Baker92]) in the datatype lattice representation. We expect `n` to range from 32-64, hence we will require between 1024 and 8192 32-bit words to simply represent the relation. Binary operations like Common Lisp's `floor` operation, which takes two arguments and produces two results, will require n^4 bits, requiring between 32,768 and 524,288 32-bit words for its representation. However, should this amount of space be considered excessive, then `floor` can instead be represented by two different functions—one for each different result—for an amount of space between 2048 and 16384 32-bit words. While this is a substantial savings in space, there is some loss of resolution due to the lack of dependence between the two different results of the `floor` function in the "two-function" representation.

The success of the Nimble type inferencer is due, in a large part, to its ability to completely encode the type complexity of Common Lisp primitive functions without actually interpreting the underlying implementation code. The type complexity of some Common Lisp functions is quite high. The exponential function `(expt base power)`, for example, is required to return a rational result if base is rational and power is an integer, and may return a floating-point approximation (possibly complex) otherwise. On the other hand, if the arguments to a library function are rational and the true mathematical result is rational, then an implementation is free to return either a floating-point number or the actual rational number result [CLtL,p.203]. For example, some Common Lisp implementations of `sqrt` go to the trouble to detect integer perfect squares, and in these cases return integer (rather than floating-point) results! Given the number of cases to consider, especially when various subranges of numbers are considered, the exact representation for the result type of `expt` or `sqrt` becomes a nightmarish expression in traditional Common Lisp type specifier syntax.

The use in the Nimble type inferencer of bit-arrays to represent the type behavior of Common Lisp primitive functions is inelegant compared with the clever "type variable" method to handle polymorphism in ML. However, the bit-array method is capable of handling the type complexity of Common Lisp while the type variable method of ML would not extend to this higher level of complexity and polymorphism. Without type variables, ML can represent only "rectangular approximations" to the true t-functions, while type variables add the ability to represent "diagonal approximations". If a t-function cannot be decomposed into rectangular and diagonal regions, then the methods of ML will not achieve the same resolution as our bit-array method. Common Lisp's `sqrt` function cannot be easily decomposed into diagonal or rectangular regions, as can be seen by the chart below.

t-function for sqrt

result ⇒ ⇓ argument	integer	ratio	c-rational	float	c-float
integer	X		X	X	X
ratio		X	X	X	X
c-rational			X	X	X
float				X	X
c-float					X

Given the type complexity of the Common Lisp builtin functions, it becomes difficult to construct the correct bit-arrays for the type inferencer. For example: x-y is equivalent to x+(-y) in normal algebra and even in normal computer arithmetic. However, the t-function for binary "-" cannot be derived from the t-functions for "+" and unary "-" unless the underlying lattice is symmetric about 0. However, given a datatype lattice which is symmetric about 0, we must then derive the t-function of binary "/" from that of "*" and unary "/", which requires a datatype lattice which is symmetric about 1. Since it is impossible to produce a lattice which is symmetric about both 0 and 1 at the same time (unless it is the indiscrete lattice of all rational numbers!), we cannot, in general, derive accurate t-functions of functions from the t-functions of their functional factors. Nor can we construct these t-functions by hand; the number of Common Lisp builtin functions is very large (on the order of 500), and each of these bit-arrays contains a large number of bits. From these observations, it can be seen that we must somehow automate the process of constructing these large bit-arrays.

We can automate the production of the bit arrays for our t-functions by using the machinery of Common Lisp itself. If we are given a sufficient number of "representative elements", then we can simply execute the function on all of the representative elements, and note into which class the result falls. With properly chosen representative elements, we can elicit the complete behavior of the function over the finite set of atomic datatypes in our Boolean lattice. Of course, the *choice* of representative elements cannot be performed automatically, but must be made carefully and intelligently. The lack of a proper representative may result in the lack of a bit in the t-function array, and hence an improper inference may be made. Furthermore, the appropriate set of representatives may be different for each function. Nevertheless, the number of representatives required is still far less than the total number of bits in the bit-array. Additionally, the ability to get the representatives correct is much less difficult than getting every bit right in the t-function bit-arrays by hand.

Certain functions like +, `min`, `gcd`, etc., satisfy certain commutativity properties. In these cases, we can eliminate half of the work for constructing t-functions by considering only non-commutative pairs of arguments and then or'ing the resulting bit-array with one of its transposes. However, even with such short-cuts, the production of these bit-arrays is a time-consuming process. The problem is somewhat eased, since the production of these bit-arrays need be done only once for a particular combination of Lisp implementation and datatype lattice, and so we can precompute these arrays.

An interesting problem occurs when we attempt to precompute large bit-arrays in Common Lisp. There is no efficient mechanism to read or write large bit-vector arrays in a compressed form (i.e., as individual bits) in Common Lisp! If the array is output using the `print` routine, the array is printed out as integers—i.e., the digits 0 or 1 followed by a single space—or 16 bits for every one bit in the array! Writing the underlying single-dimensioned bit-vector results in the vector being printed in #*10110...01 format—or 8 bits for every one bit in the array. Since the bit-arrays for the normal binary operations such as + may require 32K bytes internally, the prospect of reading 1/4 megabyte for each of these functions seems excessive.

There is no builtin mechanism in Common Lisp to convert bit-vectors into integers. Even if such a mechanism existed, we could at best print out the bit-vector in hexadecimal format—or 4 bits for every one bit in the array. For the Nimble type inferencer, we overcame this problem by tricking Common Lisp into thinking that a bit-vector was really a character vector. In this way, we achieved a one-to-one correspondence between external and internal bits. (Lest advocates of other languages be smug about this point, consider the same problem in Ada. The standard mechanism to read and write boolean arrays in Ada involves the use of character strings `"TRUE"` and `"FALSE"`, which require an average of 5.5 characters (= 44 bits) for each internal bit read or written!)

6.5 INCORPORATING USER TYPE DECLARATIONS

User type declarations for program variables are trivially incorporated into the Kaplan-Ullman type inference algorithm by using them to initialize the upper bounds instead of initializing these bounds to "top". Since the Kaplan-Ullman algorithm uniformly *meets* any new bounds with the upper bounds, the minimal datatype it determines will always be consistent with the declared datatype. The Nimble type inferencing algorithm follows the same procedure.

6.6 INSERTING TYPE CHECKS

The Nimble type inferencer puts `(the <type> <exp>)` expressions around every expression, including constants and variables. If the actual value of <exp> at run-time is an element of <type>, then this expression acts as an identity function, while if the actual value is not of the appropriate type, then an error message will be generated and the program is (usually) aborted. If, after type inference, the type of <exp> can be proved to be of a subtype of <type>, then the run-time type check is superfluous. If the compiler used to compile the program uses a complete decision procedure for `subtypep` [Baker92], then it will eliminate all of the type checks that the Nimble type inferencer was able to prove superfluous.

6.7 EXTENDING KAPLAN & ULLMAN TO WORK ON USER-DEFINED FUNCTIONS

The Kaplan-Ullman type inference algorithm dealt with only primitive functions, and did not treat user-defined functions. In order to extend Kaplan-Ullman to handle user-defined functions, we must make some approximations. The most important approximation we make is to identify all instances of the same lexical variable or the same lexical function. Thus, the argument "X" in a recursive function will always be given the same type, regardless of where or when it was called—whether from outside the function or as a recursive call from inside the function. This approximation is reasonable, since the compiler will be generating only one instance of the lexical variable or the lexical function, and thus the instance generated must deal with all situations which might occur during run-time.

This approximation collapses the control structure for the program being analyzed into a Fortran-like structure in which the formal parameters and local variables of a function behave more like Fortran variables (or Algol "own" variables) than like local stack-allocated or heap-allocated variables. Some approximation of this type is necessary in order to ensure convergence of the type inferencing algorithm. This particular approximation is the most "natural" one to use, as well as one of the easiest to explain to the programmer. If collapsing a higher order control structure into an iterative control structure loses too much resolution to make the desired distinctions, then the programmer can always "inline" the function or "unroll" the loop one or more times, to achieve higher resolution. Any other policy would be too arbitrary to understand or control.

This collapsing policy also has the advantage of treating Common Lisp lexically scoped free variables (including global variables) in a uniform and reasonably intuitive manner.

6.8 CONTROL FLOW INFERENCING

In addition to infering the types of variables, the Nimble type inferencer also infers whether a portion of the program is executed or not—i.e., it performs *control flow inferencing*. This is normally used for *dead code elimination*. By utilizing more sophisticated information to determine deadness, we can in more cases infer that a conditional will only execute one (or neither) of its arms. This is important, since dead code can ruin the sharpness of our type deductions. Consider, for example, the following code:

```
(let ((x 3))
  (if (integerp x) x (setq x 3.0)))
```

Note that the ability to tell that the assignment will *not* be executed is essential to determining that `x` will hold only integer values. A sloppier algorithm would conclude that `x` could be either an integer or a floating-point number, thus foregoing a significant optimization. Performing simultaneous control flow and type inferencing is important when processing the bodies of functions expanded in-line, because such in-lined functions are significant sources of dead code.

We can handle dead code elimination within our algorithm by incorporating control information in addition to our datatype information in the "flowstate" lattice during the forward inferencing loops. The control information basically indicates whether a particular node will ever be executed. This information is inferred by induction, using the fact that the first node in the program is executed as an initial condition. Nodes that have not yet been marked as executing do not participate in inferencing. If a node is still unmarked when a forward inferencing loop converges, then it must be the case that the node is dead code, and it (along with the "conditional" branch that goes to it) can be immediately eliminated.

6.9 TWO-PHASE INFERENCING

The implemented Nimble type inference algorithm actually involves *two* separate type inference processes. This two-phase implementation arises from the limitations of the Kaplan-Ullman algorithm discussed earlier. The two phases are quite similar, except for the following differences:

- Phase I uses a higher resolution lattice which does not satisfy the finite chain condition
- Phase I does forward inferencing only
- Phase I does not loop

There are several reasons for performing type inferencing with *two* different lattices. Kaplan-Ullman type inferencing is an algorithm by which most of the progress in producing sharp lattice upper bounds comes in the early iterations of the outer loop, while most of the effort in the later stages produces only minor improvements. Thus, the higher resolution lattice (involving a correspondingly larger amount of computational effort) is used in the beginning to achieve a better starting point for a more classical Kaplan-Ullman inferencer. We also use the first forward inference pass to produce very sharp *control flow* information; this approach is essential to the removal of dead code which would make the subsequent inferencing of sharp bounds impossible. Thus, we utilize the higher resolution indiscrete lattice where it will be the most productive—during the first forward inferencing pass—yet we use it only once in order to avoid its non-convergence problem.

The indiscrete lattice used in the first phase of the Nimble type inferencer is the same as that described earlier in [Baker92]. Basically, the lack of discreteness in this lattice is a result of the *interval* representation of rational and floating-point ranges. One end of an interval can grow without bound, or can have an irrational limit, thus, this lattice is not discrete. The use of an indiscrete lattice in a Kaplan-Ullman inferencer would prevent it from converging.

The fact that the first phase does not loop means that minor modifications must be made to the inferencer to ensure that the "looping/recursion principle" stated above is not violated. In the classical Kaplan-Ullman algorithm, the carrying of simultaneous upper/lower bounds enhanced the speed of convergence of the algorithm but not its final result. The carrying of simultaneous upper/lower bounds, however, is *essential* to obtaining a reasonable answer from the first phase algorithm.

Forward inferencing using the indiscrete lattice of [Baker92] subsumes a number of traditional static analyses. Within the limitations of this lattice—that higher order functions and data types are not modeled, except by `cons` and `function`—this inferencing can subsume constant propagation for scalars and "interval arithmetic" on primitive arithmetic functions. Therefore, the bounds for scalar numeric values can be surprisingly tight.

6.10 SUMMARY OF THE NIMBLE TYPE INFERENCE ALGORITHM

The Nimble type inference algorithm conceptually operates in several passes, although they are not implemented that way. The first pass converts Common Lisp into a somewhat simpler language by eliminating several complex notions from the language less ruthlessly than [Kranz86]. Macros are expanded; "special" (dynamically-scoped) variables are translated into semantically equivalent implementations; the dynamic operations `catch`, `throw` and `unwind-protect` (analogous to "exceptions" and "signalling" in Ada) are translated into semantically equivalent operations [Haynes87]; functions declared "inline" are expanded; some constant propagation/dead code elimination is performed and many syntactic variations are regularized. The second pass performs forward and control flow inferencing using the indiscrete lattice of [Baker92]. During the second pass, we eliminate a substantial amount of dead code which was accumulated through macro and inline function expansion as well as from the translation of the undesired special forms of Common Lisp. The third pass (which consumes most of the time required for inferencing) performs forward, backward and control flow inferencing using the discrete lattice. The results are then passed directly to a code generator, or can be output as source code decorated with complete declarations. Due to the macro and function expansions and to the issues discussed in [Baker92], this output is not particularly readable.

7. THE ANALYSIS OF THE TAK FUNCTION

We show below the analysis of the `TAK` function [Gabriel85] by the Nimble type inferencer:

```
(defun do-tak ()
  (labels
    ((tak (x y z)
       (if (not (< y x))
           z
           (tak (tak (1- x) y z)
                (tak (1- y) z x)
                (tak (1- z) x y)))))
    (tak 18 12 6)))
```

Our algorithm is able to make the following inferences:
1. `x`,`y`,`z` start as integers and are only affected by `1-`, thus are always integers.
2. `x`,`y`,`z` are always decremented, so they can never grow in the positive direction.
3. The value of the function comes eventually from `z`, and hence is an integer.

However, the algorithm cannot determine whether the function will ever stop, or whether there is any limit to the size of `x`,`y`,`z` in the negative direction. Thus, although the algorithm can be optimized to assume only *integer* arithmetic, it must still allow for the possibility of negative *bignums*, even if they are never used. Unfortunately, the *possibility* of bignums means the *possibility* of garbage collection, hence the compiler must be careful about keeping clean calling sequences.

Another inference algorithm would have to be *much* smarter than the Nimble algorithm in order to determine better range information. Proving a restricted range is equivalent to proving termination, which cannot be proved within the limits of the Kaplan-Ullman framework. Proving termination for `TAK` requires that relationships among variables be tracked, which is not done by the Kaplan-Ullman algorithm.

Due to these considerations, we believe that the Nimble analysis is about the best that can be expected for an automatic inferencing process given reasonable time and memory constraints.

8. COMPLEXITY

The complexity of the Nimble type inference algorithm is relatively difficult to analyze. As we have shown in several examples above, it is difficult to determine upper bounds upon the number of iterations in the nested loops that define the Kaplan-Ullman algorithm. The program text alone is not sufficient to determine the number of iterations, since the number will depend upon the resolution of the datatype lattice being used to analyze the program. It is easy to show examples where the number of iterations of an inner loop is equal to the *height* of the lattice (i.e., the length of the longest chain). It should also be possible to demonstrate that the outer loop can be forced to descend the entire height of the lattice. If nested loops of this type can be demonstrated, then the complexity of the Kaplan-Ullman algorithm would be at least exponential in size of the input program.

We have used the Nimble type inference algorithm on a number of small examples, but have not yet used it on large programs. As we have pointed out, the first few iterations of the outer loop achieve most of the progress at producing sharp bounds, with the additional iterations "polishing" the bounds within one of the loops. While it seems obvious that one should simply cut off the process at some point, the exact point to cut it off is not yet clear. For example, it is likely that the additional polishing done by the later loops is improving the performance of the program's *inner* loops, which usually occupy most of the program's execution time, and therefore what appears to be minor progress may actually result in substantial reductions in execution time.

Nevertheless, the Nimble algorithm seems to converge quite rapidly on the examples tried to date. This experience is consistent with the general experience of dataflow analyzers; that on most programs, they terminate more quickly than theory would predict.

The Nimble type inferencer required about 30 seconds to type the `TAK` function given above on a 4 Megabyte Macintosh Plus with a 16MHz 68020 accelerator running Coral Common Lisp. While this is quite slow, it would have been about 100 times slower without the careful tuning of the bit-vector and bit-array manipulation routines described in [Baker90c]. While 30 seconds seems excessive to some, the types of computations performed by the Nimble inferencer could be easily mapped onto parallel architectures: SIMD architectures, such as the Connection Machine [Hillis85], for high performance on bit operations; MIMD architectures, for high performance on inferencing different parts of the program simultaneously.

9. PREVIOUS WORK

Schwartz and Tenenbaum [Schwartz75][Tenenbaum74] were early researchers in type inferencing for the high level language SETL. They utilized dataflow techniques on a lattice that included higher order data structures which included both "forward" and "backwards" inferencing. Their task was eased by the functional (side-effect free) nature of their higher order data structures.

The resolution of overloaded function symbols in strongly-typed languages such as Ada [Ada83] bears much resemblance to type inference. Early researchers feared that overload resolution would require an iterative forwards-and-backwards process similar to that described here [Ichbiah79,7.5.1], but these fears proved groundless when algorithms were found that performed this resolution in a single forward and a single backward pass [Pennello80].

Abstract interpretation is the name given to the generic process of "executing" a program on a lattice which is much simpler than the standard execution lattice. This process produces a kind of "homomorphic image" of the real computation, and is often used for various kinds of static analysis [Cousot77, Mycroft81, Burn87]. Most "forward" type inference, including that performed by Kaplan-Ullman, Beer and ourselves, can be viewed as a form of abstract interpretation. However, as Tanenbaum [Tanenbaum74], Kaplan-Ullman [Kaplan80] and we show, forward inferencing, and hence abstract interpretation, is not strong enough by itself to provide the information which is desired.

Constant propagation [Aho86] can be seen as a form of forward type inference or abstract interpretation [Callahan86]. This technique detects and propagates compile-time constants by evaluating expressions (including function calls, if possible) to perform as much of the computation as possible during compilation. A complete implementation of constant propagation subsumes actual program execution, since the provision of a complete set of input data would enable the computation of all output at compile time. Since constant propagation necessitates a violation of the order of evaluation, it has much in common with *strictness* analysis in lazy functional languages [Burn87].

Kaplan and Ullman [Kaplan80] provide an algorithm and a characterization of a type inference algorithm for a run-time data-typed language such as APL or Lisp. Their algorithm is optimum, in that for a class of languages and programs that he characterizes, it provides the best possible information on the range of types that a variable can assume. Kaplan shows that both "forward" inferencing (in the normal direction of computation) and "backward" inferencing (contrary to the normal direction of computation) is required in order to extract the maximum information. Forward type inferencing propagates the type information from subexpressions to the whole expression by restricting the possibilities for the mathematical range of the subexpression functions; e.g., knowledge about the non-negativity of a "square" function might be useful to restrict the possible results from the next step in the computation. Backward type inferencing propagates the type information about the mathematical domain of functions within subexpressions; e.g., if a function computes the reciprocal of a number, then the requirement of non-zeroness of that argument must be fed backward through the computation to make sure that the reciprocal function will never see a zero argument.

Kaplan's algorithm provides the maximal amount of information, but it depends upon a rather simplified model for a programming language: a language with variables and iteration, but no recursion or data structures. Furthermore, he does not tackle the problem of functional arguments, which makes control flow analysis difficult in Lisp [Shivers88]. The Nimble type inference algorithm extends Kaplan's algorithm to handle the constructs of Common Lisp.

Most existing Lisp implementations utilize a simple forward inferencing scheme in which declaration information is propagated forwards from variables to values, function arguments to function values [Moon74, Teitelman78, Marti79, Brooks82, Yuasa85]. These schemes are not state-based, and hence cannot handle case-based inferencing. Furthermore, the lattice typically used tends to be trivial—e.g., "integer/short-float/long-float/other". Beer [Beer88] has implemented the forward portion of Kaplan's algorithm for Common Lisp using a more precise, hence indiscrete, lattice to infer types and numeric bounds. He finds that it is successful at determining the types of 80% of the variables and expressions at compile-time for an interesting benchmark. More importantly, the program ran 136% faster after type inferencing, while only an additional 3.5% improvement was realized when the rest of the declarations were inserted by hand. We believe that the Nimble two-phase approach is strictly more powerful than the Beer algorithm, although they are difficult to compare because the Beer algorithm uses heuristics to terminate its loops.

[Bauer74] pointed out the possibility of type inferencing in APL. [Budd88] has implemented an APL compiler which is successful at inferring the types of most variables and subexpressions within the APL language.

[Suzuki81] and [Borning82] attack the problem of type inferencing in the Smalltalk language. In Smalltalk, control flow and data flow analysis must be done simultaneously, since in many cases, the code executed depends upon the type and values of the data, and vice versa. They find that Smalltalk also has enough redundancy to make type inferencing quite successful.

Range inferencing is similar in concept to type inferencing. Here, we would like to narrow the range of values assumed by a variable or an expression to be less than the whole universe of values of the particular data type. For example, if a variable is inferred to be an integer, we would like to determine whether its values are restricted to a small set of integers, perhaps 0-255, so that additional optimization can be performed. Range inferencing is particularly important in reducing the need for array bounds checking, because bounds checking can slow down and possibly defeat several array indexing optimizations.

[Harrison77] is one of the first to report on compile-time range inferencing, with [Suzuki] and [Markstein82] following. Even though the results these researchers reported were positive, very few commercial compilers incorporate this sort of analysis, except for Ada compilers [Taffs85], in which range checks are required unless they can be proved redundant. To avoid the overhead of array bounds checking in those compilers which do not perform the analysis, the user must turn off all array bounds checking. This practice is too dangerous for applications where an error could cause loss of property or life.

Even in those cases where array-bounds checking cannot be eliminated, a competent type checker can still be beneficial. The programmer may already have performed his own range check to obtain a more graceful error recovery than the language system normally provides, and in some of these cases, the type checker can usually conclude that an additional check inserted by the compiler would be redundant.

Array bounds checking demonstrates one significant weakness of the Nimble type inference algorithm relative to strongly-typed languages like Ada [AdaLRM]. Ada is a strongly typed language which has a substantial amount of machinery for declaring and manipulating variables subject to range constraints. However, unlike Nimble ranges, whose endpoints must be numeric constants, Ada ranges can have *variables* as endpoints, meaning that the size of the range is not known until run-time. Thus, an Ada compiler can relatively painlessly determine that the array bounds of v are never violated in the following code, by relying on Ada's strong typing system:

```
type vector is array(natural range <>) of float;
function sum foo(v: vector) return float is
   total: float := 0;
   begin
      for i in v'range loop
            total := total + v(i);
            end loop;
      return total;
   end sum;
```

On the other hand, the current Nimble type inferencer cannot eliminate the bounds checking on `v` in the following equivalent Common Lisp code due to its inability to represent such *variable* ranges:

```
(defun sum(v &aux (total 0))
   (dotimes (i (length v) total)
      (incf total (aref v i))))
```

ML-style type inferencing [Milner78] elegantly solves two problems—typing higher order functions and data structures, and avoiding the forward-backward iterations of the dataflow techniques. However, ML-style type inferencing also has several deficiencies. It cannot handle case-based inferencing due to its lack of state and it cannot handle full Lisp-like polymorphism.

The ML-style unification algorithm which comes closest in goals to ours is that of [Suzuki81] for Smalltalk-76. Suzuki extends the ML algorithm to handle *unions* of base types, which are quite similar to our techniques for representing Common Lisp types. He uses Milner-style unification to solve a set of simultaneous inequalities on the datatypes of the variable instances instead of the more precise (and slower) Scott-style least-fixed-point limit steps. The Suzuki method may be somewhat faster than our method and it easily extends to higher-order functions, but it does not produce bounds which are as tight as those produced by the Nimble algorithm. For example, it cannot conclude that the argument to the factorial function remains a non-negative fixnum if it starts as a non-negative fixnum, nor can it conclude that the value is always a positive integer if the argument is a non-negative integer.

[Wand84] describes an ML-style type checker for Scheme, another dialect of Lisp. It handles straight-forward ML-style polymorphism, and is best characterized as "ML with parentheses". However, this method is not nearly as powerful as that in [Suzuki81], because it cannot handle the unions of datatypes introduced by Suzuki, and cannot therefore handle the polymorphism of real Lisp programs.

The Nimble type inference algorithm could be used in a functional programming environment, where it could infer sharper information than the ML unification algorithm. This is because the Nimble algorithm can handle polymorphism and case-based reasoning in a way that would be impossible for a unification-based algorithm. Its ability to type builtin functions more accurately than ML will also produce sharper type information. While it *may* be more expensive to run than a unification-based inference algorithm (although ML typing is itself known to be DEXPTIME-complete [Mairson90]), its better information may yield more efficient programs—a reasonable trade-off in some situations.

10. CONCLUSIONS AND FUTURE DIRECTIONS

Type inferencing in a run-time data typed language such as Lisp or APL is not needed for simple execution. If the goal is optimized execution, however, then more specific information as to the types of variables and expressions is necessary. Type inferencing cannot be dispensed with through additional declarations; e.g., declarations force the same type for an argument in all calls to a procedure, and eliminate the possibility of *polymorphism*, or execution of the same code at different times with different types [Cardelli85]. Type inferencing can be a real boon in checking types across procedure call interfaces, and allow for different types to be inferred within a procedure depending upon the actual arguments.

Generalized type inferencing would seem to be hopeless. However, while many examples can be contrived to show the impossibility of assigning a distinct type to an expression, most real programs have more than enough redundancy in the use of the built-in functions and operators to enable most data types to be unambiguously assigned [Beer88]. The consequences of an ambiguous assignment in Lisp is not necessarily an error, but it does reduce the possibilities for optimization; hence the more tightly the datatypes are constrained, the more efficiently the code will run.

We have described a type inference algorithm for Common Lisp which has evolved from the Kaplan-Ullman algorithm [Kaplan80] to the point that it can handle the entire Common Lisp-84 language [CLtL84]. We have shown, through a number of examples, that this algorithm uses case-based and state-based reasoning to deduce tight lattice bounds on polymorphic functions, including recursive functions. We have described a number of novel techniques for engineering an efficient implementation of the lattice manipulations required by this algorithm. We have shown how this algorithm is strictly more powerful than other popular techniques such as unification-based techniques [Milner78] on some examples, and seems more appropriate for highly polymorphic languages such as Lisp. While the algorithmic complexity of our inferencer is higher than usual for Lisp compilers, its better information can be used for a greater than usual degree of optimization. The fact that this information can be extracted in a completely mechanical fashion, and the fact that the kind of processing required can be greatly accelerated by parallel computers, mean that the cost of type inference will decrease quickly over time.

A possible improvement that could be made to the basic Kaplan-Ullman type inference machinery is the employment of a larger number of lattices. So long as every inner loop in the Kaplan-Ullman algorithm is allowed to complete, the computed bound can be used as an upper bound on the next stage execution of the inner loop. If this next stage uses a more refined lattice, then tighter bounds can be inferred. Therefore, we could conceivably start with a coarse lattice, distinguishing only between scalars, list cells, functions, etc. The next stage could distinguish various kinds of numbers, various kinds of list cells, etc. Only in the latest stages would we distinguish among the higher order kinds of data structures and their components. A large amount of space and time in type inferencing could be saved by reserving a higher resolution lattice for numbers, only for those variables which have already been shown to be numbers; a higher resolution lattice for different kinds of list cells could be reserved just for those variables shown to be only list cells; and so forth. In this way, we could utiliize different lattices for different variables, which is an improvement that we could also have achieved through strong typing. However, our lattice approach allows far more flexibility, because not all variables need be resolved to the same level of refinement.

Since the Nimble type inferencer must deal with the entirety of the Common Lisp-84 language, it must have a reasonably deep understanding of every one of its datatypes, constructs and functions. One may ask whether the enormous effort involved in incorporating this knowledge into a static analyzer is worth the effort. The answer is yes, if there exist important Common Lisp programs which would be expensive to modify which need to be statically analyzed.

In most cases, the Lisp community would be better served by a language which is *much* smaller than Common Lisp, since the many different and often redundant features of the language do not contribute either to its efficiency or to its ease of use. For example, the polymorphic type complexity of the Common Lisp library functions is mostly gratuitous, and both the efficiency of compiled code and the efficiency of the programmer could be increased by rationalizing this complexity. Notions such as dynamic floating-point contagion, multiple-values, complex argument-passing, and special variables are obsolete in today's world. Most strings and lists in Lisp are used in a functional manner, yet they are heavily penalized in performance by the remote possibility of side-effects. A major advance in the run-time efficiency and ease of static analysis of Lisp-like languages could be achieved if Lisp programs and argument lists were constructed from some functional data structure instead of from cons cells.

ACKNOWLEDGMENTS

The author wishes to thank the Department of Energy for their support and James J. Hirsch for his help in editing this manuscript.

REFERENCES

AdaLRM: *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, U.S. Government Printing Office, Wash., DC, 1983.

Aho, Alfred V.; Sethi, Ravi; and Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

Baker92: Baker, Henry. "A Decision Procedure for Common Lisp's SUBTYPEP Predicate". *Lisp and Symbolic Computation 5*,3 (Sept. 1992), 157-190.

Baker90b: Baker, Henry. "Unify and Conquer (Garbage, Updating, Aliasing...) in Functional Languages". *Proc. 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

Baker90c: Baker, Henry. "Efficient Implementation of Bit-vector Operations in Common Lisp". ACM *Lisp Pointers 3,*2-3-4 (April-June 1990), 8-22.

Bauer, Alan M., and Saal, Harry J. "Does APL really need run-time checking?" *Software Practice and Experience*, v.4, 1974,pp.129-138.

Beer, Randall D. "The compile-time type inference and type checking of Common Lisp programs: a technical summary". TR 88-116, Ctr. for Automation and Intelligent Sys. Research, Case Western Reserve Univ., May 1988; also *LISP Pointers 1*,2 (June-July 1987),5-11.

Bobrow, et al. "Common Lisp Object System Specification X3J13", *ACM SIGPLAN Notices*, v.23, Sept. 1988; also X3J13 Document 88-002R, June 1988.

Borning, Alan H. and Ingalls, Daniel H. H. "A Type Declaration and Inference System for Smalltalk" *ACM POPL 9*, 1982, pp.133-141.

Brooks, R., Gabriel, R., Steele, G. "S-1 Common Lisp Implementation". *Proc. of '82 ACM Symp. on Lisp and Funct. Prog.*, (Aug. 1982),108-113.

Brooks, R., et al. "Design of an Optimizing, Dynamically Retargetable Compiler for Common Lisp". *Proc. of '86 ACM Conf. on Lisp and Funct. Prog.*, (Aug. 1986),67-85.

Budd, Timothy. *An APL Compiler*. Springer-Verlag, NY, 1988.

Burn, G.L. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. Ph.D. Thesis, Imperial College, London, 1987.

Callahan, D., Cooper, K.D., Kennedy, K., and Torczon, L. "Interprocedural Constant Propagation". *Proc. Sigplan '86 Symp. on Compiler Constr.*, also *Sigplan Notices 21*, 7 (July 1986),152-161.

Cardelli, L., and Wegner, P. "On Understanding Types, Data Abstraction, and Polymorphism". ACM Comput. Surv. 17,4 (Dec. 1985),471-522.

Cartwright, R. "User-defined Data Types as an Aid to Verifying Lisp Programs". In Michaelson, S., and Milner, R. (eds.). *Automata, Languages and Programming*, Edinburgh Press, Edinburgh,228-256.

CLtL: Steele, Guy L., Jr. *Common Lisp: The Language*. Digital Press, 1984.

Cousot, P., and Cousot, R. "Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". *4'th ACM POPL*, 1977,238-252.

Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

Ellis, John R. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, Cambridge, MA, 1986.

Ferrante, Jeanne, and Rackoff, Charles W. "A decision procedure for the first order theory of real addition with order". *SIAM J. Comput. 4*, 1 (1975),69-76.

Gabriel, Richard P. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, MA, 1985.

Harper, R., *et al*. "Standard ML". Technical Report ECS-LFCS-86-2, Dept. of Computer Science, Edinburgh, UK, March, 1986.

Harrison, William. "Compiler Analysis of the Value Ranges for Variables". *IEEE Trans. Soft. Eng. SE-3*,3 (May 1977),243-250.

Haynes, Christopher T., and Friedman, Daniel P. "Embedding Continuations in Procedural Objects". *ACM TOPLAS 9*,4 (Oct. 1987),582-598.

Hillis, W. Daniel. *The Connection Machine*. The MIT Press, Cambridge, MA, 1985.

Ichbiah, J. "Rationale for the design of the Ada programming language." *ACM Sigplan Notices 14*,6 (June 1979),part B.

Intel Corporation. *i860 64-bit Microprocessor Programmer's Reference Manual.* Order #240329, Intel Corporation, Santa Clara, CA, 1989.

Jones, N.D., and Muchnick, S. "Binding time optimization in programming languages". *3'rd ACM POPL*, Atlanta, GA (1976),77-94.

Kanellakis, P.C., and Mitchell, J.C. "Polymorphic unification and ML typing". *ACM Funct. Prog. Langs. and Comp. Arch. (FPCA)*, 1989,54-74.

Kaplan, Marc A., and Ullman, Jeffrey D. "A Scheme for the Automatic Inference of Variable Types". *ACM JACM 27*,1 (Jan. 1980),128-145.

Katayama, Takuya. "Type Inference and Type Checking for Functional Programming Languages—A Reduced Computation Approach". *1984 ACM Conf. on Lisp and Funct. Prog.*, Aug. 1984,263-272.

Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. "ORBIT: An Optimizing Compiler for Scheme". *Proc. Sigplan '86 Symp. on Compiler Constr., also Sigplan Notices 21*, 7 (July 1986),219-233.

Ma, Kwan-Liu, and Kessler, Robert R. "TICL—A Type Inference System for Common Lisp". *SW—Prac. & Exper. 20*,6 (June 1990),593-623.

MacLane, Saunders and Birkhoff, Garrett. *ALGEBRA*. Macmillan, 1967.

Mairson, H.G. "Deciding ML Typability is Complete for Deterministic Exponential Time". *17'th ACM POPL* (Jan. 1990),382-401.

Markstein, Victoria; Cocke, John; and Markstein, Peter. "Optimization of Range Checking". *ACM POPL '82*,114-119.

Marti, J., Hearn, A.C., Griss, M.L., and Griss, C. "Standard LISP Report". *Sigplan Notices 14,* 10 (Oct. 1979),48-68.

Milner, Robin. "A Theory of Type Polymorphism in Programming" *JCSS 17*, 1978,pp.348-375.

Moon, D. *MACLISP Reference Manual Rev. 0*. Proj. MAC—M.I.T., Camb., MA, April 1974.

Morris, J.H. "Types are Not Sets". *ACM POPL*, 1973, pp.120-124.

Mycroft, Alan. *Abstract Interpretation and Optimising Transformation for Applicative Programs*. Ph.D. Thesis, Univ. Edinburgh, Scoitland, 1981.

Pennello, T., and Meyers, R. "A Simplified Operator Identification Scheme in Ada". ACM Sigplan Notices 15, 7&8 (July-Aug. 1980),82-87.

Schwartz, J.T. "Optimization of very high level languages—I. Value transmission and its corollaries". *J. Computer Lang. 1* (1975),161-194.

Scott, D. "Data types as lattices". *SIAM J. Computing*, 5,3 (Sept. 1976), 522-587.

Sethi, Ravi. "Conditional Expressions with Equality Tests". *J. ACM 25*,4 (Oct. 1978),667-674.

Shivers, O. "Control Flow Analysis in Scheme". *ACM Sigplan Conf. '88*,164-174.

Steele, Guy L., Jr. *Rabbit: A Compiler for SCHEME (A Study in Compiler Optimization)*. AI-TR-474, Artificial Intelligence Laboratory, MIT, May 1978.

Suzuki, Norihisa. "Implementation of an array bound checker". *ACM POPL* ???.

Suzuki, Norihisa. "Inferring Types in Smalltalk". *ACM POPL 8,* 1981,pp.187-199.

Taffs, D.A., Taffs, M.W., Rienzo, J.C., Hampson, T.R. "The ALS Ada Compiler Global Optimizer". in Barnes & Fisher, "Ada in Use": Proc. Ada Int'l. Conf., Camb. Univ. Press, 1985,355-366.

Teitelman, W., *et al*. *InterLISP Reference Manual*. Xerox PARC, Palo Alto, CA, 1978.

Tenenbaum, A. "Type determination for very high level languages". Ph.D. Thesis, Rep. NSO-3, Courant Inst. Math. Sci., New York U., New York, 1974.

Thatte, Satish R. "Quasi-static Typing". *17'th ACM POPL '90*, Jan. 1990,367-381.

Wand, M. "A Semantic Prototyping System". *Proc. ACM Sigplan '84 Symp. on Compiler Constr., Sigplan Notices 19*,6 (June 1984),213-221.

Yuasa, T., and Hagiya, M. Kyoto Common Lisp Report. Research Institute for Mathematical Sciences, Kyoto University, 1985.