Type Inference in the Icon Programming Language*

Kenneth Walker and Ralph E. Griswold

TR 93-32

October 13, 1992

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants DCR-8502015 and CCR-8901573.

Type Inference in the Icon Programming Language

1. Introduction

Programming languages that do not have a strong compile-time type system present a variety of problems during program execution. On the other hand, strong compile-time type checking precludes several valuable programming language features, such as polymorphous procedures that can perform operations on values of several different types.

In the absence of compile-time type checking, if types are not checked during program execution, errors may occur that are difficult to diagnose, locate, and correct. Run-time type checking is expensive, however, and in the most general case it must be done repeatedly.

Even in the absence of features that enforce strong compile-time type checking, a compiler may be able to infer the types actually used and hence be able to avoid much of the run-time type checking that otherwise would be necessary.

The Icon programming language presents an interesting case in this regard. In Icon, there are no type declarations and no types associated with variables. Instead, type is a property of a value. The original, interpretive implementation of Icon performs rigorous run-time type checking and incurs significant overhead as a result [1]. A new optimizing compiler for Icon, on the other hand, has a type inferencing system that is effective in determining type usage and in eliminating much of the run-time type checking that otherwise would be required [2].

This report describes the features of Icon that are significant for type inference, how the type inferencing system in the compiler works, its implementation, its complexity, and the improvement in execution speed that results from type inference.

2. The Icon Programming Language

Icon is a high-level, general-purpose programming language [3]. It is an expression-based procedural language with a large repertoire of operations for processing strings and structures. Icon's data structures and how they are used pose a challenge in designing an effective type inference system for the language. Backtracking also poses a challenge. While other programming languages such as Prolog [4] utilize backtracking, type inferencing systems for those languages are quite different from one suitable for Icon.

The main features of Icon that are relevant to type inferencing are:

- There are no type declarations. Variables are not typed and a variable can have a value of any type. Values, on the other hand, contain type identification. Different types can be assigned to a variable at different times. All variables have a special null value initially.
- Some expressions are polymorphous, performing different operations depending on the types of their operands. Type checking is performed on the operands of operations that are type sensitive. Except in a few easily recognized cases, assignment to a variable is insensitive to the type of the value assigned. Inappropriate types for the operands of operations are coerced to appropriate ones if possible; otherwise error termination occurs.
- Some expressions, called generators, can produce a sequence of values through a suspension/resumption evaluation mechanism. Goal-directed evaluation causes control backtracking and the resumption of suspended generators. The results produced by a generator need not all be of the same type.
- The evaluation of an expression may produce a value (*success*) or not produce any value at all (*failure*). If evaluation of an operand expression fails, the operation is not performed.

- Procedures are first-class values. Procedure parameters are not typed; the arguments of procedures can have different values at different times. The value returned by a procedure need not always have the same type.
- Structures are created during program execution. The size of a structure may grow or shrink after it is created. Structure values are pointers. Structures can be ''recursive'', containing pointers to other structures and to themselves. Such recursive structures preclude a finite type system that includes component types. Structures can be heterogeneous, containing values of different types.

Because of the freedom Icon offers to mix and change the types of values assigned to variables and contained in structures, it might seem that type usage in Icon programs would be hopelessly confused and hence prevent effective use of type inference. Fortunately, this is not the case. An empirical study using the type inference system described here was conducted on a large number of Icon programs written by many different authors for a wide variety of applications. The results, which are conservative, show a range of type consistency from about 60 to 100%, with an average of about 80%. That is, on the average, the operands of about 80% of the operators in these programs always have the same type.

This degree of type consistency is partly a natural consequence of the programming process in Icon and partly a reflection of generally good programming style. While it is possible to write an Icon program with no type consistency at all, such programs appear not to occur in practice. A low degree of type consistency is not necessarily an indication of poor programming style. It may occur for a very good reason, such as the use of polymorphous procedures.

Some examples may help in understanding how the features listed above affect type usage in Icon. Consider the expressions

```
line := read()
write(trim(line))
```

The expression read() fails if there is no data to read. If this happens, the assignment to line is not performed and it retains its former value. Consequently, the value of line after the evaluation of the assignment expression can be a string (if read() succeeds) or whatever it was before (if read() fails). If there was no prior assignment to line, its value after can be either a string or its initial null value. Of course, if line was a string before the assignment, it is a string afterward, whether or not read() fails.

Therefore, depending on what is known about the type of line before the assignment, it may or may not be necessary to perform type checking for the operand of trim(), which requires a string operand. On the other hand, in

```
while line := read() do
   write(trim(line))
```

the operand of trim() is necessarily a string, since the expression in the do clause is not evaluated unless read() succeeds.

Of all the features of Icon that affect type usage and hence type inference, structures present the most difficult problems. Consider

```
text := []
while line := read() do
   put(text, line)
```

The first expression creates an empty list and assigns it to text. The loop adds the strings read to this list, increasing its size accordingly.

In this case, text is a list of strings (or it is empty if no lines are read). Values of other types can be added to text, as in

```
put(text, 1991)
```

which adds and integer, or even

```
put(text, text)
```

which adds (a pointer to) text to text. As a result, text contains (possibly) strings, an integer and a list.

While such usage is improbable in the form shown here, it occurs frequently in Icon programs that manipulate graphs and in any event must be dealt with by the type inference system.

3. Type Inference in Icon

Two basic approaches have been taken when developing type inferencing schemes. Schemes based on unification [5-7] construct type signatures for procedures; schemes based on global data flow analysis [8-11] propagate the types variables may take on throughout a program. One strength of the unification approach is that it is effective at handling polymorphous procedures. Such schemes have properties that make them effective in implementing flexible compile-time type systems. Much of the research on them focuses on this fact. The primary purpose of the type inferencing system for the Icon compiler is to eliminate most of the run-time type checking rather than to report on type inconsistencies at compile time, so these properties have little impact on the choice of schemes used in the compiler. Type inferencing systems based on unification have a significant weakness. Procedure type-signatures do not describe side effects to global variables. Type inferencing schemes based on unification must make crude assumptions about the types of these variables.

Schemes based on global data flow analysis handle global variables effectively. Many Icon programs make significant use of global variables; this is a strong argument in favor of using this kind of type inferencing scheme for Icon. These schemes do a poor job of inferring types in the presence of polymorphous procedures. It is generally too expensive for them to compute the result type of a call in terms of the argument types of that specific call, so result types are computed based on the aggregate types from all calls. Poor type information only results if polymorphism is actually exploited within a program.

The primary use of polymorphous procedures is to implement abstract data types. Icon, on the other hand, has a rich set of built-in data types. While Icon programs make heavy use of these built-in data types and of Icon's polymorphous built-in operations, they seldom make use of user-written polymorphous procedures. While a type inferencing scheme based on global data flow analysis is not effective in inferring the precise behavior of polymorphous procedures, it is effective in utilizing the predetermined behavior of built-in polymorphous operations. These facts, combined with the observation that Icon programs often make use of global variables, indicate that global data flow analysis is the approach of choice for type inferencing in the Icon compiler.

A number of type inferencing systems handle recursion in applicative data structures [9, 12, 13]; the system described here handles Icon data types that have pointer semantics and handles destructive assignment to components of data structures. Analyses have been developed to handle pointer semantics for problems such as allocation optimizations and determining pointer aliasing to improve other analyses. However, most of these analyses lose too much information on heterogeneous structures of unbounded depth (such as the mutually referencing syntax trees and symbol tables commonly found in a translator) to be effective type inferencing systems [11, 14].

The Approach Taken

As mentioned above, the purpose of type inferencing for Icon is to provide information for use in eliminating run-time type checking. Consequently, a global flow analysis technique is used here. Its goal is to determine what types of values expressions may produce during the execution of a program. The most challenging part of this process is determining the types of variables and of the components of data structures.

The type inferencing system associates with each variable usage a set of the possible types of values that the variable might have when execution reaches the usage. In computing this set, type inferencing must take into account all code that might be executed before reaching the usage. Type inferencing computes similar sets of types for usages of data structure components. Each set may be a conservative estimate (overestimate) of the actual set of possible types that a variable may take on because the actual set may not be computable, or because an analysis to compute the actual set may be too expensive. However, a good type inferencing system operating on realistic programs can determine the exact set of types for most operands and the majority of these sets in fact contain single types, which is the information needed to generate code without type checking. The Icon compiler has an effective type inferencing system based on data flow analysis techniques.

4. Abstract Interpretation

Data flow analysis can be viewed as a form of abstract interpretation [15]. This is particularly useful for understanding type inferencing. A ''concrete'' interpreter for a language implements the standard (operational) semantics of the language, producing a sequence of states, where a state consists of an execution point, bindings of program variables to values, and so forth. An abstract interpreter does not implement the semantics, but rather computes information related to the semantics. For example, abstract interpretation may compute the sign of an arithmetic expression rather than its value. Often it computes a ''conservative'' estimate for the property of interest rather than computing exact information. Data flow analysis is simply a form of abstract interpretation that is guaranteed to terminate. This section presents a sequence of approximations to Icon semantics, culminating in one suitable for type inferencing.

Consider a simplified operational semantics for Icon, consisting only of program points (with the current execution point maintained in a program counter) and variable bindings (maintained in an environment). As an example of these semantics, consider the following program. Four program points are annotated with numbers using comments (there are numerous intermediate points that are not annotated).

```
procedure main()
   local s, n

   # 1:
   s := read()
   # 2:
   every n := 1 to 2 do {
      # 3:
      write(s[n])
      }
   # 4:
end
```

If the program is executed with an input of abc, the following states are included in the execution sequence (only the annotated points are listed). States are expressed in the form *program point*: *environment*.

```
1:  [s = null, n = null]
2:  [s = "abc", n = null]
3:  [s = "abc", n = 1]
3:  [s = "abc", n = 2]
4:  [s = "abc", n = 2]
```

It is customary to use the *collecting semantics* of a language as the first abstraction (approximation) to the standard semantics of the language. The collecting semantics of a program is defined in Cousot and Cousot [15] (they use the term *static semantics*) to be an association between program points and the sets of environments that can occur at those points during all possible executions of the program.

Consider the previous example. In general, the input to the program is unknown, so the read() function is assumed to be capable of producing any string. Representing this general case, the set of environments (once again showing only variable bindings) that can occur at point 3 is

```
[s = "", n = 1],
[s = "", n = 2],
[s = "a", n = 1],
[s = "a", n = 2],
  ...
[s = "abcd", n = 1],
[s = "abcd", n = 2],
  ...
```

A type inferencing abstraction further approximates this information, producing an association between each variable and a type at each program point. The actual type system chosen for this abstraction must be based on the

language and the use to which the information is put. The type system used here is based on Icon's run-time type system. For structure types, the system used retains more information than a simple use of Icon's type system would retain; this is explained in detail later. For atomic types, Icon's type system is used as-is. For point 3 in the preceding example, the associations between variables and types are

[s = string, n = integer]

The type inferencing system presented here is best understood as the culmination of a sequence of abstractions to the semantics of Icon, where each abstraction discards certain information. For example, the collecting semantics discards sequencing information among states in the preceding program, collecting semantics determine that, at point 3, states may occur with n equal to 1 and with n equal to 2, but it does not determine the order in which they must occur. This sequencing information is discarded because the desired type information is a static property of the program.

The first abstraction beyond the collecting semantics discards dynamic control flow information for goal-directed evaluation. The second abstraction collects, for each variable, the value associated with the variable in each environment. It discards information such as, ''x has the value 3 when y has the value 7'', replacing it with ''x may have the value 3 sometime and y may have the value 7 sometime.'' It effectively decouples associations between variables.

This second abstraction associates a set of values with a variable, but this set may be any of an infinite number of sets and it may contain an infinite number of values. In general, this precludes either a finite computation of the sets or a finite representation of them. The third abstraction defines a type system that has a finite representation. This abstraction discards information by increasing the set associated with a variable (that is, making the set less precise) until it matches a type. This third model can be implemented with standard iterative data flow analysis techniques.

To simplify the discussion, this section assumes that an Icon program consists of a single procedure and that all invocations are to built-in functions. The section on implementation describes the handing of the more general case.

Collecting Semantics

The collecting semantics of an Icon program is defined in terms a *flow graph* of the program. A flow graph is a directed graph used to represent the flow of control in a program. Nodes in the graph represent the executable primitives in the program. An edge exists from node **A** to node **B** if it is possible for execution to pass directly from the primitive represented by node **A** to the primitive represented by node **B**. Cousot and Cousot [15] prove that the collecting semantics of a program can be represented as the least fixed point of a set of equations defined over the edges of the program's flow graph. These equations operate on sets of environments.

For an example of a flow graph, consider an Icon program that consists of an expression that repeatedly resumes a generator and writes out the results:

```
procedure main()
   every write(1 to 3)
end
```

The diagram below on the left shows the abstract syntax tree for this procedure, including the implicit failure that occurs when control flows off the end of a procedure without an explicit return. The invoke node in the syntax tree represents procedure invocation. Its first argument must evaluate to the procedure to be invoked; in this case the first argument is the global variable write. The rest of the arguments are used as the arguments to the procedure. pfail represents failure of an invocation of the procedure (as opposed to expression failure within a procedure). Nodes corresponding to operations that produce values are numbered for purposes explained below.

A flow graph can be derived from the syntax trees a shown at the right: