Type Inference in the Icon Programming Language*

Kenneth Walker and Ralph E. Griswold

TR 93-32a

March 12, 1996

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

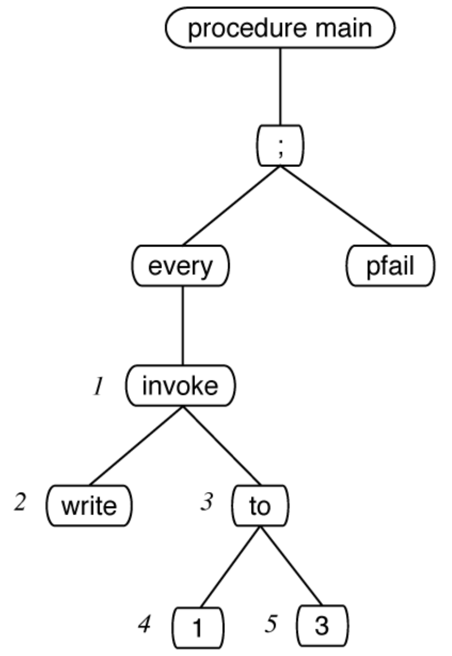
The collecting semantics of an Icon program is defined in terms a *flow graph* of the program. A flow graph is a directed graph used to represent the flow of control in a program. Nodes in the graph represent the executable primitives in the program. An edge exists from node **A** to node **B** if it 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.

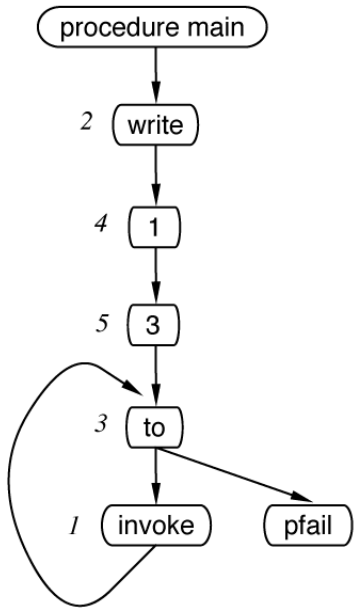
For an example of a flow graph, consider an Icon program that consists of an expression that repeatedly resumes a generator and writes out the results:

```
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:





The node labeled procedure main is the *start node* for the procedure; it performs any necessary initializations to establish the execution environment for the procedure. The edge from invoke to to is a resumption path induced by the control structure every. The path from to to pfail is the failure path for to. It is a forward execution path rather than a resumption path because the compound expression (indicated by ;) limits backtracking out of its left-hand sub-expression. The section on implementation describes how to determine the edges of the flow graph for an Icon program.

Both the standard semantics and the abstract semantics must deal with the intermediate results of expression evaluation. A temporary-variable model is used because it is more convenient for this analysis than a stack model. This analysis uses a trivial assignment of temporary variables to intermediate results. Temporary variables are not reused. Each node that produces a result is assigned some temporary variable r_i in the environment. Assuming that temporary variables are assigned to the example according to the node numbering, the to operation has the effect of

$$r_3 := r_4 \text{ to } r_5$$

Expressions that represent alternate computations must be assigned the same temporary variable, as in the following example for the subexpression x := ("a" | "b"), where the alternation control structure generates first "a" and then "b". The syntax tree below on the left and the and the flow graph are shown on the right.

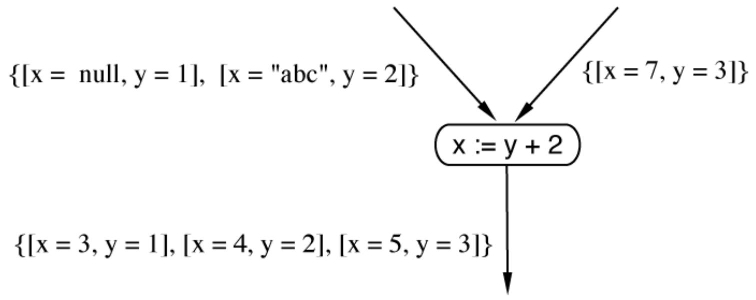
The equations that determine the collecting semantics of the program are derived directly from the standard semantics of the language. The set of environments on an edge of the flow graph is related to the sets of environments on edges coming into the node at the head of this edge. This relationship is derived by applying the meaning of the node (in the standard semantics) to each of the incoming environments.

It requires a rather complex environment to capture the full operational semantics (and collecting semantics) of a language like Icon. For example, the environment needs to include a representation of the external file system. However, later abstractions only use the fact that the function read() produces strings. This discussion assumes that it is possible to represent the file system in the environment, but does not give a representation. Other complexities of the environment are discussed later. For the moment, examples only show the bindings of variables to unstructured (atomic) values.

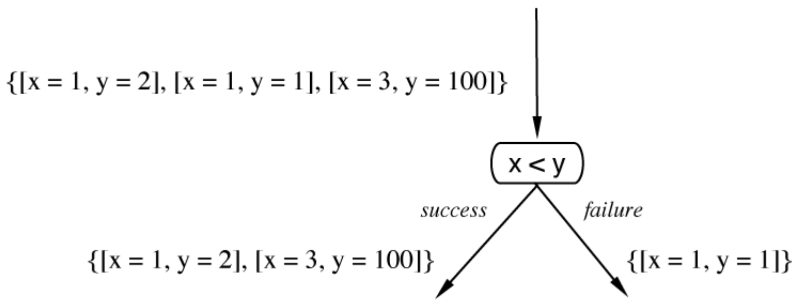
As an example of environments associated with the edges of a flow graph, consider the assignment at the end of the following code fragment. The comments in the if expression are assertions that are assumed to hold at those points in the example.

```
if x = 7 then {
   ...
   # x is 7 and y is 3
   }
else {
   ...
   # (x is null and y is 1) or (x is "abc" and y is 2)
   }
x := y + 2
```

Because of the preceding if expression, there are two paths reaching the assignment. The diagram below shows the flow graph and accompanying environments for the expression; the diagram ignores the fact that the assignment expression requires several primitive operations to implement.


{[x = null, y = 1], [x = "abc", y = 2]}
{[x = 7, y = 3]}
x := y + 2
{[x = 3, y = 1], [x = 4, y = 2], [x = 5, y = 3]}

For a conditional expression, an incoming environment is propagated to the path that it would cause execution to take in the standard semantics. This requires distinguishing the paths to be taken on failure (backtracking paths) from those to be taken on success. The following diagram shows an example of this.


{[x = 1, y = 2], [x = 1, y = 1], [x = 3, y = 100]}
x < y
success
failure
{[x = 1, y = 2], [x = 3, y = 100]}
{[x = 1, y = 1]}

In general there may be several possible backtracking paths. The environments in the standard and collecting semantics need to include a stack of current backtracking points and control flow information, and the flow graph needs instructions to maintain this stack. The Icon interpreter is an example of how this information can be maintained [1]. However, the first abstraction to the collecting semantics eliminates the need for this information, so the information is not presented in detail here.

Model 1: Eliminating Control Flow Information

The first abstraction involves taking the union of the environments propagated along all the failure paths from a node in the collecting semantics and propagating that union along each of the failure paths in the new abstraction. This abstraction eliminates the stack of backtracking points from the environment.

A more formal definition for this model requires taking a closer look at Icon data values, especially those values with internal structure. In order to handle Icon data objects with pointer semantics, an environment needs more than variable bindings. This fact is important to type inferencing. The problem is handled by including two components in the environment. The first is the *store*, which maps variables to values. Variables include *named* variables, *temporary* variables, and *structure* variables. Named variables correspond to program identifiers. Temporary variables hold intermediate results as discussed above. Structure variables are elements of structures such as lists. Note that the sets of named variables and temporary variables are each finite (based on the assumption that a program consists of a single non-recursive procedure; see the section on implementation for the handling of the general case), but for some non-terminating programs, the set of structure variables may be infinite. *Program* variables include both named variables and structure variables but not temporary variables.

Values include atomic data values such as integers, csets, and strings. They also include *pointers* that reference objects with pointer semantics. In addition to the values just described, temporary variables may contain references to program variables. These *variable references* may be used by assignments to update the store or they may be dereferenced by other operations to obtain the values stored in the variables.

The second part of the environment is the *heap*. It maps pointers to the corresponding data objects. For simplicity, the only data type with pointer semantics included in this discussion is the list. A list is a partial mapping from integers to variables. Representing other data types with pointer semantics is straightforward.

The first abstraction is called Model 1. The notations $\text{envir}_{[n]}$, $\text{store}_{[n]}$, and $\text{heap}_{[n]}$ refer to the sets of possible environments, stores, and heaps respectively in model *n*. For example, $\text{envir}_{[1]}$ is the set of possible environments in the first abstraction. In the following set of definitions, $X \times Y$ is the set of ordered pairs where the first value in the pair is from *X* and the second value is from *Y*. $X \to Y$ is the set of partial functions from *X* to *Y*. The definition of the set possible environments for Model 1 is

$$\text{envir}_{[1]} = \text{store}_{[1]} \times \text{heap}_{[1]}$$

$$\text{store}_{[1]} = \text{variables} \to \text{values}$$

$$\text{values} = \text{integers} \cup \text{strings} \cup \cdots \cup \text{pointers} \cup \text{variables}$$

$$\text{heap}_{[1]} = \text{pointers} \to \text{lists}, \ \text{where lists} = \text{integers} \to \text{variables}$$

For example, the expression

```
x := ["abc"]
```

creates a list of one element whose value is the string "abc" and assigns the list to the variable x. Let p_1 be the pointer to the list and let v_1 be the (anonymous) variable within the list. The resulting environment, $e \in \text{envir}_{[1]}$, might be

$$e = (s, h), \ \text{where } s \in \text{store}_{[1]},\ h \in \text{heap}_{[1]}$$

$$s(\text{x}) = p_1$$
$$s(v_1) = \text{"abc"}$$

$$h(p_1) = L_1, \ \text{where } L_1 \in \text{lists}$$

$$L_1(1) = v_1$$

If the statement

```
x[1] := "xyz"
```

is executed, the subscripting operation dereferences x producing p_1, then uses the heap to find L_1, which it applies to 1 to produce the result v_1. The only change in the environment at this point is to temporary variables that are not shown. The assignment then updates the store, producing

$$e_1 = (s_{1,}\ h)$$

$$s_1(\mathrm{x}) \;\; = p_1$$
$$s_1(v_1) = \text{"xyz"}$$

Assignment does not change the heap. On the other hand, the expression

```
put(x, "xyz")
```

adds the string "xyz" to the end of the list; if it is executed in the environment e, it alters the heap along with adding a new variable to the store.

$$e_1 = (s_{1,}\ h_1)$$

$$s_1(\mathrm{x}) \;\; = p_1$$
$$s_1(v_1) = \text{"abc"}$$
$$s_1(v_2) = \text{"xyz"}$$

$$h_1(p_1) = L_2$$

$$L_2(1) = v_1$$
$$L_2(2) = v_2$$

If a formal model were developed for the collecting semantics, it would have an environment similar to the one in Model 1. However, it would need a third component with which to represent the backtracking stack.

Model 2: Decoupling Variables

The next approximation to Icon semantics, Model 2, takes all the values that a variable might have at a given program point and gathers them together. In general, a variable may have the same value in many environments, so this, in some sense, reduces the amount of space required to store the information (though the space may still be unbounded). The ‘‘cost’’ of this reduction of storage is that any information about relationship of values between variables is lost.

Model 2 is also defined in terms of environments, stores, and heaps, although they are different from those of Model 1. A store in Model 2 maps sets of variables to sets of values; each resulting set contains the values associated with the corresponding variables in environments in Model 1. Similarly, a heap in Model 2 maps sets of pointers to sets of lists; each of these sets contains the lists associated with the corresponding pointers in environments in Model 1. An environment in Model 2 contains a store and a heap, but unlike in Model 1, there is only one of these environments associated with each program point. The environment is constructed so that it effectively ‘‘contains’’ the environments in the set associated with the point in Model 1.

The definition of Model 2 is

$$envir_{[2]} = store_{[2]} \times heap_{[2]}$$

$$store_{[2]} = 2^{\text{variables}} \rightarrow 2^{\text{values}}$$

$$heap_{[2]} = 2^{\text{pointers}} \rightarrow 2^{\text{lists}}$$

In Model 1, operations produce elements from the set *values*. In Model 2, operations produce subsets of this set. It is in this model that read() is taken to produce the set of all strings and that the existence of an external file system can be ignored.

Suppose a program point is annotated with the set containing the following two environments from Model 1.

$$e_{1,}\ e_2 \in\ envir_{[1]}$$

$$e_1 = (s_{1,}\ h_1)$$

$$s_1(\mathrm{x}) = 1$$
$$s_1(\mathrm{y}) = p_1$$

$$h_1(p_1) = L_1$$

$e_2 = (s_2, h_2)$

$s_2(\mathsf{x}) = 2$
$s_2(\mathsf{y}) = p_1$

$h_2(p_1) = L_2$

Under Model 2 the program point is annotated with the single environment $\hat{e} \in \mathrm{envir}_{[2]}$, where

$\hat{e} = (\hat{s}, \hat{h})$

$\hat{s}(\{\mathsf{x}\}) = \{1, 2\}$
$\hat{s}(\{\mathsf{y}\}) = \{p_1\}$
$\hat{s}(\{\mathsf{x}, \mathsf{y}\}) = \{1, 2, p_1\}$

$\hat{h}(\{p_1\}) = \{L_1, L_2\}$

Note that a store in Model 2 is distributive over union. That is,

$\hat{s}(X \cup Y) = \hat{s}(X) \cup \hat{s}(Y)$

so listing the result of $\hat{s}(\{\mathsf{x}, \mathsf{y}\})$ is redundant. A heap in Model 2 also is distributive over union.

In going to Model 2 information is lost. In the last example, the fact that $\mathsf{x} = 1$ is paired with $p_1 = L_1$ and $\mathsf{x} = 2$ is paired with $p_1 = L_2$ is not represented in Model 2.

Just as read() is extended to produce a set of values, so are all other operations. These ‘‘extended’’ operations are then used to set up the equations whose solution formally defines Model 2. This extension is straightforward. For example, the result of applying a unary operator to a set is the set obtained by applying the operator to each of the elements in the operand. The result of applying a binary operator to two sets is the set obtained by applying the operator to all pairs of elements from the two operands. Operations with more operands are treated similarly. For example

$$\begin{aligned}\{1, 3, 5\} + \{2, 4\} &= \{1 + 2, 1 + 4, 3 + 2, 3 + 4, 5 + 2, 5 + 4\}\\ &= \{3, 5, 5, 7, 7, 9\}\\ &= \{3, 5, 7, 9\}\end{aligned}$$

The loss of information mentioned above affects the calculation of environments in Model 2. Suppose the addition in the last example is from

```
z := x + y
```

and that Model 1 has the following three environments at the point before the calculation

$[\mathsf{x} = 1, \mathsf{y} = 2, \mathsf{z} = 0]$
$[\mathsf{x} = 3, \mathsf{y} = 2, \mathsf{z} = 0]$
$[\mathsf{x} = 5, \mathsf{y} = 4, \mathsf{z} = 0]$

After the calculation the three environments will be

$[\mathsf{x} = 1, \mathsf{y} = 2, \mathsf{z} = 3]$
$[\mathsf{x} = 3, \mathsf{y} = 2, \mathsf{z} = 5]$
$[\mathsf{x} = 5, \mathsf{y} = 4, \mathsf{z} = 9]$

If these latter three environments are translated into an environment of Model 2, the result is

$[\mathsf{x} = \{1, 3, 5\}, \mathsf{y} = \{2, 4\}, \mathsf{z} = \{3, 5, 9\}]$

However, when doing the computation using the semantics of + in Model 2, the value for z is $\{3, 5, 7, 9\}$. The solution to the equations in Model 2 overestimates (that is, gives a conservative estimate for) the values obtained by computing a solution using Model 1 and translating it into the domain of Model 2.

Consider the following code with respect to the semantics of assignment in Model 2. (Assume that the code is executed once, so only one list is created.)

```
x := [10, 20]
i := if read() then 1 else 2
x[i] := 30
```

After the first two assignments, the store maps x to a set containing one pointer and maps i to a set containing 1 and 2. The third assignment is not as straightforward. Its left operand evaluates to two variables; the most that can be said about one of these variables after the evaluation of the assignment expression is that it might have been assigned 30. If (s, h) is the environment after the third assignment then

$s(\{x\}) = \{p_1\}$
$s(\{i\}) = \{1, 2\}$
$s(\{v_1\}) = \{10, 30\}$
$s(\{v_2\}) = \{20, 30\}$

$h(\{p_1\}) = \{L_1\}$

$L_1(1) = v_1$
$L_1(2) = v_2$

Clearly all assignments could be treated as *weak updates* [16], where a weak update is an update that may or may not take place. However, this would involve discarding too much information; assignments would only add to the values associated with variables and not replace the values. Therefore assignments where the left-hand side evaluates to a set containing a single variable are treated as special cases. These are implemented as *strong updates*.

Model 3: A Finite Type System

The environments in Model 2 can contain infinite amounts of information, as in the program

```
x := 1
repeat x +:= 1
```

where the set of values associated with x in the loop consists of all the counting numbers. Because equations in Model 2 can involve arbitrary arithmetic, no algorithm can find the least fixed point of an arbitrary set of these equations.

The final step is to impose a finitely representable type system on values. A type is a (possibly infinite) set of values. The type system presented here includes three classifications of basic types. The first classification consists of the Icon types without pointer semantics: integers, strings, csets, etc. The second classification groups pointers together according to the lexical point of their creation. This is similar to the method used to handle recursive data structures in Jones and Muchnick [12]. Consider the code

```
x := []
every put(x, [1 to 5])
```

If this code is executed once, five lists are created in the second line, but they are all created at the same point in the program, so they all belong to the same type. The intuition behind this choice of types is that structures created at the same point in a program are likely to have components of the same type, while structures created at different points in a program are more likely to have components of different types.

The third classification of basic types handles variable references. Each named variable and temporary variable is given a type to itself. Therefore, if x is a named variable, {x} is a type. Structure variables are grouped into types according to the program point where the pointer to the structure is created. This is not necessarily the point where the variable is created; in the following code, a pointer to a list is created at one program point, but variables are added to the list at different points

```
x := []
push(x, 1)
push(x, 2)
```

References to these variables are grouped into a type associated with the program point for [], not the point for the corresponding push.

If a program contains k non-structure variables and there are n locations where pointers can be created, then the basic types for the program are integer, string, ..., P_1, ..., P_n, V_1, ..., V_n, $\{v_1\}$, ..., $\{v_k\}$ where P_i is the pointer type created at location i, V_i is the variable type associated with P_i, and v_i is a named variable or a temporary variable. Because programs are lexically finite they each have a finite number of basic types. The set of all types for a program is the smallest set that is closed under union and contains the empty set along with the basic types:

$$\text{types} = \{ \{\}, \text{integers}, \text{strings},\ldots, (\text{integers} \cup \text{strings}),\ldots, (\text{integers} \cup \text{strings} \cup \cdots \cup \{v_k\}) \}$$

Model 3 replaces the arbitrary sets of values of Model 2 by types. This replacement reduces the precision of the information, but allows for a finite representation and allows the information to be computed in finite time.

In Model 3, both the store and the heap map types to types. This store is referred to as the *type store*. The domain of type store is *variable types*, that is, those types whose only values are variable references. Similarly, the domain of the heap is *pointer types*. Its range is the set types containing only structure variables. A set of values from Model 2 is converted to a type in Model 3 by mapping that set to the smallest type containing it. For example, the set

{1, 4, 5, "23", "0"}

is mapped to

$\text{integer} \cup \text{string}$

The definition of $\text{envir}_{[3]}$ is

$$\text{envir}_{[3]} = \text{store}_{[3]} \times \text{heap}_{[3]}$$

$$\text{store}_{[3]} = \text{variable–types} \rightarrow \text{types}$$

$$\text{heap}_{[3]} = \text{pointer–types} \rightarrow \text{structure–variable–types}$$

$$\text{types} \subseteq 2^{\text{values}}$$

$$\text{variable–types} \subseteq \text{types}$$

$$\text{structure–variable–types} \subseteq \text{variable–types}$$

$$\text{pointer–types} \subseteq \text{types}$$

There is exactly one variable type for each pointer type in this model. The heap simply consists of this one-to-one mapping; the heap is of the form

$$h(P_i) = V_i$$

This mapping is invariant over a given program. Therefore, the type equations for a program can be defined over $\text{store}_{[3]}$ rather than $\text{envir}_{[3]}$ with the heap embedded within the type equations.

Suppose an environment from Model 2 is

$$e \in \text{envir}_{[2]}$$

$$e = (s, h)$$

$$s(\{x\}) = \{p_1, p_2\}$$
$$s(\{v_1\}) = \{1, 2\}$$
$$s(\{v_2\}) = \{1\}$$
$$s(\{v_3\}) = \{12.03\}$$

$$h(\{p_1\}) = \{L_1, L_2\}$$
$$h(\{p_2\}) = \{L_3\}$$

$$L_1(1) = v_1$$

$$L_2(1) = v_1$$
$$L_2(2) = v_2$$

$$L_3(1) = v_3$$

Suppose the pointers p_1 and p_2 are both created at program point 1. Then the associated pointer type is P_1 and the associated variable type is V_1. The corresponding environment in Model 3 is

$$\hat{e} \in \text{envir}_{[3]}$$

$$\hat{e} = (\hat{s}, \hat{h})$$

$$\hat{s}(\{x\}) \;\; = P_1$$
$$\hat{s}(V_1) = \text{integer} \cup \text{real}$$

$$\hat{h}(P_1) = V_1$$

The collecting semantics of a program establishes a set of (possibly) recursive equations between the sets of environments on the edges of the program's flow graph. The collecting semantics of the program is the least fixed point of these equations in which the set on the edge entering the start state contains all possible initial environments. Similarly, type inferencing establishes a set of recursive equations between the type stores on the edges of the flow graph. The least fixed point of these type inferencing equations is computable using iterative methods. See the section on implementation. The fact that these equations have solutions is due to the fact that the equations in the collecting semantics have a solution and the fact that each abstraction maintains the ‘‘structure’’ of the problem, simply discarding some details.

5. Implementation of the Type Inferencing System

The implementation of type inferencing is based on Model 3 of the last section. The implementation issues include:

- the representation of types and stores
- the handling of procedure calls
- the determination of edges in the flow graph
- the computation of a fixed point for type information

The Representation of Types and Stores

A type consists of a set of basic types (technically, it is a union of basic types, but the constituents of the basic types are not explicitly represented). The operations needed for these sets are: add a basic type to a set, form the union of two sets, form the intersection of two sets, test for membership in a set, and generate members of a subrange of basic types (for example, generate all members that are list types). A bit vector is used for the set representation, with a basic type represented by an integer index into the vector. The required operations are simple and efficient to implement using this representation. Unless the sets are large and sparse, this representation is also space efficient. While the sets of types are often sparse, for typical programs they are not large.

A store is implemented as an array of pointers to types. A mapping is established from variable references to indexes in the store. In Model 3 there is one kind of store that contains all variables. In the actual implementation, temporary variables need not be kept in this store. The purpose of this store is to propagate a change to a variable to the program points affected by the change. A temporary variable is set in one place in the program and used in one place; there is nothing to determine dynamically. It is both effective and efficient to store the type of a temporary variable in the corresponding node of the syntax tree.

Another level of abstraction can be developed that requires much less memory than Model 3, but it must be modified to produce good results. This abstraction abandons the practice of a store for every edge in the flow graph. Instead it has a single store that is a merger of all the stores in Model 3 (the type associated with a variable in a merged store is the union of the types obtained for that variable from each store being merged). For variables that are truly of one type throughout execution, this abstraction works well. Named variables do not have this property. They have an initial null value and usually are assigned a value of another type during execution. Because assignments to named variables are treated as strong updates, Model 3 can often deduce that a variable does not contain the null type at specific points in the flow graph.

For structure variables this further abstraction works well in practice. These variables are initialized to the empty type (that is, no instances of these variables exist at the start of program execution), so the problem of the initial null type does not occur. Sometimes instances of these variables are created with the null type and later changed.

However, the fact that assignments to these variables must be treated as weak updates means that type inferencing must assume that these variables can always retain their earlier type after an assignment. Propagating type information about structure variables through the syntax tree does not help much in these circumstances. While it is possible to construct example programs where Model 3 works better for structure variables than this further abstraction, experiments with prototype type inferencing systems indicate that the original system seldom gives better information for real programs [17].

Type inferencing in the compiler is implemented with two kinds of stores: local stores that are associated with edges in the flow graph and contain named variables (both local and global variables) and a global store that contains structure variables (in the implementation, the global store is actually broken up by structure-variable type into several arrays).

A Full Type System

Model 3 includes no structure types other than lists, nor does it consider how to handle types for procedure values to allow effective type inferencing in their presence. This section develops a complete and effective type system for Icon. Some of the details are omitted here to avoid obscuring the central issues. See [2] for a more detailed description.

Most of the structure types of Icon are assigned several subtypes in the type inferencing system. As explained for lists in the preceding section, these subtypes are associated with the program points where values of the type are created. The exception to this approach is records. One type is created per record declaration. While it is possible to assign a subtype to each use of a record constructor, in practice a given kind of record usually is used consistently with respect to the types of its fields throughout a program. The extra subtypes would require more storage while seldom improving the resulting type information.

For efficiency, the size of the bit vectors representing types and the size of the stores need to remain fixed during abstract interpretation. This means that all subtypes must be determined as part of the initialization of the type inferencing system. In order to avoid excessive storage usage, it is important to avoid creating many subtypes for program points where structures are not created. An invocation optimization [2] helps determine where structures can and cannot be created by determining for most invocations what operation is used. The type inferencing system determines what structures an operation can create by examining the abstract type computations associated with the operation. See [2] for an explanation of how these abstract type computations are represented.

The type system contains representations for all run-time values that must be modeled in the abstract interpretation. These run-time values can be divided into three categories, each of which is a superset of the previous category:

- the first-class Icon values
- the intermediate values of expression evaluation
- the values used internally by Icon operations

Just as these categories appear in different places during the execution of an Icon program, the corresponding types appear in different places during abstract interpretation. If certain types cannot appear as the result of a particular type computation, it is not necessary to have elements in the bit vector produced by the computation to represent those types. It is particularly important to minimize the memory used for stores associated with edges of the flow graph (this affects both space and time costs and is discussed more in the section on complexity later in this report). These stores contain only the types of the smallest set listed above: the first-class values.

Types (or subtypes) are allocated bit vector indexes. The first-class types may appear as the result of any of the three classes of computation. They are allocated indexes at the front of the bit vectors. If they are the only types that can result from an abstract computation, the bit vector for the result has no elements beyond that of the last first-class types. The first-class types include atomic types such as null, string, and integer, structure subtypes, and procedure subtypes.

The structure subtypes are allocated with

- one for each easily recognized creation point
- one representing all other structures of the type

The record subtypes are allocated with one for each record declaration. The procedure subtypes are allocated with

- one for each procedure
- one for each record constructor
- one for each built-in function

Procedure subtypes are allocated after most procedure and function values have been eliminated by invocation optimizations (the procedures and functions are still present, they are just not Icon values). Therefore, few of these subtypes are actually allocated.

The bit vectors used to hold the intermediate results of performing abstract interpretation on expressions must be able to represent the basic types plus the variable reference types. Variable reference types are allocated bit vector indexes following those of the basic types. The bit vectors for intermediate results are just long enough to contain these two classifications of types.

Identifiers are variables and are reflected in the type system. There is one type for each global variable except those eliminated by the invocation optimization. The local variable reference types are allocated with one for each variable, but the bit vector indexes and store indexes are reused between procedures. The next section describes why this reuse is possible.

Icon's operators use a number of internal values that never ''escape'' as intermediate results. Some of these internal values are reflected in the type system in order to describe the semantics of the operations in the abstract interpretation [2]. These types are allocated bit vector indexes at the end of the type system. The only bit vectors large enough to contain them are the temporary bit vectors used during interpretation of the abstract type computations of built-in operations.

Procedure Calls

A type inferencing system for the full Icon language must handle procedures. As noted above, each procedure is given its own type. This allows the type inferencing system to accurately determine which procedures are invoked.

The standard semantics for procedures can be implemented using stacks of procedure activation frames. The type inferencing system uses a trivial abstraction of these procedure frame stacks, while capturing the possible transfers of control induced by procedure calls.

The type inferencing system, in effect, uses an environment that has one frame per procedure, where that frame is a summary of all frames for the procedure that could occur in a corresponding environment of an implementation of the standard semantics. The frame is simply a portion of the store that contains local variables. Because no other procedure can alter a local variable, it is unnecessary to pass the types of local variables into procedure calls. If the called procedure returns control via a return, suspend, or fail, the types are unchanged, so they can be passed directly across the call. This allows the type inferencing system to keep only the local variables of a procedure in the stores on the edges of the flow graph for the procedure, rather than keeping the local variables of all procedures. Global variables must be passed into and out of procedure calls.

A flow graph for an entire program is constructed from the flow graphs for its individual procedures. An edge is added from every invocation of a procedure to the head of that procedure and edges are added from every return, suspend, and fail back to the invocation. In addition, edges must be added from an invocation of a procedure to all the suspends in the procedure to represent resumption. When it is not possible to predetermine which procedure is being invoked, edges are effectively added from the invocation to all procedures whose invocation cannot be ruled out based on the naive invocation optimizations. Information is propagated along an edge when type inferencing deduces that the corresponding procedure call might actually occur. The representation of edges in the flow graph is discussed below.

Type inferencing must reflect the initializations performed when a procedure is invoked. Local variables are initialized to the null value. The initialization code for the standard semantics is similar to

```
initialize locals
if (first_call) {
   user initialization code
   }
```

In type inferencing, the variables are initialized to the null *type* and the condition on the if is ignored. Type inferencing simply knows that the first-call code is executed sometimes and not others. Before entering the main procedure, global variables are set to the null type.

The Flow Graph and Type Computations

In order to determine the types of variables at the points where they are used, type inferencing must compute the contents of the stores along edges in the flow graph. Permanently allocating the store on each edge can use a large amount of memory. The usage is

$$M = E * (G + L) * T / 8$$

where

M = total memory, expressed in bytes, used by stores
E = the number of edges in the program flow graph
G = the number of global variables in the program
L = the maximum number of local variables in any procedure
T = the number of types in the type system

Large programs with many structure creation points can produce thousands of edges, dozens of variables, and hundreds of types, requiring megabytes of memory to represent the stores.

The code generation phase of the compiler just needs the (possibly dereferenced) types of operands, not the stores. If dereferenced types are kept at the expressions where they are needed, it is not necessary to keep a store with each edge of the flow graph.

Consider a section of straight-line code with no backtracking. Abstract interpretation can be performed on the graph starting with the initial store at the initial node and proceeding in execution order. At each node, the store on the edge entering the node is used to dereference variables and to compute the next store if there are side effects. Once the computations at a node are done, the store on the edge entering the node is no longer needed. If updates are done carefully, they can be done in-place, so that the same memory can be used for both the store entering a node and the one leaving it.

In the case of branching control paths (as in a `case` expression), abstract interpretation must proceed along one path at a time. The store at the start the branching of paths must be saved for use with each path. However, it need only be saved until the last path is interpreted. At that point, the memory for the store can be reused. When paths join, the stores along each path must be merged. The merging can be done as each path is completed; the store from the path can then be reused in interpreting other paths. When all paths have been interpreted, the merged store becomes the current store for the node at the join point.

The start of a loop is a point where control paths join. Unlike abstract interpretation for the joining of simple branching paths, abstract interpretation for the joining of back edges is not just a matter of interpreting all paths leading to the join point before proceeding. The edge leaving the start of the loop is itself on a path leading to the start of the loop. Circular dependencies among stores are handled by repeatedly performing the abstract interpretation until a fixed point is reached. In this type inferencing system, abstract interpretation is performed in iterations, with each node in the flow graph visited once per iteration. The nodes are visited in execution order. For back edges, the store from the previous iteration is used in the interpretation. The stores on these edges must be kept throughout the interpretation. These stores are initialized to map all variables to the empty type. This represents the fact that the abstract interpretation has not yet proven that execution can reach the corresponding edge.

The type inferencing system never explicitly represents the edges of the flow graph in a data structure. Icon is a structured programming language. Many edges are implicit in a tree walk performed in forward execution order — the order in which type inferencing is performed. The location of back edges must be predetermined in order to allocate stores for them, but the edges themselves are effectively recomputed as part of the abstract interpretation.

There are two kinds of back edges. The back edges caused by looping control structures can be trivially deduced from the syntax tree. A store for such an edge is kept in the node for the control structure. Other back edges are induced by goal-directed evaluation. These edges are determined with the same techniques used in liveness analysis [2]. A store for such an edge is kept in the node of the suspending operation that forms the start of the loop. Because the node can be the start of several nested loops, this store is actually the merged store for the stores that theoretically exist on each back edge.

At any point in abstract interpretation, three stores are of interest. The *current store* is the store entering the node on which abstract interpretation is currently being performed. It is created by merging the stores on the incoming edges. The *success store* is the store representing the state of computations when the operation succeeds. It is usually created by modifying the current store. The *failure store* is the store representing the state of computations when the operation fails.

In the presence of a suspended operation, the failure store is the store kept at the node for that operation. A new failure store is established whenever a resumable operation is encountered. This works because abstract interpretation is performed in forward execution order and resumption is LIFO. Control structures, such as if-then-else, with branching and joining paths of execution, cause difficulties because there may be more than one possible suspended operation when execution leaves the control structure. This results in more than one failure store during abstract interpretation. Rather than keeping multiple failure stores when such a control structure has operations that suspend on multiple paths, type inferencing pretends that the control structure ends with an operation that does nothing other than suspend and then fail. It allocates a store for this operation in the node for the control structure. When later operations that fail are encountered, this store is updated. The failure of this imaginary operation is the only failure seen by paths created by the control structure and the information needed to update the failure stores for these paths is that in the store for this imaginary operation. This works because the imaginary operation just passes along failure without modifying the store.

In the case where a control structure transforms failure into forward execution, as in the first subexpression of a compound expression, the failure store is allocated (with empty types) when the control structure is encountered and deallocated when it is no longer needed. If no failure can occur, no failure store need be allocated. The lack of possible failure is noted while the location of back edges is being computed during the initialization of type inferencing. Because a failure store may be updated at several operations that can fail, these are weak updates. Typically, a failure store is updated by merging the current store into it.

The interprocedural flow graph described earlier in this section has edges between invocations and returns, suspends, and fails. Type inferencing does not maintain separate stores for these theoretical edges. Instead it maintains three stores per procedure that are mergers of stores on several edges. One store is the merger of all stores entering the procedure because of invocation; this store contains parameter types in addition to the types of global variables. Another store is the merger of all stores entering the procedure because of resumption. The third store is the merger of all stores leaving the procedure because of returns, suspends, and fails. There is also a result type associated with the procedure. It is updated when abstract interpretation encounters returns and suspends.

When type inferencing encounters the invocation of a built-in operation, it performs abstract interpretation on a simplified representation of the operation. An operation is re-interpreted at each invocation. This allows type inferencing to produce good results for polymorphous operations. All side effects within these operations are treated as weak updates; the only strong updates recognized by type inferencing are assignments to named variables [2].

A global flag is set any time an update changes type information that is used in the next iteration of abstract interpretation. The flag is cleared between iterations. If the flag is not set during an iteration, a fixed point has been reached and the interpretation halts.

6. Complexity of the Type Inferencing System

An upper bound on the space requirements of type inferencing is determined by the memory required to represent the stores kept across iterations of inferencing. The number of these stores can be proportional to the size of the program.

The size of a bit vector representing a type is determined by the number of types in the type system chosen for the program being analyzed. Because a structure sub-type is allocated for every structure creation point, the number of structure sub-types can be proportional to the size of the program. Some other types, such as procedures, can also be this large. The size of each bit vector therefore is $O(n)$.

Stores are arrays of types, with one type for every variable. The number of variables in a program also can be proportional to the size of the program. The number of bits needed to represent a store in the worst case is $O(n^2)$. Therefore, the total number of bits used by type inferencing can be $O(n^3)$.

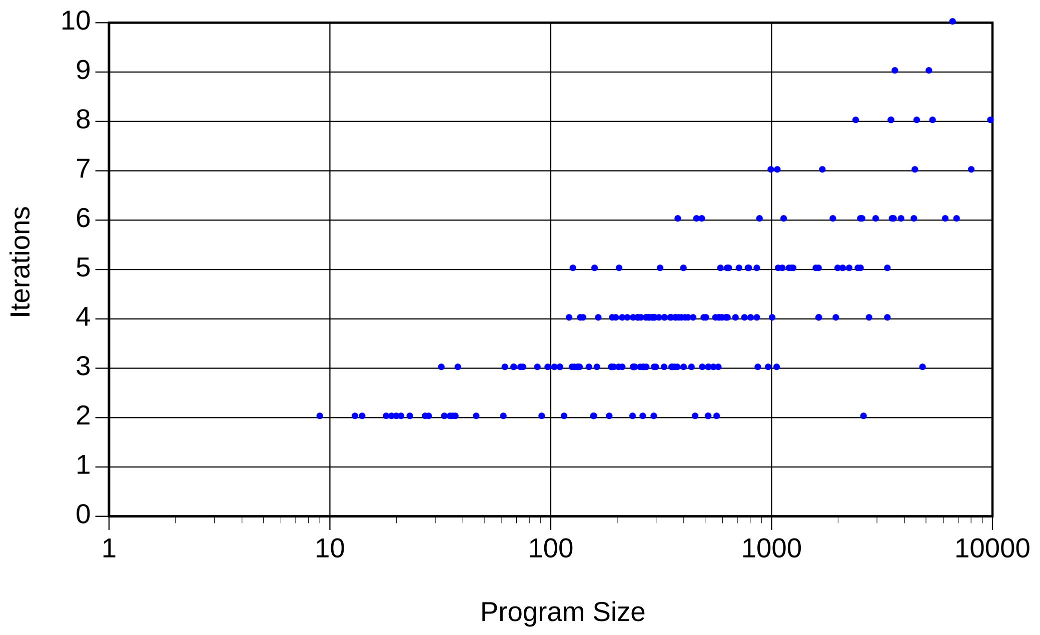
The upper bound on the time complexity of type inferencing is the worst-case number of iterations required to reach a fixed point times the worst-case cost per iteration. An iteration turns on bits in the vectors representing types but never turns them off. Therefore the number of iterations is bounded by the number of bits in the bit vectors, which is $O(n^3)$.

The worst-case cost of each iteration is the number of operations in the program times the worst-case cost of abstract interpretation on an operation. The number of operations is proportional to the size of the program and contributes a factor of $O(n)$. The cost of abstract interpretation consists of the cost of implementing the abstract semantics of the operation plus the cost of propagating stores along edges of the flow graph. The former cost is no more than a constant times the size of a type times the number of arguments. In the case of an operation that takes an arbitrary number of arguments, the cost can be charged back to the argument expressions. The cost of implementing the abstract semantics of an operation is $O(n)$ but is dominated by the cost of propagating stores to and from the operation. The most expensive operation in this respect is an invocation that type inferencing concludes may involve any procedure in the program. Stores must be propagated to and from each of these procedures. The number of procedures can be proportional to the size of the program and, as explained above, stores can be $O(n^2)$ in size. Therefore the worst case cost of an iteration is $O(n^4)$, and the time complexity of type inferencing is $O(n^7)$.

While it is possible to contrive programs that demonstrate the worst-case behavior of type inferencing, inferencing performs much better on ‘‘real’’ programs. In fact it requires a certain amount of ingenuity to construct programs the demonstrate even a number of iterations that is linear in the size of the program. Programs that demonstrate $O(n^3)$ iterations look like exercises from the theory of computation; practical programs are never that convoluted.

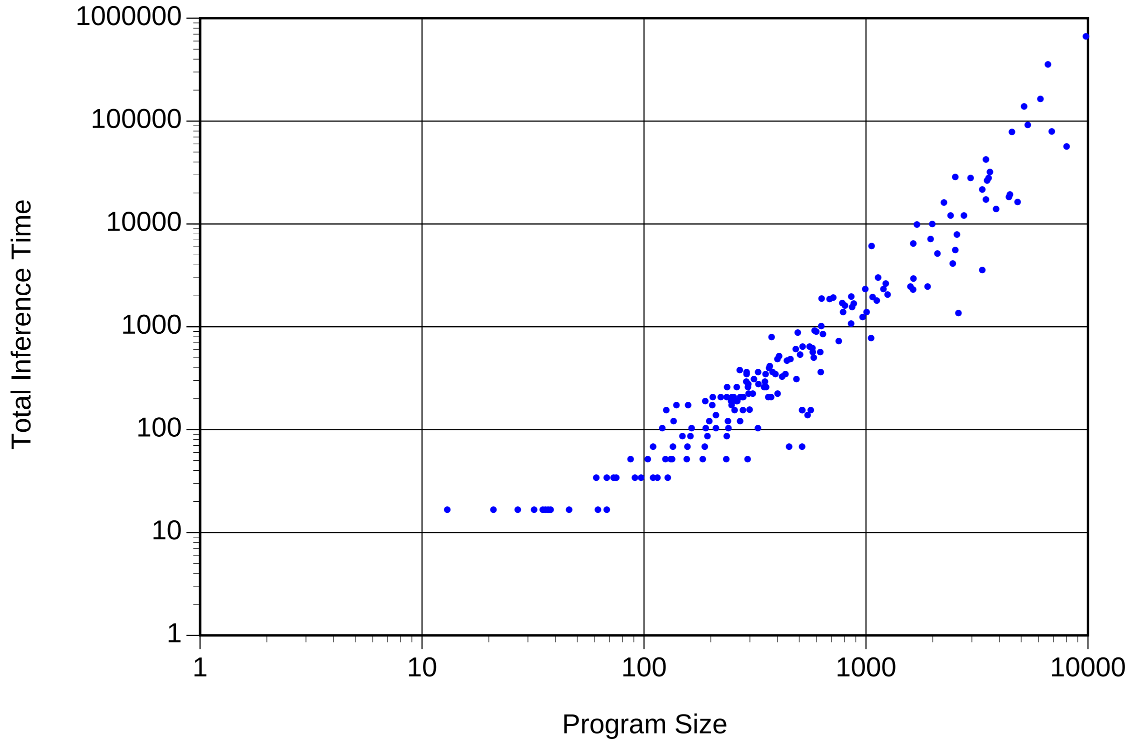
Measurements of the cost of type inference have been made on a large suite of Icon programs written by many different authors for a wide variety of applications. Many of these programs are very large by Icon standards. In these measurements, program size is determined by counting the number of syntactic elements in a program.

For this suite, the number of iterations performed by type inference shows only a slight dependency on program size. However, the number does appear to show a logarithmic upper-bound as illustrated by the following graph where program size is plotted on a logarithmic scale.



As explained in the analysis, the worst-case cost per iteration is only obtained with a flow graph containing $O(n^2)$ edges. Practical programs have flow graphs with $O(n)$ edges. If practical programs also have $O(n)$ global variables and type inferencing assigns $O(n)$ subtypes to the programs, then type inferencing will cost $O(n^3)$ per iteration. It is reasonable to expect some programs to exhibit this behavior and demonstrate a total cost of $O(n^3 \log n)$. Whether this is a problem depends on the size of the programs and on the constants associated with the higher order terms of the cost function.

Experimental results are plotted in the following graph on a log-log scale. Times are in milliseconds and were measured on a workstation-class machine. The points cluster around a curve whose slope suggests that lower order terms predominate for typical sized programs.



From a user’s point of view, the important question is the expected time for type inferencing in real programs. As the graph shows, type inference requires at most a few seconds for typical programs and requires more than a minute only for a few of the largest ones. The extreme time of nearly 11 minutes for one program is still a reasonable amount of time to spend in analysis for a final production compilation of such a huge program.

7. Performance of the Type Inference System

The effectiveness of type inferencing in improving the performance of a compiled Icon program depends on many factors, including the degree of type consistency. As mentioned earlier, the degree of type consistency in practical programs varies from about 60% to 100%, with an average of about 80%.

The results of type inference interact with other optimizations. For example, if type inference determines that type checking does not need to be performed for the operands of an operation, the code bulk for that operation may be reduced to the extent that it can be implemented by in-line code instead of a call to a subroutine. It therefore is not possible to measure the effect of type inference in isolation. Instead, the effectiveness of type inference can be determined by measuring the difference in program performance with type inference enabled and disabled.

Consider the expression:

```
1 + 2 + 3 + 4
```

In this expression, type inference can determine that the arguments of the addition operations are all integers, so that no type-checking code is needed. This expression executes about 5.8 times faster with type inference than without it, even though addition in Icon is checked for overflow.

While figures such as this are valid, they do not tell the complete story. Some Icon programs spend a significant amount of time in operations such as associative look up and storage management, where the quality of compiled code has no effect. Consequently, the improvement in the overall performance of Icon programs as the results of compiler optimizations varies widely, with a figure about about 3.5 being typical.

Measurement of the performance of a large number of Icon programs shows that type inference contributes about half of the improvement, on the average.

8. Conclusions

The type inferencing system described here was first developed in prototype [17] to test its soundness. This prototype system was then used to determine that Icon programs exhibit sufficient type consistency to justify a production type inferencing system.

As with any complex program analysis, the cost of type inference is a very important consideration. Working independently in the field of pointer analysis, Chase, Wegman, and Zadeck [16] conceived and rejected techniques similar to ones developed here to handle pointer semantics in type inference. They speculated that this type of technique would be too expensive and proceeded to develop cheaper abstractions suited to their needs. The Icon compiler proves that, if implemented carefully, such techniques are practical for type inference in Icon.

While the type inferencing system developed here does not perform perfect inferencing, the degree of type consistency displayed in Icon programs, combined with the observed effectiveness of the existing inferencing system, suggests that further work on increasing the accuracy of the type inferencing system probably is not justified. Instead, it probably would be more beneficial to try to make better use of the information provided by the current type inferencing system.

If the new optimizing compiler encourages programmers to write larger Icon programs, it may be that the machine resources needed by type inference will exceed what can be practically supplied by current workstations. Future work includes developing techniques for reducing the cost of type inference through a controlled reduction in the accuracy of information computed. This reduction must be determined on a program-by-program basis to allow accurate information to continue to be computed for typical programs.

Currently, type information is used only by the compiler’s code generator. Another interesting possibility is to use the type inferencing system to supply programmers with information that will enable them to write better programs. Experience indicates that the raw output of type inference is too voluminous and unfocused for this purpose and that the effects of program changes attempting to use this information are not intuitive. Effective tools are needed to guide the programmer in the use of such information.

Acknowledgements

We would like to thank Saumya Debray for sharing his knowledge of abstract interpretation. We also would like to thank Janalee O’Bagy, Kelvin Nilsen, and David Gudeman for helpful comments on the presentation of the material in Section 4. We also would like to thank Gregg Townsend for assistance in evaluating experimental data.

References

1. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language,* Princeton University Press, 1986.
2. K. Walker, *The Implementation of an Optimizing Compiler for Icon*, Doctoral Dissertation, The University of Arizona, July 1991.
3. R. E. Griswold and M. T. Griswold, *The Icon Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1990.
4. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
5. R. Milner, ‘‘A Theory of Type Polymorphism in Programming’’, *Journal of Computer and System Sciences 17,* 3 (Dec. 1978), 348-375.
6. N. Suzuki, ‘‘Inferring Types in Smalltalk’’, *Eighth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1981, 187-199.

7. J. A. Robinson, ‘‘A Machine-Oriented Logic Based on the Resolution Principle’’, *J. ACM 12*, 1 (Jan. 1965), 23-41.

8. M. A. Kaplan and J. D. Ullman, ‘‘A General Scheme for the Automatic Inference of Variable Types’’, *Fifth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1978, 60-75.

9. G. Weiss and E. Schonberg, *Typefinding Recursive Structures: A Data-Flow Analysis in the Presence of Infinite Type Sets*, Technical Report #235, Courant Inst. of Mathematical Sciences, New York Univ, Aug. 1986.

10. M. S. Hecht, *Flow Analysis of Computer Programs*, North-Holland, New York, NY, 1977.

11. S. S. Muchnick and N. D. Jones, *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

12. N. D. Jones and S. S. Muchnick, ‘‘A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures’’, *Ninth ACM Symposium on Principles of Programming Languages*, 1982, 66-74.

13. P. Mishra, ‘‘Towards a Theory of Types in Prolog’’, *Proceedings 1984 Symposium on Logic Programming*, Montvale, NJ, 1984, 289-298.

14. S. Horwitz, P. Pfeiffer and T. Reps, ‘‘Dependence Analysis for Pointer Variables’’, *Proceeding of the 1989 Conference on Programming Language Design and Implementation, SIGPLAN Notices 24*, 7 (July 1989), 28-40.

15. P. Cousot and R. Cousot, ‘‘Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints’’, *Fourth ACM Symposium on Principles of Programming Languages*, 1977, 238-252.

16. D. R. Chase, M. Wegman and F. K. Zadeck, ‘‘Analysis of Pointers and Structures’’, *Proceeding of the 1990 Conference on Programming Language Design and Implementation, SIGPLAN Notices 25*, 6 (June 1990), 296-310.

17. K. Walker, *A Type Inference System for Icon*, The Univ. of Arizona Tech. Rep. 88-25, 1988.