A Type Inference System for Icon*

Kenneth Walker

TR 88-25

July 5, 1988

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8502015.

A Type Inference System for Icon

1. Introduction

Variables in the Icon programming language are untyped [1]. That is, a variable may take on values of different types as the execution of a program proceeds. In the following example, x contains a string after the read (if the read succeeds), but it is then assigned an integer or real, provided the string can be converted to a numeric type.

```
x := read()
if numeric(x) then x +:= 4
```

In general, it is impossible to know the type of an operator's operands at translation time, so some type checking must be done at run time. This type checking may result in type conversions, run-time errors, or the selection among polymorphous operations (for example, the selection of integer verses real addition). In the current implementation of Icon [2], all operators check all of their operands at run time. This incurs significant overhead.

Much of this run-time type checking is unnecessary. An examination of typical Icon programs reveals that the types of most variables remain consistent throughout execution (except for the initial null value) and that these types can often be determined by inspection. Consider

```
if x := read() then
   y := x || ";"
```

Clearly both operands of || are strings so no checking or conversion is needed.

The goal of a type inference system [3] is to determine what types variables may take on during the execution of a program. It associates with each variable usage a set of the possible types of values that variable might have when execution reaches the usage. This 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 inference 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.

Icon has several types of non-applicative data structures. They all can be heterogeneous and can be used in a recursive manner. An effective type inference system must handle these data structures without loosing too much information through crude assumptions.

Type inference for non-applicative languages such as Icon can be accomplished using data flow analysis techniques [4-6]. This report describes the particular data flow analysis framework used to perform type inference on Icon programs. It does a good job of type inference in the presence of recursive data structures. A number of type inference systems handle recursion in applicative data structures [7-9]; the system described here deals with Icon data types that have pointer semantics and deals with destructive assignment to components of data structures. In the chapter *Flow Analysis and Optimization of Lisp-like Structures* in [5], Jones and Muchnick describe a data flow analysis technique to determine the "shape" of non-applicative data structures, but that analysis looses too much information in structures of unbounded depth (such as trees) to be an effective type inference system.

Any data flow analysis system for Icon programs must deal with Icon's goal-directed evaluation and its unique control structures. Determining possible execution paths through an Icon program is more complicated than it is for programs written in more conventional languages. This report includes the description of a method for computing such paths.

2. Abstract Interpretation

Data flow analysis can be viewed as a form of abstract interpretation [10]. This can be particularly useful for understanding type inference. A "concrete" interpreter for a language implements the (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, an 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 inference.

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 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 *static semantics* (also referred to as *collecting semantics*) of a language as the first abstraction (approximation) to the operational semantics of the language. The statics semantics of a program is defined in Cousot and Cousot [10] 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.

Once again, consider the previous example. In general, the input to the program is unknown, so the read function is assumed cabable of produce 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 inference abstraction further approximates this information, producing an association between each variable and a type at each program point. For point 3 in the preceding example the associations are

[s = string, n = integer]

The type inference system presented in this paper 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 static semantics discards sequencing information among states; in the preceding program, statics semantics determine that, at point 3, states may occur with n equal to 1 and with n equal to 2, but not the order in which they must occur. This sequencing information is discarded because desired type information is a static property of the program.

This section proceeds with a more detailed description of static semantics for Icon, then defines a sequence of three increasingly less precise abstractions of these static semantics. The first abstraction discards dynamic control flow information. Its purpose is to simplify the semantic model, making further abstractions easier.

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.

The 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.

This section assumes that an Icon program consists of a single procedure and that all invocations are to built-in functions. See Section 4 for information on how to extend the abstractions to multiple procedures.

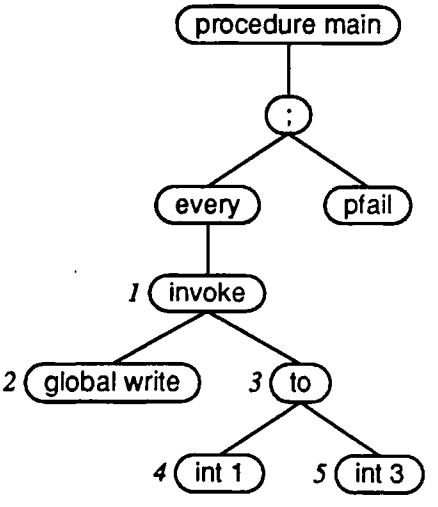
2.1 Static Semantics

The static 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 [10] prove that the static 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 the Icon program

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

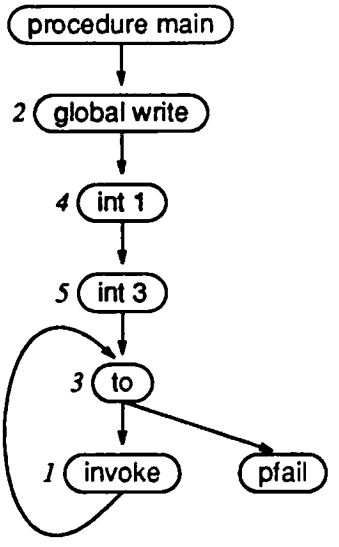
With the addition of the implicit fail at the end of the procedure, its parse tree is



Some of the node labels are based loosely on the *ucode* generated by the Icon translator for the corresponding

language constructs (see [2]). invoke corresponds to 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. int *n* corresponds to the integer literal *n*. pfail corresponds to fail (which causes procedure failure). Nodes corresponding to operations that produce values are numbered for purposes explained below.

A flow graph can be derived from the parse tree:

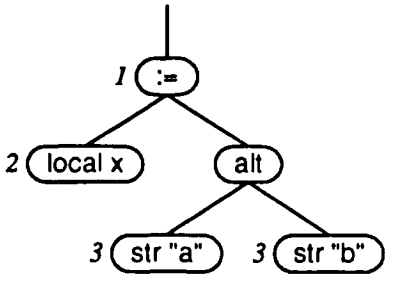


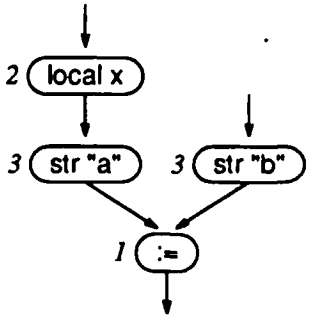
The node labeled procedure main is the *start node* for the procedure; it performs any necessary initializations to establish the execution environment for the procedure. The edge from invoke to to is a resumption path induced by the control structure every. The path from to to pfail is the failure path 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. Section 3 of this paper describes how to construct a flow graph from an Icon program.

The ucode instructions labeling the nodes of the graphs reflect the fact that the implementation of Icon uses a stack-based virtual machine: there are no explicit operands. The static semantics must deal with the intermediate results of expressions. However, a temporary variable (register) model is more convenient for this than a stack model. Each node that produces a result is assigned some 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 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"). Both the parse tree and the flow graph are shown.





The if and case control structures are handled similarly. In addition to temporaries for intermediate results, some

generators may need additional temporary variables to hold internal state during suspension; it is easy to devise a scheme to allocate them where they are needed. The parse tree is kept during abstract interpretation and used to determine the temporary variables associated with an operation and its operands.

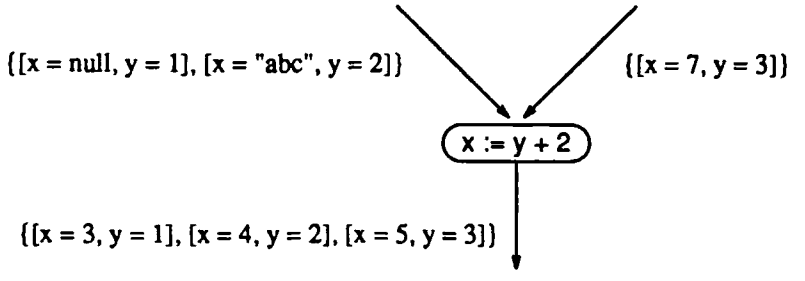
The equations that determine the static semantics of the program are derived directly from the semantics of the language. The set of environments on an edge of the flow graph is related to the sets of environments on edges coming into the node at the head of this edge. This relationship is derived by 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 static 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 paper 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 will only show the bindings of variables to unstructured (atomic) values.

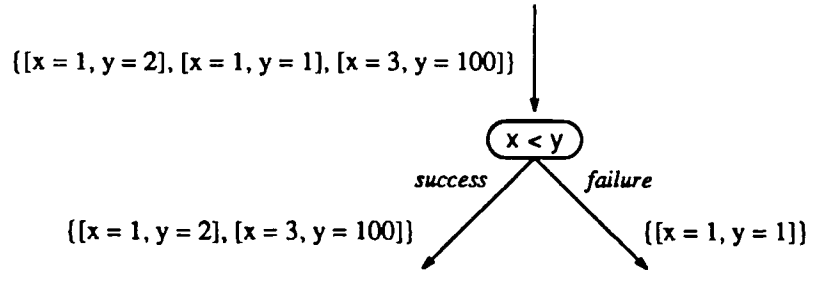
As an example of environments associated with the edges of a flow graph, consider the assignment at the end of the following code fragment. The comments in the if expression are assertions that are assumed to hold at those points in the example.

```
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. Ignoring the fact that the assignment expression requires several primitive operations to implement, the flow graph and accompanying environments for the expression are



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

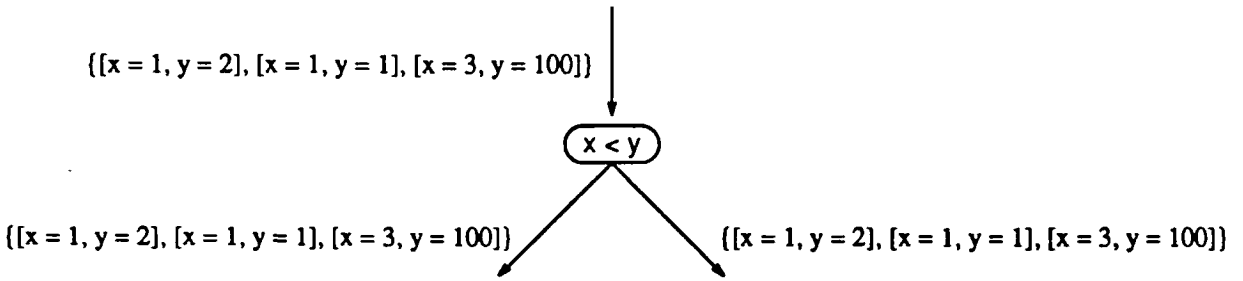


In general there may be several possible backtracking paths. The environments in the concrete and static semantics

need to include a stack of current backtracking points and control flow information, and the flow graph needs instructions to maintain this stack. See [2] for Icon implementation techniques. However, the first abstraction eliminates the need for this information (and for distinguishing success and failure paths), so the information is not presented in detail here.

2.2 Model 1: Eliminating Control Flow Information

The first abstraction involves taking the union of the environments propagated along the success and failure paths from a node in the static semantics and propagating that union along each of the paths in the new abstraction. The last example in the preceding section becomes:



A more formal definition for this model will be developed. This 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 inference. 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 consisting of a single non-recursive procedure), 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 (be mapped 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 (this differs from the heap in the Icon implementation in that that heap also contains some data objects that do not have pointer semantics). For simplicity, the only data type with pointer semantics included in this discussion is the list. It is a partial mapping from integers to variables. Representing other such data types is straightforward. The environment definition for the first abstraction is called Model 1, indicated by a subscript of [1]. Its formal definition is:

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

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

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

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

For example, the expression

```
a := ["abc"]
```

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

$e = (s, h)$, where s: $store_{[1]}$, h: $heap_{[1]}$

$s(a) = p_1$
$s(v_1) =$ "abc"

$h(p_1) = L_1$, where L_1: list

$L_1(1) = v_1$

If the statement

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

is executed, the subscripting operation dereferences a 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

$e_1 = (s_1, h)$

$s_1(a) = p_1$
$s_1(v_1) =$ "xyz"

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

```
put(a, "xyz")
```

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

$e_1 = (s_1, h_1)$

$s_1(a) = p_1$
$s_1(v_1) =$ "abc"
$s_1(v_2) =$ "xyz"

$h_1(p_1) = L_2$

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

2.3 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 infinite). The "cost" of this reduction of storage is that the any information about relationship of values between variables is lost.

Model 2 involves constructing new types of environment, store, and heap. The new store maps sets of variables to a sets of values; each resulting set contains the values associated with the corresponding variables in environments in Model 1. Similarly, the new heap 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. The new environment contains a store and heap, but unlike the old environment, there is only one of these environments associated with each program point. The envirionment is constructed so that it effectively "contains" the environments in the set associated with the point in Model 1.

Actually, the definitions of the store and heap are extended to operate on arbitrary sets of values; here, values include temporary variables, even though only program variables are values in Icon (and then only as intermediate results). In the actual implementation of the type system (see Section 5), temporary variables are handled separately, but for these models it is convenient to treat them like other variables. If the store in Model 2 is applied to a set containing a non-variable value, that value does not participate in the result; this corresponds to the fact that applying the store in Model 1 to such a value is undefined. Similarly, non-pointer values do not participate in results from the heap in Model 2 (note that the heap is only applied to dereferenced values, so variables will never appear).

The definition of Model 2 is

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

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

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

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.

$$e_1, e_2 \in envir_{[1]}$$

$$e_1 = (s_1, h_1)$$

$$s_1(x) = 1$$
$$s_1(y) = p_1$$

$$h_1(p_1) = L_1$$

$$e_2 = (s_2, h_2)$$

$$s_2(x) = 2$$
$$s_2(y) = p_1$$

$$h_2(p_1) = L_2$$

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

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

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

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

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 the entry for $\hat{s}(\{x, y\})$ is redundant. A store applied to a non-variable is the empty set, as in

$$\hat{s}(\{5\}) = \{\}$$

Consequently, such values need not be explicitly represented. A heap in Model 2 is also distributive over union; in this case it is non-pointers that need not be explicitly represented.

In going to Model 2 information is lost. In the last example, the fact that $x = 1$ is paired with $p_1 = L_1$ and $x = 2$ 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

$[x = 1, y = 2, z = 0]$
$[x = 3, y = 2, z = 0]$
$[x = 5, y = 4, z = 0]$

after the calculation the three environments will be

$[x = 1, y = 2, z = 3]$
$[x = 3, y = 2, z = 5]$
$[x = 5, y = 4, z = 9]$

If this is translated into the environment of Model 2, the result is

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

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, give a conservative estimate) for the values obtained by obtaining 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 assignment 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 being "uncertain", but 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.

2.4 Model 3: A 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. No algorithm can find the least fixed point of an arbitrary set of equations in Model 2.

The final step is to impose a finite type system on values. A type is a (possibly infinite) set of values. This 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 [7]. Consider the code

```
every insert(x, [1 to 5])
```

If this code is executed once, five lists are created, 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 may 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. 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. The store is only concerned with the variable types in its input. The heap is only concerned with the pointer types in its input and only produces types associated with structure variables. A set of values from Model 2 is converted to a type in Model 3 by mapping that to set the smallest type containing it. For example, the set

$$\{1, 4, 5, \text{"23"}, \text{"0"}\}$$

is mapped to

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

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

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

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

$$\text{heap}_{[3]} = \text{types} \rightarrow \text{types}$$

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

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

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

$$e = (s, h)$$

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

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

$L_1(1) = v_1$

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

$L_3(1) = v_3$

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

$\hat{e} \in envir_{[3]}$

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

$\hat{s}(\{a\}) = P_1$
$\hat{s}(V_1) = integer \cup real$

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

3. Constructing Flow Graphs

Icon's goal-directed evaluation scheme and its unique control structures pose a challenge for constructing flow graphs for Icon programs. This section presents a solution to that problem in the form of an attribute grammar. The attribute grammar is presented here using a convenient notation; it is implemented by a hand-coded walk of the syntax tree. Productions in the attribute grammar are of the form

```
<non-terminal> ::= <sequence of symbols> {
        <attribute calculations>
        }
```

If there are several instances of a non-terminal in a production, they are distinguished by subscripts. These subscripts do not affect the underlying context-free grammar, but are important in the attribute calculations. An attribute of a symbol is specified using dot notation. For example, the succ attribute of $expr_1$ is denoted

$expr_1$.succ

The nodes in the flow graph are taken to be a subset of the nodes in the syntax tree. This is convenient to implement and, for the type inference system presented here, it is adequate. Every symbol in the attribute grammar is assumed to have an attribute named node whose value is the corresponding node in the syntax tree.

Edges in the flow graph are represented by arrows. Standard mathematical set notation is used in the attribute computations. For example, the set of all possible edges from expr.node to nodes in the set expr.succ is

{x→y : x = expr.node, y ∈ expr.succ}

The use of braces here is distinguished from those surrounding the attribute computations.

Symbols have several *synthesized* attributes and several *inherited* attributes. The synthesized attributes contain information which is passed to the surrounding expression. The inherited attibutes contain information which comes from the surrounding expression.

The symbol expr has a synthesized attribute named begin. The value of begin is the first node to be executed in the expression represented by the symbol. For primitive expressions like literals and identifiers, it is the node associated with that symbol. For example, the attribute computation for a literal is (in this case, the node for expr and literal are identical):

```
expr ::= literal {
        expr.begin := expr.node
          ...
        }
```

For most operators it is the first node to be executed in the first subexpression:

```
expr ::= expr1 + expr2 {
        expr.begin := expr1.begin
          ...
        }
```

The computation of the begin attribute is straightforward and is omitted in the following examples.

Expressions have two inherited attributes, succ and fail. The value of the succ attribute for an expression is a set of nodes and contains all nodes that might be executed next if the expression succeeds. Similarly, the value of the fail attribute is a set of nodes that contains all nodes that might be executed next if the expression fails.

Expressions have two other synthesized attributes, resume and edges. The value of the resume attribute is a set of nodes. It contains the nodes where execution might continue if control backtracks to this expression. For expressions that cannot be resumed, the set of nodes is synthesized from the nodes of an expression preceding this one in execution order. For example, if the expression

```
2
```

is in the context

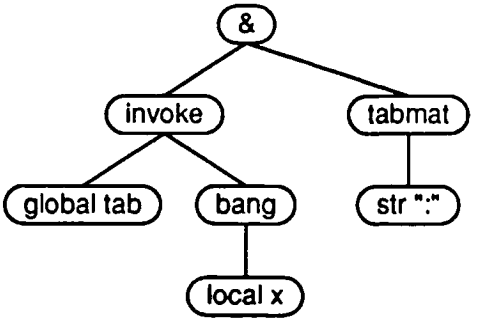
```
!x + 2
```

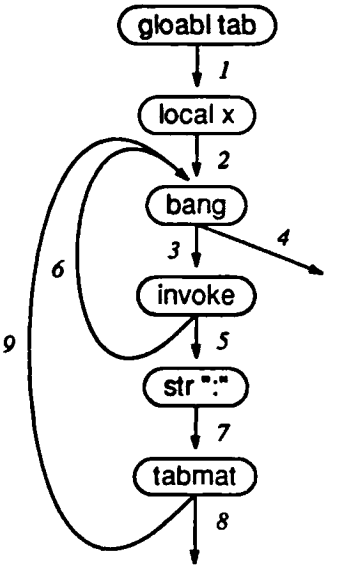
then its resume set consists of the node for !.

The value of the edges attribute for an expression is the set of edges in the flow graph that originate within the expression (the edges may or may not terminate within the expression). Consider the expression

```
tab(!x) & =":"
```

Its parse tree and flow graph (with edges numbered) are





Each node of the parse tree is attributed by the edges set of the corresponding sub-expression. For example, the edges set for local x contains edge 2, the set for bang contains edges 2-4, the set for invoke contains 1-6, etc.

In the actual implementation, the edges attribute is not explicitly represented. Instead the edges of the flow graph are stored in a more efficient manner.

Literals and identifiers cannot be resumed, so their resume nodes are the fail nodes supplied by the context. These expressions always succeed, so they contribute edges from their node to all nodes in the succ set supplied by the context.

```
expr ::= literal {
        expr.resume := expr.fail
        expr.edges := {x→y : x = expr.node, y ∈ expr.succ}
        }
```

The keyword &fail always fails, so it is impossible for control to backtrack into it; it has no resume nodes. It contributes edges from its node to all nodes in its fail set.

```
expr ::= &fail {
        expr.resume := {}
        expr.edges := {x→y : x = expr.node, y ∈ expr.fail}
        }
```

Icon operators can be categorized by the number of results they can produce. A *monogenic* operator produces exactly one result. A *conditional* operator produces either 0 or 1 result (that is it either fails or succeeds with one result). A *generative* operator can produce several results.

Addition is a typical monogenic binary operator. If its first operand succeeds, control passes to the beginning of the second operand. If the first operand fails, the entire expression fails. If the second operand succeeds, the operation is executed. If the second operand fails, the first operand is resumed (the operands are responsible for producing the correct resume set even if it is something before them that is actually resumed). If the expression is resumed, then the second operand is resumed, because the operation itself cannot produce a second result. The edges originating from this expression are those from the two operands plus all edges from the addition node to nodes in the succ set.

```
expr ::= expr1 + expr2 {
        expr1.succ := {expr2.begin}
        expr1.fail := expr.fail
        expr2.succ := {expr.node}
        expr2.fail := expr1.resume
        expr.resume := expr2.resume
        expr.edges := expr1.edges ∪ expr2.edges ∪ {x→y : x = expr.node, y ∈ expr.succ}
        }
```

The unary operator / ("is null") is conditional. If the operand succeeds, the operator is executed. If the operand fails, the expression fails. If the expression is resumed, the operand is resumed. This operator contributes edges from the operator node to nodes in both the succ set of the expression and, because the operator can fail, to the resume set of the operand.

```
expr ::= /expr1 {
        expr1.succ := {expr.node}
        expr1.fail := expr.fail
        expr.resume := expr1.resume
        expr.edges := expr1.edges ∪ {x→y : x = expr.node, y ∈ (expr.succ ∪ expr1.resume)}
        }
```

The unary operator ! ("bang") is generative. It is similar to "is null", except that the operator, rather than the operand, is resumed first during backtracking:

```
expr.resume := {expr.node}
```

Other operators follow patterns similar to the three just discussed. Control structures, on the other hand, present unique attribute computations. Many control structures do not contribute nodes to the graph. Instead they just affect

what edges are put in the graph. This is because they do not perform any data computations; they just affect the flow of control. The productions for several control structures follow.

The not control structure reverses the roles of the succ and fail sets. not *bounds* the expression it is applied to; that is, it prevents backtracking into the expression. This bounding is reflected in the fact that the resume set of the expression is discarded.

```
expr ::= not expr1 {
        expr1.succ := expr.fail
        expr1.fail := expr.succ
        expr.resume := expr.fail
        expr.edges := expr1.edges
        }
```

The semicolon causes control to pass from its first operand to its second regardless of whether the first one succeeds or fails. The first operand is bounded so its resume set is discarded. The second operand shares the succ, fail, and resume attributes with the expression as a whole.

```
expr ::= expr1 ; expr2 {
        expr1.succ := {expr2.begin}
        expr1.fail := {expr2.begin}
        expr2.succ := expr.succ
        expr2.fail := expr.fail
        expr.resume := expr2.resume
        expr.edges := expr1.edges ∪ expr2.edges
        }
```

If the first operand of alternation succeeds, the entire expression succeeds. If the first operand fails, the second operand is executed. The succ nodes and fail nodes of the second expression correspond to those of the entire expression. If the expression is resumed, either operand could be resumed depending on which one produced the last result.

```
expr ::= expr1 | expr2 {
        expr1.succ := expr.succ
        expr1.fail := {expr2.begin}
        expr2.succ := expr.succ
        expr2.fail := expr.fail
        expr.resume := expr1.resume ∪ expr2.resume
        expr.edges := expr1.edges ∪ expr2.edges
        }
```

If the control clause of an if expression succeeds, the then expression is executed. If the control clause fails, the else expression is executed. The resume points of the control clause are discarded. The succ nodes and fail nodes of both the then expression and the else expression are the same as those of the entire expression. If the if expression is resumed, either the then or the else clause could be resumed depending on which was executed.

```
expr ::= if expr1 then expr2 else expr3 {
        expr1.succ := {expr2.begin}
        expr1.fail := {expr3.begin}
        expr2.succ := expr.succ
        expr2.fail := expr.fail
        expr3.succ := expr.succ
        expr3.fail := expr.fail
        expr.resume := expr2.resume ∪ expr3.resume
        expr.edges := expr1.edges ∪ expr2.edges ∪ expr3.edges
        }
```

Constructing the flow graph for a case expression requires some additional attributes beyond those used for simpler expressions. There are two ways for control to leave an expression, but there are three ways for control to leave a

case clause. If the comparison between the case control expression and a case clause selection expression succeeds, the clause is selected and control ultimately passes to a node in either succ or fail for the case expression. If the comparison fails, control passes to the selection expression of the next case clause if there is one and to a node in fail for the case expression if there is no next clause. The inherited attribute cmpfail is the set of nodes to which control may pass if the comparison fails. The succ and fail attributes are the same as those for the entire case expression. The node for the case clause is used to represent the comparison operation for the clause.

```
case–clause ::= expr1 : expr2 {
        expr1.succ := {case–clause.node}
        expr1.fail := case–clause.cmpfail
        expr2.succ := case–clause.succ
        expr2.fail := case–clause.fail
        case–clause.resume := expr2.resume
        case–clause.edges := expr1.edges ∪ expr2.edges ∪
        {x→y : x = case–clause.node, y ∈ (expr1.resume ∪ {expr2.begin})}
        }
```

The resume set for the entire case expression is the union of these sets for the clauses.

Additional attributes are needed to indicate the presence of the default clause and to pass its begin node to the end of the case list. This involves some conditional attribute computations. The productions for the default clause, the case list, and the case expression are rather involved and have been omitted.

In the discussion so far, the effects of break and next expressions have been ignored. The attributes of a loop and those of enclosed break and next expression are related, so it is necessary to communicate between them. Consider:

```
while
   repeat
      break break !"abc"
```

The first break has the effect of replacing the repeat loop with

```
break !"abc"
```

which in turn has the effect of replacing the while loop with

```
!"abc"
```

The succ set and fail set for the while loop must be passed down to !"abc" and the resume set of !"abc" (the ! node) must be passed up to the while loop.

These loop attributes must be maintained in a stack-like manner. A loop must push loop-succ and loop-fail values. A break expression must use the top values on these stacks and pop them. Similarly, a break expression must push a loop-resume value. A loop must use the top value on this stack and pop it. An additional inherited attribute, loop-next, is needed to supply the destination for next expressions. This must also be maintained as a stack with loops pushing values and break expressions poping them.

Other expressions must pass loop-succ, loop-fail, and loop-next attributes onto their sub-expressions. They must also merge, on an element-wise basis, the loop-resume stacks of the sub-expressions. Note that all the stack operations must be applicative.

4. Interprocedural Analysis

The analysis described so far ignores the fact that Icon programs can be made up of several possibly recursive procedures. Procedure calls can be implemented in an interpreter for the real semantics by including in the environment a stack of activation frames containing local variables. In the abstract interpreter for type inference there is only one frame for each procedure. This frame is a summary of all possible frames for the real semantics. In Model 3, there is still only one store; part of it contains global variables and part of it contains local variables, with the local part changing from procedure to procedure.

This summarizing of procedure frames is effective as long as procedures are used consistently, that is, always called with parameters of the same type. While it is quite easy to write polymorphous procedures in Icon, this does not seem to be exploited a great deal. A major use of polymorphous procedures would be to implement an abstract data type such as a queue, which could be used in different contexts to contain different types of elements, but Icon already has powerful built-in data structures along with functions to manipulate them. It is typically only these built-in functions that are polymorphous and the type inference system takes their polymorphous nature into account. The type system does handle polymorphous procedures "correctly", but discards potentially useful information.

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 all the invocations of that procedure. In addition, edges must be added from an invocation of a procedure to all the suspends in the procedure to represent resumption.

These edges are different from other edges in that only types for variables with global lifetimes are transmitted over them. Because static variables have global lifetimes, they are transmitted throughout the program, even though they are only used in one procedure. Local variables are transmitted around an invocation, unaffected by the procedure being invoked.

The head of a procedure initializes local variables to the null type. It is also treated as if it contains initialization code to set static variables to the null type on the "first call". This code would look something like

```
initialize locals
if (first-call) then {
  initialize statics
  user initialization code
  }
```

However, in the flow graph this just looks like a branching of paths; the first-call condition effectively disappears like other success/failure information. Before entering the main procedure, global variables are set to the null type and all static variables are set to the empty type. In some sense the empty type represents an impossible execution path. There is a path in the flow graph from the start of the program to the body of a procedure that never passes through the initialization code. However, static variables will have an empty type along this path and no operation on them is valid.

Procedures in Icon are first-class values and procedure names are global variables whose values can be changed during execution; it may not be possible to determine the call graph of a program (the interprocedural part of the flow graph) without first doing type inference. However, type inference needs the call graph. This "chicken and egg" problem is solved by effectively constructing the call graph during type inference.

The soundness of an iterative data flow analysis system is not dependent on the order in which edges are visited; what is important is that all edges are eventually visited. This visiting of all edges must then be repeated until a fixed point is reached. Thus it is not necessary to know all the edges at the start of the analysis, as long as they are all known before the end of the analysis.

Each procedure is given its own basic type. Thus at an invocation, the type of what is being invoked indicates all procedures that might actually be invoked. This type is used to construct the interprocedural edges associated with the invocation. This type may not be complete during the early part of type inference, but it is guaranteed to be complete by the end.

It is also useful to know invocation information when constructing flow graphs within procedures. This is because some procedure invocations can fail and others cannot; it is best to eliminate as many failure edges as is practical from the flow graph. It is also helpful to eliminate unneeded resumption paths. It would be possible to wait and add these edges during type inference when they were found necessary, but a simpler (if less exact) approach is used.

In the actual implementation of the type inference system, it is assumed that assignments are not made to procedure names. If such an assignment is detected, type inference gives up (and no type information is known other than that any operand can be of any type). This seems to be a reasonable assumption; programmers sometimes assign to the names of built-in functions, but this is almost always a subtle error and not an intentional use of the ability to make such an assignment. Thus when the name of a procedure is used in an invocation, it is known exactly

what is being invoked. In cases where something other than a simple procedure name is being invoked, it is conservative to assume that the invocation can both fail and generate values.

5. Implementation

A prototype implementation of the type inference system has been written in Icon. It takes as input the parse tree of an Icon program and outputs information about the types of operands. The prototype constructs a flow graph, determines the set of basic types for the program, and then solves the type inferencing equations by iterating over the flow graph, computing successive approximations until the type information converges.

The prototype type inference systems differs in several details from Model 3. These details include more efficient representations for some information along with the implementation of things not covered in the model. The prototype has been tested on a large number of Icon programs and the quality of the results has been compared to that of a very simple type-determining system. Finally, a more efficient version of type inference has been implemented and tested. The rest of this section discusses these aspects of the implementation in more detail.

5.1 Differences Between the Model and the Prototype

In the abstract interpretation models presented in this paper, temporary variables are kept in the store and references to them are values in the type system. This is unnecessary. A temporary variable has exactly one *definition* where its value is set and at most one *use* where its value is extracted, and the definition always precedes the use. Therefore, only one copy of the temporary is needed. In the implementation, this copy is kept in the node of the parse tree that the temporary is associated with. All references to temporary variables are implicit in the parse tree, so temporary variable references are not included as values in the implemented type system.

Icon has several structure types with pointer semantics. In addition to lists, there are sets, tables, and records. Sets are collections of values. Values may be added or deleted from a set. The members of a set are not variables. However, it is convenient to put them in the store as "variables" which are never changed. This allows them to be treated uniformly with the components of other structures. A type is allocated for each set creation point and a variable type is created for each set type. This variable type contains all the "variables" in all the sets in the corresponding set type.

Tables are mappings from values to variables. When used as mappings, they act as total mappings. The variables that have not been explicitly assigned contain the default value specified when the table was created. However, when the elements of a table are generated or when the table is sorted, only the elements that have been explicitly assigned values are produced. Therefore, a table needs to be implemented as a partial mapping from values to variables (those elements defined by assignments) along with a default value. A type is allocated for each creation point of a table and variable types are allocated for the key, entry, and default value for each table type.

A record is a fixed set of variables (fields). Individual variables can be selected by field name or by integer position. Each record declaration introduces a record "type". However, as with lists, a type is added to the type system for each creation point (call to the record constructor) for a given record "type". A variable type is created for each field for each record type. This variable type contains the corresponding field from each record in the record type. This choice of variable types reflects the fact that a field is likely to be used consistently with respect to type across all records of a given record type, but may be used differently from other fields in these records.

Because record constructors and built-in functions can be assigned to variables and then invoked through those variables, it may not be possible to identify all structure creation points until type inference is in progress. However, it is easiest to allocate all variable types before type inferencing starts. This problem is solved by allocating an extra type along with the associated variable type(s) for each category of structure. If creation points are found that were not identified before type inference started, the structures produced at those points are put in the corresponding extra type.

Assignments are variable definitions, but in addition, some variable uses can also be treated as definitions. Consider the expression

```
x ||| ["end"]
```

If x does not contain a list, execution will terminate with an error message. Therefore, the abstract list concatenation in the type inference system can intersect the type of x with the set of list types, defining a new type for x in its

output store. As an example of where this improves the results of type inference, suppose x is assigned a list with the statement

```
x := [read()]
```

The programmer may be confident that the read will succeed, but type inference can not make that assumption. If x is the null type before the assignment, type inference gives it the type "list or null" after the assignment expression. However, after the list concatenation expression, x is given the type "list". Thus the code generation phase of a compiler must be able to produce code to check the type of x at the concatenation, but not thereafter.

5.2 Empirical Performance

The type inference system has been run on a suite of 514 Icon programs. This suite includes programs used to test Icon implementations, programs written as course assignments, and application programs written to solve a variety of problems. While no claim can be made that the suite accurately represents a cross-section of typical Icon programs, it does include the work of many different programmers with a variety of programming styles.

Type information was accumulated for all the operands where such type information could be useful in a code generator. Types were classified as consisting of a unique Icon type or consisting of multiple Icon types. Icon types differ from the types in the type inference system in that they only contain dereferenced values, there is only one list type, etc. The results of performing type inference on the suite is

unique types:	22973
multiple types:	2333

Thus, the type inference system determined a unique type for about 90% of all operands.

During testing, a number of operands with no type were found. These either represent unreachable code or meaningless expressions. In this way, type inference pointed out a number of programming errors. (None of these programs were included in the above results.)

In order to find out how much information is being gained by using a powerful type inference system, a simple type-determining system was written for comparison. It does a bottom-up pass of the parse tree. It assigns literals and keywords their corresponding type, but assigns variables "any type", that is, the union of all types. It treats operations very much like the type inference system does, computing a result type based on operand types. In order for this to be a fair comparison, the simple type determining system assumes that procedure names, function names, and record constructor names cannot be assigned to, just as the type inference system makes this assumption. The following results are for the same programs and operands as are the above results for type inference. Unique types have been broken into several categories (note that procedure names also include record constructor names and function names).

unique types	
literal constants:	6116
keywords:	337
procedure names:	6745
other:	3101
	16299
multiple types:	9007

Thus, without type inference, it is only possible to determine unique types for about 64% of all operands.

5.3 A Variation on Type Inference

One variation of the type inference system has been implemented. It is based on the observation that structure variable types always represent a set of variables so that assignments can only add to the associated type and never restrict it. For this reason, propagating variable/type associations for these structure variables from node to node through the flow graph may not add to the precision of the type information.

In this variation of type inference, named variables are still put in the stores that are propagated through the flow graph, but structure variables are put in a "global" store that is shared by all nodes. When the suite of programs was run through this system, only one operand among all of the programs went from having a unique type to having

multiple types. On programs with structure types, this system runs significantly faster than the theoretically more precise system.

6. Conclusions

An earlier data flow analysis framework for type inference of Icon was developed in a rather *ad hoc* manner. The present type inference system evolved out of an attempt to explain and justify that earlier system. Abstract interpretation has proven useful not only for describing type inference, but also for understanding and reasoning about it. The current system is simpler and more closely reflects the semantics of Icon than the earlier system.

The current implementation is rather slow and uses a lot of memory. Both of these problems are due to the fact that type information about every variable is propagated to every node. The usual solution to this problem is to precompute *definition-use* chains that link variable definitions to variable uses. Because structure variables can reasonably be put in a global store, definition-use chains are only needed for named variables. These definition-use chains can be computed before type inference begins, although invoking a non-procedure variable may occasionally require making pessimistic assumptions about the definition-use chains of global variables. With this information, variable types need only be kept at variable definitions. In addition, a type need only be recomputed when another type it depends on changes. These dependencies are reflected in the definition-use chains and in the parse trees of expressions. Using these techniques and implementing type inference in C should result in a system efficient enough for use in a production compiler.

The information produced by this type inference system presents numerous opportunities for a compiler to eliminate type checking from generated code. In the present Icon implementation (based around an interpreter), all of the type checking is done in run-time routines. With type inference, type checking is only needed for 10% of all operands, making it practical (at least in a compiler) to remove type checking from most run-time routines and generate it in-line instead. In addition, as a byproduct of type inference, it is possible to identify operands that are not variable references. Using this information, unnecessary dereferencing can also be eliminated without any additional analysis. Beyond aiding in the generation of good code, type inference can detect cases where operands cannot be valid, allowing the production of better diagnostic information.

Acknowledgements

Ralph Griswold served as the research advisor for this project. Saumya Debray provided help with the concepts of abstract interpretation. Ralph Griswold, Saumya Debray, Janalee O'Bagy, Kelvin Nilsen, and David Gudeman provided useful comments on this report.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
2. R. E. Griswold and M. T. Griswold, *The Implementation of The Icon Programming Language*, Princeton University Press, 1986.
3. 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.
4. M. S. Hecht, *Flow Analysis of Computer Programs*, North-Holland, New York, NY, 1977.
5. S. S. Muchnick and N. D. Jones, *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
6. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.
7. 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.

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

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. 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.

Appendix A — Proof of Consistency for the Abstractions

This appendix contains a proof that Model 1 is consistent with the static semantics and proofs for two operations that Model 3 is consistent with Model 1. The conditions used to guarantee that a model is consistent with a less abstract model depend on imposing a lattice on the domain of each model. The domain of static semantics and Model 1 is sets of environments ($envir_{[1]}$). The lattice used here is the usual lattice for a power set: subset is the ordering relation ($\leq$), union is the join operation ($\vee$), the empty set is bottom, and the set of all environments is top.

The domain of Model 3 is $store_{[3]}$. Assume

$$s_1, s_2 \in store_{[3]}$$

then the ordering relation and join operation are defined by

$$s_1 \leq s_2 \text{ iff } \forall t \in types, s_1(t) \subseteq s_2(t)$$

$$s_1 \vee s_2 = s_3, \text{ where } \forall t \in types, s_3(t) = s_1(t) \cup s_2(t)$$

The first step in proving the that $store_{[3]}$ is a lattice under this ordering relation is to show that the type system forms a lattice under the subset relation. The basic types are non-empty and disjoint. The type system is constructed to be the smallest set closed under union containing the basic types and the empty set. Using these facts, it can be shown that each type can be uniquely represented as a union of zero or more basic types and conversely, that each such union is a type. Thus the type system with the subset relation is isomorphic to the power set of type names with the subset relation and the type system forms a lattice. Is is then easy to show that the lattice of the type system induces a lattice structure on $store_{[3]}$.

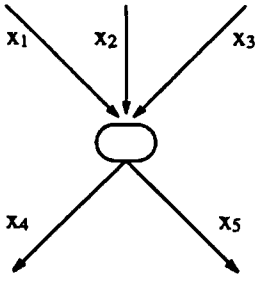
Let D_i be the domain of abstraction i. Given an edge m of the flow graph with n being the node at the head of m, a *transfer function*

$$f_{m,i}: D_i \rightarrow D_i$$

implements the portion of the semantics in abstraction i of node n that causes control to transfer to m. The input to the function is the join of the values on the incoming edges. Suppose

$$x_1, x_2, x_3, x_4, x_5 \in D_i$$

are the values associated with the edges of the graph



and assume that the transfer functions of the outgoing edges are $f_{4,i}$ and $f_{5,i}$, then the equations for this graph are

$$x_4 = f_{4,i}(x_1 \vee x_2 \vee x_3)$$

$$x_5 = f_{5,i}(x_1 \vee x_2 \vee x_3)$$

A transfer function needs to be monotone, that is

$$\forall x_1, x_2 \in D_i \text{ s.t. } x_1 \leq x_2, f_{m,i}(x_1) \leq f_{m,i}(x_2)$$

A model i is said to be consistent with a less abstract model j, if there exists an *abstraction function*

$$\alpha: D_j \rightarrow D_i$$

and a *concretization function*

$$\gamma: D_i \rightarrow D_j$$

such that the following four conditions hold

1. $\forall x_1, x_2 \in D_j$ s.t. $x_1 \leq x_2$, $\alpha(x_1) \leq \alpha(x_2)$
2. $\forall \hat{x}_1, \hat{x}_2 \in D_i$ s.t. $\hat{x}_1 \leq \hat{x}_2$, $\gamma(\hat{x}_1) \leq \gamma(\hat{x}_2)$
3. $\forall \hat{x} \in D_i$, $\hat{x} = \alpha(\gamma(\hat{x}))$
4. $\forall x \in D_j$, $x \leq \gamma(\alpha(x))$

and, in addition, for every edge m

5. $\forall x \in D_j$, $\alpha(f_{m,j}(x)) \leq f_{m,i}(\alpha(x))$

The next step is to show that Model 1 is consistent with the static semantics. The domain of each of these abstractions is the same and both α and γ are the identity functions. Thus conditions 1-4 are trivially true. Condition 5 degenerates to

$$f_{m,0}(x) \leq f_{m,1}(x)$$

Where $f_{m,0}$ and $f_{m,1}$ are corresponding transfer functions in the static semantics and Model 1, respectively. By the definition of Model 1, each $f_{m,1}$ can be written as

$$f_{m,1}(x) = f_{m,0}(x) \cup F(x)$$

where F(x) is the union of the results of the transfer functions for the other edges leaving the same node. Condition 5 follows immediately.

So far the problem of "memory allocation", that is, the naming of pointers and structure variables, has been ignored. A naming scheme must be chosen in order to prove correctness of list operations. A simple scheme is to have two "system" variables, A_p and A_v, which are used to assign integer subscripts to new pointers and structure variables, respectively. For example, suppose the current enironment is (s_1, h_1) (in Model 1) and

$$s_1(A_p) = 3$$
$$s_1(A_v) = 7$$

and the expression

```
["a", "b", "c"]
```

is executed. This expression allocates a new pointer and 3 new structure variables, so the resulting environment, (s_2, h_2), has

$$s_2(A_p) = 4$$
$$s_2(A_v) = 10$$
$$s_2(v_7) = \text{"a"}$$
$$s_2(v_8) = \text{"b"}$$
$$s_2(v_9) = \text{"c"}$$
$$h_2(p_3) = L$$
$$L(1) = v_7$$
$$L(2) = v_8$$
$$L(3) = v_9$$

However, the type system presented here depends on being able to identify where in the program pointers are created. It is also necessary to identify the creation point of a pointer associated with a structure variable. Therefore each pointer and structure variable is also be subscripted by the program point at which the pointer is created. If the previous example occurs at point n in the flow graph then p_3 becomes $p_{n,3}$ and v_7 becomes $v_{n,7}$. This is just a renaming with respect to the simple allocation scheme in Model 1, but is important to later models.

The following conventions are maintained in the proof that Model 3 is consistent with Model 1.

$i, j, k \in$ positive integers

$n \in$ nodes

$v \in$ variables (non-structure variables unless otherwise stated)

$v_{n,i}, v_{n,j} \in$ structure variables

$r_i \in$ temporary variables

$V_n \in$ variables types

$p_{n,k} \in$ pointers

$P_n \in$ pointer types

$t \in$ types

Let Type: $2^{values} \rightarrow 2^{values}$ be the function that maps an arbitrary set of values to the smallest type containing those values. Type has the following properties (given without proof)

- it is monotone
- it is distributive over union
- $\forall x,y \in 2^{values}$, $Type(x \cap y) \subseteq Type(x) \cap Type(y)$
- $\forall x \in 2^{values}$, $x \subseteq Type(x)$
- $\forall x \in types$, $x = Type(x)$
- $\forall x \subseteq variables$, $Type(x) \subseteq variables$
- $\forall x$, s.t. $x \cap variables = \{\}$, $Type(x) \cap variables = \{\}$

The definition of

$$\alpha: 2^{envir_{[3]}} \rightarrow store_{[3]}$$

is

let $x \in 2^{envir_{[3]}}$

$\alpha(x) = \hat{s}$, where

$$\forall t,\ \hat{s}(t) = Type(\{d : v \in t \cap variables, (s, h) \in x, d = s(v)\})$$

The function $\hat{s} = \alpha(x)$ is distributive over union. Proof, let

$t_1, t_2 \in$ types

$$\begin{aligned}\hat{s}(t_1 \cup t_2) &= Type(\{d : v \in (t_1 \cup t_2) \cap variables, (s, h) \in x, d = s(v)\}) \\ &= Type(\{d : v \in t_1 \cap variables, (s, h) \in x, d = s(v)\} \cup \{d : v \in t_2 \cap variables, (s, h) \in x, d = s(v)\}) \\ &= Type(\{d : v \in t_1 \cap variables, (s, h) \in x, d = s(v)\}) \cup \\ &\quad Type(\{d : v \in t_2 \cap variables, (s, h) \in x, d = s(v)\}) \\ &= \hat{s}(t_1) \cup \hat{s}(t_2)\end{aligned}$$

Also, if $t \cap variables = \{\}$ then

$$\hat{s}(t) = Type(\{\}) = \{\}$$

All transfer functions in Model 3 transform stores in such a way as to preserve this distributivity over union and to preserve the fact that a store produces the empty set when applied to a non-variable type.

The definition of

$$\gamma: store_{[3]} \rightarrow 2^{envir_{[3]}}$$

is

let $\hat{s} \in \text{store}_{[3]}$

$\gamma(\hat{s}) = \{(s, h) \in \text{envir}_{[1]}:$

$\forall\ v,\ s(v) \in \hat{s}(v)$

$\forall\ v_{n,i},\ s(v_{n,i}) \in \hat{s}(V_n)$

$\forall\ p_{n,k},\ h(p_{n,k}) = L$ where L: positive integers $\rightarrow \{v_{n,k}$: for some k$\}$ is some paritial mapping $\}$

Proof of Condition 1

Assume

$x_1, x_2 \in s^{\text{envir}_{[1]}},\ \alpha(x_1) = \hat{s}_1,\ \alpha(x_2) = \hat{s}_2$

$x_1 \leq x_2$

by the definition of this ordering relation

$x_1 \subseteq x_2$

by the definition of alpha

$\forall\ t,\ \hat{s}_1(t) = \text{Type}(\{d : v \in t \cap \text{variables}, (s, h) \in x_1, d = s(v)\})$

$\subseteq \text{Type}(\{d : v \in t \cap \text{variables}, (s, h) \in x_2, d = s(v)\})$ $x_1 \subseteq x_2$, Type is monotone

$= \hat{s}_2(t)$ definition of α

therefore by the definition of the ordering relation over $\text{store}_{[3]}$

$\hat{s}_1 \leq \hat{s}_2$

$\alpha(x_1) \leq \alpha(x_2)$

Proof of Condition 2

Assume

$\hat{s}_1, \hat{s}_2 \in \text{store}_{[3]}$

$\hat{s}_1 \leq \hat{s}_2$

by the definition of γ

$\gamma(\hat{s}_1) = \{(s, h):$

$\forall\ v,\ s(v) \in \hat{s}_1(v)$

$\forall\ v_{n,i},\ s(v_{n,i}) \in \hat{s}_1(V_n)$

$\forall\ p_{n,k},\ h(p_{n,k}) = L$ where L: positive integers $\rightarrow \{v_{n,k}$: for some k$\}$ $\}$

by the definition of the ordering relation on $\text{store}_{[3]}$

$\forall\ v,\ \hat{s}_1(v) \leq \hat{s}_2(v)$

$\forall\ V_n,\ \hat{s}_1(V_n) \leq \hat{s}_2(V_n)$

therefore

$\gamma(\hat{s}_1) \subseteq \{(s, h):$

$\forall\ v,\ s(v) \in \hat{s}_2(v)$

$\forall\ v_{n,i},\ s(v_{n,i}) \in \hat{s}_2(V_n)$

$$\forall\, p_{n,k},\ h(p_{n,k}) = L \text{ where } L\text{: positive integers} \rightarrow \{v_{n,k} : \text{for some } k\}\ \}$$

by the definition of γ, the right-hand side is $\gamma(\hat{s}_2)$

$$\gamma(\hat{s}_1) \leq \gamma(\hat{s}_2)$$

Proof of Condition 3

Let

$$\hat{s}, \hat{s}_1 \in \text{store}_{[3]} \text{ s.t. } \hat{s}_1 = \alpha(\gamma(\hat{s}))$$

If

$$\forall\, t \in \text{basic types}, \hat{s}_1(t) = \hat{s}(t)$$

then it follows from the fact that these stores are distributive over union and induction on the number of basic types in t that

$$\forall\, t \in \text{types}, \hat{s}_1(t) = \hat{s}(t)$$

There are three classes of basic types to consider.

Case 1: t is a non-variable type

$$\hat{s}_1(t) = \{\} = \hat{s}(t)$$

Case 2: $t = \{v\}$ for some variable v

$$\begin{aligned}
\hat{s}_1(t) &= \text{Type}(\{d : (s, h) \in \gamma(\hat{s}), d = s(v)\}) && \text{by definition of } \alpha \\
&= \text{Type}(\{d : d \in \hat{s}(\{v\})\}) && \text{by definition of } \gamma \\
&= \text{Type}(\hat{s}(\{v\})) \\
&= \hat{s}(\{v\}) \\
&= \hat{s}(t)
\end{aligned}$$

Case 3: $t = V_n$

$$\begin{aligned}
\hat{s}_1(t) &= \text{Type}(\{d : v_{n,i} \in V_n, (s, h) \in \gamma(\hat{s}), d = s(v_{n,i})\}) && \text{by definition of } \alpha \\
&= \text{Type}(\{d : d \in \hat{s}(V_n)\})\}) && \text{by definition of } \gamma \\
&= \text{Type}(\hat{s}(V_n)) \\
&= \hat{s}(V_n) \\
&= \hat{s}(t)
\end{aligned}$$

Proof of Condition 4

Assume

$$x \in 2^{\text{envir}_{[3]}}$$

by the definition of α

$$\gamma(\alpha(x)) = \gamma(\hat{s}) \text{ where}$$

$$\forall\, t, \hat{s}(t) = \text{Type}(\{d : v \in t \cap \text{variables}, (s, h) \in x, d = s(v)\})$$

by the definition of γ

$$\gamma(\alpha(x)) = \gamma(\hat{s}) = \{(s_1, h_1) :$$

$$\forall\, v, s_1(v) \in \hat{s}(\{v\}) = \text{Type}(\{d : (s, h) \in x, d = s(v)\})$$

$$\forall\ v_{n,i},\ s_1(v_{n,i}) \in \hat{s}(V_n) = \mathrm{Type}(\{d : v_{n,j} \in V_n,\ (s, h) \in x,\ d = s(v_{n,j})\})$$

$$\forall\ p_{n,k},\ h(p_{n,k}) = L \text{ where } L\text{: positive integers} \rightarrow \{v_{n,k} : \text{for some } k\}\ \}$$

let

$$(s_1, h_1) \in x$$

$$\forall\ v,\ s_1(v) \in \mathrm{Type}(\{\ s_1(v)\})$$

$$\subseteq \mathrm{Type}(\{d : (s, h) \in x,\ d = s(v)\})$$

$$\forall\ v_{n,i},\ s_1(v_{n,i}) \in \mathrm{Type}(\{\ s_1(v_{n,i})\})$$

$$\subseteq \mathrm{Type}(\{d : v_{n,j} \in V_n,\ (s, h) \in x,\ d = s(v_{n,j})\})$$

therefore

$$(s_1, h_1) \in \gamma(\alpha(x))$$

$$x \subseteq \gamma(\alpha(x))$$

$$x \leq \gamma(\alpha(x))$$

Proof of Condition 5 for selected transfer functions

Consistency between Model 1 and Model 3 will be shown for assignment and list creation. A transfer function is constructed for the operation at a node based on the temporary variables associated with that node and its operands. These functions need to dereference the contents of temporary variables. Let the dereferencing function for Model 1 be

$$\mathrm{Deref}_1\text{: store}_{[1]} \times \text{temporary variables} \rightarrow \text{values}$$

$$\mathrm{Deref}_1(s, r_i) = \begin{cases} s(s(r_i)) & \text{if } s(r_i) \in \text{variables} \\ s(r_i) & \text{otherwise} \end{cases}$$

and let the dereferencing function for Model 3 be

$$\mathrm{Deref}_3\text{: store}_{[3]} \times \text{temporary variables} \rightarrow 2^{\text{values}}$$

$$\mathrm{Deref}_3(\hat{s}, r_i) = (\hat{s}(\{r_i\}) - \text{variables}) \cup \hat{s}(\hat{s}(\{r_i\}))$$

Assignment

Let some node of a flow graph be labeled with assignment. Let r_0 be the temporary variable associated with the node and r_1 and r_2 be the temporary variables associated with the left and right operands, respectively. Let assign_1 be the transfer function associated with the node in Model 1.

$$\mathrm{assign}_1(x) = \{(s_1, h) : (s, h) \in x,\ s(r_1) \in \text{variables},\ \forall\ v \in \text{variables}$$

$$s_1(v) = \begin{cases} s(v) & \text{if } v \neq s(r_1) \text{ and } v \neq r_0 \\ s(r_1) & \text{if } v = r_0 \\ \mathrm{Deref}_1(s, r_2) & \text{if } v = s(r_1) \end{cases} \}$$

(There are situations where assignment in Icon is more complicated than this, but they are not dealt with in this proof.)

Assignment in Model 3 needs to distinguish between "certain" assignment, when there is only one variable on the left hand side, and "uncertain" assignment, when there might be more than one variable. Let assign_3 be the transfer function in Model 3 using the same temporary variables as assign_1. Let

$$\mathrm{assign}_3(\hat{s}) = \hat{s}_1$$

where, for $t = \{\}$

$$\hat{s}_1(t) = \{\}$$

for $t \in$ basic types, let $t_1 = \hat{s}(\{r_1\}) \cap$ variables

$$\hat{s}_1(t) = \begin{cases} \hat{s}(t) & \text{if } t \neq \{r_0\} \text{ and } t \not\subseteq t_1 \\ t_1 & \text{if } t = \{r_0\} \\ \text{Deref}_3(\hat{s}, r_2) & \text{if } t = t_1 = \{v\}, \text{ for some variable } v \neq r_0 \\ \text{Deref}_3(\hat{s}, r_2) \cup \hat{s}(t) & \text{if } t \neq \{r_0\}, t \subseteq t_1, \text{ and } \forall\, v, t_1 \neq \{v\} \end{cases}$$

for $t_1 = t_2 \cup t_3$

$$\hat{s}_1(t_1) = \hat{s}_1(t_2) \cup \hat{s}_1(t_3)$$

Condition 5 for assignment is

$$\forall\, x \in 2^{\text{envir}_{[3]}},\ \alpha(\text{assign}_1(x)) \leq \text{assign}_3(\alpha(x))$$

let

$$\alpha(\text{assign}_1(x)) = \hat{s}_1$$

$$\text{assign}_3(\alpha(x)) = \hat{s}_2$$

by the definition of the ordering on $\text{store}_{[3]}$, this requires proving

$$\forall\, t,\ \hat{s}_1(t) \subseteq \hat{s}_2(t)$$

Only proofs for the interesting cases of $t \in$ variable basic types are presented here. By the definition of α and assign_1

$$\hat{s}_1(t) = \text{Type}(\{d : v \in t, (s_1, h_1) \in \text{assign}_1(x), d = s_1(v)\})$$

$$= \text{Type}(\{d : v \in t, (s, h) \in x, s(r_1) \in \text{variables}, d = \begin{cases} s(v) & \text{if } v \neq s(r_1) \text{ and } v \neq r_0 \\ s(r_1) & \text{if } v = r_0 \\ \text{Deref}_1(s, r_2) & \text{if } v = s(r_1) \end{cases}\})$$

Let

$$\alpha(x) = \hat{s}$$

$$t_1 = \hat{s}(\{r_1\}) \cap \text{variables}$$

$$= \text{Type}(\{d : (s, h) \in x, d = s(r_1)\}) \cap \text{variables}$$

Then by the definition of assign_3, for $t \in$ variable basic types

$$\hat{s}_2(t) = \begin{cases} \hat{s}(t) & \text{if } t \neq \{r_0\} \text{ and } t \not\subseteq t_1 \\ t_1 & \text{if } t = \{r_0\} \\ \text{Deref}_3(\hat{s}, r_2) & \text{if } t = t_1 = \{v\}, v \neq r_0 \\ \text{Deref}_3(\hat{s}, r_2) \cup \hat{s}(t) & \text{if } t \neq \{r_0\}, t \subseteq t_1, \text{ and } \forall\, v, t_1 \neq \{v\} \end{cases}$$

The proof is driven by the cases in the preceding formula.

Case 1: $t \neq \{r_0\}$ and $t \not\subseteq t_1$

suppose $\exists\, (s, h) \in x$ s.t. $s(r_1) \in t$. t is a variable basic type so

$$t = \text{Type}(\{s(r_1)\})$$

$$\subseteq \text{Type}(\{d : (s, h) \in x, d = s(r_1)\}) \cap \text{variables}$$

$$= t_1$$

but this contradicts the assumption $t \not\subseteq t_1$, thus

$$\forall (s, h) \in x,\ s(r_1) \notin t$$

$$\hat{s}_1(t) = \text{Type}(\{d : v \in t, (s, h) \in x, s(r_1) \in \text{variables}, d = s(v)\})$$

$$\subseteq \text{Type}(\{d : v \in t, (s, h) \in x, d = s(v)\})$$

$$\hat{s}_2(t) = \hat{s}(t)$$

$$= \text{Type}(\{d : v \in t, (s, h) \in x, d = s(v)\})$$

$$\hat{s}_1(t) \subseteq \hat{s}_2(t)$$

Case 2: $t = \{r_0\}$

$$\hat{s}_1(t) = \text{Type}(\{d : (s, h) \in x, s(r_1) \in \text{variables}, d = s(r_1)\})$$

$$= \text{Type}(\{d : (s, h) \in x, d = s(r_1)\} \cap \text{variables})$$

$$\subseteq \text{Type}(\{d : (s, h) \in x, d = s(r_1)\}) \cap \text{Type}(\text{variables}))$$

$$= \text{Type}(\{d : (s, h) \in x, d = s(r_1)\}) \cap \text{variables}$$

$$\hat{s}_2(t) = t_1$$

$$= \text{Type}(\{d : (s, h) \in x, d = s(r_1)\}) \cap \text{variables}$$

$$\hat{s}_1(t) \subseteq \hat{s}_2(t)$$

Case 3: $t = t_1 = \{v\}, v \neq r_0$

suppose $\exists (s, h) \in x$, s.t. $s(r_1) \in$ variables and $s(r_1) \neq v$

$$\{s(r_1)\} \subseteq \text{Type}(\{s(r_1)\})$$

$$\subseteq \text{Type}(\{d : (s_1, h_1) \in x, d = s_1(r_1)\}) \cap \text{variables}$$

$$= t_1$$

$$= \{v\}$$

but this implies $s(r_1) = v$, which contradicts the assumption, so

$$\forall (s, h) \in x \text{ s.t. } s(r_1) \in \text{variables}, s(r_1) = v$$

$$\hat{s}_1(t) = \text{Type}(\{d : (s, h) \in x, d = \text{Deref}_1(s, r_2)\}$$

$$= \text{Type}(\{d : (s, h) \in x, d = \begin{cases} s(s(r_2)) & \text{if } s(r_2) \in \text{variables} \\ s(r_2) & \text{otherwise} \end{cases}\})$$

$$= \text{Type}(\{d : (s, h) \in x, s(r_2) \in \text{variables}, d = s(s(r_2))\}) \cup \text{Type}(\{d : (s, h) \in x, d = s(r_2) \notin \text{variables}\})$$

$$\hat{s}_2(t) = \text{Deref}_3(\hat{s}, r_2)$$

$$= (\hat{s}(\{r_2\}) - \text{variables}) \cup \hat{s}(\hat{s}(\{r_2\}))$$

$$\hat{s}(\{r_2\}) - \text{variables} = \text{Type}(\{d : (s, h) \in x, d = s(r_2)\}) - \text{variables}$$

$$= (\text{Type}(\{d : (s, h) \in x, d = s(r_2) \in \text{variables}\}) \cup$$

$$\text{Type}(\{d : (s, h) \in x, d = s(r_2) \notin \text{variables}\})) - \text{variables}$$

$$= \text{Type}(\{d : (s, h) \in x, d = s(r_2) \notin \text{variables}\})$$

$$\hat{s}(\hat{s}(\{r_2\})) = \text{Type}(\{d : v \in \text{Type}(\{d_1 : (s_1, h_1) \in x, d_1 = s_1(r_2)\}) \cap \text{variables}, (s, h) \in x, d = s(v)\}$$

$$\supseteq \text{Type}(\{d : v \in \{d_1 : (s_1, h_1) \in x, d_1 = s_1(r_2)\}) \cap \text{variables}, (s, h) \in x, d = s(v)\}$$

$$= \text{Type}(\{d : (s, h) \in x, (s_1, h_1) \in x, s_1(r_2) \in \text{variables}, d = s(s_1(r_2))\})$$

$$\supseteq \text{Type}(\{d : (s, h) \in x, s(r_2) \in \text{variables}, d = s(s(r_2))\})$$

$$\hat{s}_1(t) \subseteq \hat{s}_2(t)$$

Case 4: $t \neq \{r_0\}$, $t \subseteq t_1$, and $\forall v, t_1 \neq \{v\}$

$$\hat{s}_1(t) = \text{Type}(\{d: v \in t, (s, h) \in x, s(r_1) \in \text{variables}, d = \begin{cases} s(v) & \text{if } v \neq s(r_1) \\ \text{Deref}_1(s, r_2)) & \text{if } v = s(r_1) \end{cases}\})$$

$$= \text{Type}(\{d: v \in t, (s, h) \in x, s(r_1) \in \text{variables}, d = \begin{cases} s(v) & \text{if } v \neq s(r_1) \\ s(s(r_2)) & \text{if } v = s(r_1) \text{ and } s(r_2) \in \text{variables} \\ s(r_2) & \text{if } v = s(r_1) \text{ and } s(r_2) \notin \text{variables} \end{cases}\})$$

$$= \text{Type}(\{d : v \in t, (s, h) \in x, s(r_1) \in \text{variables}, v \neq s(r_1), d = s(v)\}) \cup$$
$$\text{Type}(\{d : v \in t, (s, h) \in x, s(r_1) \in \text{variables}, v = s(r_1), s(r_2) \in \text{variables}, d = s(s(r_2))\}) \cup$$
$$\text{Type}(\{d : v \in t, (s, h) \in x, s(r_1) \in \text{variables}, v = s(r_1), s(r_2) \notin \text{variables}, d = s(r_2)\})$$

$$\subseteq \text{Type}(\{d : v \in t, (s, h) \in x, s(r_1) \in \text{variables}, d = s(v)\}) \cup$$
$$\text{Type}(\{d : (s, h) \in x, s(r_2) \in \text{variables}, d = s(s(r_2))\}) \cup$$
$$\text{Type}(\{d : (s, h) \in x, s(r_2) \notin \text{variables}, d = s(r_2)\})$$

Using intermediate results from Case 3

$$\hat{s}_2(t) = \text{Deref}_3(\hat{s}, r_2) \cup \hat{s}(t)$$

$$\supseteq \text{Type}(\{d : (s, h) \in x, d = s(r_2) \notin \text{variables}\}) \cup$$
$$\text{Type}(\{d : (s, h) \in x, s(r_2) \in \text{variables}, d = s(s(r_2))\}) \cup$$
$$\text{Type}(\{d : v \in t, (s, h) \in x, d = s(v)\})$$

$$\hat{s}_1(t) \subseteq \hat{s}_2(t)$$

List Creation

Let node n of a flow graph be labeled with list creation. Let r_0 be the temporary variable associated with the node and r_1 through r_m be the temporary variables associated with the m operands, whose values will be put in the list. Let list_1 be the transfer function associated with the node in Model 1.

$$\text{list}_1(x) = \{(s_1, h_1) : (s, h) \in x, s_1(v) = \begin{cases} s(v) & \text{if } v \notin \{r_0, A_p, A_v\} \text{ and } \forall i, 1 \le i \le m,\ v \neq v_{n,s(A_v)+i-1} \\ p_{n,s(A_p)} & \text{if } v = r_0 \\ \text{Deref}_1(s, r_i) & \text{if } 1 \le i \le m \text{ and } v = v_{n,s(A_v)+i-1} \\ s(A_p) + 1 & \text{if } v = A_p \\ s(A_v) + m & \text{if } v = A_v \end{cases}\}$$

$$h_1(p_{j,k}) = \begin{cases} h(p_{j,k}) & \text{if } j \neq n \text{ or } k \neq s(A_p) \\ L & \text{if } j = n \text{ and } k = s(A_p), \text{ where } \forall i, 1 \le i \le m, L(i) = v_{n,s(A_v)+i-1} \end{cases}\}$$

Let list_3 be the transfer function in Model 3 using the same temporary variables as list_1. Then

$$\text{list}_3(\hat{s}) = \hat{s}_1$$

where, for $t = \{\}$

$$\hat{s}_1(t) = \{\}$$

for $t \in$ basic types

$$\hat{s}_1(t) = \begin{cases} \hat{s}(t) & \text{if } t \neq \{r_0\} \text{ and } t \neq V_n \\ P_n & \text{if } t = \{r_0\} \\ \hat{s}(V_n) \cup \bigcup_{i=1}^{m} Deref_3(\hat{s}, r_i) & \text{if } t = V_n \end{cases}$$

for $t_1 = t_2 \cup t_3$

$$\hat{s}_1(t_1) = \hat{s}_1(t_2) \cup \hat{s}_1(t_3)$$

Note that in Model 3, $\hat{s}(\{A_p\})$ = integer and $\hat{s}(\{A_p\})$ = integer for all $\hat{s}$. These variables play no part in Model 3, but are retained to keep α and γ simple.

Let

$$\alpha(list_1(x)) = \hat{s}_1$$

$$list_3(\alpha(x)) = \hat{s}_2$$

then condition 5 for list creation requires proving

$$\forall\, t,\ \hat{s}_1(t) \subseteq \hat{s}_2(t)$$

As with assignment, only proofs for the interesting cases of $t \in$ variable basic types are presented here. In addition, it is assumed that $t \neq \{A_p\}$ and $t \neq \{A_v\}$ as these cases are trivial. By the definition of α and $list_1$

$$\hat{s}_1(t) = Type(\{d : v \in t, (s_1, h_1) \in list_1(x), d = s_1(v)\})$$

$$= Type(\{d : v \in t, (s, h) \in x, d = \begin{cases} s(v) & \text{if } v \neq r_0 \text{ and } \forall\, i,\ 1 \leq i \leq m,\ v \neq v_{n,s(A_v)+i-1} \\ p_{n,s(A_p)} & \text{if } v = r_0 \\ Deref_1(s, r_i) & \text{if } 1 \leq i \leq m \text{ and } v = v_{n,s(A_v)+i-1} \end{cases}\})$$

Let

$$\alpha(x) = \hat{s}$$

Then by the definition of $list_3$, for $t \in$ variable basic types

$$\hat{s}_2(t) = \begin{cases} \hat{s}(t) & \text{if } t \neq \{r_0\} \text{ and } t \neq V_n \\ P_n & \text{if } t = \{r_0\} \\ \hat{s}(V_n) \cup \bigcup_{i=1}^{m} Deref_3(\hat{s}, r_i) & \text{if } t = V_n \end{cases}$$

The proof is driven by the cases in the preceding formula.

Case 1: $t \neq \{r_0\}$ and $t \neq V_n$

$$\hat{s}_1(t) = Type(\{d : v \in t, (s, h) \in x, d = s(v)\})$$

$$\hat{s}_2(t) = \hat{s}(t)$$

$$= Type(\{d : v \in t, (s, h) \in x, d = s(v)\})$$

$$\hat{s}_1(t) = \hat{s}_2(t)$$

Case 2: $t = \{r_0\}$

$$\hat{s}_1(t) = Type(\{d : (s, h) \in x, d = p_{n,s(A_p)}\})$$

$\subseteq \text{Type}(\{d : d \in P_n\})$

$= \text{Type}(P_n)$

$= P_n$

$\hat{s}_2(t) = P_n$

$\hat{s}_1(t) \subseteq \hat{s}_2(t)$

Case 3: $t = V_n$

$$\hat{s}_1(t) = \text{Type}(\{d : (s, h) \in x, v_{n,s(A_n)+i-1} \in V_n, d = \begin{cases} s(v_{n,s(A_n)+i-1}) & \text{if } i < 1 \text{ or } i > m \\ \text{Deref}_1(s, r_i) & \text{if } 1 \le i \le m \end{cases}\})$$

$= \text{Type}(\{d : (s, h) \in x, v_{n,s(A_n)+i-1} \in V_n, i < 1 \text{ or } i > m, d = s(v_{n,s(A_n)+i-1})\}) \cup$

$\text{Type}(\{d : (s, h) \in x, 1 \le i \le m, d = \text{Deref}_1(s, r_i)\})$

$\subseteq \text{Type}(\{d : (s, h) \in x, v \in V_n, d = s(v)\}) \cup \bigcup_{i=1}^{m} \text{Type}(\{d : (s, h) \in x, d = \text{Deref}_1(s, r_i)\})$

$\hat{s}_2(t) = \hat{s}(V_n) \cup \bigcup_{i=1}^{m} \text{Deref}_3(\hat{s}, r_i)$

$= \text{Type}(\{d : (s, h) \in x, v \in V_n, d = s(v)\}) \cup \bigcup_{i=1}^{m} \text{Deref}_3(\hat{s}, r_i)$

Using the same arguments used in Case 3 for assignment

$\text{Type}(\{d : (s, h) \in x, d = \text{Deref}_1(s, r_i)\}) \subseteq \text{Deref}_3(\hat{s}, r_i)$

$\hat{s}_1(t) \subseteq \hat{s}_2(t)$