Chapter 8

Compile-Time Type Prediction and Type Checking for Common Lisp Programs

Randall Beer

Department of Computer Engineering and Science
Case Western Reserve University, Cleveland, OH 44106
email: beer@alpha.ces.cwru.edu

Abstract

This paper describes a system for both predicting and checking the types of Common Lisp programs at compile-time. The system is capable of deriving type declarations from constraints implicit in the code and identifying potential run-time type errors. These capabilities can be used to improve the efficiency of Common Lisp on general-purpose architectures without sacrificing safety or ease of use. The system operates by using the type constraints of primitive functions to propagate known information throughout a dataflow graph of the program.

8.1 Introduction

One of the most distinctive features of the programming language Lisp is its run-time typing of data. This feature allows typeless variables, run-time type checking, generic functions* and source-level processing of type information. However, this feature is also expensive to implement on general-purpose computers. A recent study [19] found that Lisp programs running on these architectures spend an average of 11-24% of their time in type checking and dispatch. This problem is particularly bad in Common Lisp [17] because of the proliferation of generic functions for such types as numbers and sequences. For example, any of eight potentially distinct types of numbers may be added in any combination using the single function +. It is well-known that this overhead may be greatly reduced by the use of special-purpose hardware [14]. However, the problem remains critical for implementations

*Here the term generic function refers only to the fact that these functions work for multiple primitive data types. In the Common Lisp Object System specification, which is part of the emerging ANSI Common Lisp, this term has a more specific technical meaning [18].
(defun distance-between-points (x1 y1 x2 y2)
  (declare (fixnum x1 y1 x2 y2))
  (let ((dx (the fixnum (- x2 x1)))
        (dy (the fixnum (- y2 y1))))
    (declare (fixnum dx dy))
    (sqrt (the fixnum (+ (the fixnum (* dx dx))
                       (the fixnum (* dy dy))))))

**Figure 8.1:** The distance-between-points function

running on stock hardware. Indeed, this situation played a crucial role in at least one wide-ranging critique of Common Lisp [4].

One possible solution to this problem is for the programmer to make use of the optional type declaration facility provided by Common Lisp in order to inform the compiler of the types of values certain variables and expressions may take on at run-time. This information may allow the compiler to optimize the affected code accordingly. Unfortunately, there are a number of disadvantages to this approach. First of all, the unfortunate side-effect of optimizing out type-checks is that no type checking is performed at either compile-time or run-time. This can lead to obscure errors when the declarations are mistaken or violated.

Another problem with declarations is the severe burden they place upon the programmer, a burden worsened by the verbosity of the information that must often be supplied for maximum efficiency. For example, consider the function distance-between-points (Figure 8.1), which computes the distance between two points whose coordinates are represented as small integers. Character for character, this function contains more declarations than it does computation (the declare and the forms declare the types of variables and expressions, respectively). Yet all of these declarations are required by a typical Lisp compiler in order to fully optimize the generated code.

This paper explores the use of compile-time type prediction and type checking as a practical alternative to these problems [2]. In this approach, a static dataflow analysis of a Lisp program is used to derive type predictions from constraints implicit in the code and to identify places where run-time type errors might occur. The former information can be used to inform the compiler where optimizations may safely be performed. The latter information can be used to inform the compiler where run-time type-checks need to be inserted, or to notify the programmer where potential problems lie so that he or she may make an informed decision as to whether or not they can safely be ignored.

Constraints on the types of expressions and variables in Common Lisp programs can come from many sources. The types of constants are trivially known. User declarations are another source of type information. In addition, certain type-conditional statements, such as (if (listp x) ...), imply the type of the conditioned variable in the true and false branches of the conditional. Finally, and perhaps most importantly, the built-in Common Lisp functions themselves carry a great deal of type information. For example, the function reverse may only be passed a sequence,
and the type of the result returned by \texttt{reverse} is always the same as the type of its argument. Such type information may be supplemented by implementation-specific constraints. For example, many implementations restrict the length of a sequence to be a \texttt{fixnum}, even though [17] states that it may be any positive integer.

8.2 The Type System

Common Lisp provides a rich and complicated lattice of data types. We assume the existence of the standard meet (greatest lower bound, denoted \( \land \)), join (least upper bound, denoted \( \lor \)) and partial order (denoted \( \leq \)) operations on this lattice. In the context of a type lattice, the partial ordering \( r \leq s \) should be interpreted to mean that type \( r \) is a subtype of type \( s \). Primitive types supported by Common Lisp include hash tables, functions, characters, streams, structures, etc. [17]. Our focus in this paper will be on numbers, sequences, and arrays, since computations involving these types typically exhibit the most significant speedup when generic operations involving them are replaced by type-specific operations.

Eight primitive numeric types are supported by Common Lisp: machine integers (\texttt{fixnums}), arbitrarily large integers (\texttt{bignums}), ratios, complex numbers and four different floating point formats. All of these numbers are operated on by the same generic arithmetic functions. Significant speedup can be achieved if these generic operations can be specialized at compile-time to type-specific instructions. It is especially important to identify numeric operations involving only \texttt{fixnums} or only short floating point numbers whenever possible, since these data types are the ones for which efficient code sequences often exist on general-purpose architectures.

A sequence is an abstract data type that denotes an ordered set of objects. It includes such data types as lists, general vectors, strings, and bit vectors as subtypes. Common Lisp provides over two dozen generic operations on sequences, such as determining the length of a sequence, reversing a sequence, extracting or modifying a designated element of a sequence, extracting a subsequence, mapping a function across a sequence, sorting a sequence, etc. All of these operations could be made significantly more efficient if the subtype of the sequence were known at compile-time.

Common Lisp supports multidimensional arrays which can contain any Lisp object. General Common Lisp arrays may also be dynamically resized, and one array may share its contents with another. Often, however, arrays are specialized to contain only a particular type of data and do not make use of these advanced features. When the type of such a simple specialized array is known at compile-time, its storage and access can be optimized accordingly.

In their simplest form, Common Lisp types are denoted by symbols. For example, the symbol \texttt{fixnum} denotes machine precision integers. The top of the type lattice is denoted by the symbol \texttt{t} and the bottom is denoted by the symbol \texttt{nil} (i.e. for all \( s \) in the lattice, \( \texttt{nil} \leq s \leq \texttt{t} \)). A rich language is provided for specializing and combining types. For example, the type specifier \( \texttt{integer 0 *} \) denotes the integers ranging from 0 to positive infinity, and the type specifier \( \texttt{simple-array single-float 3 *} \) denotes all two-dimensional simple arrays of single precision floating point numbers that have exactly 3 rows and any number of columns (within
(number representation interval)
(sequence representation element-type length)
(array complexity element-type dimensions)

Figure 8.2: Canonical forms for number, sequence and array type specifiers

type specifiers, the character * means that the given component is unspecified). Common Lisp allows type specifiers to be combined using the connectives member, not, and and or. For example, the function position, which always returns either nil or a non-negative integer, has a return type of (or null (integer 0 *)) Finally, Common Lisp allows types to be defined by arbitrary predicates through the satisfies type specifier. For example, the type specifier (and (integer 0 100) (satisfies primep)), where primep is a Lisp function that returns true if its argument is prime and false otherwise, denotes the set of prime numbers less than 100 inclusive.

Due to the complexity of Common Lisp type specifiers, the system described in this paper makes use of a somewhat simpler type scheme. Common Lisp type specifiers are automatically translated into this simpler scheme when a program is processed by the system. All of the atomic Common Lisp type specifiers are supported in this simpler type scheme. However, all number, sequence and array types are translated into the canonical forms shown in Figure 8.2.

The representation of a numeric type specifier is a symbol denoting one of the following atomic types: bignum, integer, ratio, rational, short-float, single-float, double-float, long-float, float, complex or number. The interval of a numeric type specifier gives its range. This component is unspecified for complex numbers, because the normal arithmetic ordering does not apply. We will ignore complex numbers in this paper. An example of a valid numeric type specifier would be (number fixnum [0,+INF]).

The representation of a sequence type specifier is a symbol denoting one of the following atomic types: sequence, list, simple-vector or vector. The element-type may be any valid type specifier. The length component of a sequence type specifier is an interval that bounds the possible length of the sequence. An example of a valid sequence type specifier would be (sequence simple-vector character [0,255]).

The complexity of an array type specifier is either complex or simple. The element-type component of an array type specifier may be any valid type specifier. The dimensions component of an array type specifier is a list of intervals that bound each dimension. An example of a valid array type specifier would be (array simple single-float ([0,10] [0,10])).

The simpler type scheme also allows type specifiers to be combined using or and member. However, the connectives not, and and satisfies are not supported, with the single exception of (not null). Common Lisp type specifiers involving these unsupported connectives are translated in a conservative way, possibly including more objects than strictly necessary.

An important property of the Common Lisp type lattice that has serious implications for type inference is that the finite chain condition does not hold. That is,
this lattice contains infinite sequences of elements related by \( \leq \). The failure of the
finite chain condition means that naive fixed-point algorithms may fail to converge on
this lattice. One of many examples of such infinite paths provided by the integers
is shown below:

\[
\text{integer} 0 \leq \text{integer} 1 \leq \cdots \leq \text{integer} \ast \leq \text{integer}
\]

8.3 The Representation of Programs

In the standard application of flow analysis to the problem of static type deter-
mination (e.g. [10]), type is viewed as an attribute of a variable. In this approach,
flow analysis is used to track the types of each variable through the sequence of
assignments determined by the control flow graph of the program. However, in Lisp,
it is the values that are typed, not the variables. Indeed, the same variable may
hold values of different types at different points in a program. For the purpose of
Lisp compiler optimization, we are interested in the types of the data values that
can reach each parameter of every function call.

For this reason, we represent a Lisp function as a surface dataflow graph of
function calls. This representation makes explicit every source and sink of data in the
function and the links between them. The graph contains a node for each function
call and an arc for each possible dataflow between them. There are also a number of
special-purpose nodes. The constant, global, and parameter nodes represent the
sources of dataflow resulting from constants, free variables, and function parameters,
respectively. The return-value node represents the dataflow sink that consumes
any data the function returns to its caller. The join and split nodes represent the
fan-in and fan-out of dataflow, respectively. No control flow is explicitly represented
in this graph, only its net effect on the dataflow.

The dataflow graph of a given Lisp function is constructed by symbolically evalu-
ating that function using dataflow arcs rather than actual values. Each constant,
function parameter and free variable creates a new dataflow arc. Each function call
consumes existing dataflow arcs as arguments, creates a new node in the dataflow
graph, and returns a new dataflow arc as its value. When control transfers back to
a previously evaluated form (e.g. as a result of a \texttt{go}), the current values of program
variables are merged with their previous values at that point (perhaps creating join
nodes in the process). Special care must be taken when symbolically evaluating
conditional statements such as the \texttt{if} special form in Common Lisp. Because the
truth value of a condition will not, in general, be known at compile-time, a symbolic
evaluator must separately evaluate all possible paths and then merge the results (once
again, possibly creating join nodes in the process).

Certain conditional statements can be used by the dataflow analyzer to extract
additional type constraints from a Lisp program. For example, in the true branch
of an \texttt{if} form whose condition is \( (> n 0) \), we know that the type of \( n \) is \( \text{number} \ (0, +\text{INF}) \). By the same token, we know that the type of \( n \) is \( \text{number} \ (-\text{INF}, 0) \) in the false branch. Such implicit type information (called a
conditional filter) is automatically extracted by the dataflow analyzer when it sym-
bolically evaluates conditional expressions.
(defun ifact (n)
  (declare (fixnum n))
  (do ((counter n (1- counter))
       (result 1 (+ counter result)))
     ((<= counter 0) result)))

Figure 8.3: The iterative factorial function ifact and its dataflow graph

The dataflow graph of the iterative factorial function ifact is shown in Figure 8.3. The label on the top arc reflects the (declare (fixnum n)) type declaration made by the programmer. The limits minf and maxf represent the values of the Common Lisp constants most-negative-fixnum and most-positive-fixnum, respectively. The other two dataflow labels represent constraints on counter that are implicit in the structure of the code. All other dataflows are assumed to have a declared type of t.

8.4 Type Inference Rules

In the type inference system, each primitive function in Common Lisp potentially has three pieces of information associated with it: a description of its maxtypes, a forward inference rule, and a backward inference rule. The maxtype of an argument to a function is the least upper bound of all types in the lattice which can validly be passed through that argument and similarly for the maxtype of its result. A forward inference predicts the type of a function's result from the types of its arguments. A backward inference infers the types of a function's arguments from the type of its result. These two kinds of rules provide very different sorts of information. A forward inference makes predictions which are as correct as the information from
(define-mxotypes + (arest (number number [-INF,+INF])))
  (number number [-INF,+INF]))

(define-forward-type + (arest (number ?representations ?intervals))
  '(number , (numeric-least-upper-bound representations)
    ,(reduce #'+ _ intervals)))

(define-mxotypes length ((sequence sequence * [0,#.sequence-length-limit]))
  (number integer [0,#.sequence-length-limit]))

(define-forward-type length ((sequence ? ? ?length-interval))
  '(number integer ,length-interval))

(define-backward-type length (? (number integer ?length-interval)
  '((sequence sequence * ,length-interval)))

Figure 8.4: Sample type inference rules for + and length

which they are derived, whereas a backward inference only provides information
which is required to be correct for the program to be free from type errors at run-
time. We will refer to the results of a forward inference as a predicted type and
the results of a backward inference will be referred to as a required type. As we
will see later, this distinction can be used to warn the programmer about type
inconsistencies or to automatically generate minimal run-time type-checks which
guarantee the safety of the code [12].

Sample type inference rules are shown in Figure 8.4. The syntax of these rules
resembles the function definitions that one might give for each of these functions,
except that these definitions operate over type specifiers rather than actual data
values. Each argument is defined by a template in which ? matches any single
element. If the question mark is followed by a symbol, that symbol will be bound to
the matching value. Rather than invoking a general pattern matcher, these rules are
compiled into Lisp functions which explicitly perform the required destructuring.

The rules for + illustrate the basic features of the forward inference rules. The
mxtype definition states that + accepts zero or more numbers and returns a num-
ber as a result. The forward inference rule states that + returns a number whose
representation and interval can be computed from those of its arguments. The func-
tion numeric-least-upper-bound implements the various coercion rules specified
by Common Lisp for arithmetic operations on numeric types. The function +_ is a
binary version of + which operates on intervals and properly handles positive and
negative infinity.

The inference rules for length illustrate the format of a backward inference rule.
A backward inference rule takes the predicted types of the arguments (not used in
this example) and the required type of the result, and produces required types for
each of the arguments. The backward inference rule for length states that if the
result is required to be an integer lying within some interval, then the argument is
required to be a sequence whose length lies within that same interval. The symbol
sequence-length-limit is an implementation-specific constant which specifies the
maximum length of sequences.

These type inference rules can become quite complex. Type inference rules
support the full lambda list syntax, including the lambda list keywords \texttt{#rest},
\texttt{#optional} and \texttt{#key}. In the case of a \texttt{#rest} parameter, the associated template is
matched to the types of each of the actual arguments to a function in turn, and any
pattern variables in the template are bound to lists of the matching components.
(see the forward inference rule for \texttt{+} in Figure 8.4). In the case of \texttt{#optional}
and \texttt{#key} parameters, the types of the default values may be specified. In addition, type
inference rules can perform some fairly complex deductions. For example, backward
inference rules can also perform what might be called “sideways” inference. That
is, the predicted type of one parameter can be used to constrain the required type
of another.

Some types have more than one representation in our type system. For
example, simple strings containing 256 characters or less might be expressed
as \texttt{(sequence\ simple-vector\ string-char [0,255])} or as \texttt{(array\ simple\ string-char ([0,255]))}. This leads to a problem when matching the parameters of type inference rules to type specifiers, because a simple string represented as
an array would not match the rule for \texttt{length} in Figure 8.4, even though it should.
However, if we always represented such strings as sequences, then they would not
match the inference rules for array manipulation functions such as \texttt{aref}, even though they should. The problem is that the portion of the Common Lisp type lattice involving vectors is nonhierarchical. A vector is both a kind of array and a kind of sequence. Since there is no obvious canonical form for such types, the system must translate between the equivalent forms as necessary when matching type specifiers
to type inference rules.

## 8.5 The Inference System

The purpose of the type inference system is to assign a predicted type and a
required type to every arc in the dataflow graph. These assignments are made
by forward and backward inference systems, respectively. A forward inference rule
computes the predicted type of a function’s return value from the predicted types
of its arguments. The forward inference system uses these rules to propagate known
type information forward throughout the dataflow graph. A backward inference rule
computes the required type of a function’s arguments from the required type of its
return value and the predicted types of its arguments. The backward inference sys-
tem uses these rules to propagate requirements backwards throughout the dataflow
graph. Finally, the predicted and required type assignments are combined to generate
a final safe and efficient assignment of types. This section briefly describes each
of these processes in turn.
8.5.1 The Forward Inference System

The forward system derives predictions about the types of values each dataflow will carry at run-time. In deriving these predictions, we make use of programmer-supplied declarations, implicit declarations provided by conditional filters, and the type constraints of the primitive Common Lisp functions. The forward inference algorithm operates as follows. The predicted types of all dataflow sources (constant, global, and parameter nodes) are set to their declared types. All other predicted types are set to nil, the bottom of the Common Lisp type lattice. This information is then propagated forward throughout the dataflow graph until a minimal fixed-point is reached using the forward inference rules for the primitive Common Lisp functions. The predicted type of each dataflow is set to the join of its previous prediction and the meet of the prediction of the function at its source with any declarations or conditional filters on the dataflow.

As mentioned earlier, the failure of the finite chain condition for the Common Lisp type lattice implies that fixedpoint algorithms such as that described above may fail to converge. For this reason, we modify the basic forward inference algorithm to detect potentially infinite sequences of type predictions and to accelerate their convergence by decreasing the precision of the prediction. For example, if the interval of an integer type specifier of some dataflow arc takes on the sequence of values \{[0,0], [0,1], [0,2], ...\} as the forward inference algorithm iterates, it would be replaced by [0, \text{+INF}). In effect, we dynamically abstract the Common Lisp type lattice as necessary in order to guarantee convergence.

8.5.2 The Backward Inference System

The backward system derives requirements each dataflow must satisfy if the code is to be safe from run-time errors. The required types of all dataflows are set to \text{v}, the top of the Common Lisp type lattice. The requirements generated by each function call node are then propagated backward throughout the graph until a maximal fixedpoint is reached using the backward prediction rules for the primitive Common Lisp functions. The required type of each dataflow is set to the meet of its previous requirement and the requirement generated by the function at its sink unless the current prediction for this dataflow is already a subtype of this requirement. In other words, the required type of a dataflow represents a requirement which is not yet satisfied by the current prediction.

8.5.3 Combining the Forward and Backward Systems

We now consider how the predictions derived by the forward inference system and the requirements derived by the backward inference system can be combined. If we are not concerned about type safety, we can simply use the forward predictions as the basis for compiler optimization. In fact, we can sometimes achieve even better (but potentially unsafe) predictions by combining the two systems in the following manner. First, the forward inference system is run to convergence. Then the predicted types that it derives are taken as requirements and the backward system is run to convergence. Next, the required types derived by the backward
system are taken as predictions, and the forward system is run once again. This process of alternating between forward and backward inference can itself be iterated until convergence.

If, however, we are concerned with the run-time safety of a Lisp function, then we cannot mix the requirements generated by backward inference and the predictions generated by forward inference [12]. Any dataflow whose predicted type is not a subtype of its required type represents a potential run-time error and must be verified by the programmer or with an explicit run-time type check. One simple approach to code safety would be to generate a run-time type check for every such dataflow. However, this is potentially inefficient because one type-check may eliminate the need for another. In order to minimize the number of run-time type checks required to guarantee code safety, we can combine the forward and backward systems somewhat differently than described above.

We will call a dataflow verified if its predicted type is a subtype of its required type. We begin by verifying the dataflows originating from constant, global, and parameter nodes. For each dataflow which is not already verified, we mark it as requiring verification and then set its predicted type to the meet of this type and its required type and propagate this new information forward. The rationale here is that, since we are going to have to verify this requirement anyway, and since, by definition, it is more precise than the original prediction, we might as well try to use this new information to derive other more precise predictions. This new information may verify ancestor dataflows which were previously unverified. Next, we consider dataflows downstream from those we have just considered, and so on until all dataflows have been examined.

Finally, the system must analyze the information it has derived. An error message is printed for any dataflow for which the meet of its predicted type and its required type is nil. Such dataflows represent type errors. Any declarations that were actually used to make predictions that could not otherwise be derived also need to be verified (since user declarations are only promises). However, the programmer may set a *trust-declarations* flag to ignore these. For any dataflows marked as requiring verification, a warning message is printed. Such dataflows represent potential run-time safety problems. This information could also be used by an optimizing compiler to identify places in the code where run-time type checks need to be inserted.

8.6 Examples

This section presents three examples of nontrivial Common Lisp functions whose efficiency can be significantly improved without sacrificing safety by the techniques described in this paper. All timings were performed in VAX LISP, Digital Equipment Corporation's implementation of Common Lisp.

8.6.1 fact

Our first example is the iterative factorial function fact, shown in Figure 8.3. It is interesting to note that the fixnum declaration provided by the programmer for
(defun ifact (n)
    (declare (fixnum n))
    (do ((counter n (the fixnum (1- counter)))
          (result 1 (the integer (* counter result)))
          (<= counter 0) result)
        (declare (fixnum counter)
                 (integer result))))

Figure 8.5: Derived declarations for the ifact function

ifact is completely useless to any classical Lisp compiler because n is not directly involved in any computation. However, it is the kind of declaration a programmer might be reasonably expected to give since a Lisp compiler cannot in general make any assumptions about the types of the parameters a function will be passed. Furthermore, this declaration, coupled with the dataflow structure of the code, certainly does constrain the types of counter and result. This is the sort of situation we would like our system to be able to improve. Using the approach described in this paper, we can infer the additional type declarations shown in Figure 8.5. Furthermore, assuming that the *trust-declarations* flag is set, every function call in ifact is determined to be safe.

For n equal to 10, these additional declarations halve the execution time of ifact over the original version in VAX LISP. For large n, the computation becomes dominated by the bignum multiplications and the difference contributes little to the total execution time. In general, however, fixnum declarations for loop variables are very important. It should also be noted that the system can actually do a bit better than shown in Figure 8.5. It can infer that the type of the result of the 1- is (integer 0 most-positive-fixnum - 1) and that the type of the result of the *, and therefore result, is (integer 1 *). However, few Lisp compilers could make any use of this additional precision.

8.6.2 longest-word

As a second example, consider the longest-word function, which returns the longest sequence of nonseparat character in a given string (Figure 8.6). For example, (longest-word "this is a test" "$\star$) would return "this". Given the declarations for string and separator, the type inference system can derive all of the additional declarations shown in Figure 8.6. The fact that VAX LISP restricts the length of a sequence to be a fixnum is crucial to the inferences the system achieves for this function. Furthermore, the system cannot verify that start is always small enough to serve as a valid sequence index (in principle, start can take on the value most-positive-fixnum, which is not a valid sequence index). Therefore, a run-time type-check will need to be inserted into the code or the programmer will need to take responsibility for the value of this variable. All other function calls are found to be safe (assuming that the *trust-declarations* flag is set).
(defun longest-word (string separator)
  (declare (simple-string string)
    (string-char separator))
  (let ((longest-start 0)
        (longest-end 0)
        (longest-length 0))
    (declare (fixnum longest-start longest-end longest-length))
    (do* ((start 0 (the fixnum (1+ (the fixnum end))))
          (end (position separator string)
               (position separator string :start start)))
         ((null end)
          (when (> (the fixnum (- (length string) start))
                   longest-length)
           (setq longest-start start)
           (setq longest-end (length string))))
     (declare (fixnum start) (type (or fixnum null) end))
     (when (> (the fixnum (- (the fixnum end) start))
              longest-length)
       (setq longest-start start)
       (setq longest-end end)
       (setq longest-length
            (the fixnum (- (the fixnum end) start))))
     (subseq string longest-start longest-end))

Figure 8.6: Derived declarations for the longest-word function
8.6.3 word-count

As a final example, consider the word-count function shown below, which writes a report of the number of times each word appearing in a given input file to a given output file. This function is generally considered to be a "representative" Common Lisp function because it exercises so many different features of Common Lisp [22]. The dataflow graph for this function contains 130 dataflows.

(defun alpha-or-quote-p (ch)
  (or (alpha-char-p ch) (char= ch \\')(char= ch \\n))

(defun word-count (infile-name outfile-name)
  (declare (simple-string in-file-name out-file-name))
  (let ((hashtable (make-hash-table :test #'equal :size 1000))
        (total 0))
    (declare (hash-table hashtable) (integer total))
    (with-open-file (inf in-file-name :direction :input)
      (do ((buf (read-line inf nil nil)
              (read-line inf nil nil)))
          ((null buf))
        (declare (simple-string buf))
        (setq buf (nstring-upcase buf))
        (do ((start (position-if #'alpha-or-quote-p buf :start 0)
              (position-if #'alpha-or-quote-p buf
                            :start (the fixnum end)))
             (null start))
            (declare (type (or null fixnum) start end))
          (setq end (or (position-if-not
                         #'alpha-or-quote-p buf
                         :start (the fixnum (1+ (the fixnum start))))
                   (length buf)))
        (incf total)
        (incf (gethash (subseq buf
                         (the fixnum start)
                         (the fixnum end))
                      hashtable 0))))
    (let ((words (list ((list nil))
                       (maphash #'(lambda (key val)
                                    (push (cons key val) list))
                                 hashtable)))
      (declare (list words))
      (princ "Sorting ... ")
      (setq words (sort words #'> :key #'cdr))
      (princ "Writing word list ... ")
      (with-open-file (outf out-file-name :direction :output
                             (quote-to-string words)))
Given the declarations for *infilename* and *outfilename*, the system can infer all of the additional type declarations shown above. These declarations cut the execution time of this function for a given test file roughly in half over the undeclared version. It is interesting to note that the system has been able to infer all but two of the declarations which appeared in [22]. The first missing declaration is that *total* is a fixnum. Though it was able to infer that *total* was an integer, there was no way for the system to determine how large that integer might become. The second missing declaration was that the result of the call to `gethash` is a fixnum. Even though only fixnums are stored in the hash table, the system is not capable of reasoning about the contents of mutable data structures. Interestingly, these two missing declarations make very little difference.

In terms of code safety, the type checking phase identified only two places where run-time type checks would need to be inserted to guarantee the safety of the code. First, the result of the call to `gethash` must be verified to be a number. Second, `pair` must be verified to be a list in the functional argument to `mapc`. With these two run-time type checks, `word-count` is free from the possibility of type errors at run-time (assuming that the `*trust-declarations*` flag is set).

### 8.7 Related Work

Compile-time type checking and type inference have been examined for a number of languages which have dynamic properties analogous to Lisp. Bauer and Saal [1] showed that 80% of the run-time type, rank, and shape checks performed by a naive APL interpreter could be performed at compile-time by a static analysis of the code. Budd [7] constructed an APL compiler based upon some of these principles. Suzuki [20] and Borning and Ingalls [3] have developed systems which infer and check the types of objects in Smalltalk, allowing invalid messages to be detected and message lookup to be performed at compile-time.

The general use of flow analysis for the compile-time determination of types was introduced by Tannenbaum [21]. Kaplan and Ullman [10] presented a more powerful algorithm and proved that it is optimal in an important sense. Miller [12] used the distinction between forward and backward inference in these systems to perform type checking. Abstract interpretation provides a unified theoretical framework in which to consider the static analysis of programs [8].

An alternative approach to compile-time type checking and inference involves the use of generic type signatures for primitive functions to set up a set of equations
Chapter 8. Type Predication and Type Checking

for the types of a user program. These equations can often be solved using unification. This approach is particularly well suited to strongly-typed, purely functional languages and can handle polymorphic and recursive types. The development of the type system for ML has been influential in this approach [13].

Since the work described in this paper first appeared [2], a number of authors have begun to examine the application of techniques for compile-time type checking and type inference to Lisp. Shivers [16] has developed an approach to type recovery in Scheme. A major concern of this work is the correct flow-analysis of higher-order functions. Johnson [9] has developed an approach to verifying the type safety of Common Lisp programs. The principal goal of this work is to improve the robustness of exploratory software development without sacrificing its advantages.

8.8 Conclusion

In this paper, a system for the compile-time type prediction and type-checking of Common Lisp programs has been presented. This work has demonstrated that there is a significant amount of type information implicit in Common Lisp programs. The system described in this paper is capable of automatically recovering a great deal of this information from realistic Common Lisp programs. We have shown that declarations derived from this recovered information can make a significant difference to the execution time of these programs. We have also shown how forward and backward inference can be combined to identify the minimum number of locations where run-time type checks need to be inserted in order to guarantee that the code will be free from type errors at run-time. Thus, the efficiency benefits of exhaustive programmer declarations can often be achieved without burdening the programmer, and without sacrificing run-time safety.

Acknowledgements

This work was carried out in the Center for Automation and Intelligent Systems Research, which Yo-Han Pao created. In a very real sense, much of both this and my later work would not have been possible without Yo-Han’s support. I would also like to thank Gary Brown, who first suggested the problem of Common Lisp type inference to me. This work was supported by grants from the Digital Equipment Corporation and the Cleveland Advanced Manufacturing Program through the Ohio Edison Program.

References


