Preliminary Report on

A Practical Type Inference System for Common Lisp[*]

Randall D. Beer
Center for Automation and Intelligent Systems Research and the Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106
beer%case@CSNet-Relay.ARPA

1 Introduction

While the combination of dynamic typing and generic functions in Lisp have always presented a challenge to optimizing Lisp compilers for stock hardware, the situation has never been more difficult than in Common Lisp [7]. For example, one may add any of eight distinct primitive types of numbers in any combination using the single function +. While the overhead of sorting this type information out at run-time may be largely alleviated by the use of special-purpose hardware or microcode, the problem remains critical for implementations running on conventional general-purpose computers. Indeed, this situation played a crucial role in at least one wide-ranging critique of Common Lisp [2].

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 range of values certain variables and expressions may take on at run-time, allowing the compiler to optimize the affected code accordingly. Unfortunately, there are a number of disadvantages to this approach. Chief among them is the severe burden placed upon the programmer, a burden worsened by the verbosity of the information which must often be supplied for maximum efficiency. For example, consider the following function, which computes the distance between two points whose coordinates are represented as small integers. In VAX LISP [3], these declarations make nearly a 40% difference in speed.

```
(defun dist-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)))))))
```

[*]This paper describes research in progress. It is offered in this preliminary form due to its potential interest to the Common Lisp Community. Some additional background may be found in [1]. Comments, criticisms, and suggestions are welcome. VAX LISP and MicroVAX are registered trademarks of Digital Equipment Corporation. This work is supported by grants from the Digital Equipment Corporation and the Cleveland Advanced Manufacturing Program.

An alternative approach to this problem is to have the compiler itself automatically infer as much of the necessary type information as possible. While this is clearly a difficult problem in general, a great deal of theoretical work has been done on developing this approach and examining its limitations. There have also been a number of suggestions for incorporating this technique into compilers for various dialects of Lisp, APL, Smalltalk, and Prolog. However, to our knowledge, no compiler in widespread use for any of these languages currently performs any nontrivial type inference.

The goal of our research is to develop a practical type inference system for Common Lisp. By practical, we mean that the system should be able to infer useful type declarations from type constraints implicit in the code as well as any partial declarations provided by the programmer with a "reasonable" amount of computation. It is perhaps best thought of as a kind of "declaration amplifier" which minimizes the amount of effort a programmer must invest in order to achieve efficient code. This work is characterized by a very pragmatic flavor. Type inference is an extraordinarily difficult problem in general and an important aspect of our research continues to be the separation of the realistic inferences from the unrealistic ones.

In its present form, the system is designed as a preprocessor to the compiler rather than as an integral part of it. Thus the type inference system accepts Common Lisp source code as input and returns that source possibly annotated with additional declarations. This approach was chosen so as not to burden the research with the details of any particular compiler and to make the results of our work as widely applicable as possible. We are currently focusing on inferences involving numbers, sequences, and arrays, because these are the types for which specialized code can often be generated on stock hardware. However, the system can handle the full Common Lisp language in the sense that any inferences it makes can only err on the conservative side. Simplifying the language in order to simplify the type inference process was not considered to be a viable option.

2 Background

We may distinguish between two major approaches to type inference, which we term the *functional* approach and the *imperative* approach, respectively. The functional approach has enjoyed the most attention, both theoretically and practically. Perhaps the most visible success of this work has been the functional programming language ML [6] which is capable of type checking programs containing no type declarations. In this approach, a program is represented as a collection of generic signatures for the functions in its body. These signatures may contain variables, thus supporting polymorphism. A generic signature for the program as a whole is obtained by unification from the constituent signatures and the manner in which they are assembled in the program. Type checking is then performed by determining whether the signature of a given call on a function is a substitution instance of its generic signature. While this approach is powerful, simple, and elegant, we have found it to be insufficient for a type system as complex as the Common Lisp type lattice.

The imperative approach [4],[5], on the other hand, represents a program as a flowgraph whose nodes are assignment statements of the form $Z \leftarrow \oplus(X_1,X_2,...,X_k)$. The types of all variables in a program are determined by repeatedly performing forward and backward inferences across these statements until fixed points are achieved. Though this approach suffers from a number of drawbacks (particularly in the area of polymorphism) and has not found many practical applications, we feel that it holds the most promise for dealing with the complexity of the Common Lisp type lattice. For this reason, our system is essentially of the imperative variety, though with a number of important differences, particularly in the area of program representation.

3 The Representation of Programs

Rather than representing a program as a control flow graph of assignment nodes, as in the classical imperative approach, we use a surface dataflow graph of function calls. For example, compare the following definition of the iterative factorial function, which will serve as an example in Section 5, to its dataflow graph representation in Figure 1.

```
(defun ifact (n)
  (declare (fixnum n))
  (do ((counter n (1- counter))
       (result 1 (* counter result)))
      ((<= counter 0) result)))
```

This graph contains a node for each function call in the code and an arc for each possible dataflow between these calls. A small number of specialized nodes are also included in order to simplify the processing of the graph. A **splice** node always appears at the head of a loop, joining the initial value of the associated variable with the values of any of its update forms. A **split** node occurs whenever a given dataflow needs to go to more than one place. In addition, a **join** node may occur for a conditional statement in order to merge any corresponding dataflows from each of the arms of the conditional. The **fixnum** label on the top arc reflects the type declaration made by the programmer. The α labels on two of the dataflows will be explained shortly. All other dataflows are assumed to have a declared type of **t**.

This graph is constructed by a code analyzer which walks through a Common Lisp program in much the same way as a compiler, building function call nodes and dataflow arcs as it goes. It is called a *surface* dataflow because it only captures dataflow which can be determined by a static analysis of the code. It does not include such "deeper" dataflow as that caused by side-effecting shared structures, which in general can only be determined at run-time.

This representation offers a number of advantages over a control flow graph of assignment statements. It reflects the natural symmetry between variables and expressions with respect to the flow of data through a program. It also makes explicit the importance of dataflow to the type inference problem rather then leaving this implicit in the inference algorithm. This dataflow graph forms a kind of type constraint network which is then solved for its "minimal" fixed point by the type inference algorithm. A solution is an assignment of as precise a type as possible to each dataflow arc in the graph.

4 The Type Inference System

In our 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 rule deduces something about the type of a function's result from the types of its arguments. A backward inference rule performs the reverse inference. These two kinds of rules provide very different sorts of information. A forward inference makes predictions which are guaranteed to be correct, whereas backward inferences only provide information which is required to be correct for the program to succeed at run-time. This distinction can be used to warn the programmer about type inconsistencies or to automatically generate minimal run-time time checks which guarantee the safety of the code [5]. We have currently focused primarily on forward inference rules.

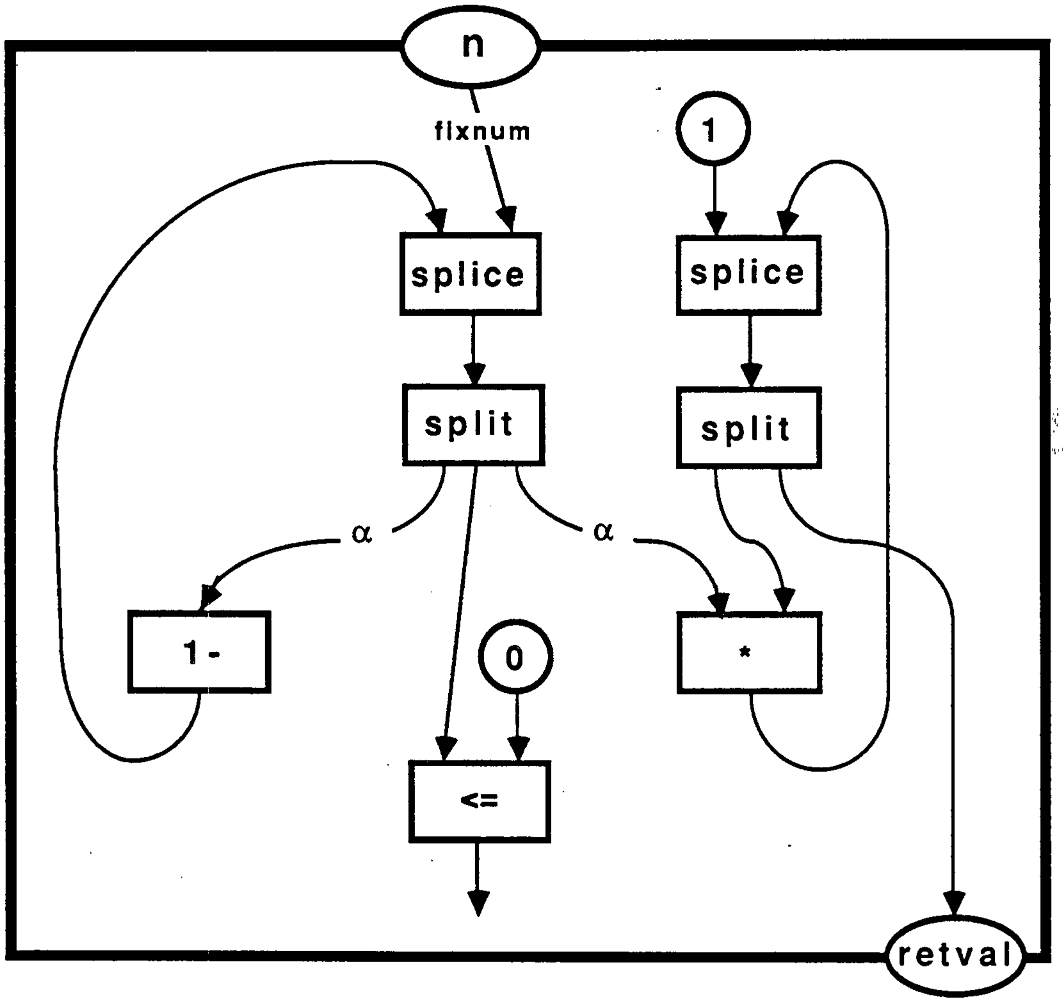

Figure 1 - The Dataflow Graph for **ifact**.

These rules operate over an internal type language which is mostly isomorphic to Common Lisp's, but has a more rigid format and sometimes contains additional information. For example, the general scheme for a numeric type internally is (**number** *representation interval*). Thus, the **fixnum** declaration in Figure 1 would actually be represented as (**number integer** [*mnf*, *mpf*]), where *mnf* and *mpf* represent the Common Lisp constants **most-negative-fixnum** and **most-positive-fixnum**, respectively. The α labels in Figure 1 represent the internal type (**number number (0,+inf]**), which is a declaration implicit in the structure of the conditional since the body of the loop is only evaluated when **counter** is greater than **0**. Such implicit declarations may be extracted automatically by the dataflow analyzer in a number of special cases such as this one, but these are currently entered by hand. The maxtype definition and forward inference rule which operate on these types for * are given below:

```
(define-maxtypes * (&rest (number number [-inf,+inf]))
  (number number [-inf,+inf]))

(defun-forward-type * (&rest (number ?reps ?ints))
  `(number ,(numeric-maxtypes reps) ,(reduce #'*__ ints)))
```

The above forms bear a close resemblance to the function definition one might give for * except that each argument in the lambda list is replaced by a template which destructures the type of the corresponding actual parameter. These rules are compiled into functions which explicitly perform the destructuring, rather than invoking a general pattern matcher. The function **numeric-maxtypes** implements the various coercion rules specified by Common Lisp for arithmetic operations on numeric types, as described in [7]. The function *__ is a binary version of * which works with intervals.

The actual forward inference algorithm is a fairly standard iterative fixed-point algorithm on lattices. The predicted types of all dataflows except those originating from parameters and constants are set to **nil**. The predicted types of constant dataflows are set to the types of the constants. The predicted types of parameter dataflows are either set to their declared types (if we are inferring internal declarations for the function) or to the types of the actual parameters (if we are making an inference across a call on this function in the course of inferring the types of another function). The constant and parameter dataflows are then pushed onto a stack and their types are propagated forward in a depth-first manner by primitive forward inference rules such as that above or by recursive invocations of the algorithm for user functions. An inference is made across a function call node only when the types of all of its inputs are **nonnil**. The types of a **split** are the same as the type of its input . The type of a **join** is the union of the types of its inputs. The type of a **splice** is also the union of the types of its inputs except that a **splice** node does not wait for all of its inputs to become non**nil** before propagating forward. This is to avoid a deadlock situation in which forward inferences across a **splice** are delayed until a foward inference across the same **splice** occurs because of the cycle involved.

Each new prediction for a dataflow is intersected with the declared type (which thus acts as a kind of filter for the maximal type that dataflow will carry) and unioned with the previous prediction. Whenever a dataflow's predicted type changes, its immediate children are pushed onto the stack. This process continues until the stack is empty, in which case a fixed-point solution has been achieved. The precision of this solution depends upon a variety of factors such as the precision of the internal type language and the forward inference rules.

Another important limiting factor to the precision of the solution is the ability of the forward inference algorithm to maintain its desirable properties, such as consistency and convergence, in the face of increasing precision. For example, it is relatively straightfoward to insure that the algorithm is well-behaved if only the "principle" numeric types of the Common Lisp type lattice (**number**, **integer**, **single-float**, etc.) are included. However, the basic algorithm outlined above may fail to converge in the presence of loops when numeric intervals are involved. The declarations implicit in conditionals, such as those labelled by α in Figure 1, can often prevent this situation, but in general nonconvergence, or at lease extremely long paths to convergence, is possible. To address this, we have extended the algorithm to detect potentially nonconvergent situations and replace the offending limit with either **-inf** or **+inf**, as appropriate. While this local technique will often sacrifice precision, it is far simpler than any global technique which seeks to recognize what the loop as a whole is doing.

5 Examples

It is interesting to note that the **fixnum** declaration provided by the programmer for **ifact** above is completely useless to any normal 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 we have sketched above, we can infer the following additional type declarations:

```
(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))))
```

These additional declarations result in approximately a 50% increase in speed in VAX LISP over the original version for **n** equal to **10**. For large **n**, the computation becomes bounded 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 above. It can infer that the type of the result of the **1-** is **(integer 0** *mpf-1*) and that the type of the result of the *****, and therefore **result**, is **(integer 1 *)**. However, few compilers can make any use of this additional information. Note that the declarations implicit in the conditional are crucial to these inferences. A much less precise solution would be achieved if the termination test had been **(zerop counter)** instead of **(<= counter 0)**.

As a second example, consider the function **longest-word** shown below, which returns the longest nonwildcard substring of a given string. For example, **(longest-word "*this*s a*t" #\*)** would return **"this"**. The fact that many Common Lisp implementations restrict the length of a sequence to be a **fixnum** even though [7] specifies that it may be any positive integer is crucial to the inferences we can achieve with this function. The type inference system represents this information as implementation-dependent constants such as **sequence-length-limit** and **string-length-limit**.

```
(defun longest-word (string wildcard)
  (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 wildcard string)
               (position wildcard string :start start)))
         ((null end)
          (when (> (the fixnum (- (length string) start)) longest-length)
            (setq longest-start start)
            (setq longest-end end)))
      (declare (fixnum start))
      (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)))
```

6 Current Status

We are currently in the process of implementing a prototype of the system described above in VAX LISP. The dataflow analyzer and the forward inference system are running and a number of forward inference rules have been entered. The **ifact** example has been run successfully on this system, but at this point the **longest-word** example has only been worked out on paper. The system required 0.38 CPU seconds on a MicroVAX to build the dataflow graph for **ifact** and 0.1 CPU seconds to infer the declarations shown in Section 5. While it is far too early to draw any specific conclusions from these numbers, they do suggest that the type inference system is not unreasonably computationally expensive.

Acknowledgements

I would like to thank Lars Nyman and Mark Guzowski for their many useful comments and suggestions. I would also like to express my appreciation for the environment Dr. Yoh-Han Pao has fostered here at the Center under the auspices of CAMP. Special thanks to Gary Brown, who suggested this problem to me, and the Digital Equipment Corporation, which has supported my work from the beginning.

References

[1] Beer, Randall D. "Practical Type Inference for Common Lisp," Technical Report TR 110-86, Center for Automation and Intelligent Systems Research, Case Western Reserve University, July, 1986.

[2] Brooks, Rodney A. and Gabriel, Richard P. "A Critique of Common Lisp," in *Proceedings of the 1984 Symposium on Lisp and Functional Programming*, Austin, Texas, pp. 1-8.

[3] Digital Equipment Corporation, *VAX LISP/VMS User's Guide*, Maynard, Massachusetts, May, 1986.

[4] Kaplan, Marc A. and Ullman, Jeffrey D., "A Scheme for the Automatic Inference of Variable Types," *Journal of the Association of Computing Machinery*, Vol. 27, No.1, January, 1980, pp. 128-145.

[5] Miller, Terrence C., "Type Checking in an Imperfect World," *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, 1979, pp. 237-243.

[6] Milner, Robin, "A Theory of Type Polymorphism in Programming," *J. of Computer and System Sciences* **17**, 1978, pp. 348-375.

[7] Steele, Guy L., *Common Lisp: The Language*, Digital Press, 1984.