The Implementation of an Optimizing Compiler for Icon*

Kenneth Walker

TR 91-16

August 9, 1991

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

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

This technical report has been submitted as a dissertation to the faculty of the Department of Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Graduate College of the University of Arizona.

Table of Contents

Abstract

There are many optimizations that can be applied while translating Icon programs. These optimizations and the analyses needed to apply them are of interest for two reasons. First, Icon's unique combination of characteristics requires developing new techniques for implementing them. Second, these optimizations are useful in variety of languages and Icon can be used as a medium for extending the state of the art.

Many of these optimizations require detailed control of the generated code. Previous production implementations of the Icon programming language have been interpreters. The virtual machine code of an interpreter is seldom flexible enough to accommodate these optimizations and modifying the virtual machine to add the flexibility destroys the simplicity that justified using an interpreter in the first place. These optimizations can only reasonably be implemented in a compiler. In order to explore these optimizations for Icon programs, a compiler was developed. This dissertation describes the compiler and the optimizations it employs. It also describes a run-time system designed to support the analyses and optimizations.

Icon variables are untyped. The compiler contains a type inferencing system that determines what values variables and expression may take on during program execution. This system is effective in the presence of values with pointer semantics and of assignments to components of data structures.

The compiler stores intermediate results in temporary variables rather than on a stack. A simple and efficient algorithm was developed for determining the lifetimes of intermediate results in the presence of goal-directed evaluation. This allows an efficient allocation of temporary variables to intermediate results.

The compiler uses information from type inferencing and liveness analysis to simplify generated code. Performance measurements on a variety of Icon programs show these optimizations to be effective.

CHAPTER 1

Introduction

Motivation

This dissertation describes the implementation of an optimizing compiler for the Icon programming language [1,2]. This is a practical and complete compiler for a unique and complex programming language. This dissertation describes the theory behind several parts of the compiler and describes the implementation of all interesting aspects of the compiler.

The motivation for developing a compiler for the Icon programming language is to have a vehicle for exploring optimization techniques. Some performance improvements can be obtained by modifying the run-time system for the language, for example by implementing alternative data structures or storage management techniques. These improvements may apply to a broad class of programs and the techniques can reasonably be implemented in an interpreter system. However, other techniques, such as eliminating unnecessary type checking, apply to expressions within specific programs. The Icon interpreter [3] is based on a virtual machine with a relatively small instruction set of powerful operations. A small instruction set is easier to implement and maintain than a large one, and the power of many of the individual operations insures that the overhead of the decoding loop is not excessive. The disadvantage of this instruction set is that an Icon translator that generates code for the interpreter does not have enough flexibility to do many of the possible program-specific optimizations. It is possible to devise a set of more primitive virtual machine instructions that expose more opportunities for these optimizations. Increasingly primitive instruction sets provide increasingly more opportunities for optimizations. In the extreme, the instruction set for a computer (hardware interpreter) can be used and the translator becomes a compiler. A compiler was chosen for this research because it is a good vehicle for exploring program-specific optimizations and eliminates the overhead of a software

interpreter which might otherwise become excessive.

Type Inferencing

Most Icon operations require operands with specific types. The types of the actual operands in an expression must be checked and possibly converted to the required types. However, Icon variables are untyped; in general, this checking cannot done at translation time. The Icon interpreter takes the simple approach to the problem and performs all of the type checking for an expression every time it is executed. For most programs, a *type inferencing system* can provide the information needed to do much of the checking at translation time, eliminating the need for these checks at run time. A type inferencing system determines the types that elements of a program (variables, expression, procedures, etc) can take on at run time. The Icon compiler contains an effective and practical type inferencing system, and implements code generation optimizations that make use of the information produced by the type inferencing system.

Two basic approaches have been taken when developing type inferencing schemes. Schemes based on unification [4-6] construct type signatures for procedures; schemes based on global data flow analysis [7-10] propagate throughout a program the types variables may take on. 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.

Icon has several types of non-applicative data structures with pointer semantics. They all can be heterogeneous and can be combined to form arbitrary graphs. An effective type inferencing system must handle these data structures without losing too much information through crude assumptions. These composite data structures typically consist of a few basic elements used repeatedly and they logically have a recursive structure. A number of type inferencing systems handle recursion in applicative data structures [8, 11, 12]; 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 [10, 13].

Work by Chase, Wegman, and Zadeck [14] published subsequent to the original technical report on the Icon type inferencing system [15] presents a technique similar to the one used in this type inferencing system. They use a minimal language model to describe the use of the technique for pointer analysis. They speculate that the technique might be too slow for practical use and propose methods of improving the technique in the context of pointer analysis. Use of the prototype Icon type inferencing system described in the original technical report indicates that memory usage is more of a problem than execution time. This problem is addressed in the implementation of type inferencing in the Icon compiler.

Liveness Analysis

Type checking optimizations can be viewed as forms of argument handling optimizations. Other argument handling optimizations are possible. For example, when it is safe to do so, it is more efficient to pass a variable argument by reference than to copy it to a separate location and pass a reference to that location (this particular opportunity for optimization arises because of implementation techniques borrowed from the Icon interpreter — Icon values are larger than pointers and Icon parameter passing is built on top of C parameter passing). Such optimizations are not possible in a stack-based execution model; a temporary-variable model is needed and such a model is used by the Icon compiler. Icon's goal-directed evaluation can extend the lifetime of the intermediate values stored in temporary variables. Icon presents a unique problem in *liveness analysis*, which is the static determination of the lifetime of values in a program [10, 16]. While this problem, like other liveness problems, can be solved with traditional techniques, it has enough structure that it can be solved without precomputing a flow graph or using expensive forms of data flow analysis.

The only previous implementation of Icon using a temporary-variable model is a partial implementation by Christopher [17]. Christopher uses the fact that Icon programs contain many instances of bounded goal-directed evaluation to deduce limits for the lifetimes of intermediate values. However, this approach produces a very crude estimate for these lifetimes. While

overestimating the lifetime of intermediate values results in a safe allocation of temporary variables to these values, a fine-grained liveness analysis results in the use of fewer temporary variables. The Icon compiler addresses this problem of fine-grained liveness analysis in the presence of goal-directed evaluation and addresses the problem of applying the information to temporary variable allocation.

Analyzing Goal-Directed Evaluation

Many kinds of analyses of Icon programs must deal with Icon's goal-directed evaluation and its unique control structures. These analyses include type inferencing, liveness analysis, and the control flow analyses in O'Bagy's prototype compiler [18]. Determining possible execution paths through an Icon program is more complicated than it is for programs written in more conventional languages. The implementation of the type inferencing system and liveness analysis here explore variations on the techniques presented by O'Bagy.

Dissertation Organization

This dissertation is logically divided into three parts. Chapters 2 through 4 present the main ideas upon which the compiler is based, Chapters 5 through 10 describe the implementation of these ideas, and Chapter 11 presents performance measurements of compiled code.

Chapter 2 describes the code generated by the compiler. It explains how Icon data values, variables, and goal-directed evaluation are implemented, independent of the actual translation process. Chapter 3 presents a theoretical model of the type inferencing system used in the compiler. The model includes the important ideas of the type inferencing system, while ignoring some purely pragmatic details. Chapter 4 explains the liveness analysis problem and presents the solution used in the compiler.

The Icon compiler is designed to be a production-quality system. The compiler system consists of the compiler itself and a run-time system. The fact that these two components are not entirely independent must be carefully considered in the design of such a production-quality

system. Chapter 5 describes the system as a whole and how the interactions between the components are handled.

Chapter 6 presents the organization of the compiler itself. This chapter describes some parts of the compiler in detail, but defers major topics to other chapters. Chapter 7 builds on the model presented in Chapter 3 and describes the full type inferencing system used in the compiler and its implementation. Chapter 8 describes the translation techniques used to produce code from expressions that employ Icon's goal-directed evaluation scheme and its unique control structures. It also describes the allocation of temporary variables using the information produced by liveness analysis.

The code generator does no look-ahead and as a result it often produces code that is poor when taken in context of subsequent code. This problem is shared with most code generators as are some of the solutions used in this compiler. The unique code generation techniques required by Icon's goal-directed evaluation produce unusual variations of this problem and require some innovative solutions in addition to the standard ones. Chapter 9 describes the various techniques employed to handle this problem. Chapter 10 describes the optimizations that can be done using the results of type inferencing. These optimizations also make use of liveness information.

Chapter 11 demonstrates the effects of the various optimizations used in the compiler on the performance of specific kinds of expressions. It also presents measurements of the performance of compiled code for a variety of complete programs, comparing the performance to that of the Icon interpreter. In addition, the sizes of the executable code for the complete programs are presented.

The conclusions, Chapter 12, summarize what has been done and lists some work that remains to be explored.

CHAPTER 2

The Translation Model

A compiler translates programs written in a particular programming language into code that can be executed by the hardware of a computer, perhaps with the aid of an operating system. The design of a compiler involves deciding how features of the source language are represented on the target machine. These features include data types, variables, intermediate values, operations, and control structures.

Modern compilers seldom produce machine code directly. They translate a program into a form closer to machine code than the source language and depend on other tools to finish the translation. If the compiler produces an object module, it depends on a linker and a loader to produce executable code. If the compiler produces assembly language, it also depends on an assembler. A recent trend among compilers produced in research environments has been to produce C code [19, 20], adding a C compiler to the list of tools required to finish the translation to machine code [21-27]. The Icon compiler takes this approach and generates C code.

There are several advantages to compiling a language into C. Low-level problems such as register allocation and the selection and optimization of machine instructions are handled by the C compiler. As long as these problems are outside the scope of the research addressed by the compiler, it is both reasonable and effective to allow another compiler to deal with them. In general, it is easier to generate code in a higher-level language, just as it is easier to program in a higher-level language. As long as the target language lies on a "nearly direct path" from the source language to machine code, this works well. C is closely matched to most modern machine architectures, so few tangential translations must be done in generating C code from Icon.

Another advantage of generating C code is that it greatly increases the portability of the compiler and facilitates cross-compilation. The popularity of C in recent years has resulted in production-quality C compilers for most systems. While the implementation of Icon in C

contains some machine and system dependencies, C's conditional compilation, macro, and file inclusion facilities make these dependencies relatively easy to deal with when they arise. These facts make possible the development of a highly portable Icon compiler, allowing the compiler's effectiveness to be tested by Icon's large user community.

Data Representation

Because the target language is C, Icon data must be represented as C data. The careful representation of data and variables is important to the performance of an implementation of a high-level language such as Icon. In addition, information provided by type inferencing can be used to optimize these representations. However, such considerations are largely outside the scope of this current research. For this reason, the representations used in code produced by this compiler and the compiler's run-time system are largely unchanged from those of the Icon interpreter system [3]. The interpreter's run-time system is written in C. Therefore borrowing its data representations for the compiler system is simple. This choice of representation means that the run-time system for the compiler could be adapted directly from the run-time system for the interpreter, and it allowed the compiler development to concentrate on parts of the system addressed by this research. In addition, this choice of representation allows a meaningful comparison of the performance of compiled code to the performance of interpreted code.

An Icon value is represented by a two-word descriptor [3]. The first word, the *d-word*, contains type information. In the case of a string value, the type is indicated by zero in a high-order bit in the d-word, and the length of a string is stored in low-order bits of the d-word. All other types have a one in that bit and further type information elsewhere in the d-word. The *v-word* of a descriptor indicates the value. The v-word of the null value is zero, the v-word of an Icon integer is the corresponding C integer value, and v-words of other types are pointers to data. A descriptor is implemented with the following C structure:

```
struct descrip {
   word dword;          /*  type field */
   union {
      word integr;      /*  integer value */
      char *sptr;       /*  pointer to character string */
      union block *bptr;/*  pointer to a block */
      dptr descptr;     /*  pointer to a descriptor */
      } vword;
   };
```

word is defined to be a C integer type (one that is at least 32-bits long), block is a union of structures implementing various data types, and dptr is a pointer to a descrip structure.

Intermediate Results

While the representation of data in the compiler is the same as in the interpreter, the method of storing the intermediate results of expression evaluation is not. Two basic approaches have been used in language implementations to store intermediate results. A stack-based approach is simple and dynamic. It requires no pre-analysis of expressions to allocate storage for the intermediate results, but the simple rigid protocol allows little room for optimization. For Icon there is an additional problem with a stack-based approach. Goal-directed evaluation extends the lifetime of some intermediate results, requiring that the top elements of the evaluation stack be copied at critical points in execution [3, 18]. In spite of the need for this extra copying, most previous implementations of Icon have been implemented with an evaluation stack.

An alternative to using a stack is to pre-allocate a temporary variable for each intermediate result. In this model, operations take explicit locations as arguments. Therefore an operation can directly access program variables as arguments; there is no need to perform the extra operations of pushing addresses or values on a stack. In addition, the lifetime of a temporary variable is not

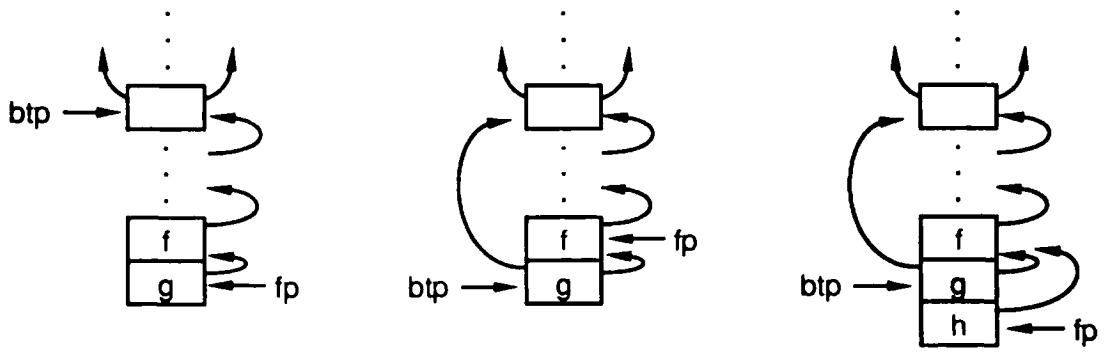
determined by a rigid protocol. The compiler can assign an intermediate result to a temporary variable over an arbitrary portion of the program, eliminating the copying needed to preserve a value beyond the lifetime imposed by a stack-based approach. This compiler uses the temporary-variable model because it allows more opportunities to optimize parameter handling, a major goal of this research.

Icon's automatic storage management dictates the use of a garbage collector in the run-time system. When this garbage collector is invoked, it must be able to locate all values that may be used later in the program. In the current interpreter system, intermediate values and local variables are stored on the same stack. The garbage collector sweeps this stack to locate values. In the compiler, a different approach is taken to insure that all necessary values are locatable. Arrays of descriptors are allocated contiguously along with a count of the number of descriptors in the array. The arrays are chained together. An array of descriptors may be local to a C function, or it may be allocated with the malloc library function. The garbage collector locates values by following the chain and scanning the descriptors in each array. These descriptors are referred to as *tended* descriptors.

Executable Code

Even more important than where intermediate results are stored is how they are computed. Some aspects of Icon expression evaluation are similar to those of many other languages, but others aspects are not. Goal-directed evaluation with backtracking poses a particular challenge when implementing Icon expression evaluation. The Icon interpreter is based on a virtual machine that includes backtracking, as are Prolog interpreters based on the Warren Abstract Machine [28]. While details differ between the Icon and Prolog virtual machines, their implementation of control backtracking is based on the same abstract data structures and state variables. Such a virtual machine contains a stack of procedure frames, but the stack is maintained differently from that of a virtual machine that does not implement goal-directed evaluation.

The difference manifests itself when a procedure produces a result, but has alternate results that it can produce in the event of backtracking. When this occurs, the frame for the procedure remains on the stack after control returns to the caller of the procedure. This frame contains the information needed to produce the alternate results. The left stack in the following diagram shows that procedure f has called procedure g. The arrows on the left of the stack represent the *backtracking chain* of procedures that can produce alternate results. btp points to the head of the backtracking chain which currently starts further down in the stack. The arrows on the right represent the call chain of procedures. fp points to the frame of the currently executing procedure.



Suppose g produces the first of several possible results. Execution returns to f and g's frame is added to the backtracking chain. This is represented by the middle stack in the diagram. If f then calls h, its procedure frame is added to the top of the stack as shown in the right stack in the diagram.

If h produces a result and is not capable of producing more, execution returns to f and the stack again looks like the one in the middle of the diagram (the program pointer within f is different, of course). If h produces a result and is capable of producing more, execution returns to f, but h's frame remains on the stack and is added to the head backtracking chain, similar to what was done when g produced a result. If h produces no results, backtracking occurs. h's frame is removed from the stack, execution returns to the procedure g who's frame is at the head of the backtracking chain, and g's frame is removed from the head of the chain. The stack once again

looks like left stack in the diagram and g proceeds to produce another result.

Traditional languages such as Pascal or C present high-level virtual machines that contain no notion of backtracking and have no need to perform low-level stack manipulations. Icon expressions with goal-directed evaluation cannot be translated directly into such languages. This is the fundamental problem that must be addressed when designing a compiler for Icon. O'Bagy presents an elegant solution to this problem in her dissertation [18]. Her solution is used by this optimizing compiler as a basis for translating Icon expressions into C code. The rest of this section contains a brief explanation of the variation of her approach that is used in the compiler, while exploring useful ways of viewing the problem. O'Bagy's dissertation describes how control structures not covered in this discussion can be implemented using her model.

Formal semantics is one tool that can be used in understanding a language [29, 30]. The added complexity caused by Icon's goal-directed evaluation is reflected in Gudeman's description of Icon using denotational semantics [31]. While conventional programming languages can be described using one continuation for each expression, Icon requires two continuations. One continuation for an expression embodies the rest of the program if the expression succeeds, while the other embodies the rest of the program if the expression fails.

The Icon compiler uses the notion of success continuations to implement goal-directed evaluation. However, these continuations violate some of the properties traditionally associated with continuations. A continuation in denotational semantics and in the language Scheme [32, 33] is a function that never returns. However, the success continuations produced by the compiler implement backtracking by returning. In addition, these continuations implement the rest of the current bounded expression rather than the rest of the entire program. Note that unlike continuations in Scheme, these continuations are created at compile time, not at run time. Some Prolog compilers have been based on a similar continuation-passing technique [22, 34].

The C language is oriented toward an imperative style of programming. In order to produce efficient code, the Icon compiler should not generate an excessive number of function calls. Specifically, it should avoid creating continuations for every expression. A more operational

view of Icon's semantics and of C's semantics can be useful in understanding how to accomplish this. An operation in Icon can succeed or fail. In the view of denotational semantics, the question of what will be done in each case must be answered, with the answers taking the form of functions. In an operational view, the questions can take the form of where to go in each case. The answers to these questions can be any type of transfer of control supported by the C language: execute the next sequential instruction, execute a function, return from a function, or go to a label.

Most operations in Icon are *monogenic*. That is, they produce exactly one result, like operations in conventional languages. For these operations, the compiler can generate code whose execution simply falls through into the code that implements the subsequent operation.

Conditional operations are more interesting. These operations either produce a single value or fail. If such an operation succeeds, execution can fall through into code implementing the subsequent operation. However, if the operation fails, execution must transfer elsewhere in the program. This is accomplished by branching to a *failure label*. If the code for the operation is put in-line, this is straightforward. However, if the operation (either a built-in operation or an Icon procedure) is implemented by a separate C function, the function must notify the caller whether it succeeded or failed and the caller must effect the appropriate transfer of control.

By convention, C functions produced by the compiler and those implementing the run-time routines each return a signal (this convention is violated in some special cases). A signal is an integer (and is unrelated to Unix signals). If one of these C functions needs to return an Icon value, it does so through a pointer to a result location that is passed to it as an argument. Two standard signals are represented by the manifest constants A_Continue and A_Resume.

A return (either an Icon return expression or the equivalent construct in a built-in operation) is implemented with code similar to

```
*result = operation result;
return A_Continue;
```

Failure is implemented with the code

```
return A_Resume;
```

The code implementing the call of an operation consists of both a C call and signal-handling code.

```
switch (operation(args, &result)) {
   case A_Continue: break;
   case A_Resume: goto failure label;
   }
```

This code clearly can be simplified. This form is general enough to handle the more complex signal handling that can arise during code generation. Simplifying signal handling code is described in Chapter 9.

Generators pose the real challenge in implementing Icon. A generator includes code that must be executed if subsequent failure occurs. In addition, a generator, in general, needs to retain state information between suspending and being resumed. As mentioned above, this is accomplished by calling a success continuation. The success continuation contains subsequent operations. If an operation in the continuation fails, an A_Resume signal is returned to the generator, which then executes the appropriate code. The generator retains state information in local variables. If the generator is implemented as a C function, a pointer to the continuation is passed to it. Therefore, a function implementing a generative operation need not know its success continuation until run time.

Consider the operation *i* to *j*. This operation can be implemented in Icon with a procedure like

```
procedure To(i, j)
   while i <= j do {
      suspend i
      i +:= 1
      }
   fail
end
```

It can be implemented by an analogous C function similar to the following (for simplicity, C ints are used here instead of Icon values).

```
int to(i, j, result, succ_cont)
int i, j;
int *result;
int (*succ_cont)();
   {
   int signal;

   while (i <= j) {
      *result = i;
      signal = (*succ_cont)();
      if (signal != A_Resume)
         return signal;
      ++i;
      }
   return A_Resume;
   }
```

There is no explicit failure label in this code, but it is possible to view the code as if an implicit

failure label occurs before the ++i.

The Icon expression

```
every write(1 to 3)
```

can be compiled into the following code (for simplicity, the write function has been translated into printf and scoping issues for result have been ignored). Note that the every simply introduces failure.

```
switch (to(1, 3, &result, sc)) {  /* standard signal-handling code */
   ...
   }

int sc()
   {
   printf("%d\n", result);
   return A_Resume;
   }
```

The final aspect of Icon expression evaluation that must be dealt with is that of bounded expressions. Once execution leaves a bounded expression, that expression cannot be resumed. At this point, the state of the computation with respect to backtracking looks as it did when execution entered the bounded expression. This means that, in generated code, where to go on failure (either by branching to an explicit failure label or by returning an A_Resume signal) must be the same. However, this failure action is only correct in the C function containing the start of the code for the bounded expression. If a function suspended by calling a success continuation, execution is no longer in that original C function. To accommodate this restoration of failure action, execution must return to that original function.

This is accomplished by setting up a *bounding label* in the original C function and allocating a signal that corresponds to the label. When the end of the bounded expression is reached, the signal for the bounding label is returned. When the signal reaches the function containing the label, it is converted into a goto. It can be determined statically which calls must convert which signals. Note that if the bounded expression ends in the original C function, the "return signal" is already in the context of the label. In this case, it is immediately transformed into a goto by the compiler, and there is no real signal handling.

Consider the Icon expression

```
move(1);
...
```

The move function suspends and the C function implementing it needs a success continuation. In this case, move is called in a bounded context, so the success continuation must return execution to the function that called move. The continuation makes use of the fact that, like the C function for to, the one for move only intercepts A_Resume signals and passes all other signals on to its caller.

This expression can be implemented with code similar to the following. There are two possible signals that might be returned. move itself might produce an A_Resume signal or it might pass along the bounding signal from the success continuation. Note that for a compound expression, both the bounding label and the failure label are the same. In general, this is not true. In this context, the result of move(1) is discarded. The variable trashcan receives this value; it is never read.

```
    switch (move(1, &trashcan, sc)) {
       case 1:
          goto L1;
       case A_Resume:
          goto L1;
       }
L1: /* bounding label & failure label */
    ...

int sc() {
   return 1;  /* bound signal */
   }
```

Calling Conventions

This discussion has touched on the subject of calling conventions for run-time routines. In Icon, it is, in general, impossible to know until run time what an invocation is invoking. This is handled in the compiler with a standard calling convention for the C functions implementing operations and procedures. This calling convention allows a C function to be called without knowing anything about the operation it implements.

A function conforming to the standard calling convention has four parameters. These parameters are, in order of appearance, the number of Icon arguments (a C int), a pointer to the beginning of an array of descriptors holding the Icon arguments, a pointer to the descriptor used as the Icon result location, and a success continuation to use for suspension. The function itself is responsible for any argument conversions including dereferencing, and for argument list adjustment. As explained above, the function returns an integer signal. The function is allowed to return the signals A_Resume, A_Continue, and any signals returned by the success

continuation. It may ignore the success continuation if it does not suspend. The function may be passed a null continuation. This indicates that the function will not be resumed. In this case, suspend acts like a simple return, passing back the signal A_Continue (this is not shown in the examples). The outline of a standard-conforming function is

```
int function-name(nargs, args, result, succ_cont)
int nargs;
dptr args;
dptr result;
continuation succ_cont;
   {
   ...
   }
```

continuation is defined to be a pointer to a function taking no arguments and returning an integer.

Later sections of this dissertation describe the code generation process in more detail and describe optimizations of various parts of the code including parameter passing, continuations, signal handling, and branching.

CHAPTER 3

The Type Inferencing Model

Three sections of this dissertation are devoted to type inferencing: two chapters and an appendix. This chapter develops a theoretical model of type inferencing for Icon. For simplicity, it ignores some features of the language. This chapter presents intuitive arguments for the correctness of the formal model. Chapter 7 describes the actual implementation of type inferencing in the Icon compiler. The implementation handles the full Icon language and, for pragmatic reasons, differs from the theoretical model in some details. Appendix B presents more formal arguments for the correctness of the type inferencing system than given in this chapter.

This chapter starts with the motivation for performing type inferencing. It then describes the concept of *abstract interpretation*. This concept is used as a tool in this chapter to develop a type inferencing system from Icon's semantics. This chapter gives an intuitive presentation of this development process before presenting the formal models of abstract semantics for Icon. The most abstract of the formal models is the type inferencing system.

Motivation

Variables in the Icon programming language are untyped. 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 versus real addition). In the Icon interpreter system, 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 inferencing system 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 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.

Abstract Interpretation

Data flow analysis can be viewed as a form of abstract interpretation [35]. This can be 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, 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 chapter 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 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 [35] (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.

Once again, 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 in this chapter 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 does not determine the order in which they must occur. This sequencing information is discarded because 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.

This chapter assumes that an Icon program consists of a single procedure and that all invocations are to built-in functions. It also assumes that there are no co-expressions beyond the main

co-expression. See Chapter 7 for information on how to extend the abstractions to multiple procedures and multiple co-expressions.

Collecting Semantics

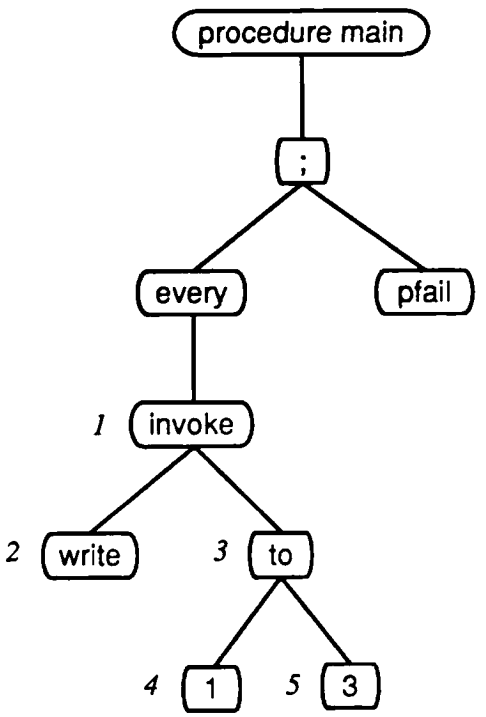
The collecting semantics of an Icon program is defined in terms a *flow graph* of the program. A flow graph is a directed graph used to represent the flow of control in a program. Nodes in the graph represent the executable primitives in the program. An edge exists from node A to node B if it is possible for execution to pass directly from the primitive represented by node A to the primitive represented by node B. Cousot and Cousot [35] 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.

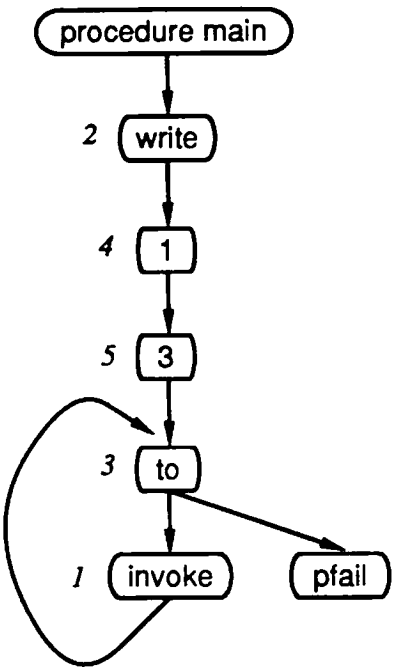
For an example of a flow graph, consider the Icon program

```
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 fail at the end of the procedure. 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 procedure failure (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 tree. This is shown on the right.



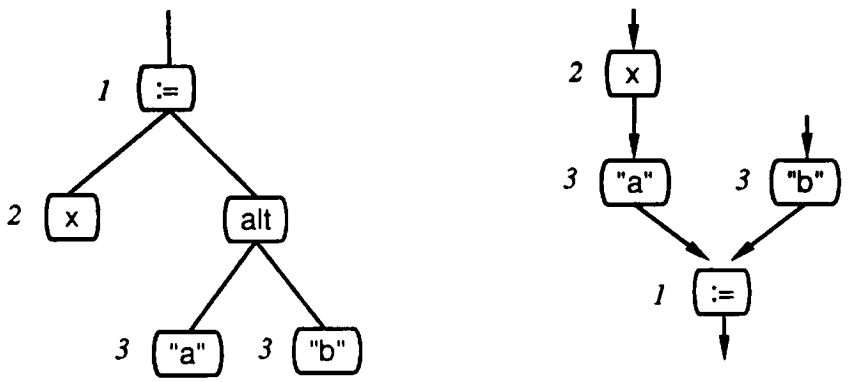


The node labeled procedure main is the *start node* for the procedure; it performs any necessary initializations to establish the execution environment for the procedure. The edge from invoke to to is a resumption path induced by the control structure every. The path from to to pfail is the failure path 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. Chapter 7 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 decision is unrelated to the use of a temporary-variable model in the compiler. 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 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"). The syntax tree below on the left and the and the flow graph are shown on the right.



The if and case control structures are handled similarly. In addition to temporary variables for intermediate results, some generators may need additional temporary variables to hold internal states during suspension. It is easy to devise a scheme to allocate them where they are needed; details are not presented here. The syntax 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 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.

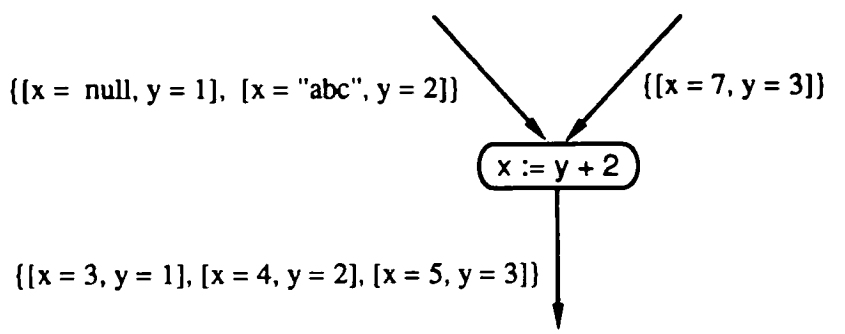
It requires a rather complex environment to capture the full operational semantics (and collecting semantics) of a language like Icon. For example, the environment needs to include a representation of the external file system. However, later abstractions only use the fact that the function read produces 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.

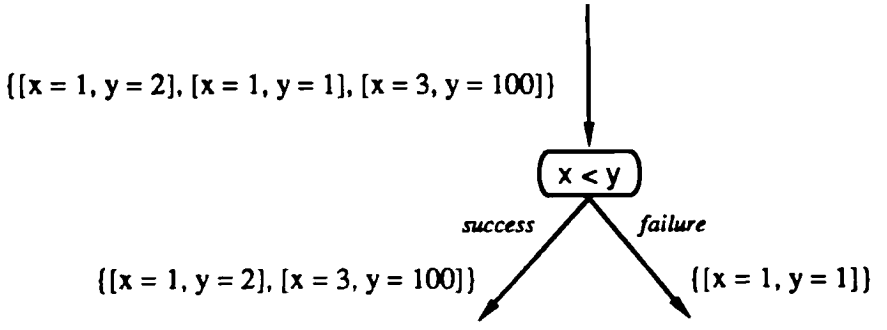
```
if x = 7 then {
   ...
   # x is 7 and y is 3
   }
else {
   ...
   # (x is null and y is 1) or (x is "abc" and y is 2)
   }
x := y + 2
```

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.



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.



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 existing Icon interpreter system is an example of how this information can be maintained [3]. 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; as mentioned earlier, this assumption is removed in Chapter 7), 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 (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. A list is a partial mapping from integers to variables. Representing other data types with pointer semantics is straightforward; this is discussed in Chapter 7.

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 respectives 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. $X \rightarrow Y$ is the set of partial functions from X to Y. The definition of the set possible environments for model 1 is

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

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

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

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

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, $e \in envir_{[1]}$, might be

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

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

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

$$L_1(1) = v_1$$

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

$$e_1 = (s_1, h)$$

$$s_1(a) = p_1$$
$$s_1(v_1) = \text{"xyz"}$$

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 environment 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"

$h_1(p_1) = L_2$

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

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

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

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

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

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 listing the result of $\hat{s}(\{x, y\})$ 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 $x = 1$ 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

$$\{1, 3, 5\} + \{2, 4\} = \{1+2, 1+4, 3+2, 3+4, 5+2, 5+4\}$$
$$= \{3, 5, 5, 7, 7, 9\}$$
$$= \{3, 5, 7, 9\}$$

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 these latter three environments are translated into an 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, 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 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 *weak updates* [14], 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 [11]. 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. Therefore, if a is a named variable, {a} 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

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

is mapped to

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

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

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

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

$$\text{heap}_{[3]} = \text{poiner-types} \rightarrow \text{structure-variable-types}$$

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

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

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

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

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 $store_{[3]}$ rather than $envir_{[3]}$ with the heap embedded within the type equations.

Suppose an environment from Model 2 is

$e \in envir_{[2]}$

$e = (s, h)$

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

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

$L_1(1) = v_1$

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

$L_3(1) = v_3$

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

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

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

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

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

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. This is discussed in Chapter 7. 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 the each abstraction maintains the "structure" of the problem, simply discarding some details.

Chapter 7 also extends type inferencing to handle the entire Icon language. Chapter 10 uses the information from type inferencing to optimize the generated code.

CHAPTER 4

Liveness Analysis of Intermediate Values

The maintenance of intermediate values during expression evaluation in the Icon programming language is more complicated than it is for conventional languages such as C and Pascal. O'Bagy explains this in her dissertation [18]:

> "Generators prolong the lifetime of temporary values. For example, in
>
> ```
> i = find(s1,s2)
> ```
>
> the operands of the comparison operation cannot be discarded when find produces its result. If find is resumed, the comparison is performed again with subsequent results from find(s1,s2), and the left operand must still be available."

In some implementation models, it is equally important that the operands of find still be available if that function is resumed (this depends on whether the operand locations are used during resumption or whether all needed values are saved in the local state of the function).

As noted in Chapter 2, a stack-based model handles the lifetime problem dynamically. However, a temporary-variable model like the one used in this compiler requires knowledge at compile-time of the lifetime of intermediate values. In a straightforward implementation of conventional languages, liveness analysis of intermediate values is trivial: an intermediate value is computed in one place in the generated code, is used in one place, and is live in the contiguous region between the computation and the use. In such languages, determining the lifetime of intermediate values only becomes complicated when certain optimizations are performed, such as code motion and common subexpression elimination across basic blocks [10, 16]. This is not true in Icon. In the presence of goal-directed evaluation, the lifetime of an intermediate value can extend beyond the point of use. Even in a straightforward implementation, liveness analysis is

not trivial.

In its most general form, needed in the presence of the optimizations mentioned above, liveness analysis requires iterative methods. However, goal-directed evaluation imposes enough structure on the liveness problem that, at least in the absence of optimizations, iterative methods are not needed to solve it. This chapter presents a simple and accurate method for computing liveness information for intermediate values in Icon. The analysis is formalized in an attribute grammar.

Implicit Loops

Goal-directed evaluation extends the lifetime of intermediate values by creating implicit loops within an expression. In O'Bagy's example, the start of the loop is the generator find and the end of the loop is the comparison that may fail. An intermediate value may be used within such a loop, but if its value is computed before the loop is entered, it is not recomputed on each iteration and the temporary variable must not be reused until the loop is exited.

The following fragment of C code contains a loop and is therefore analogous to code generated for goal-directed evaluation. It is used to demonstrate the liveness information needed by a temporary variable allocator. In the example, *v1* through *v4* represent intermediate values that must be assigned to program variables.

```
v1 = f1();
while (--v1) {
    v2 = f2();
    v3 = v1 + v2;
    f3(v3);
    }
v4 = 8;
```

Separate variables must be allocated for *v1* and *v2* because they are both needed for the addition.

Here, x is chosen for *v1* and y is chosen for *v2*.

```
x = f1();
while (--x) {
    y = f2();
    v3 = x + y;
    f3(v3);
    }
v4 = 8;
```

x cannot be used to hold *v3*, because x is needed in subsequent iterations of the loop. Its lifetime must extend through the end of the loop. y, on the other hand, can be used because it is recomputed in subsequent iterations. Either variable may be used to hold *v4*.

```
x = f1();
while (--x) {
    y = f2();
    y = x + y;
    f3(y);
    }
x = 8;
```

Before temporary variables can be allocated, the extent of the loops created by goal-directed evaluation must be estimated. Suppose O'Bagy's example

```
i = find(s1, s2)
```

appears in the following context

```
procedure p(s1, s2, i)
   if i = find(s1, s2) then return i + *s1
   fail
end
```

The simplest and most pessimistic analysis assumes that a loop can appear anywhere within the procedure, requiring the conclusion that an intermediate value in the expression may live to the end of the procedure. Christopher's simple analysis [17] notices that the expression appears within the control clause of an if expression. This is a bounded context; implicit loops cannot extend beyond the end of the control clause. His allocation scheme reuses, in subsequent expressions, temporary variables used in this control clause. However, it does not determine when temporary variables can be reused within the control clause itself.

The analysis presented here locates the operations within the expression that can fail and those that can generate results. It uses this information to accurately determine the loops within the expression and the intermediate values whose lifetimes are extended by those loops.

Liveness Analysis

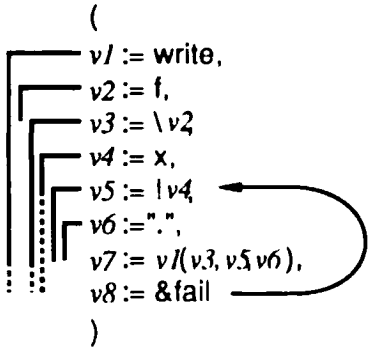
It is instructive to look at a specific example where intermediate values must be retained beyond (in a lexical sense) the point of their use. The following expression employs goal-directed evaluation to conditionally write sentences in the data structure x to an output file. Suppose f is either a file or null. If f is a file, the sentences are written to it; if f is null, the sentences are not written.

```
every write(\f, !x, ".")
```

In order to avoid the complications of control structures at this point in the discussion, the following equivalent expression is used in the analysis:

```
write(\f, !x, ".") & &fail
```

This expression can be converted into a sequence of primitive operations producing intermediate values (*v1*, *v2*, ...). This is shown in diagram. For convenience, the operations are expressed in Icon, except that the assignments do not dereference their right-hand operands.



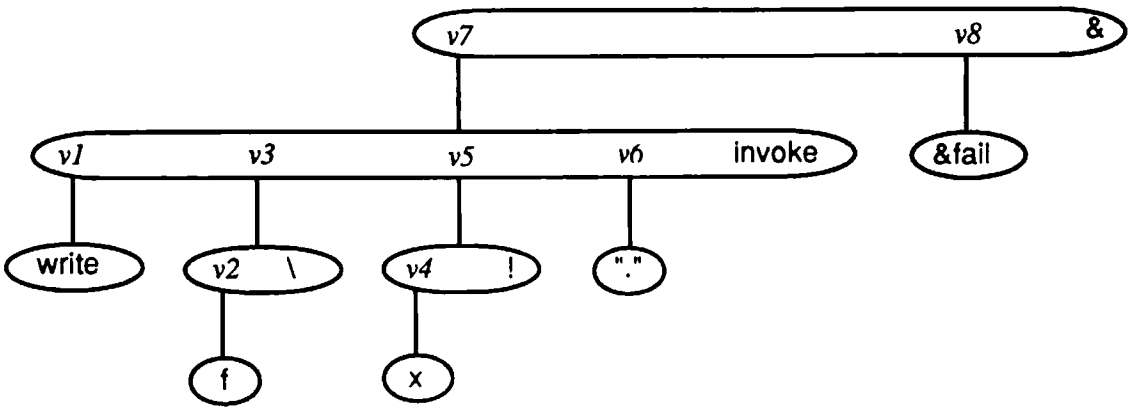
Whether or not the program variables and constants are actually placed in temporary variables depends on the machine model, implementation conventions, and what optimizations are performed. Clearly a temporary variable is not needed for &fail. However, temporary variables are needed if the subexpressions are more complex; intermediate values are shown for all subexpressions for explanatory purposes.

When &fail is executed, the ! operation is resumed. This creates an implicit loop from the ! to &fail, as shown by the arrow in the above diagram. The question is: What intermediate values must be retained up to &fail? A more instructive way to phrase the question is: After &fail is executed, what intermediate values could be reused without being recomputed? From the sequence of primitive operations, it is clear that the reused values include *v1* and *v3*, and, if the element generation operator, !, references its argument after resumption, then the reused values include *v4*. *v2* is not used within the loop, *v5* and *v6* are recomputed within the loop, and *v7* and *v8* are not used. The lines in the diagram to the left of the code indicate the lifetime of the intermediate values. The dotted portion of each line represents the region of the lifetime beyond what would exist in the absence of backtracking.

Liveness information could be computed by making the implicit loops explicit then performing a standard liveness analysis in the form of a global data flow analysis. That is unnecessarily expensive. There is enough structure in this particular liveness problem that it can be solved during the simple analysis required to locate the implicit loops caused by goal-directed evaluation.

Several concepts are needed to describe analyses involving execution order within Icon expressions. *Forward execution order* is the order in which operations would be executed at run time in the absence of goal-directed evaluation and explicit loops. Goal-directed evaluation involves both failure and the resumption of suspended generators. The control clause of an if-then-else expression may fail, but instead of resuming a suspending generator, it causes the else clause to be executed. This failure results in forward execution order. Forward execution order imposes a partial ordering on operations. It produces no ordering between the then and the else clauses of an if expression. *Backtracking order* is the reverse of forward execution order. This is due to the LIFO resumption of suspended generators. The backward flow of control caused by looping control structures does not contribute to this liveness analysis (intermediate results used within a looping control structure are also computed within the loop), but is dealt with in later chapters. The every control structure is generally viewed as a looping control structure. However, it simply introduces failure. Looping only occurs when it is used with a generative control clause, in which case the looping is treated the same as goal-directed evaluation.

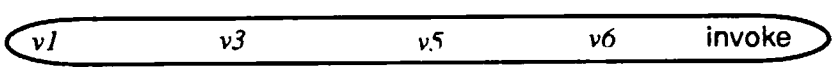
A notation that emphasizes intermediate values, subexpressions, and execution order is helpful for understanding how liveness is computed. Both postfix notation and syntax trees are inadequate. A postfix notation is good for showing execution order, but tends to obscure subexpressions. The syntax tree of an expression shows subexpressions, but execution order must be expressed in terms of a tree walk. In both representations, intermediate values are implicit. For this discussion, an intermediate representation is used. A subexpression is represented as a list of explicit intermediate values followed by the operation that uses them, all enclosed in ovals. Below each intermediate value is the subexpression that computes it. This representation is referred to as a *postfix tree*. The postfix tree for the example above is:



In this notation, the forward execution order of operations (which includes constants and references to program variables) is left-to-right and the backtracking order is right-to-left. In this example, the backtracking order is &fail, invoke, ".", !, x, \, f, and write.

As explained above, the use of an intermediate value must appear in an implicit loop for the value to have an extended lifetime. Two events are needed to create such a loop. First, an operation must fail, initiating backtracking. Second, an operation must be resumed, causing execution to proceed forward again. This analysis computes the maximum lifetime of intermediate values in the expression, so it only needs to compute the rightmost operation (within a bounded expression) that can fail. This represents the end of the farthest reaching loop. Once execution proceeds beyond this point, no intermediate value can be reused.

The intermediate values of a subexpression are used at the end of the subexpression. For example, invoke uses the intermediate values *v1*, *v3*, *v5*, and *v6*; the following figure shows these intermediate results and the operation in isolation.



In order for these uses to be in a loop, backtracking must be initiated from outside; that is, beyond the subexpression (in the example, only &fail and & are beyond the subexpression).

In addition, for an intermediate value to have an extended lifetime, the beginning of the loop must start after the intermediate value is computed. Two conditions may create the beginning of a loop. First, the operation itself may be resumed. In this case, execution continues forward within the operation. It may reuse any of its operands and none of them are recomputed. The operation does not have to actually generate more results. For example, reversible swap (the operator <->) can be resumed to reuse both of its operands, but it does not generate another result. Whether an operation actually reuses its operands on resumption depends on its implementation. In the Icon compiler, operations implemented with a C function using the standard calling conventions always use copies of operands on resumption, but implementations tailored to a particular use often reference operand locations on resumption. Liveness analysis is presented here as if all operations reuse their operands on resumption. In the actual implementation, liveness analysis computes a separate lifetime for values used internally by operations and the code generator decides whether this lifetime applies to operands. This internal lifetime may also be used when allocating tended descriptors for variables declared local to the in-line code for an operation. The behavior of the temporary-variable model presented in this dissertation can be compared with one developed by Nilsen and Martinek [36]; it also relies on the liveness analysis described in this chapter.

The second way to create the beginning of a loop is for a subexpression to generate results. Execution continues forward again and any intermediate values to the left of the generative subexpression may be reused without being recomputed. Remember, backtracking is initiated from outside the expression. Suppose an expression that can fail is associated with *v6*, in the previous figure. This creates a loop with the generator associated with *v5*. However, this particular loop does not include invoke and does not contribute to the reuse of *v1* or *v3*.

A resumable operation and generative subexpressions are all *resumption points* within an expression. A simple rule can be used to determine which intermediate values of an expression have extended lifetimes: If the expression can be resumed, the intermediate values with extended lifetimes consist of those to the left of the rightmost resumption point of the expression. This

rule refers to the "top level" intermediate values. The rule must be applied recursively to subexpressions to determine the lifetime of lower level intermediate values.

It sometimes may be necessary to make conservative estimates of what can fail and of resumption points (for liveness analysis, it is conservative to overestimate what can fail or be resumed). For example, invocation may or may not be resumable, depending on what is being invoked and, in general, it cannot be known until run time what is being invoked (for the purposes of this example analysis, it is assumed that the variable write is not changed anywhere in the program).

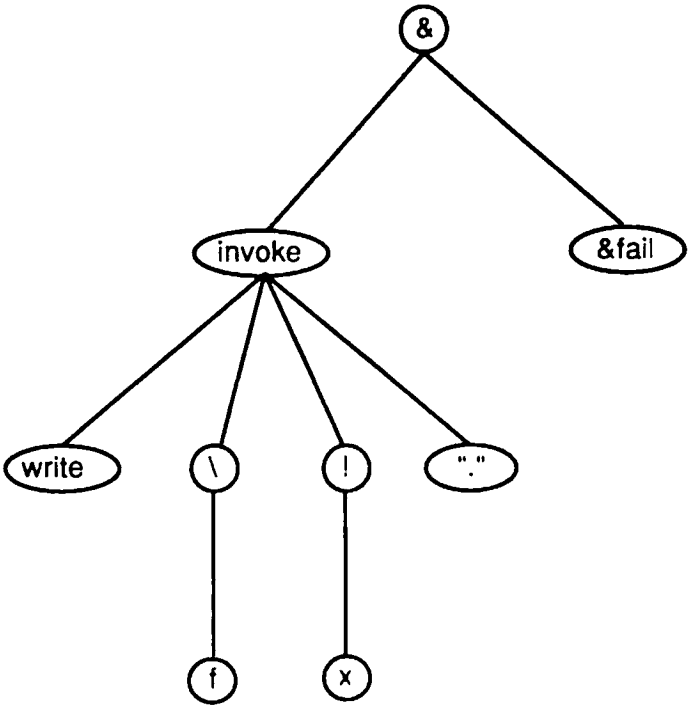
In the example, the rightmost operation that can fail is &fail. Resumption points are ! and the subexpressions corresponding to the intermediate values *v5* and *v7*.

Once the resumption points have been identified, the rule for determining extended lifetimes can be applied. If there are no resumption points in an expression, no intermediate values in that expression can be reused. Applying this rule to the postfix tree above yields *v1*, *v3*, and *v4* as the intermediate values that have extended lifetimes.

Similar techniques can be used for liveness analysis of Prolog programs, where goal-directed evaluation also creates implicit loops. One difference is that a Prolog clause is a linear sequence of calls. It does not need to be "linearized" by construction a postfix tree. Another difference is that all intermediate values in Prolog programs are stored in explicit variables. A Prolog variable has a lifetime that extends to the right of its last use if an implicit loops starts after the variable's first use and ends after the variable's last use.

An Attribute Grammar

To cast this approach as an attribute grammar, an expression should be thought of in terms of an abstract syntax tree. The transformation from a postfix tree to a syntax tree is trivial. It is accomplished by deleting the explicit intermediate values. A syntax tree for the example is:



Several interpretations can be given to a node in a syntax tree. A node can be viewed as representing either an operation, an entire subexpression, or an intermediate value.

This analysis associates four attributes with each node (this ignores attributes needed to handle break expressions). The goal of the analysis is to produce the lifetime attribute. The other three attributes are used to propagate information needed to compute the lifetime.

- resumer is either the rightmost operation (represented as a node) that can initiate backtracking into the subexpression or it is null if the subexpression cannot be resumed.
- failer is related to resumer. It is the rightmost operation that can initiate backtracking that can continue past the subexpression. It is the same as resumer, unless the subexpression itself contains the rightmost operation that can fail.

- gen is a boolean attribute. It is true if the subexpression can generate multiple results if resumed.
- lifetime is the operation beyond which the intermediate value is no longer needed. It is either the parent node, the resumer of the parent node, or null. The lifetime is the parent node if the value is never reused after execution leaves the parent operation. The lifetime is the resumer of the parent if the parent operation or a generative sibling to the right can be resumed. A lifetime of null is used to indicate that the intermediate value is never used. For example, the value of the control clause of an if expression is never used.

Attribute computations are associated with productions in the grammar. The attribute computations for failer and gen are always for the non-terminal on the left-hand side of the production. These values are then used at the parent production; they are effectively passed up the syntax tree. The computations for resumer and lifetime are always for the attributes of non-terminals on the right-hand side of the production. resumer is then used at the productions defining these non-terminals; it is effectively passed down the syntax tree. lifetime is usually saved just for the code generator, but it is sometimes used by child nodes.

Primary Expressions

Variables, literals, and keywords are primary expressions. They have no subexpressions, so their productions contain no computations for resumer or lifetime. The attribute computations for a literal follow. A literal itself cannot fail, so backtracking only passes beyond it if the backtracking was initiated before (to the right of) it. A literal cannot generate multiple results.

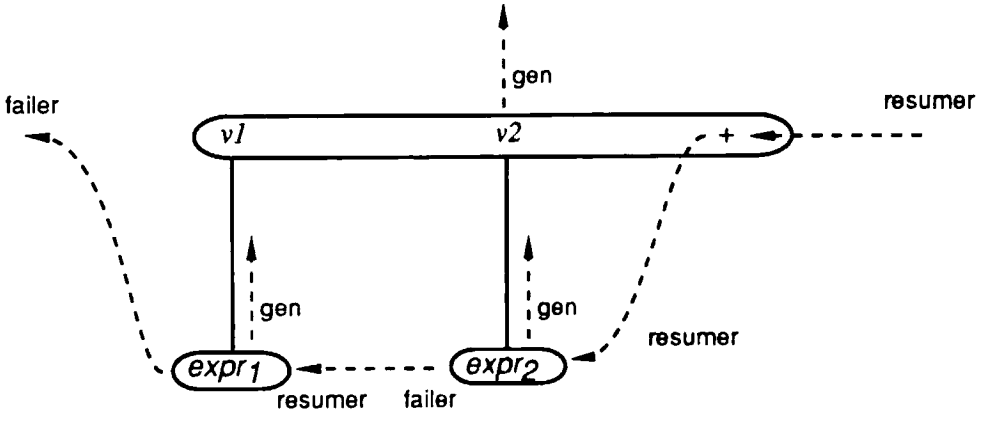
```
expr ::= literal {
            expr.failer := expr.resumer
            expr.gen := false
            }
```

Another example of a primary expression is the keyword &fail. Execution cannot continue past &fail, so it must be the rightmost operation within its bounded expression that can fail. A pre-existing attribute, node, is assumed to exist for every symbol in the grammar. It is the node in the syntax tree that corresponds to the symbol.

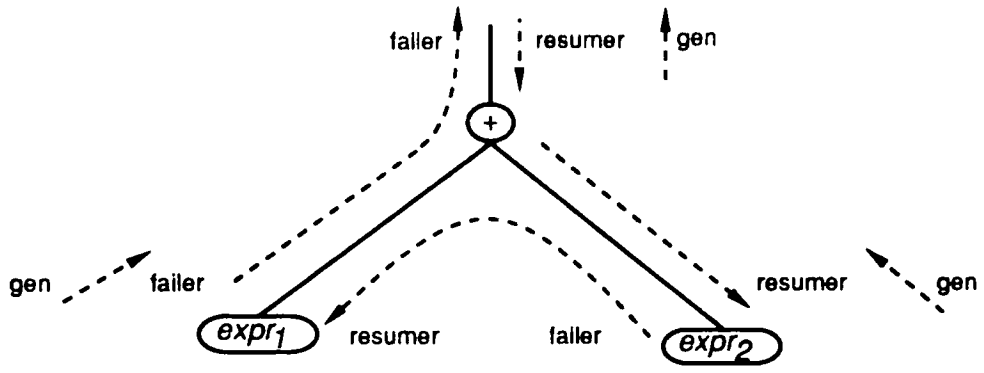
```
expr ::= &fail {
            expr.failer := expr.node
            expr.gen := false
            }
```

Operations with Subexpressions

Addition provides an example of the attribute computations involving subexpressions. The following diagram shows how resumer, failer, and gen information would be passed through the postfix tree.



This information would then be used to compute lifetime information for *v1* and *v2*. The next figure shows how the attribute information is actually passed through the syntax tree.



The lifetime attributes are computed for the roots of the subtrees for $expr_1$ and $expr_2$.

The details of the attribute computations depend, in part, on the characteristics of the individual operation. Addition does not fail, so the rightmost resumer, if there is one, of $expr_2$ is the rightmost resumer of the entire expression. The rightmost resumer of $expr_1$ is the rightmost operation that can initiate backtracking that continues past $expr_2$. Addition does not suspend, so the lifetime of the value produced by $expr_2$ only extends through the operation (that is, it always is recomputed in the presence of goal-directed evaluation). If $expr_2$ is a generator, then the result of $expr_1$ must be retained for as long as $expr_2$ might be resumed. Otherwise, it need only be retained until the addition is performed. $expr_1$ is the first thing executed in the expression, so its failer is the failer for the entire expression. The expression is a generator if either $expr_1$ or $expr_2$ is a generator (note that the operation | is logical *or*, not Icon's alternation control structure):

```
expr ::= expr1 + expr2 {
                expr2.resumer := expr.resumer
                expr2.lifetime := expr.node
                expr1.resumer := expr2.failer
```

```
        if expr2.gen & (expr.resumer ≠ null) then
            expr1.lifetime := expr.resumer
        else
            expr1.lifetime := expr.node
        expr.failer := expr1.failer
        expr.gen := (expr1.gen | expr2.gen)
        }
```

/expr provides an example of an operation that can fail. If there is no rightmost resumer of the entire expression, it is the rightmost resumer of the operand. The lifetime of the operand is simply the operation, by the same argument used for $expr_2$ of addition. The computation of failer is also analogous to that of addition. The expression is a generator if the operand is a generator:

```
expr ::= /expr1 {
        if expr.resumer = null then
            expr1.resumer := expr.node
        else
            expr1.resumer := expr.resumer
        expr1.lifetime := expr.node
        expr.failer := expr1.failer
        expr.gen := expr1.gen
        }
```

!expr differs from /expr in that it can generate multiple results. If it can be resumed, the result of the operand must be retained through the rightmost resumer:

```
expr ::= !expr1 {
            if expr.resumer = null then {
               expr1.resumer := expr.node
               expr1.lifetime := expr.node
               }
            else {
               expr1.resumer := expr.resumer
               expr1.lifetime := expr.resumer
               }
            expr.failer := expr1.failer
            expr.gen := true
            }
```

Control Structures

Other operations follow the general pattern of the ones presented above. Control structures, on the other hand, require unique attribute computations. In particular, several control structures bound subexpressions, limiting backtracking. For example, not bounds its argument and discards the value. If it has no resumer, then it is the rightmost operation that can fail. The expression is not a generator:

```
expr ::= not expr1 {
            expr1.resumer := null
            expr1.lifetime := null
```

```
        if expr.resumer = null then
            expr.failer :=  expr.node
        else
            expr.failer :=  expr.resumer
        expr.gen := false
        }
```

$expr_1$; $expr_2$ bounds $expr_1$ and discards the result. Because the result of $expr_2$ is the result of the entire expression, the code generator makes their result locations synonymous. This is reflected in the lifetime computations. Indeed, all the attributes of $expr_2$ and those of the expression as a whole are the same:

```
expr ::= expr1 ; expr2 {
        expr1.resumer := null
        expr1.lifetime := null
        expr2.resumer := expr.resumer
        expr2.lifetime := expr.lifetime
        expr.failer := expr2.failer
        expr.gen := expr2.gen
        }
```

A reasonable implementation of alternation places the result of each subexpression into the same location: the location associated with the expression as a whole. This is reflected in the lifetime computations. The resumer of the entire expression is also the resumer of each subexpression. Backtracking out of the entire expression occurs when backtracking out of $expr_2$ occurs. This expression is a generator:

```
expr ::= expr1 | expr2 {
            expr2.resumer:= expr.resumer
            expr2.lifetime := expr.lifetime
            expr1.resumer := expr.resumer
            expr1.lifetime := expr.lifetime
            expr.failer := expr2.failer
            expr.gen := true
            }
```

The first operand of an if expression is bounded and its result is discarded. The other two operands are treated similar to those of alternation. Because there are two independent execution paths, the rightmost resumer may not be well-defined. However, it is always conservative to treat the resumer as if it is farther right than it really is; this just means that an intermediate value is kept around longer than needed. If there is no resumer beyond the if expression, but at least one of the branches can fail, the failure is treated as if it came from the end of the if expression (represented by the node for the expression). Because backtracking out of an if expression is rare, this inaccuracy is of little practical consequence. The if expression is a generator if either branch is a generator:

```
expr ::= if expr1 then expr2 else expr3 {
            expr3.resumer := expr.resumer
            expr3.lifetime := expr.lifetime
            expr2.resumer := expr.resumer
            expr2.lifetime := expr.lifetime
            expr1.resumer := null
            expr1.lifetime := null
```

```
        if expr.resumer = null & (expr1.failer ≠ null | expr2.failer ≠ null) then
            expr.failer := expr.node
        else
            expr.failer = expr.resumer
        expr.gen := (expr2.gen | expr3.gen)
        }
```

The do clause of every is bounded and its result discarded. The control clause is always resumed at the end of the loop and can never be resumed by anything else. The value of the control clause is discarded. Ignoring break expressions, the loop always fails:

```
expr ::= every expr1 do expr2 {
        expr2.resumer := null
        expr2.lifetime := null
        expr1.resumer := expr.node
        expr1.lifetime := null
        expr.failer := expr.node
        expr.gen := false
        }
```

Handling break expressions requires some stack-like attributes. These are similar to the ones used in the control flow analyses described in O'Bagy's dissertation [18] and the ones used to construct flow graphs in the original technical report on type inferencing.

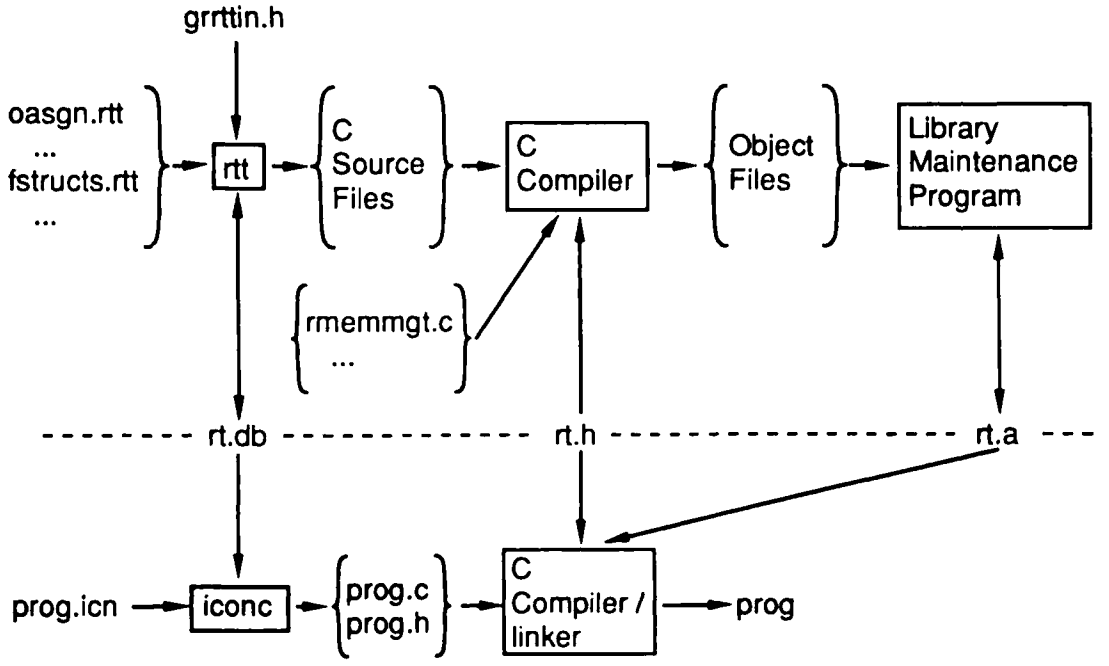
The attributes presented here can be computed with one walk of the syntax tree. At a node, subtrees are processed in reverse execution order: first the resumer and lifetime attributes of a subtree are computed, then the subtree is walked. Next the failer and gen attributes for the node itself are computed, and the walk moves back up to the parent node.

CHAPTER 5

Overview of the Compiler

Components of the Compiler

The Icon compiler is divided into two components: a run-time system and the compiler itself. This organization is illustrated below. In the diagram, labeled boxes represent programs, other text (some of it delimited by braces) represents files, and arrows represent data flow.



The run-time system appears above the dotted line and the compiler itself appears below the line. The programs shown with the run-time system are executed once when the run-time system is installed or updated. They build a data base, rt.db, and an object-code library, rt.a, for use by the compiler. The general definition of the term "data base" is used here: a collection of related data. rt.db is stored as a text file. It is accessed and manipulated in internal tables by the

programs rtt and iconc. The rtt program is specific to the Icon compiler system and is described below. The C compiler and the library maintenance program are those native to the system on which the Icon compiler is being used. The format of the object-code library is dictated by the linker used with the C compiler. The file rt.h contains C definitions shared by the routines in the run-time system and code generated by the compiler.

The diagram of the compiler itself reflects the fact that the Icon compiler uses a C compiler and linker as a back end. However, iconc automatically invokes these programs, so the Icon programmer sees a single tool that takes Icon source as input and produces an executable file.

The Run-time System

As with the run-time system for the interpreter, the run-time system for the compiler implements Icon's operations. However, the compiler has needs beyond those of the interpreter. In the interpreter's run-time system, all operations are implemented as C functions conforming to certain conventions. The interpreter uses the same implementation of an operation for all uses of the operation. While this approach results in acceptable performance for many Icon programs, the purpose of an optimizing compiler is to obtain better performance. A major goal in the development of iconc is to use information from type inferencing to tailor the parameter passing and parameter type conversions of an operation to particular uses of the operation and to place code in line where appropriate. The compiler needs a mechanism to support this tailored operation invocation. In addition, the compiler needs information about the properties of operations for use in performing type inferencing and other analyses.

In addition to supporting the analyses and optimizations of iconc, there are several other major goals in the design of the compiler's run-time system. These include

- Specification of all information about an operation in one place.
- Use of one coding to produce both general and tailored implementation of an operation.
- Use of the pre-existing run-time system as a basis for the new one.

Most of the design goals for the run-time system would best be served by a special-purpose language in which to implement the run-time system. Such a language would allow the properties of an operation needed by various analyses to be either explicitly coded or easily inferred from parts of the code used to produce an implementation of the operation. The language would also allow easy recognition and manipulation of parts of the code that need to be tailored to individual uses of an operation. In addition, the language would provide support for features of Icon such as its data types and goal-directed evaluation.

While a special-purpose language is consistent with most design goals, it is not consistent with using the interpreter's run-time system written in C as a basis for that of the compiler. A special-purpose language also has the problem that it requires a large effort to implement. These conflicting goals are resolved with a language that is a compromise between an ideal special-purpose implementation language and C. The core of the language is C, but the language contains special features for specifying those aspects of a run-time operation that must be dealt with by the compiler. This language is called the *implementation language* for the Icon compiler's run-time system. Because the implementation language is designed around C, much of the detailed code for implementing an operation can be borrowed from the interpreter system with only minor changes. The important facets of the implementation language are discussed here. A full description of the language can be found in the reference manual for the language [37]. The core material from this reference manual is included as Appendix A of this dissertation.

The Implementation Language

The implementation language is used to describe the operators, keywords, and built-in functions of Icon. In addition to these operations, the run-time system contains routines to support other features of Icon such as general invocation, co-expression activation, and storage management. These other routines are written in C.

The program rtt takes as input files containing operations coded in the implementation language and translates the operations into pure C. rtt also builds the data base, rt.db, with information about the operations. In addition to operations written in the implementation language, rtt input may contain C functions. These C functions can use several of the extensions available to the detailed C code in the operations. These extensions are translated into ordinary C code. While not critical to the goals of the run-time system design, the ability to use these extensions in otherwise ordinary C functions is very convenient.

The definition of an operation is composed of three layers. The outer layer brackets the code for the operation. It consists of a header at the beginning of the code and the reserved word end at the end of the code. The header may be preceded by an optional description of the operation in the form of a string literal; this description is used only for documentation. The second layer consists of type checking and type conversion code. Type checking code may be nested. The inner layer is the detailed C code, abstract type computations, and code to handle run-time errors. An abstract type computation describes the possible side-effects and result types of the operation in a form that type inferencing can use. This feature is needed because it is sometimes impractically difficult to deduce this information from the C code. The code to handle run-time errors is exposed; that is, it is not buried within the detailed C code. Because of this, type inferencing can easily determine conditions under which an operation terminates without either producing a value or failing. (A further reason for exposing this code is explained in the implementation language reference manual in the section on scoping.)

An operation header starts with one of the three reserved words operator, function, or keyword. The header contains a description of the operation's *result sequence*, that is, how many

results it can produce. This includes both the minimum and maximum number of results, with * indicating an unbounded number. In addition, it indicates, by a trailing +, when an operation can be resumed to perform a side-effect after it has produced its last result. This information is somewhat more detailed than can easily be inferred by looking at the returns, suspends, and fails in the detailed C code. The information is put in the data base for use by the analysis phases of iconc.

An operation header also includes an identifier. This provides the name for built-in functions and keywords. For all types of operations, rtt uses the identifier to construct the names of the C functions that implement the operations. The headers for operators also include an operator symbol. The parser for iconc is not required to use this symbol for the syntax of the operation, but for most operations it does so; list creation, [...], is an example of an exception. The headers for built-in functions and operators include a parameter list. The list provides names for the parameters and indicates whether dereferenced and/or undereferenced versions of the corresponding argument are needed by the operation. It also indicates whether the operation takes a variable number of arguments.

The following are five examples of operation headers.

```
function{0,1+} move(i)

function{*} bal(c1,c2,c3,s,i,j)

operator{1} [...] llist(elems[n])

operator{0,1} / null(underef x -> dx)

keyword{3} regions
```

move is a function that takes one argument. It may produce zero or one result and may be resumed to produce a side effect after its last result. bal is a function that takes six arguments. It produces an arbitrary number of results. The list-creation operator is given the symbol [...]

(which may be used for string invocation, if string invocation is enabled) and is given the name llist. It takes an arbitrary number of arguments with elems being the array of arguments and n being the number of arguments. List creation always produces one result. The / operator is given the name null. It takes one argument, but both undereferenced and dereferenced versions are needed by the operation. It produces either zero or one result. &regions is a keyword that produces three results.

Type checking and type conversion constructs consist of an if-then construct, an if-then-else construct, a type_case construct that selects code to execute based on the type of an argument, and a len_case construct that selects code to execute based on the number of arguments in the variable part of a variable-length argument list. The conditions in the if-then and if-then-else constructs are composed of operations that check the types of arguments or attempt to convert arguments to specific types.

A type check starts with 'is:'. This is followed by the name of a type and an argument in parentheses. For example, the then clause of the following if is executed if x is a list.

```
if is:list(x) then ...
```

A type conversion is similar to a type check, but starts with 'cnv:'. For example, the following code attempts to convert s to a string. If the conversion succeeds, the then clause of the if is executed.

```
if cnv:string(s) then ...
```

There are forms of conversion that convert a null value into a specified default value, forms that put a converted value in a location other than the parameter, and forms that convert Icon values into certain types of C values. The later type of conversion is convenient because the detailed code is expressed in C. In addition, exposing conversions back and forth between Icon and C values leaves open the possibility of future optimizations to eliminate unnecessary conversions to Icon values. The control clause of an if may also use limited forms of expressions involving

boolean operators. The full syntax and semantics of conversions are described in the implementation language reference manual.

Detailed code is expressed in a slightly extended version of C and is introduced by one of two constructs. The first is

inline { *extended C* }

This indicates that it is reasonable for the Icon compiler to put the detailed code in line. The second construct is

body { *extended C* }

This indicates that the detailed code is too large to be put in line and should only appear as part of a C function in the run-time library. The person who codes the operation is free to decide which pieces of detailed code are suitable to in-lining and which are not. The decision is easily changed, so an operation can be fine tuned after viewing the code produced by the compiler.

One extension to C is the ability to declare variables that are tended with respect to garbage collection. Another extension is the ability to use some constructs of the implementation language, such as type conversions, within the C code. An important extension is the inclusion of return, suspend, and fail statements that are analogous to their Icon counterparts. This extension, combined with the operation headers, makes the coding of run-time routines independent of the C calling conventions used in the compiler system. The return and suspend statements have forms that convert C values to Icon values, providing inverses to conversions in the type checking code of the implementation language.

This mechanism is more than is necessary for those keywords that are simple constants. For keywords that are string, cset, integer, or real constants there is a special form of definition. The Icon compiler treats keywords coded with these definitions as manifest constants. When future versions of the Icon compiler implement constant folding, that optimization will be automatically applied to these keywords.

Standard and Tailored Operation Implementations

For every operation that it translates, except keywords, rtt creates a C function conforming to the standard calling conventions of the compiler system. With the help of the C compiler and library maintenance routine, rtt puts an object module for that function in the compiler system's run-time library. This function is suitable for invocation through a procedure block. It is used with unoptimized invocations.

rtt creates an entry in the data base for every operation it translates, including keywords. This entry contains the code for the operation. The code is stored in the data base in a form that is easier to parse and process than the original source, and the body statements are replaced by calls to C functions. These C functions are in the run-time library and implement the code from the body statement. The calling conventions for these functions are tailored to the needs of the code and do not, in general, conform to the standard calling conventions of the compiler system.

When the compiler can determine that a particular operation is being invoked, it locates the operation in the data base and applies information from type inferencing to simplify or eliminate the code in the operation that performs type checking and conversions on arguments. These simplifications will eliminate detailed code that will never be executed in this invocation of the operation. The compiler can attempt the simplification because the type checking code is in the data base in a form that is easily dealt with. If enough simplification is possible, a tailored version of the operation is generated in line. This tailored version includes the simplified type checking code, if there is any left. For detailed code that has not been eliminated by the simplification, the tailored version also includes the C code from inline statements and includes calls to the functions that implement the code in body statements. The process of producing tailored versions of built-in operations is described in more detail in a later chapter.

Ideally, when unique types can be inferred for the operands of an operation, the compiler should either produce a small piece of type-specific in-line C code or produce a call to a type-specific C function implementing the operation. It is possible to code operations in the implementation language such that the compiler can do this. In addition, this is the natural way to

code most operations. For the few exceptions, there are reasonable compromises between ideal generated code and elegant coding in the implementation language. This demonstrates that the design and implementation of the run-time system and its communication with the compiler is successful.

CHAPTER 6

Organization of Iconc

The Icon compiler, iconc, takes as input the source code for an Icon program and, with the help of a C compiler and linker, produces an executable file. The source code may be contained in several files, but iconc does not support separate compilation. It processes an entire program at once. This requirement simplifies several of the analyses, while allowing them to compute more accurate information. Without the entire program being available, the effects of procedures in other files is unknown. In fact, it is not possible to distinguish built-in functions from missing procedures. Type inferencing would be particularly weakened. It would have to assume that any call to an undeclared variable could have any side effect on global variables (including procedure variables) and on any data structure reachable through global variables or parameters.

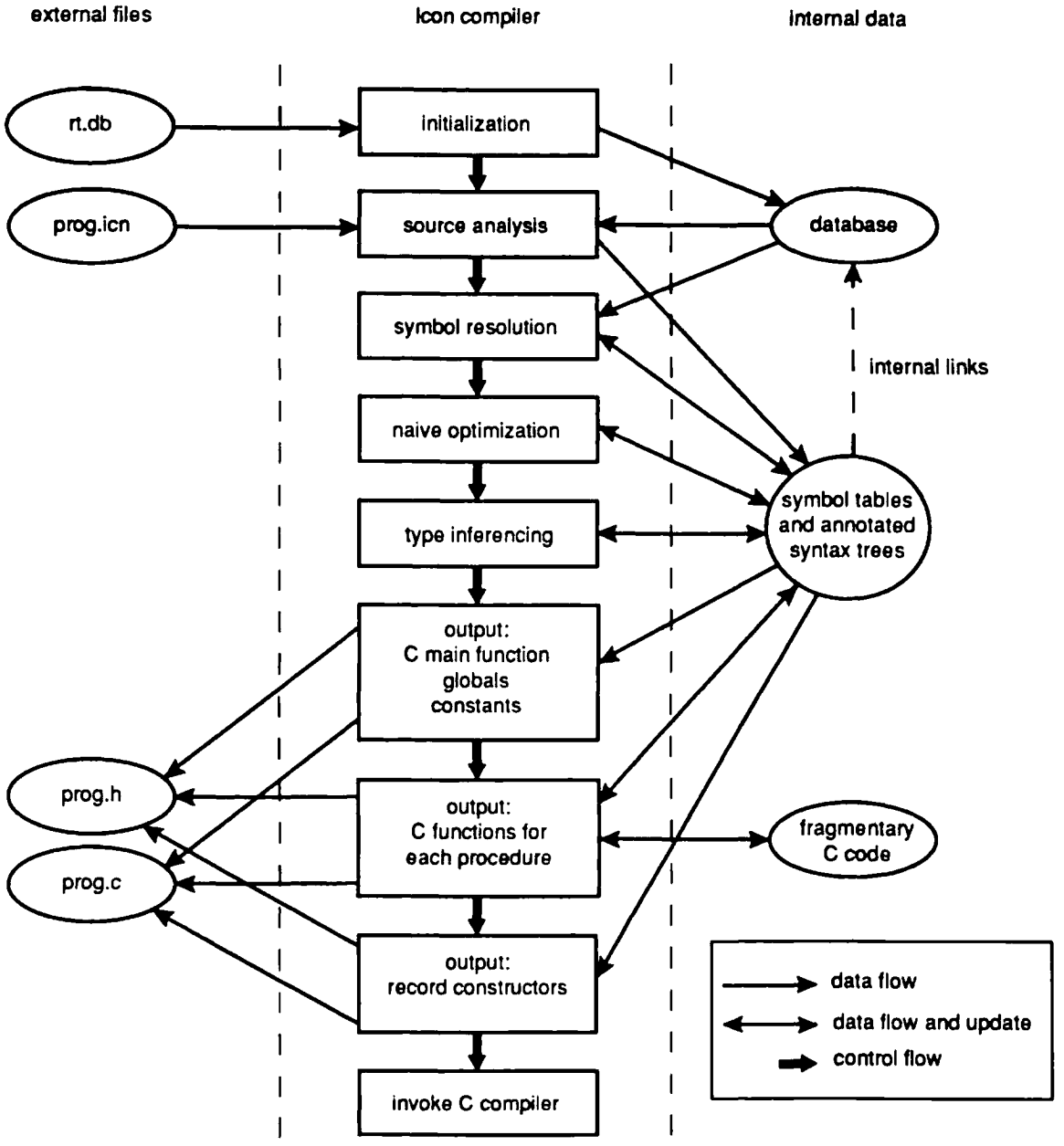
Compiler Phases

Iconc is organized into a number of phases. These are illustrated in the diagram on the following page.

The initialization phase includes reading a data base of information about run-time routines into internal tables. This information is used in many of the other phases.

The source analysis phase consists of a lexical analyzer and parser. These are adapted from those used in the interpreter system. The parser generates abstract syntax trees and symbol tables for all procedures before subsequent phases are invoked. The symbol resolution phase determines the scope of variables that are not declared in the procedures where they are used. This resolution can only be done completely after all source files for the program are read. If a variable does not have a local declaration, the compiler checks to see whether the variable is declared global (possibly as a procedure or record constructor) in one of the source files. If not, the compiler checks to see whether the variable name matches that of a built-in function. If the

name is still not resolved, it is considered to be a local variable.



Naive Optimizations

Naive optimizations involve invocation and assignment. These optimizations are done before type inferencing to aid that analysis. Certain "debugging features" of Icon such as the variable function interfere with these optimizations. By default, these features are disabled. If the user of iconc requests the debugging features, these optimizations are bypassed. While these optimizations are being done, information is gathered about whether procedures suspend, return, or fail. This information is used in several places in the compiler.

The invocation optimization replaces general invocation by a direct form of invocation to a procedure, a built-in function, or a record constructor. This optimization involves modifying nodes in the syntax tree. It only applies to invocations where the expression being invoked is a global variable initialized to a value in one of the three classes of procedure values. First, the Icon program is analyzed to determine which variables of this type appear only as the immediate operands of invocations. No such variable is ever assigned to, so it retains its initial value throughout the program (a more exact analysis could be done to determine the variables that are not assigned to, but this would seldom yield better results in real Icon programs because these programs seldom do anything with procedure values other that invoke them). This means that all invocations of these variables can be replaced by direct invocations. In addition, the variables themselves can be discarded as they are no longer referenced.

The invocation optimization improves the speed of type inferencing in two ways, although it does nothing to improve the accuracy of the information produced. Performing type inferencing on direct invocations is faster than performing it on general invocations. In addition, type inferencing has fewer variables to handle, which also speeds it up.

The invocation optimization does improve code generated by the compiler. In theory, the optimization could be done better after type inferencing using the information from that analysis, but in practice this would seldom produce better results. On most real Icon programs, this optimization using the naive analysis replaces all general invocations with direct ones.

As noted in Chapter 3, it is important for type inferencing to distinguish strong updates from weak updates. The data base contains a general description of assignment, but it would be difficult for a type inferencing system to use the description in recognizing that a simple assignment or an augmented assignment to a named variable is a strong update. It is much easier to change general assignments where the left hand side is a named variable to a special assignment and have type inferencing know that the special assignment is a strong update. Special-casing assignment to named variables is also important for code generation. General optimizations to run-time routines are not adequate to produce the desired code for these assignments. The optimizations to assignment are described in Chapter 10.

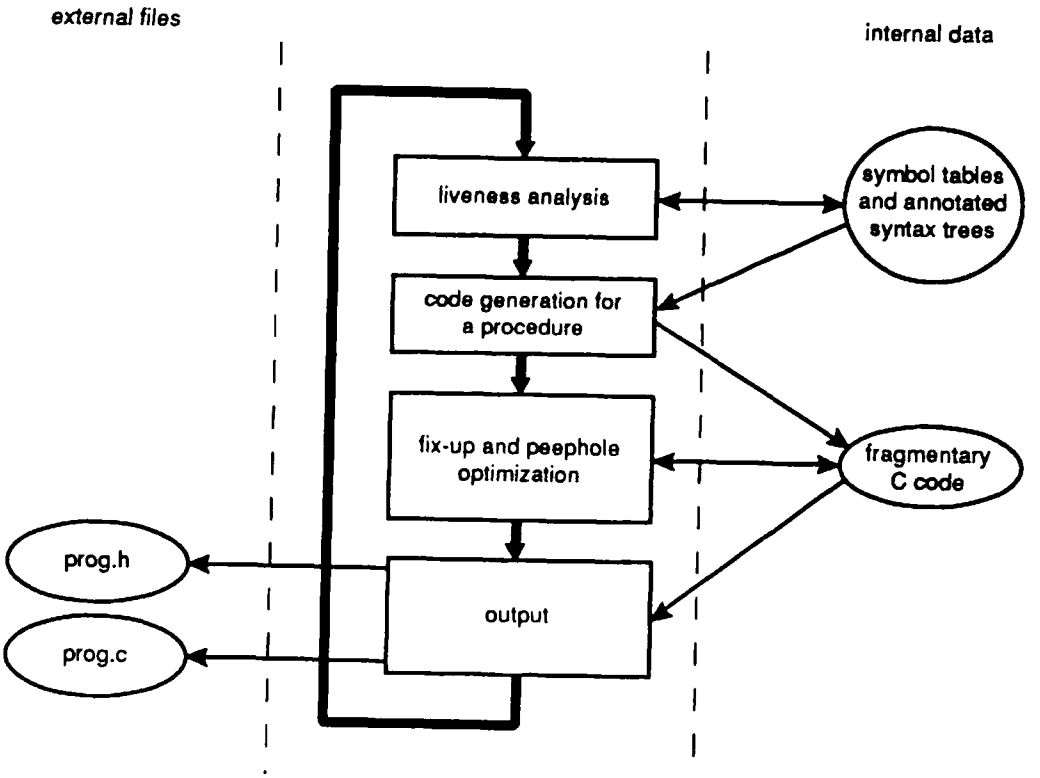
The details of type inferencing is described in other chapters. Producing code for the C main function, global variables, constants, and record constructors is straightforward. C code is written to two files for organizational purposes; it allows definitions and code to be written in parallel.

Code Generation for Procedures

Producing code for procedures involves several sub-phases. The sub-phases are liveness analysis, basic code generation, fix-up and peephole optimization, and output. During this phase of code generation, procedures are processed one at at time.

These sub-phases are described in later chapters. The code fix-up phase and peephole optimization are performed during the same pass over the internal representation of the C code. Some clean-up from peephole optimization is performed when the code is written. The logical organization of the compiler places the fix-up phase as a pass in code generation with peephole optimization being a separate phase. The organization of this dissertation reflects the logical organization of the compiler rather than its physical organization.

The physical organization of this phase is shown in the following diagram.


external files
internal data
liveness analysis
symbol tables and annotated syntax trees
code generation for a procedure
fix-up and peephole optimization
fragmentary C code
prog.h
prog.c
output

CHAPTER 7

The Implementation of Type Inferencing

Chapter 3 develops a theoretical type inferencing model for Icon, called Model 3. The purpose of that chapter is to explain the relationship between type inferencing and the semantics of Icon; for simplicity, some features of the language along with certain questions of practical importance are ignored in that chapter. This chapter describes the implementation of the type inferencing system used in the Icon compiler. The implementation is based on Model 3. This chapter describes solutions to those issues that must be addressed in developing a complete practical type inferencing system from Model 3. The issues include:

- the representation of types and stores
- the development of a type system for the full Icon language
- the handing of procedure calls and co-expression activation
- the determination of edges in the flow graph
- the computation of a fixed point for type information

In addition, performance of the abstract interpretation must be considered. This includes both speed and memory usage.

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 the type inferencing model, Model 3, presented in Chapter 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 does work 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 [15].

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 from Chapter 3 includes no structure type other than lists, nor does it consider how to handle types for procedure and co-expression values to allow effective type inferencing in their presence. This section develops a complete and effective type system for Icon.

Most of the structure types of Icon are assigned several subtypes in the type inferencing system. As explained for lists in Chapter 3, 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. The invocation optimization described in Chapter 6 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 in the data base. A new clause in an

abstract type computation indicates that a structure can be created. The following example is the abstract type computation from the built-in function list. It indicates the the function creates and returns a new list with elements whose type is the same as that of the parameter x (the second parameter).

```
abstract {
   return new list(type(x))
   }
```

This is the clause as written by the programmer developing the run-time library; it is translated into an internal form for storage in the data base.

Invocation optimizations may not identify the operation associated with some invocations. The initialization phase of type inferencing skips these invocations. Type inferencing may later discover that one of these invocations can create a structure. Each structure type is given one subtype that is used for all of these later discoveries. While it is possible for there to be several of these creation points representing logically distinct subtypes, this seldom occurs in practice. If it does happen, type inferencing produces a correct, but less precise, result.

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 is discussed more in the last section of this chapter). 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 are:

- null
- string
- cset
- integer
- real
- file
- list subtypes
- set subtypes
- table subtypes
- record subtypes
- procedure subtypes
- co-expression subtypes

The list subtypes are allocated with

- one for the argument to the main procedure
- one for each easily recognized creation point
- one representing all other lists

The set subtypes are allocated with

- one for each easily recognized creation point
- one representing all other sets

The table subtypes are allocated with

- one for each easily recognized creation point
- one representing all other tables

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
- one representing operators available for string invocation

Note that procedure subtypes are allocated after most procedure and function values have been eliminated by invocation optimizations (the procedures and functions are still there, they are just not Icon values). Therefore, few of these subtypes are actually allocated. The co-expression subtypes are allocated with

- one for the main co-expression
- one for each create expression

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. The variable reference types are

- integer keyword variable types
- &pos
- &subject
- substring trapped variable types
- table-element trapped variable types
- list-element reference types
- table assigned-value reference types
- field reference types
- global variable reference types
- local variable reference types

&random and &trace behave the same way from the perspective of the type inferencing system, so they are grouped under one type as integer keyword variables. The fact that &pos can cause assignment to fail is reflected in the type inferencing system, so it is given a separate type. &subject is the only string keyword variable so it is in a type by itself.

Substring trapped variables and table-element trapped variables are "hidden" structures in the implementation of Icon. They appear as intermediate results, but are only indirectly observable in the semantics of Icon. In order to reflect these semantics in the type inferencing system, there are substring trapped variable types and table-element trapped variable types. These are structure types similar to sets, but are variable reference types rather than first-class types. The substring trapped variable types are allocated with

- one for each easily recognized creation point
- one representing all other substring trapped variables

The table-element trapped variable types are allocated with

- one for each easily recognized creation point
- one representing all other table-element trapped variables

List elements, table assigned-values, and record fields are all variables that can appear as the intermediate results of expression evaluation. The type system has corresponding variable reference types to represent them. The list-element reference types are allocated with one for each list types. The table assigned-value reference types are allocated with one for each table type. The field reference types are allocated with one for each record field declaration.

Identifiers are variables and are reflected in the type system. The global variable reference types are allocated with

- one for each global variable (except those eliminated by invocation optimizations).
- one for each static variable

The local variable reference types are allocated with one for each local 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. The full set of types that can be used to express these semantics are presented in the syntax of the abstract type computations of the run-time implementation language; see Appendix A. In addition to the types of intermediate results, these types include

- set-element reference types
- table key reference types
- table default value reference types
- references to the fields within substring trapped variables that reference variables
- references to fields within table-element trapped variables that reference tables

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.

Set elements, table keys, and table default values do not appear as variable references in the results of expression evaluation. However, it is necessary to refer to them in order to describe the effects of certain Icon operations. For this reason, they are included in the type system. The set-element reference types are allocated with one for each set type. The table key reference types are allocated with one for each table type. The table default value reference types are allocated with one for each table type.

Substring trapped variable types contain references to the variable types being trapped and table-element trapped variable types contain references to the table types containing the element being trapped. These references are fields within these trapped variable types. There is one field reference type for each trapped variable type.

Procedure Calls and Co-Expression Activations

A type inferencing system for the full Icon language must handle procedures and co-expressions. As noted above, each procedure and each `create` expression is given its own type. This allows the type inferencing system to accurately determine which procedures are invoked and what co-expressions might be activated by a particular expression.

The standard semantics for procedures and co-expressions can be implemented using stacks of procedure activation frames, with one stack per co-expression. The first frame, on every stack

except that of the main co-expression, is a copy of the frame for the procedure that created the co-expression. The local variables in this frame are used for evaluating the code of the co-expression. The type inferencing system uses a trivial abstraction of these procedure frame stacks, while capturing the possible transfers of control induced by procedure calls and co-expression activations.

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. Because static variables may be altered in recursive calls, they must also 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 and co-expressions. 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. Edges are effectively added between all co-expressions and all activations, and between all activations. Information is propagated along an edge when type inferencing deduces that the corresponding procedure call or co-expression activation 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. On the first call to the procedure, static variables are also initialized to the null value. The initialization code for the standard semantics is similar to

```
initialize locals
if (first_call) {
    initialize statics
    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 and all static variables are set to the empty type. In some sense, the empty type represents an impossible execution path. Type inferencing sees paths in the flow graph from the start of the program to the body of a procedure that never pass through the initialization code. However, static variables have an empty type along this path and no operation on them is valid. Invalid operations contribute nothing to type information.

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

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

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

S = the number of static 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. 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 chapter 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 and static 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.

Two stores are associated with each co-expression. One is the merger of all stores entering the co-expression and the other is the merger of all stores leaving the co-expression. Execution can not only leave through an activation operator, it can also re-enter through the activation. The store entering the activation is a merger of the stores entering all co-expressions in which the activation can occur. Because a procedure containing an activation may be called from several co-expressions, it is necessary to keep track of those co-expressions. A set of co-expressions is associated with each procedure for this purpose. Each co-expression also contains a type for the values transmitted to it. The result type of an activation includes the result types for all co-expressions that might be activated and the types of all values that can be transmitted to a co-expression that the activation might be executed in.

When type inferencing encounters the invocation of a built-in operation, it performs abstract interpretation on the representation of the operation in the data base. It interprets the type-checking code to see what paths might be taken through the operation. The interpretation uses the abstract type computations and ignores the detailed C code when determining the side effects and result type of the operation. Because the code at this level of detail contains no loops, it is not necessary to save stores internal to operations. An operation is re-interpreted at each invocation. This allows type inferencing to produce good results for polymorphous operations. At this level, the code for an operation is simple enough that the cost of re-interpretation is not prohibitive. All side effects within these operations are treated as weak updates; the only strong updates recognized by type inferencing are the optimized assignments to named variables (see Chapter 6).

The abstract semantics of control structures are hard-coded within the type inferencing system. The system combines all the elements described in this chapter to perform the abstract interpretation. 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.

CHAPTER 8

Code Generation

This chapter describes the code generation process. The examples of generated code presented here are produced by the compiler, but some cosmetic changes have been made to enhance readability.

Code generation is done on one procedure at a time. An Icon procedure is, in general, translated into several C functions. There is always an *outer function* for the procedure. This is the function that is seen as implementing the procedure. In addition to the outer function, there may be several functions for success continuations that are used to implement generative expressions.

The outer function of a procedure must have features that support the semantics of an Icon call, just as a function implementing a run-time operation does. In general, a procedure must have a procedure block at run time. This procedure block references the outer function. All functions referenced through a procedure block must conform to the compiler system's standard calling conventions. However, invocation optimizations usually eliminate the need for procedure variables and their associated procedure blocks. When this happens, the calling conventions for the outer function can be tailored to the needs of the procedure.

As explained in Chapter 2, the standard calling convention requires four parameters: the number of arguments, a pointer to the beginning of an array of descriptors holding the arguments, a pointer to a result location, and a success continuation to use for suspension. The function itself is responsible for dereferencing and argument list adjustment. In a tailored calling convention for an outer function of a procedure, any dereferencing and argument list adjustment is done at the call site. This includes creating an Icon list for the end of a variable-sized argument list. The compiler produces code to do this that is optimized to the particular call. An example of an optimization is eliminating dereferencing when type inferencing determines that an argument

cannot be a variable reference.

The number of arguments is never needed in these tailored calling conventions because the number is fixed for the procedure. Arguments are still passed via a pointer to an array of descriptors, but if there are no arguments, no pointer is needed. If the procedure returns no value, no result location is needed. If the procedure does not suspend, no success continuation is needed.

In addition to providing a calling interface for the rest of the program, the outer function must provide local variables for use by the code generated for the procedure. These variables, along with several other items, are located in a procedure frame. An Icon procedure frame is implemented as a C structure embedded within the frame of its outer C function (that is, as a local struct definition). Code within the outer function can access the procedure frame directly. However, continuations must use a pointer to the frame. A global C variable, pfp, points to the frame of the currently executing procedure. For efficiency, continuations load this pointer into a local register variable. The frame for a main procedure might have the following declaration.

```
struct PF00_main {
   struct p_frame *old_pfp;
   dptr old_argp;
   dptr rslt;
   continuation succ_cont;
   struct {
      struct tend_desc *previous;
      int num;
      struct descrip d[5];
      } tend;
   };
```

with the definition

```
struct PF00_main frame;
```

in the procedure's outer function. A procedure frame always contains the following five items: a pointer to the frame of the caller, a pointer to the argument list of the caller, a pointer to the result location of this call, a pointer to the success continuation of this call, and an array of tended descriptors for this procedure. It may also contain C integer variables, C double variables, and string and cset buffers for use in converting values. If debugging is enabled, additional information is stored in the frame. The structure p_frame is a generic procedure frame containing a single tended descriptor. It is used to define the pointer old_pfp because the caller can be any procedure.

The argument pointer, result location, and success continuation of the call must be available to the success continuations of the procedure. A global C variable, argp, points the argument list for the current call. This current argument list pointer could have been put in the procedure frame, but it is desirable to have quick access to it. Quick access to the result location and the success continuation of the call is less important, so they are accessed indirectly through the procedure frame.

The array of descriptors is linked onto the chain used by the garbage collector to locate tended descriptors. These descriptors are used for Icon variables local to the procedure and for temporary variables that hold intermediate results. If the function is responsible for dereferencing and argument list adjustment (that is, if it does not have a tailored calling convention), the modified argument list is constructed in a section of these descriptors.

The final thing provided by the outer function is a *control environment* in which code generation starts. In particular, it provides the bounding environment for the body of the procedure and the implicit failure at the end of the procedure. The following C function is the tailored outer function for a procedure named p. The procedure has arguments and returns a result. However, it does not suspend, so it needs no success continuation.

```
static int P01_p(args, rslt)
dptr args;
dptr rslt;
{
    struct PF01_p frame;
    register int signal;
    int i;

    frame.old_pfp = pfp;
    pfp = (struct p_frame *)&frame;
    frame.old_argp = argp;
    frame.rslt = rslt;
    frame.succ_cont = NULL;

    for (i = 0; i < 3; ++i)
        frame.tend.d[i].dword = D_Null;
    argp = args;
    frame.tend.num = 3;
    frame.tend.previous = tend;
    tend = (struct tend_desc *)&frame.tend;

    translation of the body of procedure p

L10: /* bound */
```

```
L4: /* proc fail */
    tend = frame.tend.previous;
    pfp = frame.old_pfp;
    argp = frame.old_argp;
    return A_Resume;
L8: /* proc return */
    tend = frame.tend.previous;
    pfp = frame.old_pfp;
    argp = frame.old_argp;
    return A_Continue;
    }
```

The initialization code reflects the fact that this function has three tended descriptors to use for local variables and intermediate results. L10 is both the bounding label and the failure label for the body of the procedure. Code to handle procedure failure and return (except for setting the result value) is at the end of the outer function. As with bounding labels, the labels for these pieces of code have associated signals. If a procedure fail or return occurs in a success continuation, the continuation returns the corresponding signal which is propagated to the outer function where it is converted into a goto. The code for procedure failure is located after the body of the procedure, automatically implementing the implicit failure at the end of the procedure.

Translating Icon Expressions

Icon's goal-directed evaluation makes the implementation of control flow an important issue during code generation. Code for an expression is generated while walking the expression's syntax tree in forward execution order. During code generation there is always a *current failure action*. This action is either "branch to a label" or "return a signal". When the translation of a procedure starts, the failure action is to branch to the bounding label of the procedure body. The action is changed when generators are encountered or while control structures that use failure are

being translated.

The allocation of temporary variables to intermediate results is discussed in more detail later. However, some aspects of it will be addressed before presenting examples of generated code. The result location of a subexpression may be determined when the parent operation is encountered on the way down the syntax tree. This is usually a temporary variable, but does not have to be. If no location has been assigned by the time the code generator needs to use it, a temporary variable is allocated for it. This temporary variable is used in the code for the parent operation.

The code generation process is illustrated below with examples that use a number of control structures and operations. Code generation for other features of the language is similar.

Consider the process of translating the following Icon expression:

```
return if a = (1 | 2) then "yes" else "no"
```

When this expression is encountered, there is some current failure action, perhaps a branch to a bounding label. The return expression produces no value, so whether a result location has been assigned to it is of no consequence. If the argument of a return fails, the procedure fails. To handle this possibility, the current failure action is set to branch to the label for procedure failure before translating the argument (in this example, that action is not used). The code for the argument is then generated with its result location set to the result location of the procedure itself. Finally the result location is dereferenced and control is transferred to the procedure return label. The dereferencing function, deref, takes two arguments: a pointer to a source descriptor and a pointer to a destination descriptor.

```
code for the if expression
deref(rslt, rslt);
goto L7 /* proc return */;
```

The control clause of the if expression must be bounded. The code implementing the then clause must be generated following the bounding label for the control clause. A label must also

be set up for the else clause with a branch to this label used as the failure action for the control clause. Note that the result location of each branch is the result location of the if expression which is in turn the result location of the procedure. Because neither branch of the if expression contains operations that suspend, the two control paths can be brought together with branch to a label.

```
    code for control clause
L4: /* bound */
    rslt->vword.sptr = "yes";
    rslt->dword = 3;
    goto L6 /* end if */;
L5: /* else */
    rslt->vword.sptr = "no";
    rslt->dword = 2;
L6: /* end if */
```

Using a branch and a label to bring together the two control paths of the if expression is an optimization. If the then or the else clauses contain operations that suspend, the general continuation model must be used. In this model, the code following the if expression is put in a success continuation, which is then called at the end of both the code for the then clause and the code for the else clause.

Next consider the translation of the control clause. The numeric comparison operator takes two operands. In this translation, the standard calling conventions are used for the library routine implementing the operator. Therefore, the operands must be in an array of descriptors. This array is allocated as a sub-array of the tended descriptors for the procedure. In this example, tended location 0 is occupied by the local variable, a. Tended locations 1 and 2 are free to be allocated as the arguments to the comparison operator. The code for the first operand simply builds a variable reference.

```
frame.tend.d[1].dword = D_Var;
frame.tend.d[1].vword.descptr = &frame.tend.d[0] /* a */;
```

However, the second operand is alternation. This is a generator and requires a success continuation. In this example, the continuation is given the name P02_main (the Icon expression is part of the main procedure). The continuation contains the invocation of the run-time function implementing the comparison operator and the end of the bounded expression for the control clause of the if. The function O0o_numeq implements the comparison operator. The if expression discards the operator's result. This is accomplished by using the variable trashcan as the result location for the call. The compiler knows that this operation does not suspend, so it passes a null continuation to the function. The end of the bounded expression consists of a transfer of control to the bounding label. This is accomplished by returning a signal. The continuation is

```
static int P02_main()
   {
   register struct PF00_main *rpfp;

   rpfp  = (struct PF00_main *)pfp;
   switch (O0o_numeq(2, &(rpfp->tend.d[1]), &trashcan, (continuation)NULL))
      {
      case A_Continue:
         break;
      case A_Resume:
         return A_Resume;
      }
   return 4; /* bound */
   }
```

Each alternative of the alternation must compute the value of its subexpression and call the

success continuation. The failure action for the first alternative is to branch to the second alternative. The failure action of the second alternative is the failure action of the entire alternation expression. In this example, the failure action is to branch to the else label of the if expression. In each alterative, a bounding signal from the continuation must be converted into a branch to the bounding label. Note that this bounding signal indicates that the control expression succeeded.

```
        frame.tend.d[2].dword = D_Integer;
        frame.tend.d[2].vword.integr = 1;
        switch (P02_main()) {
           case A_Resume:
              goto L2 /* alt */;
           case 4 /* bound */:
              goto L4 /* bound */;
           }
     L2: /* alt */
        frame.tend.d[2].dword = D_Integer;
        frame.tend.d[2].vword.integr = 2;
        switch (P02_main()) {
           case A_Resume:
              goto L5 /* else */;
           case 4 /* bound */:
              goto L4 /* bound */;
           }
```

The code for the entire return expression is obtained by putting together all the pieces. The result is the following code (the code for P02_main is not repeated).

```
    frame.tend.d[1].dword = D_Var;
    frame.tend.d[1].vword.descptr = &frame.tend.d[0] /* a */;
    frame.tend.d[2].dword = D_Integer;
    frame.tend.d[2].vword.integr = 1;
    switch (P02_main()) {
       case A_Resume:
          goto L2 /* alt */;
       case 4 /* bound */:
          goto L4 /* bound */;
       }
L2: /* alt */
    frame.tend.d[2].dword = D_Integer;
    frame.tend.d[2].vword.integr = 2;
    switch (P02_main()) {
       case A_Resume:
          goto L5 /* else */;
       case 4 /* bound */:
          goto L4 /* bound */;
       }
L4: /* bound */
    rslt->vword.sptr = "yes";
    rslt->dword = 3;
    goto L6 /* end if */;
L5: /* else */
    rslt->vword.sptr = "no";
    rslt->dword = 2;
```

```
L6: /* end if */
    deref(rslt, rslt);
    goto L7 /* proc return */;
```

Signal Handling

In order to produce signal handling code, the code generator must know what signals may be returned from a call. These signals may be either directly produced by the operation (or procedure) being called or they may originate from a success continuation. Note that either the operation or the continuation may be missing from a call, but not both. The signals produced directly by an operation are A_Resume, A_Continue, and A_FallThru (this last signal is only used internally within in-line code).

The signals produced by a success continuation belong to one of three categories: A_Resume, signals corresponding to labels within the procedure the continuation belongs to, and signals corresponding to labels in procedures farther down in the call chain. The last category only occurs when the procedure suspends. The success continuation for the procedure call may return a signal belonging to the calling procedure. This is demonstrated in the following example (the generated code has been "cleaned-up" a little to make it easier to follow). The Icon program being translated is

```
procedure main()
   write(p())
end

procedure p()
   suspend 1 to 10
end
```

The generative procedure p is called in a bounded context. The code generated for the call is

```
        switch (P01_p(&frame.tend.d[0], P05_main)) {
           case 7 /* bound */:
              goto L7 /* bound */;
           case A_Resume:
              goto L7 /* bound */;
           }
     L7: /* bound */
```

This call uses the following success continuation. The continuation writes the result of the call to p then signals the end of the bounded expression.

```
        static int P05_main()
           {
           register struct PF00_main *rpfp;

           rpfp  = (struct PF00_main *)pfp;
           F0c_write(1, &rpfp->tend.d[0], &trashcan, (continuation)NULL);
           return 7; /* bound */
           }
```

The to operator in procedure p needs a success continuation that implements procedure suspension. Suspension is implemented by switching to the old procedure frame pointer and old argument pointer, then calling the success continuation for the call to p. The success continuation is accessed with the expression *rpfp->succ_cont. In this example, the continuation will only be the function P05_main. The suspend must check the signal returned by the procedure call's success continuation. However, the code generator does not try to determine exactly what signals might be returned by a continuation belonging to another procedure. Such a continuation

may return an A_Resume signal or a signal belonging to some procedure farther down in the call chain. In this example, bounding signal 7 will be returned and it belongs to main.

If the call's success continuation returns A_Resume, the procedure frame pointer and argument pointer must be restored, and the current failure action must be executed. In this case, that action is to return an A_Resume signal to the to operator. If the call's success continuation returns any other signal, that signal must be propagated back through the procedure call. The following function is the success continuation for the to operator.

```
static int P03_p()
   {
   register int signal;
   register struct PF01_p *rpfp;

   rpfp  = (struct PF01_p *)pfp;
   deref(rpfp->rslt, rpfp->rslt);
   pfp = rpfp->old_pfp;
   argp = rpfp->old_argp;
   signal = (*rpfp->succ_cont)();
   if (signal != A_Resume) {
      return signal;
      }
   pfp = (struct p_frame *)rpfp;
   argp = NULL;
   return A_Resume;
   }
```

The following code implements the call to the to operator. The signal handling code associated with the call must pass along any signal from the procedure call's success continuation. These

signals are recognized by the fact that the procedure frame for the calling procedure is still in effect. At this point, the signal is propagated out of the procedure p. Because the procedure frame is about to be removed from the C stack, the descriptors it contains must be removed from the tended list.

```
        frame.tend.d[0].dword = D_Integer;
        frame.tend.d[0].vword.integr = 1;
        frame.tend.d[1].dword = D_Integer;
        frame.tend.d[1].vword.integr = 10;
        signal = O0k_to(2, &frame.tend.d[0], rslt, P03_p);
        if (pfp != (struct p_frame *)&frame) {
           tend = frame.tend.previous;
           return signal;
           }
        switch (signal) {
           case A_Resume:
              goto L2 /* bound */;
           }
    L2: /* bound */
```

So far, this discussion has not addressed the question of how the code generator determines what signals might be returned from a call. Because code is generated in execution order, a call involving a success continuation is generated before the code in the continuation is generated. This makes it difficult to know what signals might originate from the success continuation. This problem exists for direct calls to a success continuation and for calls to an operation that uses a success continuation.

The problem is solved by doing code generation in two parts. The first part produces incomplete signal handling code. At this time, code to handle the signals produced directly by an

operation is generated. The second part of code generation is a fix-up pass that completes the signal handling code by determining what signals might be produced by success continuations.

The code generator constructs a call graph of the continuations for a procedure. Some of these calls are indirect calls to a continuation through an operation. However, the only effect of an operation on signals returned by a continuation is to intercept A_Resume signals. All other signals are just passed along. This is true even if the operation is a procedure. This call graph of continuations does not contain the procedure call graph nor does it contain continuations from other procedures.

Forward execution order imposes a partial order on continuations. A continuation only calls continuations strictly greater in forward execution order than itself. Therefore the continuation call graph is a DAG.

The fix-up pass is done with a bottom-up walk of the continuation call DAG. This pass determines what signals are returned by each continuation in the DAG. While processing a continuation, the fix-up pass examines each continuation call in that continuation. At the point it processes a call, it has determined what signals might be returned by the called continuation. It uses this information to complete the signal handling code associated with the call and to determine what signals might be passed along to continuations higher up the DAG. If a continuation contains code for a suspend, the fix-up pass notes that the continuation may return a *foreign* signal belonging to another procedure call. As explained above, foreign signals are handled by special code that checks the procedure frame pointer.

Temporary Variable Allocation

The code generator uses the liveness information for an intermediate value when allocating a temporary variable to hold the value. As explained in Chapter 4, this information consists of the furthest program point, represented as a node in the syntax tree, through which the intermediate value must be retained. When a temporary variable is allocated to a value, that variable is placed on a *deallocation list* associated with the node beyond which its value is not needed. When the

code generator passes a node, all the temporary variables on the node's deallocation list are deallocated.

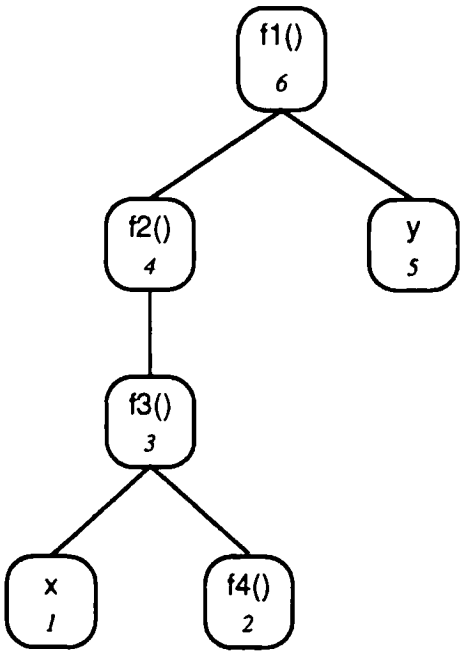
The code generator maintains a *status array* for temporary variables while it is processing a procedure. The array contains one element per temporary variable. This array is expandable, allowing a procedure to use an arbitrary number of temporary variables. In a simple allocation scheme, the status of a temporary variable is either *free* or *in-use*. The entry for a temporary variable is initially marked free, it is marked in-use when the variable is allocated, and it is marked free again when the variable is deallocated.

The simple scheme works well when temporary variables are allocated independently. It does not work well when arrays of contiguous temporary variables are allocated. This occurs when temporary variables are allocated to the arguments of a procedure invocation or any invocation conforming to the standard calling conventions; under these circumstances, the argument list is implemented as an array. All of the contiguous temporary variables must be reserved before the first one is used, even though many operations may be performed before the last one is needed. Rather than mark a temporary variable in-use before it actually is, the compiler uses the program point where the temporary variable will be used to mark the temporary variable's entry in the status array as *reserved.* A contiguous array of temporary variables are marked reserved at the same time, with each having a different reservation point. A reserved temporary variable may be allocated to other intermediate values as long as it will be deallocated before the reservation point. In this scheme, an entry in a deallocation list must include the previous status of the temporary variable as it might be a reserved status.

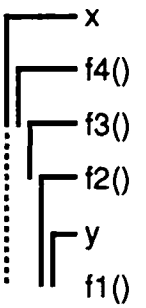
The compiler allocates a contiguous subarray of temporary variables for the arguments of an invocation when it encounters the invocation on the way down the syntax tree during its tree walk. It uses a first-fit algorithm to find a large enough subarray that does not have a conflicting allocation. Consider the problem of allocating temporary variables to the expression

```
f1(f2(f3(x, f4())), y)
```

where f1 can fail and f4 is a generator. The syntax tree for this expression is shown below. Note that invocation nodes show the operation as part of the node label and not as the first operand to general invocation. This reflects the direct invocation optimization that is usually performed on invocations. Each node in the graph is given a numeric label. These labels increase in value in forward execution order.



The following figure shows the operations in forward execution order with lines on the left side of the diagram showing the lifetime of intermediate values. This represents the output of the liveness analysis phase of the compiler. Because f4 can be resumed by f1, the value of the expression x has a lifetime that extends to the invocation of f1. The extended portion of the lifetime is indicated with a dotted line.



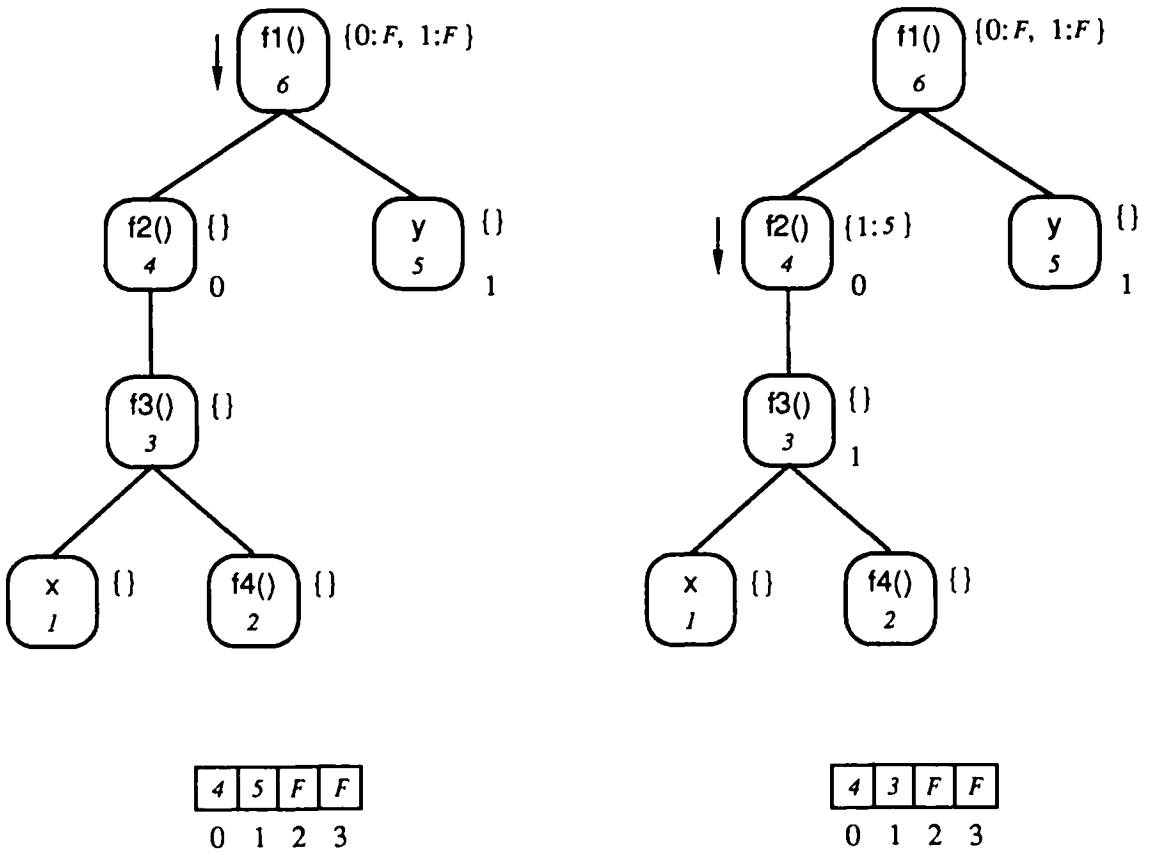
The following series of diagrams illustrate the process of allocating intermediate values. Each diagram includes an annotated syntax tree and a status array for temporary variables. An arrow in the tree shows the current location of the tree walk. A deallocation list is located near the upper right of each node. An element in the list consists of a temporary variable number and the status with which to restore the variable's entry in the status array. If a temporary variable has been allocated to an intermediate value, the variable's number appears near the lower right of the corresponding node.

The status array is shown with four elements. The elements are initialized to *F* which indicates that the temporary variables are free. A reserved temporary variable is indicated by placing the node number of the reservation point in the corresponding element. When a temporary variable is actually in use, the corresponding element is set to *I*.

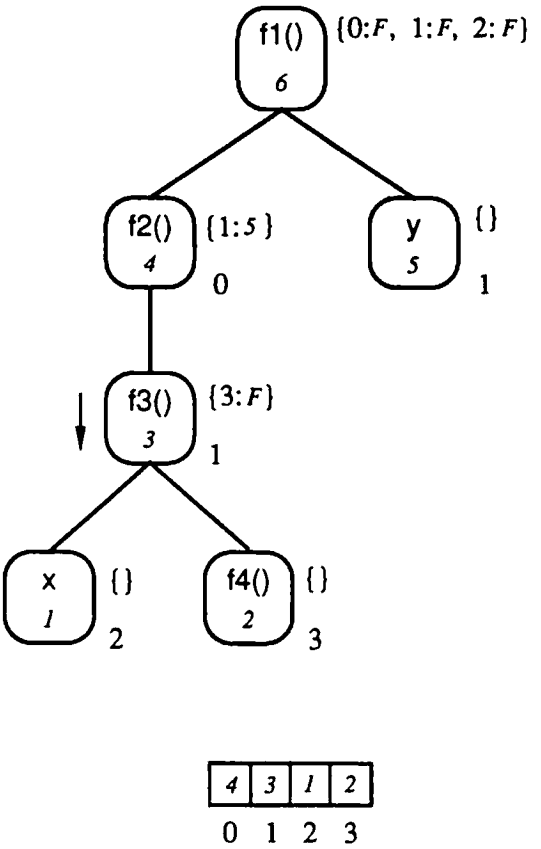
Temporary variables are reserved while walking down the syntax tree. The tree illustrated below on the left shows the state of allocation after temporary variables have been allocated for the operands of f1. Two contiguous variables are needed. All variables are free, so the first-fit algorithm allocates variables 0 and 1. The status array is updated to indicate that these variables are reserved for nodes *4* and *5* respectively, and the nodes are annotated with these variable numbers. The lifetime information in the previous figure indicates that these variables should be deallocated after f1 is executed, so the deallocation array for node *6* is updated.

The next step is the allocation of a temporary variable to the operand of f2. The intermediate value has a lifetime extending from node *3* to node *4*. This conflicts with the allocation of

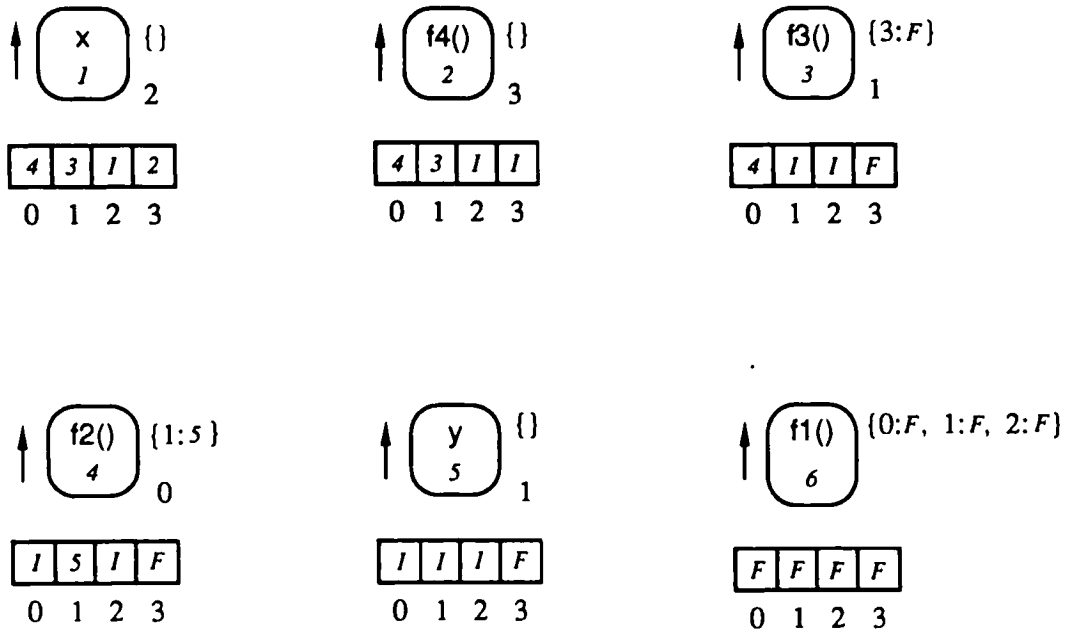
variable 0, but not the allocation of variable 1. Therefore, variable 1 is allocated to node *3* and the deallocation list for node *4* is updated. This is illustrated in the tree on the right:



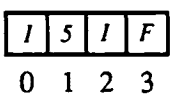
The final allocation requires a contiguous pair of variables for nodes *1* and *2*. The value from node *1* has a lifetime that extends to node *6*, and the value from node *2* has a lifetime that extends to node *3*. The current allocations for variables 0 and 1 conflict with the lifetime of the intermediate value of node *1*, so the variables 2 and 3 are used in this allocation. This is illustrated in the tree:



The remaining actions of the allocator in this example mark temporary variables in-use when the code generator uses them and restore previous allocated statuses when temporary variables are deallocated. This is done in the six steps illustrated in the following diagram. The annotations on the graph do not change. Only the node of interest is shown for each step. These steps are performed in node-number order.

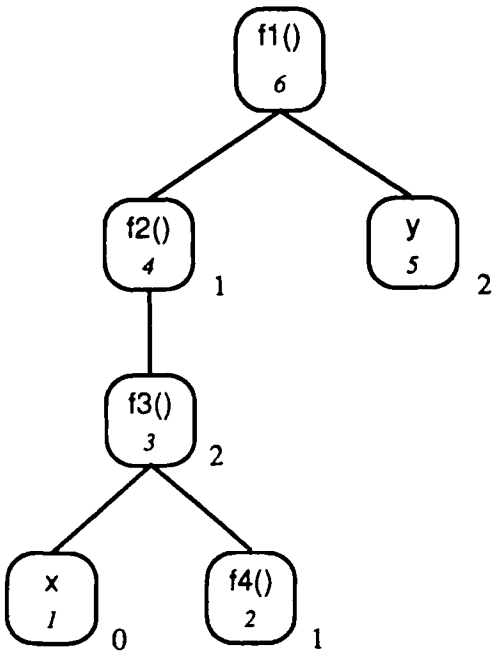


In general, the tree walk will alternate up and down the syntax tree. For example, if node *5* had children, the allocation status after the deallocation associated with node *4*,



is used to allocate temporary variables to those children. If this requires more than four temporary variables, the status array is extended with elements initialized to *F*.

This allocation algorithm is not guaranteed to produce an allocation that uses a minimal number of temporary variables. Indeed, a smaller allocation for the previous example is illustrated in the tree:



While the non-optimality of this algorithm is unlikely to have a measurable effect on the performance of any practical program, the problem of finding an efficient optimal solution is of theoretical interest. Classical results in the area of register allocation do not apply. It is possible to allocate a minimum number of registers from expression trees for conventional languages in polynomial time [16]. The algorithm to do this depends on the fact that registers (temporary variables) are dead as soon as the value they contain is used. This is not true for Icon temporary variables.

The result of Prabhala and Sethi stating that register allocation is NP-complete even in the presence of an infinite supply of registers also does not apply [38]. Their complexity result derives from performing register allocation in the presence of common subexpression elimination (that is, from performing register allocation on expression DAGS rather than trees) on a 2-address-instruction machine with optimality measured as the minimum number of instructions needed to implement the program. Goal-directed evaluation imposes more structure on lifetimes than common subexpression elimination, the machine model used here is the C language, and optimality is being measure as the minimum number of temporary variables needed.

The Icon temporary variable allocation problem is different from the Prolog variable allocation problem. Prolog uses explicit variables whose lifetimes can have arbitrary overlaps even in the absence of goal-directed evaluation. The Prolog allocation problem is equivalent to the classical graph coloring problem which is NP-complete [16, 39].

If the allocation of a subarray of temporary variables is delayed until the first one is actually needed in the generated code, an optimum allocation results for the preceding example. It is not obvious whether this is true for the general case of expression trees employing goal-directed evaluation. This problem is left for future work.

In addition to holding intermediate values, temporary variables are used as local tended variables within in-line code. This affects the pattern of allocations, but not the underlying allocation technique.

CHAPTER 9

Control Flow Optimizations

Naive Code Generation

Naive code generation does not consider the effects and needs of the immediately surrounding program. The result is often a poor use of the target language. Even sophisticated code generation schemes that consider the effects of relatively large pieces of the program still produce poor code at the boundaries between the pieces. This problem is typically solved by adding a *peephole optimizer* to the compiler to improve the generated code [16,40-42]. A peephole optimizer looks at several instructions that are adjacent (in terms of execution) and tries to replace the instructions by better, usually fewer, instructions. It typically analyzes a variety of properties of the instructions such as addressing modes and control flow.

The Icon compiler has a peephole optimizer that works on the internal form of the generated C code and deals only with control flow. The previous examples of generated code contain a number of instances of code where control flow can be handled much better. For example, it is possible to entirely eliminate the following code fragment generated for the example explaining procedure suspension.

```
        switch (signal) {
           case A_Resume:
              goto L2 /* bound */;
           }
     L2: /* bound */
```

This code is produced because the code generator does not take into account the fact that the bounding label happens to immediately follow the test.

Success Continuations

For the C code in the preceding example, it is quite possible that a C compiler would produce machine code that its own peephole optimizer could eliminate. However, it is unlikely that a C compiler would optimize naively generated success continuations. An earlier example of code generation produced the continuation:

```
static int P02_main()
   {
   register struct PF00_main *rpfp;

   rpfp  = (struct PF00_main *)pfp;
   switch (O0o_numeq(2, &(rpfp->tend.d[1]), &trashcan, (continuation)NULL))
      {
      case A_Continue:
         break;
      case A_Resume:
         return A_Resume;
      }
   return 4; /* bound */
   }
```

If the statement

```
return 4; /* bound */
```

is brought into the switch statement, replacing the break, then P02_main consists of a simple operation call (a C call with associated associated signal handling code). This operation call is

```
switch (O0o_numeq(2, &(rpfp->tend.d[1]), &trashcan, (continuation)NULL))
   {
   case A_Continue:
      return 4; /* bound */
   case A_Resume:
      return A_Resume;
   }
```

P02_main is called directly in two places in the following code.

```
   frame.tend.d[2].dword = D_Integer;
   frame.tend.d[2].vword.integr = 1;
   switch (P02_main()) {
      case A_Resume:
         goto L2 /* alt */;
      case 4 /* bound */:
         goto L4 /* bound */;
      }
L2: /* alt */
   frame.tend.d[2].dword = D_Integer;
   frame.tend.d[2].vword.integr = 2;
   switch (P02_main()) {
      case A_Resume:
         goto L5 /* else */;
      case 4 /* bound */:
         goto L4 /* bound */;
      }
L4: /* bound */
```

A direct call to a trivial function can reasonably be replaced by the body of that function. When this is done for a continuation, it is necessary to *compose* the signal handling code of the body of a continuation with that of the call. This is accomplished by replacing each return statement in the body with the action in the call corresponding to the signal returned. The following table illustrates the signal handling composition for the first call in the code. The resulting code checks the signal from O0o_numeq and performs the final action.

signal from O0o_numeq	signal from P02_main	final action
A_Continue	4	goto L4;
A_Resume	A_Resume	goto L2;

The result of in-lining P02_main is

```
    frame.tend.d[2].dword = D_Integer;
    frame.tend.d[2].vword.integr = 1;
    switch (O0o_numeq(2, &frame.tend.d[1], &trashcan, (continuation)NULL))
      {
      case A_Continue:
        goto L4 /* bound */;
      case A_Resume:
        goto L2 /* alt */;
      }
L2: /* alt */
    frame.tend.d[2].dword = D_Integer;
    frame.tend.d[2].vword.integr = 2;
```

```
    switch (O0o_numeq(2, &frame.tend.d[1], &trashcan, (continuation)NULL))
       {
       case A_Continue:
          goto L4 /* bound */;
       case A_Resume:
          goto L5 /* else */;
       }
L4: /* bound */
```

With a little more manipulation, the switch statements can be converted into if statements and the label L2 can be eliminated:

```
    frame.tend.d[2].dword = D_Integer;
    frame.tend.d[2].vword.integr = 1;
    if (O0o_numeq(2, &frame.tend.d[1], &trashcan, (continuation)NULL) ==
       A_Continue) goto L4 /* bound */;
    frame.tend.d[2].dword = D_Integer;
    frame.tend.d[2].vword.integr = 2;
    if (O0o_numeq(2, &frame.tend.d[1], &trashcan, (continuation)NULL) ==
       A_Resume) goto L5 /* else */;
L4: /* bound */
```

The Icon compiler's peephole optimizer recognizes two kinds of trivial continuations. The kind illustrated in the previous example consists of a single call with associated signal handling. The other kind simply consists of a single return-signal statement. As in the above example, continuations do not usually meet this definition of triviality until control flow optimizations are performed on them. For this reason, the Icon compiler's peephole optimizer must perform some optimizations that could otherwise be left to the C compiler.

Iconc's Peephole Optimizer

The peephole optimizer performs the following optimizations:

- elimination of unreachable code
- elimination of gotos immediately preceding their destinations
- collapse of branch chains
- deletion of unused labels
- collapse of trivial call chains (that is, in-lining trivial continuations)
- deletion of unused continuations
- simplification of signal checking

Unreachable code follows a goto or a return, and it continues to the first referenced label or to the end of the function. This optimization may eliminate code that returns signals, thereby reducing the number of signals that must be handled by a continuation call. This provides another reason for performing this traditional optimization in the Icon compiler rather than letting the C compiler do it. This code is eliminated when the fix-up pass for signal handling is being performed. gotos immediately preceding their labels also are eliminated at this time.

Unused labels usually are eliminated when the code is written out, but they may be deleted as part of a segment of unreachable code. Unused continuations are simply not written out.

A branch chain is formed when the destination of a goto is another goto or a return. A break in a switch statement is treated as a goto. There may be several gotos in a chain. Each goto is replaced by the goto or return at the end of the chain. This may leave some labels unreferenced and may leave some of the intermediate gotos unreachable. Branch chains are collapsed during the fix-up pass.

Inter-function optimization is not traditionally considered a peephole optimization. This is because human beings seldom write trivial functions and most code generators do not produce continuations. The Icon compiler, however, uses calls to success continuations as freely as it

uses gotos. Therefore collapsing trivial call chains is as important as collapsing branch chains.

There are two kinds of calls to trivial continuations: direct calls and indirect calls through an operation. A direct call always can be replaced by the body of the continuation using signal handling code that is a composition of that in the continuation and that of the call. If the continuation consists of just a return statement, this means that the call is replaced by the action associated with the returned signal: either another return statement or a goto statement. For continuations consisting of a call, the composition is more complicated, as demonstrated by the example given earlier in this chapter.

In the case of an indirect call through an operation, the continuation cannot be placed in line. However, there is an optimization that can be applied. Under some circumstances, the compiler produces a continuation that simply calls another continuation. For example, this occurs when compiling the Icon expression

```
every write(!x | "end")
```

The compiler allocates a continuation for the alternation, then compiles the expression !x. The element generation operator suspends, so the compiler allocates a continuation for it and code generation proceeds in this continuation. However, the end of the first alternative has been reached so the only code for this continuation is a call to the continuation for the alternation. The continuation for the alternation contains the code for the invocation of write and for the end of the every control structure. The code for the first alternative is

```
frame.tend.d[2].dword = D_Var;
frame.tend.d[2].vword.descptr = &frame.tend.d[0] /* x */;
```

```
    switch (O0e_bang(1, &frame.tend.d[2], &frame.tend.d[1], P02_main)) {
        case A_Resume:
            goto L1 /* alt */;
        }
L1: /* alt */
```

The code for the two continuations are

```
static int P02_main()
    {
    switch (P03_main()) {
        case A_Resume:
            return A_Resume;
        }
    }

static int P03_main()
    {
    register struct PF00_main *rpfp;

    rpfp  = (struct PF00_main *)pfp;
    F0c_write(1, &rpfp->tend.d[1], &trashcan, (continuation)NULL);
    return A_Resume;
    }
```

The call to O0e_bang can be optimized by passing the continuation P03_main in place of P02_main.

The optimizations that collapse trivial call chains are performed during the fix-up pass for signal handling.

The final peephole optimization involves simplifying the signal handling code associated with a call. In general, signals are handled with a switch statement containing a case clause for each signal. The C compiler does not know that these signals are the only values that are ever tested by the switch statement, nor is the C compiler likely to notice that some cases simply pass along to the next function down in the call chain the signal that was received. The Icon compiler can use this information to optimize the signal handling beyond the level to which the C compiler is able to optimize it. The optimizer may replace the general form of the switch statement with a switch statement utilizing a default clause or with an if statement. In some cases, the optimizer completely eliminates signal checking. This optimization is done when the code is written.

CHAPTER 10

Optimizing Invocations

Several optimizations apply to the invocation of procedures and built-in operations. These include optimizations resulting from the application of information from type inferencing, optimizations resulting from the application of lifetime information to passing parameters and returning results, and optimizations involving the generation of in-line code. There are interactions between the optimizations in these three categories.

A primary motivation in developing the Icon compiler was to explore the optimizations that are possible using information from type inferencing. These optimizations involve eliminating type checking and type conversions where type inferencing indicates that they are not needed. Dereferencing is not normally viewed as a type conversion, because variable references are not first-class values in Icon. However, variable references occur as intermediate values and do appear in the type system used by the Icon compiler. Therefore, from the perspective of the compiler, dereferencing is a type conversion.

When a procedure or built-in operation is implemented as a C function conforming to the standard calling conventions of the compiler system, that function is responsible for performing any type checking and type conversions needed by the procedure or operation. For this reason, the checking and conversions can only be eliminated from tailored implementations.

Invocation of Procedures

As explained earlier, a procedure has one implementation: either a standard implementation or a tailored implementation. If the compiler decides to produce a tailored implementation, the caller of the procedure is responsible for dereferencing. When type inferencing determines that an operand is not a variable reference, no dereferencing code is generated. Suppose p is a procedure that takes one argument and always fails. If P01_p is the tailored C function

implementing p, then it takes one argument: a pointer to a descriptor containing the dereferenced Icon argument. Without using type information, the call p(3) translates into

```
frame.tend.d[0].dword = D_Integer;
frame.tend.d[0].vword.integr = 3;
deref(&frame.tend.d[0], &frame.tend.d[0]);
P01_p(&frame.tend.d[0]);
```

With the use of type information, the call to deref is eliminated:

```
frame.tend.d[0].dword = D_Integer;
frame.tend.d[0].vword.integr = 3;
P01_p(&frame.tend.d[0]);
```

Invocation and In-lining of Built-in Operations

Icon's built-in operations present more opportunities for these optimizations than procedures, because they can contain type checking and conversions beyond dereferencing. Built-in operations are treated differently than procedures. Except for keywords, there is always a C function in the run-time library that implements the operation using the standard calling conventions. In addition, the compiler can create several tailored in-line versions of an operation from the information in the data base.

It is important to keep in mind that there are two levels of in-lining. An in-line version of an operation always involves the type checking and conversions of the operation (although they may be optimized away during the tailoring process). However, detailed code is placed in-line only if it is specified with an inline statement in the run-time system. If the detailed code is specified with a body statement, the "in-line" code is a function call to a routine in the run-time library. The difference can be seen by comparing the code produced by compiling the expression ~x to that produced by compiling the expression /x. The definition in the run-time

implementation language of cset complement is

```
operator{1} ~ compl(x)
   if !cnv:tmp_cset(x) then
      runerr(104, x)
   abstract {
      return cset
      }
   body {
      ...
      }
end
```

The conversion to tmp_cset is a conversion to a cset that does not use space in the block region. Instead the cset is constructed in a temporary local buffer. The data base entry for the operation indicates that the argument must be dereferenced. The entry has a C translation of the type conversion code with a call to the support routine, cnv_tcset, to do the actual conversion. cnv_tcset takes three arguments: a buffer, a source descriptor, and a destination descriptor. The entry in the data base has a call to the function O160_compl in place of the body statement. This function takes as arguments the argument and the result location of the operation. The code generator ignores the abstract clause. The in-line code for ~x is

```
frame.tend.d[3].dword = D_Var;
frame.tend.d[3].vword.descptr = &frame.tend.d[0] /* x */;
deref(&frame.tend.d[3], &frame.tend.d[3]);
```

```
    if (cnv_tcset(&(frame.cbuf[0]), &(frame.tend.d[3]), &(frame.tend.d[3])))
       goto L1 /* then: compl */;
    err_msg(104, &(frame.tend.d[3]));
L1: /* then: compl */
    O160_compl(&(frame.tend.d[3]) , &frame.tend.d[2]);
```

The following is the definition of the '/' operator. Note that both undereferenced and dereferenced versions of the argument are used.

```
operator{0,1} / null(underef x -> dx)
   abstract {
      return type(x)
      }
   if is:null(dx) then
      inline {
         return x;
         }
   else
      inline {
         fail;
      }
end
```

In this operation, all detailed code is specified with inline statements. The generated code for /x follows. Note that the order of the then and else clauses is reversed to simplify the test. L3 is the failure label of the expression. The return is implemented as an assignment to the result location, frame.tend.d[2], with execution falling off the end of the in-line code.

```
    frame.tend.d[3].dword = D_Var;
    frame.tend.d[3].vword.descptr = &frame.tend.d[0] /* x */;
    deref(&frame.tend.d[3], &frame.tend.d[4]);
    if (frame.tend.d[4].dword == D_Null)
       goto L2 /* then: null */;
    goto L3 /* bound */;
L2: /* then: null */
    frame.tend.d[2] = frame.tend.d[3];
```

If type inferencing determines a unique type for x in each of these expressions, the type checking is eliminated from the code. Suppose type inferencing determines that x can only be of type cset in the expression

```
a := ~x
```

If parameter passing and assignment optimizations (these are explained below) are combined with the elimination of type checking, the resulting code is

```
O160_compl(&(frame.tend.d[0] /* x */), &frame.tend.d[1] /* a */);
```

The form of this translated code meets the goals of the compiler design for the invocation of a complicated operation: a simple call to a type-specific C function with minimum parameter passing.

The implementation language for run-time operations requires that type conversions be specified in the control clause of an if statement. However, some conversions, such as converting a string to a cset, are guaranteed to succeed. If the code generator recognizes one of these conversions, it eliminates the if statement. The only code generated is the conversion and the code to be executed when the conversion succeeds. Suppose type inferencing determines that x in the preceding example can only be a string. Then the generated code for the example is

```
frame.tend.d[2] = frame.tend.d[0] /* x */;
cnv_tcset(&(frame.cbuf[0]), &(frame.tend.d[2]), &(frame.tend.d[2]));
O160_compl(&(frame.tend.d[2]) , &frame.tend.d[1] /* a */);
```

Heuristic for Deciding to In-line

The in-line code for the operators shown so far in the section is relatively small. However, the untailored in-line code for operations like the element generation operator, !, is large. If tailoring the code does not produce a large reduction in size, it is better to generate a call to the C function in the run-time library that uses the standard calling conventions. A heuristic is needed for deciding when to use in-line code and when to call the standard C function.

A simple heuristic is to use in-line code only when all type checking and conversions can be eliminated. However, this precludes the generation of in-lining code in some important situations. The operator / is used to direct control flow. It should always be used with an operand whose type can vary at run time, and the generated code should always be in-lined. Consider the Icon expression

```
if /x then x := ""
```

The compiler applies parameter-passing optimizations to the sub-expression /x. It also eliminates the return value of the operator, because the value is discarded. An implementation convention for operations allows the compiler to discard the expression that computes the return value. The convention requires that a return expression of an operation not contain user-visible side effects (storage allocation is an exception to the rule; it is visible, but the language makes no guarantees as to when it will occur). The code for /x is reduced to a simple type check. The code generated for the if expression is

```
    if ((frame.tend.d[0] /* x */).dword == D_Null)
       goto L2 /* bound */;
    goto L3 /* bound */;
L2: /* bound */
    frame.tend.d[0] /* x */.vword.sptr = "";
    frame.tend.d[0] /* x */.dword = 0;
L3: /* bound */
```

To accommodate expressions like those in the preceding example, the heuristic used in the compiler is to produce tailored in-line code when that code contains no more than one type check. Only conversions retaining their if statements are counted as a type checks. This simple heuristic produces reasonable code. Future work includes examining more sophisticated heuristics.

In-lining Success Continuations

Suspension in in-line code provides further opportunity for optimization. In general, suspension is implemented as a call to a success continuation. However, if there is only one call to the continuation, it is better not to put the code in a continuation. The code should be generated at the site of the suspension. Consider the expression

```
every p(1 to 10)
```

The implementation of the to operator is

```
operator{*} ... to(from, to)
   /*
    * arguments must be integers.
    */
   if !cnv:C_integer(from) then
      runerr(101, from)
   if !cnv:C_integer(to) then
      runerr(101, to)

   abstract {
      return integer
      }

   inline {
      for ( ; from <= to; ++from) {
         suspend C_integer from;
         }
      fail;
      }
end
```

The arguments are known to be integers, so the tailored version consists of just the code in the inline statement. The for statement is converted to gotos and conditional gotos, so the control flow optimizer can handle it (this conversion is done by rtt before putting the code in the data base). The suspend is translated into code to set the result value and a failure label used for the code of the rest of the bounded expression. This code is generated before the label and consists of a call to the procedure p and the failure introduced by the every expression. The generated code follows. The failure for the every expression is translated into goto L4, where L4 is the failure label introduced by the suspend. The control flow optimizer removes both the goto and

the label. They are retained here to elucidate the code generation process.

```
    frame.tend.d[1].dword = D_Integer;
    frame.tend.d[1].vword.integr = 1;
    frame.tend.d[2].dword = D_Integer;
    frame.tend.d[2].vword.integr = 10;

L1: /* within: to */
    if (!(frame.tend.d[1].vword.integr <= frame.tend.d[2].vword.integr) )
        goto L2 /* bound */;
    frame.tend.d[0].vword.integr = frame.tend.d[1].vword.integr;
    frame.tend.d[0].dword = D_Integer;
    P01_p(&frame.tend.d[0]);
    goto L4 /* end suspend: to */;
L4: /* end suspend: to */
    ++frame.tend.d[1].vword.integr;
    goto L1 /* within: to */;
L2: /* bound */
```

This is an example of a generator within an every expression being converted into an in-line loop. Except for the fact that descriptors are being used instead of C integers, this is nearly as good as the C code

```
for (i = 1; i <= 10; ++i)
    p(i);
```

Parameter Passing Optimizations

As mentioned above, parameter-passing optimizations are used to improved the generated code. These optimizations involve eliminating unneeded argument computations and eliminating unnecessary copying. These optimizations are applied to tailored in-line code. They must take into account how a parameter is used and whether the corresponding argument value has an extended lifetime.

In some situations, a parameter is not used in the tailored code. There are two common circumstances in which this happens. One is for the first operand of conjunction. The other occurs with a polymorphous operation that has a type-specific optional parameter. If a different type is being operated on, the optional parameter is not referenced in the tailored code. If a tailored operation has an unreferenced parameter and the invocation has a corresponding argument expression, the compiler notes that the expression result is discarded. Earlier in this chapter there are examples of optimizations possible when expression results are discarded. If the corresponding argument is missing, the compiler refrains from supplying a null value for it. Consider the invocation

```
insert(x, 3)
```

insert takes three arguments. If x is a table, the third argument is used as the entry value and must be supplied in the generated code. In the following generated code, the default value for the third argument is computed into frame.tend.d[2].dword:

```
frame.tend.d[1].dword = D_Integer;
frame.tend.d[1].vword.integr = 3;
frame.tend.d[2].dword = D_Null;
frame.tend.d[3] = frame.tend.d[0] /* x */;
F1o0_insert(&(frame.tend.d[2]), &(frame.tend.d[1]), &(frame.tend.d[3]),
   &trashcan);
```

Because F1o0_insert uses a tailored calling convention, its arguments can be in a different order from those of the Icon function. It appears that the argument expression x is computed in the wrong place in the execution order. However, this is not true; the expression is not computed at all. If it were, the result would be a variable reference. Instead, the assignment of the value in x to the temporary variable is a form of optimized dereferencing. Therefore, it must be done as part of the operation, not as part of the argument computations. This is explained below.

If the value of x in this expression is a set instead of a table, the entry value is not used. This is illustrated by the following code. Note that a different C function is called for a set than for a table; this is because a different body statement is selected.

```
frame.tend.d[1].dword = D_Integer;
frame.tend.d[1].vword.integr = 3;
frame.tend.d[2] = frame.tend.d[0] /* x */;
F1o1_insert(&(frame.tend.d[1]) , &(frame.tend.d[2]) , &trashcan);
```

In general, an operation must copy its argument to a new descriptor before using it. This is because an operation is allowed to modify the argument. Modification of the original argument location is not safe in the presence of goal-directed evaluation. The operation could be re-executed without recomputing the argument. Therefore, the original value must be available. This is demonstrated with the following expression.

```
every p(0 to (1 to 3))
```

This is a double loop. The outer to expression is the inner loop, while the inner to expression is the outer loop. to modifies its first argument while counting. However, the first argument to the outer to has an extended lifetime due to the fact that the second argument is a generator. Therefore, this to operator must make a copy of its first argument. The generated code for this every expression is

```
    frame.tend.d[2].dword = D_Integer;
    frame.tend.d[2].vword.integr = 0;
    frame.tend.d[4].dword = D_Integer;
    frame.tend.d[4].vword.integr = 1;
    frame.tend.d[5].dword = D_Integer;
    frame.tend.d[5].vword.integr = 3;

L1: /* within: to */
    if (!(frame.tend.d[4].vword.integr <= frame.tend.d[5].vword.integr))
        goto L2 /* bound */;
    frame.tend.d[3].vword.integr = frame.tend.d[4].vword.integr;
    frame.tend.d[3].dword = D_Integer;

    frame.tend.d[6] = frame.tend.d[2];
L3: /* within: to */
    if (!(frame.tend.d[6].vword.integr <= frame.tend.d[3].vword.integr))
        goto L4 /* end suspend: to */;
    frame.tend.d[1].vword.integr = frame.tend.d[6].vword.integr;
    frame.tend.d[1].dword = D_Integer;
    P01_p(&frame.tend.d[1]);
    ++frame.tend.d[6].vword.integr;
    goto L3 /* within: to */;

L4: /* end suspend: to */
    ++frame.tend.d[4].vword.integr;
    goto L1 /* within: to */;
L2: /* bound */
```

The first argument to the outer to is copied with the statement

```
frame.tend.d[6] = frame.tend.d[2];
```

The copying of the other arguments has been eliminated because of two observations: the second argument of to is never modified and the first argument of the inner to (outer loop) is never reused without being recomputed. This second fact is determined while the lifetime information is being calculated. There is no generator occurring between the computation of the argument and the execution of the operator. Even if there were, it would only necessitate copying if the generator could be resumed after the operator started executing.

As noted above, another set of optimizations involves deferencing named variables. If an operation needs only the dereferenced value of an argument and type inferencing determines that the argument is a specific named variable (recall that each named variable is given a distinct variable reference type), the code generator does not need to generate code to compute the variable reference, because it knows what it is. That is, it does not need the value of the argument. If the argument is a simple identifier, no code at all is generated for the argument.

As shown in the code presented above for

```
insert(x, 3)
```

dereferencing can be implemented as simple assignment rather than a call to the deref function:

```
frame.tend.d[3] = frame.tend.d[0] /* x */;
```

In fact, unless certain conditions interfere, the variable can be used directly as the argument descriptor and no copying is needed. This is reflected in the code generated in a previous example:

```
if /x then ...
```

x is used directly in the in-line code for /:

```
if ((frame.tend.d[0] /* x */).dword == D_Null)
   goto L2 /* bound */;
```

This optimization cannot be performed if the operation modifies the argument, nor can it be performed if the variable's value might change while the operation is executing. Performing the optimization in the presence of the second condition would violate the semantics of argument dereferencing. The compiler does two simple tests to determine if the second condition might be true. If the operation has a side effect, the compiler assumes that the side-effect might involve the named variable. Side effects are explicitly coded in the abstract type computations of the operation. The second test is to see if the argument has an extended lifetime. The compiler assumes that the variable might be changed by another operation during the extended lifetime (that is, while the operation is suspended).

Assignment Optimizations

The final set of invocation optimizations involves assignments to named variables. These includes simple assignment and augmented assignments. Optimizing these assignments is important and optimizations are possible beyond those that can easily be done working from the definition in the data base; assignments to named variables are treated as special cases. The optimizations are divided into the cases where the right-hand-side might produce a variable reference and those where it produces a simple Icon value.

There are two cases when the right-hand-side of the assignment evaluates to a variable reference. If the right-hand-side is a named variable, a dereferencing optimization can be used. Consider

```
s := s1
```

This Icon expression is translated into

```
frame.tend.d[0] /* s */ = frame.tend.d[1] /* s1 */;
```

This is the ideal translation of this expression. For other situations, the deref function must be used. For example the expression

```
s := ?x
```

is translated into

```
if (O0f2_random(&(frame.tend.d[0] /* x */), &frame.tend.d[2]) == A_Resume)
   goto L1 /* bound */;
deref(&frame.tend.d[2], &frame.tend.d[1] /* s */);
```

When the right-hand-side computes to a simple Icon value, the named variable on the left-hand-side can often be used directly as the result location of the operation. This occurs in the earlier example

```
a := ~x
```

which translates into

```
O160_compl(&(frame.tend.d[0] /* x */), &frame.tend.d[1] /* a */);
```

This optimization is safe as long as setting the result location is the last thing the operation does. If the operation uses the result location as a work area and the variable were used as the result location, the operation might see the premature change to the variable. In this case, a separate result location must be allocated and the Icon assignment implemented as a C assignment. String concatenation is an example of an operation that uses its result location as a work area. The expression

```
s := s1 || s
```

is translated into

```
if (StrLoc(frame.tend.d[1] /* s1 */) + StrLen(frame.tend.d[1] /* s1 */)
   == strfree )
   goto L1 /* within: cater */;
StrLoc(frame.tend.d[2]) = alcstr(StrLoc(frame.tend.d[1] /* s1 */),
   StrLen(frame.tend.d[1] /* s1 */));
StrLen(frame.tend.d[2]) = StrLen(frame.tend.d[1] /* s1 */);
goto L2 /* within: cater */;
L1: /* within: cater */
   frame.tend.d[2] = frame.tend.d[1] /* s1 */;
L2: /* within: cater */
   alcstr(StrLoc(frame.tend.d[0] /* s */), StrLen(frame.tend.d[0] /* s */));
   StrLen(frame.tend.d[2]) = StrLen(frame.tend.d[1] /* s1 */) +
      StrLen(frame.tend.d[0] /* s */);

   frame.tend.d[0] /* s */ = frame.tend.d[2];
```

frame.tend.d[2] is the result location. If frame.tend.d[0] (the variable s) were used instead, the code would be wrong.

There are still some optimizations falling under the category covered by this chapter to be explored as future work. For example, as shown earlier,

```
a := ~x
```

is translated into

```
frame.tend.d[2] = frame.tend.d[0] /* x */;
cnv_tcset(&(frame.cbuf[0]), &(frame.tend.d[2]), &(frame.tend.d[2]));
O160_compl(&(frame.tend.d[2]) , &frame.tend.d[1] /* a */);
```

when x is a string. The assignment to frame.tend.d[2] can be combined with the conversion to produce the code

```
cnv_tcset(&(frame.cbuf[0]), &(frame.tend.d[0] /* x */), &(frame.tend.d[2]));
O160_compl(&(frame.tend.d[2]) , &frame.tend.d[1] /* a */);
```

There is, of course, always room for improvement in code generation for specific cases. However, the optimizations in this chapter combine to produce good code for most expressions. This is reflected in the performance data presented in the Chapter 11.

CHAPTER 11

Performance of Compiled Code

The performance of compiled code is affected by the various optimizations performed by the compiler. This chapter demonstrates the effects of these optimizations on the execution speed of Icon expressions. It also presents speed improvements and memory usage for compiled code versus interpreted code for a set of complete Icon programs. All timing results used in this chapter were obtained on a Sun 4/490 and are the average of the results from three runs.

Expression Optimizations

The effects of four categories of optimization are demonstrated. These are assignment optimizations, invocation optimizations, control flow optimizations, and optimizations using information from type inferencing. Expression timings for the first three categories were made using techniques described in the August 1990 issue of *The Icon Analyst* [43]. The following program skeleton is used to construct the programs to perform these timings.

```
procedure main()
   local x, start, overhead, iters

   iters := 1000000

   start := &time
   every 1 to iters do {
      }
   overhead := &time - start
```

```
x := 0
start := &time
every 1 to iters do {
   expression to be timed (may use x)
   }

write(&time – start – overhead)
end
```

The timings are performed both with and without the desired optimizations, and the results are compared by computing the ratio of the time without optimization to the time with optimization.

The assignment optimizations are described in Chapter 10. The effect of the assignment optimizations on the expression

```
x := &null
```

is measured using the program outlined above. The analysis that produces the assignment optimization is disabled by enabling debugging features in the generated code. The only other effect this has on the assignment expression is to insert code to update the line number of the expression being executed. In this test, the line number code is removed before the C code is compiled, insuring that the assignment optimization is the only thing measured. The timing results for this test produce

Assignment Test

Time in Milliseconds Averaged over Three Runs

Unoptimized	Optimized	Ratio
1122	478	2.3

The tests were performed with type inferencing enabled. Therefore, even the "unoptimized" version of the assignment has the standard operation optimizations applied to it. This test

demonstrates the importance of performing the special-case assignment optimizations.

The next category of optimization measured is invocation optimization. This results in the direct invocation of the C functions implementing operations, or in some cases results in the operations being generated in line. The execution time for the expression

```
tab(0)
```

is measured with and without invocation optimizations. As with the assignment optimizations, these optimizations are disabled by enabling debugging features. Once again the line number code is removed before the C code is compiled. These optimizations interact with the optimizations that use information from type inferencing. The measurements are made with type inferencing disabled. Therefore, no type checking simplifications are performed. Without the invocation optimizations, the generated code consists of an indirect invocation through the global variable tab. With the invocation optimizations, the generated code consists of type checking/conversion code for the argument to tab and a call to the function implementing the body statement of tab. The timing results for tab(0) produce

Invocation Test

Time in Milliseconds Averaged over Three Runs

Unoptimized	Optimized	Ratio
8394	4321	1.9

The third category of optimization is control flow optimization. As explained in Chapter 9, these optimizations only perform improvements that a C compiler will not perform when the code contains trivial call chains. One situation that produces trivial call chains is nested alternation. The execution time for the expression

```
every x := ixor(x, 1 | 2 | 3 | 4 | 5)
```

is measured with and without control flow optimizations. The timing results for this every loop produce

Control Flow Test

Time in Milliseconds Averaged over Three Runs

Unoptimized	Optimized	Ratio
6384	4184	1.5

The final category of optimization results from type inferencing. The speed improvements result from generating operations in line, eliminating type checking, and generating success continuations in line. Use of the to operation is a good example of where these optimizations can be applied. This is demonstrated by measuring the speed of an every loop using the to operation. The program that performs the measurement is

```
procedure main()
   local x, start

   start := &time
   every x := 1 to 5000000
   write(&time - start)
end
```

The timing results for this program produce

Type Inference Test

Time in Milliseconds Averaged over Three Runs

Unoptimized	Optimized	Ratio
9233	2721	3.3

Another approach to determining the effectiveness of type inferencing is to measure how small a set it deduces for the possible types of operands to operations. This indicates whether future work should concentrate on improving type inferencing itself or simply concentrate on using type information more effectively in code generation. A simple measure is used here: the percentage of operands for which type inferencing deduces a unique Icon type. Measurements are made for operands of all operators, except optimized assignment, and for operands of all built-in functions appearing in optimized invocations. For the most part, these are the operations where the code generator can use type information. Measurements were made for a set of 14 programs (described below). Unique operand types within each program range from 63 percent to 100 percent of all operands, with an overall figure for the tests suite of 80 percent (this is a straight unweighted figure obtained by considering all operands in the test suite without regard to what program they belong to); even a perfect type inferencing system will not deduce unique types for 100 percent of all operands, because not all operands have unique types. This suggests that an improved type inferencing system may benefit some programs, but will have only a small overall impact. Future work should give priority to making better use of the type information rather than to increasing the accuracy of type inferencing.

Program Execution Speed

It has been demonstrated that the compiler optimizations are effective at improving the kinds of expressions they are directed toward. The question remains: How fast is compiled code (with and without optimizations) for complete programs as compared to interpreted code for the same programs? For some expressions, optimizations may interact to create significant cumulative speed improvements. For example, the fully optimized code for the every loop in the previous example is 30 times faster than the interpreted code; the improvement of 3.3 from type inferencing contributes one factor in the total improvement. Other expressions may spend so much time in the run-time system (which is unaffected by compiler optimizations) that no measurable improvements are produced.

A set of 14 programs was selected mostly from contributions to the Icon program library [44] for testing the performance of the compiler. These programs were selected to represent a variety of applications and programming styles (an additional requirement is that they run long enough to obtain good timing results).

The following table shows the speed improvements for the compiled code as compared to interpreted code. The compiler and interpreter used for the measurements both implement Version 8 of Icon. The execution time used to compute the speed improvements is the cpu time measured using the Bourne shell's time command. The first column in the table shows the execution time under the interpreter. The second column is for compiled code with debugging features enabled and optimizations disabled. This code is still better than what would be obtained by just removing the interpreter loop, because intelligent code generation is performed, especially for bounded expressions, and keywords are generated in line. The third column is for code with debugging features disabled and full optimization enabled.

Execution Time In Seconds Averaged Over Three Runs

Program	Interpreter	Compiler Unoptimized	Compiler Optimized
cksol	49.9	33.5 (1.48)	22.5 (2.21)
concord	31.1	18.5 (1.68)	9.8 (3.17)
iidecode	60.3	34.0 (1.77)	12.9 (4.67)
iiencode	50.4	34.4 (1.46)	10.5 (4.80)
impress	44.6	24.8 (1.79)	14.0 (3.18)
list	43.1	24.5 (1.75)	13.6 (3.16)
memfiltr	60.8	34.3 (1.77)	15.3 (3.97)
mf	30.1	18.7 (1.60)	14.7 (2.04)

pssplit	64.0	39.0 (1.64)	26.6 (2.40)
roffcmds	32.9	18.1 (1.81)	12.0 (2.74)
sentence	34.3	23.9 (1.43)	16.2 (2.11)
spandex	36.8	23.3 (1.57)	14.7 (2.50)
textcnt	36.2	18.4 (1.96)	9.9 (3.65)
wrapper	27.3	15.9 (1.71)	9.4 (2.90)

The numbers in parentheses are speed-up factors obtained by dividing the interpreter execution time by the execution time of compiled code.

Code Size

One advantage the compiler has over the interpreter is that, unless a program is compiled with full string invocation enabled, the executable code for a program need not include the full run-time system. For systems with limited memory, this can be a significant advantage.

The sizes of executable code presented here are obtained from file sizes. All executable files have had debugging information stripped from them. The size of the executable code in the interpreter system is taken to be the size of the interpreter (278,528 bytes) plus the size of the icode for the program being measured (under Unix systems, the size of the executable header, 12,800 bytes for the Sun 4, is subtracted from the size of the icode file, because it is not present during interpretation). Measurements for the 14 test programs are:

Program Sizes in Bytes

Program	Interpreter	Compiler	Ratio
cksol	282,153	81,920	0.29
concord	284,416	90,112	0.31

iidecode	285,525	98,304	0.34
iiencode	283,567	81,920	0.28
impress	295,656	114,688	0.38
list	287,376	98,304	0.34
memfiltr	296,082	114,688	0.38
mf	282,739	81,920	0.28
pssplit	279,709	73,728	0.26
roffcmds	280,797	81,920	0.29
sentence	283,249	81,920	0.28
spandex	281,843	81,920	0.29
textcnt	280,397	73,728	0.26
wrapper	279,780	73,728	0.26

Other factors create differences in memory usage between the interpreter and compiled code. For example, the interpreter allocates a stack for expression evaluation. On the Sun 4, this stack is 40,000 bytes. The compiler, on the other hand, allocates work areas on a per-procedure basis and only allocates the maximum needed at any execution point within the procedure.

CHAPTER 12

Conclusions

Summary

The underlying ideas used in type inferencing, liveness analysis, and temporary variable allocation were explored using prototype systems before work was started on the compiler described in this dissertation. The fundamental reasons for creating the compiler were to prove that these ideas could be incorporated into a complete and practical compiler for Icon, to explore optimizations that are possible using the information from type inferencing, and to determine how well those optimizations perform. The goal of proving the usefulness of ideas continues a long tradition in the Icon language project and in the SNOBOL language project before it.

The prototype type inferencing system demonstrates that a naive implementation uses too much memory; implementation techniques were developed for the compiler to greatly reduce this memory usage. As the design and implementation of the compiler progressed, other problems presented themselves, both large and small, and solutions were developed to solve them. These problems include how to elegantly produce code either with or without type checking, how to generate good code for simple assignments (a very important kind of expression in most Icon programs), how to generate code that uses the continuation-passing techniques chosen for the compilation model, and how to perform peephole optimizations in the presence of success continuations.

This dissertation describes the problems addressed by the Icon compiler and why they are important to the compiler, along with innovative solutions. It presents a complete set of techniques used to implement the optimizing compiler. Performance measurements demonstrate the improvements brought about by the various optimizations. They also demonstrate that, for most programs, compiled code runs much faster than interpreted code. Previous work has shown that

simply eliminating the interpreter loop is not enough to produce large performance improvements [18]. Therefore, the measurements show that the set of techniques, in addition to being complete, is also effective.

Future Work

The Icon compiler builds upon and adds to a large body of work done previously by the Icon project. There are many problems and ideas relating to the implementation of Icon that remain to be explored in the future. Several are presented in earlier chapters. Others are described in the following list.

- The quality of type inferencing can be improved. For example, if

  ```
  x ||| y
  ```

  is successfully executed, both x and y must contain lists. The current version of type inferencing in the compiler does not use this information; it updates the store based on result types and side effects, but not based on the argument types that must exist for successful execution without run-time error termination. Another improvement is to extend the type system to include constants and thereby perform constant propagation automatically as part of type inferencing. The type system can also be extended to distinguish between values created in allocated storage and those that are constant and do not reside in allocated storage. A descriptor that never contains values from allocated storage does not need to be reachable by garbage collection.

- In spite of large improvements in the storage requirements of type inferencing over the prototype system, this analysis requires large amounts of memory for some programs. A suggestion by John Kececioglu [45] is to explore the use of applicative data structures that share structure with their predecessors.

- Type inferencing provides information about values that do not need run-time type information associated with them. In the case of integers and reals, this information along with information from the data base about run-time operations can be used to perform computations on pure C values and to demote Icon descriptor variables to simple C integer and double variables. The current compiler makes little use of these opportunities for optimization. Numerous other optimizations using the information from type inferencing are possible beyond what is currently being done. One of them is to choose the representation of a data structure based on how the data structure is used.
- Translating constant expressions involving integer and real values into the corresponding C expressions would allow the C compiler to perform constant folding on them. For other Icon types, constant folding must be performed by the Icon compiler. This is particularly important for csets, but is not presently being done.
- O'Bagy's prototype compiler performs two kinds of control flow optimizations. It eliminates unnecessary bounding and demotes generators that can not be resumed. The code generation techniques used in this compiler combined with the peephole optimizer automatically eliminate unnecessary bounding. The peephole optimizer also automatically demotes generators that are placed in-line. Enhancements to the peephole optimizer could effect the demotion of generators that are not placed in-line.
- The compiler uses a simple heuristic to decide when to use the in-line version of an operation and when to call the function implementing the operation using the standard calling conventions. More sophisticated heuristics should be explored.
- Temporary variables can retain pointers into allocated storage beyond the time that those pointers are needed. This reduces the effectiveness of garbage collection. Because garbage collection does not know which temporary variables are active and which are not, it retains all values pointed to by temporary variables. This problem can be solved by assigning the null value to temporary variables that are no longer active. However, this incurs significant overhead. The trade off between assigning null values and the reduced

effectiveness of garbage collection should be explored.

- The Icon compiler generates C code. If it generated assembly language code, it could make use of machine registers for state variables, such as the procedure frame pointer, and for holding intermediate results. This should result in a significant improvement in performance (at the cost of a less portable compiler and one that must deal with low-level details of code generation).
- Several of the analyses in the compiler rely on having the entire Icon program available. Separate compilation is very useful, but raises problems. On possible solution is to change the analyses to account for incomplete information. They could assume that undeclared variables can be either local or global and possibly initialized to a built-in function or unknown procedures, and that calls to unknown operations can fail, or return or suspend any value and perform any side-effect on any globally accessible variable. This would significantly reduce the effectiveness of some analyses. Another approach is to do incremental analyses, storing partial or tentative results in a data base. This is a much harder approach, but can produce results as good as compiling the program at one time.
- Enhancements to the compiler can be complemented with improvements to the run-time system. One area that can use further exploration is storage management.

Acknowledgements

I would like to thank Ralph Griswold for acting as my research advisor. He provided the balance of guidance, support, and freedom needed for me to complete this research. From him I learned many of the technical writing skills I needed to compose this dissertation. I am indebted to him and the other members of the Icon Project who over the years have contributed to the Icon programming language that serves as a foundation of this research. I would like to thank Peter Downey and Saumya Debray for also serving as members on my committee and for providing insightful criticisms and suggestions for this dissertation. In addition, Saumya Debray shared with me his knowledge of abstract interpretation, giving me the tool I needed to shape the final form of the type inferencing system.

I have received help from a number of my fellow graduate students both while they were still students and from some after they graduated. Clinton Jeffery, Nick Kline, and Peter Bigot proofread this dissertation, providing helpful comments. Similarly, Janalee O'Bagy, Kelvin Nilsen, and David Gudeman proofread earlier reports that served as a basis for several of the chapters in this dissertation. Janalee O'Bagy's own work on compiling Icon provided a foundation for the compiler I developed. Kelvin Nilsen applied my liveness analysis techniques to a slightly different implementation model, providing insight into dependencies on execution models.

Appendix A — The Implementation Language

This appendix contains a description of the language used to implement the run-time operations of the Icon compiler system. Chapter 5 provides a description of the design goals of the implementation language and an introduction to it. Some of the design decisions for the language were motivated by optimizations planned for the future, such as constant folding of csets. The use of these features is presented as if the optimizations were implemented; this insures that the optimizations will be supported by the run-time system when they are implemented. This appendix is adapted from the reference manual for the language [37].

The translator for the implementation language is the program rtt. An rtt input file may contain operation definitions written in the implementation language, along with C definitions and declarations. Rtt has a built-in C preprocessor based on the ANSI C Standard, but with extensions to support multi-line macros with embedded preprocessor directives [46]. Rtt prepends a standard include file, grttin.h, on the front of every implementation language file it translates.

The first part of this appendix describes the operation definitions. C language documentation should be consulted for ordinary C grammar. The extensions to ordinary C grammar are described in the latter part of the appendix.

The grammar for the implementation language is presented in extended BNF notation. Terminal symbols are set in Helvetica. Non-terminals and meta-symbols are set in *Times-Italic*. In addition to the usual meta-symbols, *::=* for "is defined as" and | for "alternatives", brackets around a sequence of symbols indicates that the sequence is optional, braces around a sequence of symbols followed by an asterisk indicates that the sequence may be repeated zero or more times, and braces followed by a plus indicates that the enclosed sequence may be repeated one or more times.

Operation Documentation

An operation definition can be preceded by an optional description in the form of a C string literal.

documented-definition ::= [C-string-literal] operation-definition

The use of a C string allows an implementation file to be run through the C preprocessor without altering the description. The preprocessor concatenates adjacent string literals, allowing a multi-line description to be written using multiple strings. Alternatively, a multi-line description can be written using '\' for line continuation. This description is stored in the operation data base where it can be extracted by *documentation generation programs*. These documentation generators produce formatted documentation for Icon programmers and for C programmers maintaining the Icon implementation. The documentation generators are responsible for inserting newline characters at reasonable points when printing the description.

Types of Operations

Rtt can be used to define the built-in functions, operators, and keywords of the Icon language. (Note that there are some Icon constructs that fall outside this implementation specification system. These include control structures such as string scanning and limitation, along with record constructors and field references.)

operation-definition ::=

function *result-seq*	*identifier* ([*param-list*]) [*declare*]	*actions*	end	|
operator *result-seq op*	*identifier* ([*param-list*]) [*declare*]	*actions*	end	|
keyword *result-seq*	*identifier*	*actions*	end	|
keyword *result-seq*	*identifier* const *key-const*		end	

result-seq ::= { *length* , *length* [+] } |
{ *length* [+] } |
{ }

length ::= *integer* | *

result-seq indicates the minimum and maximum length of the result sequence of an operation (the operation is treated as if it is used in a context where it produces all of its results). For example, addition always produces one result so its *result-seq* is {1, 1}. If the minimum and maximum are the same, only one number need be given, so the *result-seq* for addition can be coded as {1}. A conditional operation can produce either no results (that is, it can fail) or it can produce one result, so its *result-seq* is {0, 1}. A length of * indicates unbounded, so the *result-seq* of ! is indicated by {0, *}. An * in the lower bound means the same thing as 0, so {0, *} can be written as {*, *}, which simplifies to {*}. A *result-seq* of {} indicates no result sequence. This is not the same as a zero-length result sequence, {0}; an operation with no result sequence does not even fail. exit is an example of such an operation.

A + following the length(s) in a *result-seq* indicates that the operation can be resumed to perform some side effect after producing its last result. All existing examples of such operations produce at most one result, performing a side effect in the process. The side effect on resumption is simply an undoing of the original side effect. An example of this is tab, which changes &pos as the side effect.

For functions and keywords, *identifier* is the name by which the operation is known within the Icon language (for keywords, *identifier* does not include the &). New functions and keywords can be added to the language by simply translating implementations for them. For operations, *op* is (usually) the symbol by which the operation is known within the Icon language and *identifier* is a descriptive name. It is possible to have more than one operation with the same *op* as long as they have different identifiers and take a different number of operands. In addition to translating

the implementation for an operator, adding a new operator requires updating iconc's lexical analyzer and parser to know about the symbol (in reality, an *operator* definition may be used for operations with non-operator syntax, in which case any syntax may be used; iconc's code generator identifies the operation by the type of node put in the parse tree by a parser action). In all cases, the *identifier* is used to construct the name(s) of the C function(s) which implement the operation.

A *param-list* is a comma separated list of parameter declarations. Some operations, such as the write function, take a variable number of arguments. This is indicated by appending a pair of brackets enclosing an identifier to the last parameter declaration. This last parameter is then an array containing the *tail* of the argument list, that is, those arguments not taken up by the preceding parameters. The identifier in brackets represents the length of the tail and has a type of C integer.

param-list ::= param { , param } [[identifier]]*

Most operations need their arguments dereferenced. However, some operations, such as assignment, need undereferenced arguments and a few need both dereferenced and undereferenced versions of an argument. There are forms of parameter declarations to match each of these needs.

param ::= *identifier* |
underef *identifier* |
underef *identifier* –> *identifier*

A simple identifier indicates a dereferenced parameter. underef indicates an undereferenced parameter. In the third form of parameter declaration, the first identifier represents the undereferenced form of the argument and the second identifier represents the dereferenced form. This third form of declaration may not be used with the variable part of an argument list. These identifiers are of type *descriptor*. Descriptors are implemented as C structs. See [3] for a detailed explanation of descriptors.

Examples of operation headers:

```
"detab(s,i,...) – replace tabs with spaces, with stops at columns indicated."
function{1} detab(s, i[n])
   actions
end

"x <-> y – swap values of x and y."
" Reverses swap if resumed."
operator{0,1+} <-> rswap(underef x -> dx, underef y -> dy)
   declare
   actions
end

"&fail – just fail"
keyword{0} fail
   actions
end
```

Declare Clause

Some operations need C declarations that are common to several actions. These can be declared within the declare clause.

declare ::= declare { *C declarations* }

These may include *tended* declarations, which are explained below in the section on extensions to C. If a declaration can be made local to a block of embedded C code, it is usually better to put it there than in a declare clause. This is explained below in the discussion of the body action.

Constant Keywords

Any keyword can be implemented using general *actions*. However, for constant keywords, iconc can sometimes produce more efficient code if it treats the keyword as a literal constant. Therefore, a special declaration is available for declaring keywords that can be represented as Icon literals. The constant is introduced with the word const and can be one of four literal types.

key-const ::= string-literal | cset-literal | integer-literal | real-literal

When using this mechanism, it is important to be aware of the fact that rtt tokenizes these literals as C literals, not as Icon literals. The contents of string literals and character literals (used to represent cset literals) are not interpreted by rtt except for certain situations in string concatenation (see [46]). They are simply stored, as is, in the data base. This means that literals with escape sequences can be used even when C and Icon would give them different interpretations. However, C does not recognize control escapes, so '\^', which is a valid Icon literal, will result in an error message from rtt, because the second quote ends the literal, leaving the third quote dangling. Only decimal integer literals are allowed.

Actions

All operations other than constant keywords are implemented with general *actions*.

Actions fall into four categories: type checking and conversions, detail code expressed in extended C, abstract type computations, and error reporting.

*actions ::= { action }**

action ::= *checking-conversions* |
detail-code |
abstract { *type-computations* } |
runerr(*msg_number* [, *descriptor*]) [;]
{ *actions* }

Type Checking and Conversions

The type checking and conversions are

checking-conversions ::= if *type-check* then *action* |
if *type-check* then *action* else *action* |
type_case *descriptor* of { { *type-select* }+ }
len_case *identifier* of { { *integer* : *action* }+ default : *action* }

type-select ::= { *type-name* : }+ *action* |
default : *action*

These actions specify run-time operations. These operations could be performed in C, but specifying them in the implementation language gives the compiler information it can use to generate better code.

The if actions use the result of a *type-check* expression to select an action. The type_case action selects an action based on the type of a descriptor. If a type_case action contains a default clause, it must be last. *type-select* clauses must be mutually exclusive in their selection. The len_case action selects an action based on the length of the variable part of the argument list of the operation. The *identifier* in this action must be the one representing that length.

A *type-check* can succeed or fail. It is either an assertion of the type of a descriptor, a conversion of the type of a descriptor, or a logical expression involving *type-checks*. Only limited

forms of logical expressions are supported.

type-check ::= *simple-check* { && *simple-check* }* |
! *simple-check*

simple-check ::= is: *type-name* (*descriptor*) |
cnv: *dest-type* (*source* [, *destination*]) |
def: *dest-type* (*source* , *value* [, *destination*])

dest-type ::= cset |
integer |
real |
string |
C_integer |
C_double |
C_string |
(exact)integer |
(exact)C_integer
tmp_string |
tmp_cset

The is check succeeds if the value of the descriptor is in the type indicated by *type-name*. Conversions indicated by cnv are the conversions between the Icon types of cset, integer, real, and string. Conversions indicated by def are the same conversions with a default value to be used if the original value is null.

dest-type is the type to which to a value is to be converted, if possible. cset, integer, real, and string constitute a subset of *icon-type* which is in turn a subset of *type-name* (see below). C_integer, C_string, and C_double are conversions to internal C types that are easier to

manipulate than descriptors. Each of these types corresponds to an Icon type. A conversion to an internal C type succeeds for the same values that a conversion to the corresponding Icon type succeeds. C_integer represents the C integer type used for integer values in the particular Icon implementation being compiled (typically, a 32-bit integer type). C-double represents the C double type. C-string represents a pointer to a null-terminated C character array. However, see below for a discussion of the destination for conversion to C_string. (exact) before integer or C_integer disallows conversions from reals or strings representing reals, that is, the conversion fails if the value being converted represents a real value.

Conversion to tmp_string is the same as conversion to string (the result is a descriptor), except that the string is only guaranteed to exist for the lifetime of the operation (the lifetime of a suspended operation extends until it can no longer be resumed). Conversion to tmp_string is generally less expensive than conversion to string and is never more expensive, but the resulting string must not be exported from the operation. tmp_cset is analogous to tmp_string.

The source of the conversion is the descriptor whose value is to be converted. If no destination is specified, the conversion is done ''in-place''. However, it may not actually be possible to do an argument conversion in the argument's original location, so the argument may be copied to another location as part of the conversion. Within the *scope* of the conversion, the parameter name refers to this new location. The scope of a conversion is usually only important for conversions to C types; the run-time system translator and the Icon compiler try to keep the movement of descriptor parameters transparent (see below for more details). All elements of the variable part of an argument list must be descriptors. Therefore, when an element is converted to a C type, an explicit location must be given for the destination.

The destinations for conversions to cset, integer, real, string, (exact)integer, tmp_string, and tmp_cset must be descriptors. The destinations for conversions to C_integer, C_double, and (exact)C_integer must be the corresponding C types. However, the destination for conversion to C_string must be tended. If the destination is declared as ''tended char *'', then the dword (string length) of the tended location will be set, but the operation will not have direct

access to it. The variable will look like a "char *". Because the operation does not have access to the string length, it is not a good idea to change the pointer once it has been set by the conversion. If the destination is declared as a descriptor, the operation has access to both the pointer to the string and the string's length (which includes the terminating null character). If a parameter is converted to C_string and no explicit destination is given, the parameter will behave like a "tended char *" within the scope of the conversion.

The second argument to the def conversion is the default value. The default value may be any C expression that evaluates to the correct type. These types are given in the following chart.

cset:	struct b_cset
integer:	C_integer
real:	double
string:	struct descrip
C_integer:	C_integer
C_double:	double
C_string:	char *
tmp_string:	struct descrip
tmp_cset:	struct b_cset
(exact)integer:	C_integer
(exact)C_integer:	C_integer

The numeric operators provide good examples of how conversions are used:

```
operator{1} / divide(x, y)
   if cnv:(exact)C_integer(x) && cnv:(exact)C_integer(y)  then
      actions
   else {
```

```
        if !cnv:C_double(x) then
           runerr(102, x)
        if !cnv:C_double(y) then
           runerr(102, y)
        actions
        }
end
```

Within the code indicated by *actions*, x and y refer to C values rather than to the Icon descriptors of the unconverted parameters.

The subject of any type check or type conversion must be an unmodified parameter. For example, once an in-place conversion has been applied to a parameter, another conversion may not be applied to the same parameter. This helps insure that type computations in iconc only involve the unmodified types of arguments, simplifying those computations. This restriction does not apply to type checking and conversions in C code.

Scope of Conversions

The following discussion is included mostly for completeness. The scope of conversions sounds complicated, but in practice problems seldom occur in code that "looks reasonable". If a problem does occur, the translator catches it. Normally, the intricacies of scope should be ignored and the person writing run-time routines should code conversions in a manner that seems natural.

An "in-place" conversion of a parameter can create a scope for the parameter name separate from the one introduced by the parameter list. This is because conversions to C types may require the converted value to be placed in a different location with a different type. The parameter name is then associated with this new location. The original scope of a parameter starts at the beginning of the operation's definition. The scope of a conversion starts at the conversion. A

scope extends through all code that may be executed after the scope's beginning, up to a runerr or a conversion that hides the previous scope (because the type checking portion of the implementation language does not contain loops or arbitrary gotos, scope can easily be determined lexically).

The use of an in-place conversion in the first sub-expression of a conjunction, *cnv1* && *cnv2*, has a potential for causing problems. In general, there is no way to know whether the first conversion will effectively be undone when the second conversion fails. If the first conversion is actually done in-place, the parameter name refers to the same location in both the success and failure scope of the conjunction, so the conversion is not undone. If the conversion is done into a separate location, the failure scope will refer to the original value, so the conversion will effectively be undone. Whether the conversion is actually done in-place depends on the context in which operation is used. However, conversion to C_integer and C_double always preserve the original value, so there is no potential problem using them as the first argument to a conjunction, nor is there any problem using a non-conversion test there. An example of this uncertainty:

```
if cnv:string(s1) && cnv:string(s2) then {
  /* s1 and s2 both refer to converted values */
  }
else {
  /* s2 refers to the original value. s1 may refer to either the original or the
     converted value */
  }
```

The translator issues a warning if there is a potential problem.

It is possible for scopes to overlap; this happens because scopes start within conditional actions. In rare instances, executable code using the name may appear within this overlapping scope, as in the following example, which resembles code that might be found in the definition of a string analysis function such as find.

```
if is:null(s) then {
    if !def:C_integer(i, k_pos) then
        runerr(101, i)
    }
else {
   if !def:C_integer(i, 1) then
        runerr(101, i)
   }

actions
```

Here, *actions* occurs within the scope of both conversions. Note that *actions* is not in the scope of the original parameter i. This is because that scope is ended in each branch of the outer if by the conversions and the runerrs.

If overlap does occur, the translator tries to insure that the same location is used for the name in each scope. The only situation when it cannot do this is when the type of the location is different in each scope, for instance, one is a C_integer and the other is a C_real. If a name is referenced when there is conflicting scope, the translator issues an error message.

Type Names

The *type-names* represent types of Icon intermediate values, including variable references. These are the values that enter and leave an operation; ''types'' internal to data structures, such as list element blocks, are handled completely within the C code.

type-name ::= empty_type |
icon-type |
variable-ref

icon-type ::= null |
string |
cset |
integer |
real |
file |
list |
set |
table |
record |
procedure |
co_expression

variable-ref ::= variable |
tvsubs |
tvtbl |
kywdint |
kywdpos |
kywdsubj

The *type-names* are not limited to the first-class types of Icon's language definition. The *type-names* that do not follow directly from Icon types need further explanation. empty_type is the type containing no values and is needed for conveying certain information to the type inferencing system, such as an unreachable state. For example, the result type of stop is empty_type. It may also be used as the internal type of an empty structure. Contrast this with null, which consists of the null value.

Variable references are not first-class values in Icon; they cannot be assigned to variables.

However, they do appear in the definition of Icon as arguments to assignments and as the subject of dereferencing. For example, the semantics of the expression

```
s[3] := s
```

can be described in terms of a substring trapped variable and a simple variable reference. For this reason, it is necessary to include these references in the type system of the implementation language. variable consists of all variable references. It contains five distinguished subtypes. tvsubs contains all substring trapped variables. tvtbl contains all table-element trapped variables. kywdint contains &random and &trace. kywdpos contains &pos. kywdsubj contains &subject.

Including C Code

As noted above, C declarations can be included in a declare clause. Embedded C code may reference these declarations as well as declarations global to the operation.

Executable C code can be included using one of two actions.

detail-code ::= body { *extended-C* } |
inline { *extended-C* }

body and inline are similar to each other, except that inline indicates code that is reasonable for the compiler to put in-line when it can. body indicates that for the in-line version of the operation, this piece of C code should be put in a separate function in the link library and the body action should be replaced by a call to that function. Any parameters or variables from the declare clause needed by the function must be passed as arguments to the function. Therefore, it is more efficient to declare variables needed by a body action within that body than within the declare. However, the scope of these local variables is limited to the body action.

Most Icon keywords provide examples of operations that should be generated in-line. In the following example, nulldesc is a global variable of type descriptor. It is defined in the include

files automatically included by rtt.

```
"&null – the null value."
keyword{1} null
   abstract {
      return null
      }
   inline {
      return nulldesc;
      }
end
```

Error Reporting

runerr(*msg_number [, descriptor]*) *[;]*

runerr is translated into a call to the run-time error handling routine. Specifying this as a separate action rather than a C expression within a body or inline action gives the compiler additional information about the behavior of the operation. *msg_number* is the number used to look up the error message in a run-time error table. If a descriptor is given, it is taken to be the offending value.

Abstract Type Computations

abstract { *type-computations* }

The behavior of an operation with respect to types is a simplification of the full semantics of the operation. For example, the semantics of the function image is to produce the string representing its operand; its behavior in the type realm is described as simply returning some

string. In general, a good simplification of an operation is too complicated to be automatically produced from the operation's implementation (of course, it is always possible to conclude that an operation can produce any type and can have any side effect, but that is hardly useful). For this reason, the programmer must use the abstract action to specify *type-computations*.

type-computations ::= { store [*type*] = *type* [;] }* [return *type* [;]]

type-computations consist of side effects and a statement of the result type of the operation. There must be exactly one return *type* along any path from the start of the operation to C code containing a return, suspend, or fail.

A side effect is represented as an assignment to the *store*. The store is analogous to program memory. Program memory is made up of locations containing values. The store is made up of locations containing types. A type represents a set of values, though only certain such sets correspond to types for the purpose of abstract type computations. Types may be basic types such as all Icon integers, or they may be composite types such as all Icon integers combined with all Icon strings. The rules for specifying types are given below. A location in the store may correspond to one location in program memory, or it may correspond to several or even an unbounded number of locations in program memory. The contents of a location in the store can be thought of as a conservative (that is, possibly overestimated) summary of values that might appear in the corresponding location(s) in program memory at run time.

Program memory can be accessed through a pointer. Similarly, the store can be indexed by a pointer type, using an expression of the form store[*type*], to get at a given location. An Icon global variable has a location in program memory, and a reference to such a variable in an Icon program is treated as a pointer to that location. Similarly, an Icon global variable has a location in the store and, during type inferencing, a reference to the variable is interpreted as a pointer type indexing that location in the store. Because types can be composite, indexing into the store with a pointer type may actually index several locations. Suppose we have the following side effect

store[*type1*] = *type2*

Suppose during type inferencing *type1* evaluates to a composite pointer type consisting of the pointer types for several global variables, then all corresponding locations in the store will be updated. If the above side effect is coded in the assignment operator, this situation might result from an Icon expression such as

every (x | y) := &null

In this example, it is obvious that both variables are changed to the null type. However, type inferencing can only deduce that at least one variable in the set is changed. Thus, it must assume that each could either be changed or left as is. It is only when the left hand side of the side effect represents a unique program variable that type inferencing knows that the variable cannot be left as is. In the current implementation of type inferencing, assignment to a single named variable is the only side effect where type inferencing recognizes that the side effect will definitely occur.

Indexing into the store with a non-pointer type corresponds to assigning to a non-variable. Such an assignment results in error termination. Type inferencing ignores any non-pointer components in the index type; they represent execution paths that don't continue and thus contribute nothing to the types of expressions.

A type in an abstract type computation is of the form

type ::= *type-name* |
type (*variable*) |
attrb-ref |
new *type-name* (*type* { , *type* }*) |
store [*type*] |
type ++ *type* |

```
type ** type  |
( type )
```

The type(*variable*) expression allows type computations to be expressed in terms of the type of an argument to an operation. This must be an unmodified argument. That is, the abstract type computation involving this expression must not be within the scope of a conversion. This restriction simplifies the computations needed to perform type inferencing.

This expression is useful in several contexts, including operations that deal with structure types. The type system for a program may have several sub–types for a structure type. The structure types are list, table, set, record, substring trapped variable, and table–element trapped variable. Each of these Icon types is a composite type within the type computations, rather than a basic type. Thus the type inferencing system may be able to determine a more accurate type for an argument than can be expressed with a *type–name*. For example, it is more accurate to use

```
if is:list(x) then
   abstract {
      return type(x)
      }
   actions
else
   runerr(108, x)
```

than it is to use

```
if is:list(x) then
   abstract {
      return list
      }
   actions
else
   runerr(108, x)
```

Structure values have internal ''structure''. Structure types also need an internal structure that summarizes the structure of the values they contain. This structure is implemented with type attributes. These attributes are referenced using dot notation:

attrb-ref ::= *type* . *attrb-name*

attrb-name ::= lst_elem |
set_elem |
key |
tbl_elem |
default |
all_fields |
str_var |
trpd_tbl

Just as values internal to structure values are stored in program memory, types internal to structure types are kept in the store. An attribute is a pointer type referencing a location in the store.

A list is made up of (unnamed) variables. The lst_elem attribute of a list type is a type representing all the variables contained in all the lists in the type. For example, part of the code for the bang operator is as follows, where dx is the dereferenced operand.

```
type_case dx of {
   list: {
      abstract {
         return type(dx).lst_elem
         }
      actions
      }
   ...
```

This code fragment indicates that, if the argument to bang is in a list type, bang returns some variable from some list in that type. In the type realm, bang returns a basic pointer type.

The set_elem attribute of a set type is similar. The locations of a set never "escape" as variables. That is, it is not possible to assign to an element of a set. This is reflected in the fact that a set_elem is always used as the index to the store and is never assigned to another location or returned from an operation. The case in the code from bang for sets is

```
set: {
   abstract {
      return store[type(dx).set_elem]
      }
   actions
   }
```

Tables types have three attributes. key references a location in the store containing the type of any possible key value in any table in the table type. tbl_elem references a location containing the type of any possible element in any table in the table type. default references a location containing the type of any possible default value for any table in the table type. Only tbl_elem corresponds to a variable in Icon. The others must appear as indexes into the store.

Record types are implemented with a location in the store for each field, but these locations cannot be accessed separately in the type computations of the implementation language. These are only needed separately during record creation and field reference, which are handled as special cases in the compiler. Each record type does have one attribute, all_fields, available to type computations. It is a composite type and includes the pointer types for each of the fields.

Substring trapped variables are implemented as structures. For this reason, they need structure types to describe them. The part of the structure of interest in type inferencing is the reference to the underlying variable. This is reflected in the one attribute of these types, str_var. It is a reference to a location in the store containing the pointer types of the underlying the variables that are ''trapped''. str_var is only used as an index into the store; it is never exported from an operation.

Similarly table-element trapped variables need structure types to implement them. They have one attribute, trpd_tbl, referencing a location in the store containing the type of the underlying table. The key type is not kept separately in the trapped variable type; it must be immediately added to the table when a table-element trapped variable type is created. This pessimistically assumes that the key type will eventually be put in the table, but saves an attribute in the trapped variable for the key. trpd_tbl is only used as an index into the store; it is never exported from an operation.

The type computation, new, indicates that an invocation of the operation being implemented creates a new instance of a value in the specified structure type. For example, the implementation of the list function is

```
function{1} list(size, initial)
   abstract {
      return new list(type(initial))
      }
   actions
end
```

The type arguments to the new computation specify the initial values for the *attributes* of the structure. The table type is the only one that contains multiple attributes. (Note that record constructors are created during translation and are not specified via the implementation language.) Table attributes must be given in the order: key, tbl_elem, and default.

In the type system for a given program, a structure type is partitioned into several sub-types (these sub-types are only distinguished during type inferencing, not at run time). One of these sub-types is allocated for every easily recognized use of an operation that creates a new value for the structure type. Thus, the following Icon program has two list sub-types: one for each invocation of list.

```
procedure main()
   local x

   x := list(1, list(100))
end
```

Two operations are available for combining types. Union is denoted by the operator '++' and intersection is denoted by the operator '**'. Intersection has the higher precedence. These operations interpret types as sets of values. However, because types may be infinite, these sets are treated symbolically.

C Extensions

The C code included using the declare, body, and inline actions may contain several constructs beyond those of standard C. There are five categories of C extensions: access to interface variables, declarations, type conversions/type checks, signaling run-time errors, and return statements.

In addition to their use in the body of an operation, the conversions and checks, run-time error, and declaration extensions may be used in ordinary C functions that are put through the implementation language translator.

Interface Variables

Interface variables include parameters, the identifier for length of the variable part of an argument list, and the special variable result. Unconverted parameters, converted parameters with Icon types, and converted parameters with the internal types tmp_string and tmp_cset are descriptors and within the C code have the type struct descrip. Converted parameters with the internal type of C_integer have some signed integer type within the C code, but exactly which C integer type varies between systems. This type has been set up using a typedef in the automatically included include file so it is available for use in declarations in C code. Converted parameters with the internal type of C_double have the type double within the C code. Converted parameters of the type C_string have the type char *. The length of the variable part of a argument list has the type *int* within the C code.

result is a special descriptor variable. Under some circumstances it is more efficient to construct a return value in this descriptor than to use other methods. See Section 5 of the implementation language reference manual for details.

Declarations

The extension to declarations consists of a new storage class specifier, tended (register is an example of an existing storage class specifier). Understanding its use requires some knowledge of Icon storage management. Only a brief description of storage management is given here; see the Icon implementation book for further details.

Icon values are represented by descriptors. A descriptor contains both type information and value information. For large values (everything other than integers and the null value) the descriptor only contains a pointer to the value, which resides elsewhere. When such a value is dynamically created, memory for it is allocated from one of several memory regions. Strings are allocated from the *string region*. All other relocatable values are allocated from the *block region*. The only non-relocatable values are co-expression stacks and co-expression activation blocks. On some systems non-relocatable values are allocated in the *static region*. On other systems there is no static region and these values are allocated using the C malloc function.

When a storage request is made to a region and there is not enough room in that region, a *garbage collection* occurs. All *reachable* values for each region are located. Values in the string and block regions are moved into a contiguous area at the bottom of the region, creating (hopefully) free space at the end of the region. Unreachable co-expression stacks and activator blocks are ''freed''. The garbage collector must be able to recognize and save all values that might be referenced after the garbage collection and it must be able to find and update all pointers to the relocated values. Operation arguments that contain pointers into one of these regions can always be found by garbage collection. The implementations of many operations need other descriptors or pointers into memory regions. The tended storage class identifies those descriptors and pointers that may have *live* values when a garbage collection could occur (that is, when a memory allocation is performed).

A descriptor is implemented as a C struct named descrip, so an example of a tended descriptor declaration is

```
tended struct descrip d;
```

Blocks are also implemented as C structs. The following list illustrates the types of block pointers that may be tended.

```
tended struct b_real *bp;
tended struct b_cset *bp;
tended struct b_file *bp;
tended struct b_proc *bp;
tended struct b_list *bp;
tended struct b_lelem *bp;
tended struct b_table *bp;
tended struct b_telem *bp;
tended struct b_set *bp;
tended struct b_selem *bp;
tended struct b_record *bp;
tended struct b_tvkywd *bp;
tended struct b_tvsubs *bp;
tended struct b_tvtbl *bp;
tended struct b_refresh *bp;
tended struct b_coexpr *cp;
```

Alternatively, a union pointer can be used to tend a pointer to any kind of block.

```
tended union block *bp;
```

Character pointers may also be tended. However, garbage collection needs a length associated with a pointer into the string region. Unlike values in the block region, the strings themselves do not have a length stored with them. Garbage collection treats a tended character pointer

as a zero-length string. These character pointers are almost always pointers into some string, so garbage collection effectively treats them as zero-length substrings of the strings. The string as a whole must be tended by some descriptor so that it is preserved. The purpose of tending a character pointer is to insure that the pointer is relocated with the string it points into. An example is

```
tended char *s1, *s2;
```

Tended arrays are not supported. tended may only be used with variables of local scope. tended and register are mutually exclusive. If no initial value is given, one is supplied that is consistent with garbage collection.

Type Conversions/Type Checks

Some conditional expressions have been added to C. These are based on type checks in the type specification part of the implementation language.

is: *type-name* (*source*)
cnv: *dest-type* (*source* , *destination*)
def: *dest-type* (*source* , *value* , *destination*)

source must be an Icon value, that is, a descriptor. *destination* must be a variable whose type is consistent with the conversion. These type checks may appear anywhere a conditional expression is valid in a C program. Note that is, cvn, and def are reserved words to distinguish them from labels.

The type_case statement may be used in extended C. This statement has the same form as the corresponding action, but in this context, C code replaces the *actions* in the *type-select* clauses.

Signaling Run-time Errors

runerr is used for signaling run-time errors. It acts like a function but may take either 1 or 2 arguments. The first argument is the error number. If the error has an associated value, the second argument is a descriptor containing that value.

Return Statements

There are three statements for leaving the execution of an operation. These are analogous to the corresponding expressions in the Icon language.

ret-statments ::= return *ret-value* ; |
suspend *ret-value* ; |
fail ;

ret-value ::= *descriptor* |
C_integer *expression* |
C_double *expression* |
C_string *expression* |
descript-constructor

descriptor is an expression of type struct descrip. For example

```
{
    tended struct descrip dp;
    ...
    suspend dp;
    ...
}
```

Use of C_integer, C_double, or C_string to prefix an expression indicates that the expression

evaluates to the indicated C type and not to a descriptor. When necessary, a descriptor is constructed from the result of the expression, but when possible the Icon compiler produces code that can use the raw C value (See Section 5 of the implementation language reference manual). As an example, the integer case in the divide operation is simply

```
inline {
   return C_integer x / y;
   }
```

Note that a returned C string must not be in a local (dynamic) character array; it must have a global lifetime.

A *descript-constructor* is an expression that explicitly converts a pointer into a descriptor. It is only valid in a return statement, because it builds the descriptor in the implicit location of the return value.

descript-constructor ::= string (*length* , *char-ptr*) |
cset (*block-ptr*) |
real (*block-ptr*) |
file (*block-ptr*) |
procedure (*block-ptr*) |
list (*block-ptr*) |
set (*block-ptr*) |
record (*block-ptr*) |
table (*block-ptr*) |
co_expression (*stack-ptr*) |
tvtbl (*block-ptr*) |
named_var (*descr-ptr*) |

struct_var (*descr–ptr* , *block–ptr*) |

substr (*descr–ptr* , *start* , *len*) |

kywdint (*descr–ptr*) |

kywdpos (*descr–ptr*) |

kywdsubj (*descr–ptr*)

The arguments to string are the length of the string and the pointer to the start of the string. *block–ptr*s are pointers to blocks of the corresponding types. *stack–ptr* is a pointer to a co–expression stack. *descr–ptr* is a pointer to a descriptor. named_var is used to create a reference to a variable (descriptor) that is not in a block. struct_var is used to create a reference to a variable that is in a block. The Icon garbage collector works in terms of whole blocks. It cannot preserve just a single variable in the block, so the descriptor referencing a variable must contain enough information for the garbage collector to find the start of the block. That is what the *block–ptr* is for. substr creates a substring trapped variable for the given descriptor, starting point within the string, and length. kywdint, kywdpos, and kywdsubj create references to keyword variables.

Note that returning either C_double *expression* or substr(*descr–ptr*, *start*, *len*) may trigger a garbage collection.

Appendix B — Correctness of the Type Inferencing Model

The appendix discusses the correctness of the type inferencing model with respect to the semantics of Icon. A series of abstractions is correct if each abstraction is *consistent* with the previous abstraction. The standard definition of consistency is used. This definition is given below.

A proof is given that Model 1 is consistent with the collecting semantics; this is done without examining the semantics of individual operations. Model 2 is not considered in this discussion of consistency. Instead, Model 3 is compared directly with Model 1. A full proof that Model 3 is consistent with Model 1 requires examining the semantics of each operation. Icon is too large a language for such a proof to be practical. A partial proof is given in which consistency is proven for the domain of the models and for the semantics of two operations.

$envir_{[1]}$ is defined in Chapter 3 to contain just a store and a heap. Arguments are given that the remaining information needed to represent the state of an Icon interpreter can be ignored. A proof of correctness must must explicitly discard this information later in the abstraction process. This remaining information, the *balance component* of the environment, is finite at any point in execution, so it can be encoded as an integer. $envir_{[1]}$ is redefined here to be

$$envir_{[1]} = store_{[1]} \times heap_{[1]} \times balance$$

$$balance = integer$$

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 collecting 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

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

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

The first step in proving 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. It follows that the type system with the subset relation is isomorphic to the power set of type names with the subset relation and that the type system forms a lattice. It is then easy to show that the lattice of the type system induces a lattice structure on $store_{[3]}$.

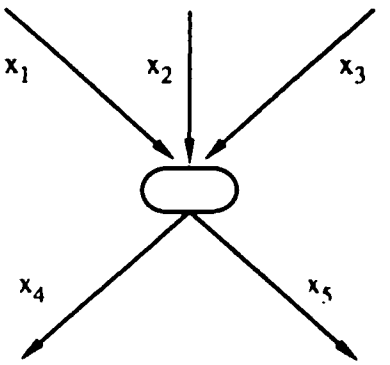
Let D_i be the domain of abstraction i. Given an edge m of the flow graph with n being the node at the head of m, a *transfer function*

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

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. $\forall\, x_1, x_2 \in D_j \text{ s.t. } x_1 \leq x_2,\ \alpha(x_1) \leq \alpha(x_2)$

2. $\forall\, \hat{x}_1, \hat{x}_2 \in D_i \text{ s.t. } \hat{x}_1 \leq \hat{x}_2,\ \gamma(\hat{x}_1) \leq \gamma(\hat{x}_2)$

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 collecting semantics (abstraction 0). 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 collecting semantics and in Model 1, respectively. The transfer function for the success edge of a node selects those environments on the incoming edges to the node that cause the operation associated with the node to succeed. The transfer function produces its result set by applying the meaning of the operation to each selected environment. This function is the same in both models, so condition 5 is true for it.

There may be several failure edges for a node; this results from the fact that different paths reaching the node may produce different backtracking paths. In both models, the transfer functions for these edges select those environments on the incoming edges that will cause the operation associated with the node to fail. The transfer functions in the collecting semantics further reduce the set of environments they operate on by selecting only those whose backtracking stack (encoded within the "balance" component of the environment) indicates that the edge should be taken. In order to avoid dealing with the actual encoding of the balance component of the environment, it is updated identically in both the the collecting semantics and in Model 1. For example, the portion of the balance component that encodes the backtracking stack is "popped" by the transfer functions for failure edges. If an update is applied to a balance component where the update is not defined, a zero balance component results (zero is chosen as an arbitrary

integer). This can happen in Model 1 because that abstraction may propagate environments to edges where they cannot occur in the collecting semantics. Each $f_{m,1}$ for failure edges 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 failure 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 environment is (s_1, h_1, b_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, b_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 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 Consistency of Model 3 with respect to Model 1

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

i, j, k ∈ positive integers

n ∈ nodes

v ∈ variables (usually non-structure variables unless otherwise stated)

$v_{n,i}$, $v_{n,j}$ ∈ structure variables

r_i ∈ temporary variables

V_n ∈ variable types

$p_{n,k}$ ∈ pointers

P_n ∈ pointer types

t ∈ types

As explained above, a proof of consistency requires establishing functions α and γ and demonstrating conditions 1 through 5. Let Type: $2^{values} \rightarrow 2^{values}$ be the function that maps an arbitrary set of values to the smallest type containing those values. This is an auxiliary function

used for defining α. 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

$\forall t \in$ variable types, $\hat{s}(t) = Type(\{d : v \in t, (s, h, b) \in x, d = s(v)\})$

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

$t_1, t_2 \in$ variable types

$\hat{s}(t_1 \cup t_2) = Type(\{d : v \in t_1 \cup t_2, (s, h, b) \in x, d = s(v)\})$

$= Type(\{d : v \in t_1, (s, h, b) \in x, d = s(v)\} \cup \{d : v \in t_2, (s, h, b) \in x, d = s(v)\})$

$$= \text{Type}(\{d : v \in t_1, (s, h, b) \in x, d = s(v)\}) \cup$$

$$\text{Type}(\{d : v \in t_2, (s, h, b) \in x, d = s(v)\})$$

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

All transfer functions in Model 3 transform stores in such a way as to preserve this distributivity over union.

The definition of

$$\gamma: \text{store}_{[3]} \rightarrow 2^{\text{envir}_{[1]}}$$

is

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

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

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

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

$$\forall\, p_{n,k},\ h(p_{n,k}) = L$$

where L: positive integers $\rightarrow \{v_{n,k}$: for some k$\}$ is some partial mapping

$$b \in \text{integers} \}$$

Proof of Condition 1 (α is monotone)

Assume

$$x_1, x_2 \in 2^{\text{envir}_{[1]}},\ \alpha(x_1) = \hat{s}_1,\ \alpha(x_2) = \hat{s}_2$$

$x_1 \leq x_2$

by the definition of this ordering relation

$x_1 \subseteq x_2$

by the definition of α

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

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

Assume

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

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

by the definition of γ

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

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

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

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

$b \in$ integers $\}$

by the definition of the ordering relation on $store_{[3]}$

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

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

therefore

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

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

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

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

$b \in$ integers $\}$

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

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

Proof of Condition 3 (mappings from abstract to concrete back to abstract preserve information)

Let

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

The goal is to prove that

$\hat{s}_1 = \hat{s}$

If it can be established that

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

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

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

There are two classes of basic types to consider in order to establish the above hypothesis.

Case 1: $t = \{v\}$ for some variable v

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

$$= \text{Type}(\{d : d \in \hat{s}(\{v\})\}) \qquad \text{by definition of } \gamma$$

$$= \text{Type}(\hat{s}(\{v\}))$$

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

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

Case 2: $t = V_n$

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

$$= \text{Type}(\{d : d \in \hat{s}(V_n)\}) \qquad \text{by definition of } \gamma$$

$$= \text{Type}(\hat{s}(V_n))$$

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

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

Proof of Condition 4 (mappings from concrete to abstract back to concrete only introduce uncertainty)

Assume

$$x \in 2^{envir_{pi}}$$

by the definition of α

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

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

by the definition of γ

$$\gamma(\alpha(x)) = \gamma(\hat{s}) = \{(s_1, h_1, b_1) :$$

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

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

$$\forall\, p_{n,k},\ h_1(p_{n,k}) = L \text{ where } L\text{: positive integers} \rightarrow \{v_{n,k} : \text{for some } k\}$$

$$b_1 \in \text{integers} \}$$

It is necessary to show that any element of x is in $\gamma(\alpha(x))$. By the above equation, $\gamma(\alpha(x))$ contains environments with all combinations of all possible heaps and balances; in particular, any heap and balance from x occur in these combinations. It must be shown that any store from x also occurs in these combinations.

Let

$$(s_1, h_1, b_1) \in x$$

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

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

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

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

therefore

$$(s_1, h_1, b_1) \in \gamma(\alpha(x))$$

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

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

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

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

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

and let the dereferencing function for Model 3 be

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

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

Lemma 1:

$$\forall\, \hat{s} = \alpha(x),\ x \in 2^{env_{[n]}}$$

$$Deref_3(\hat{s}, r_i) \supseteq Type(\{d : (s, h, b) \in x,\ d = Deref_1(s, r_i)\})$$

Proof

$$Deref_3(\hat{s}, r_i) = (\hat{s}(\{r_i\}) - variables) \cup \hat{s}(\hat{s}(\{r_i\}) \cap variables)$$

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

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

$$\hat{s}(\hat{s}(\{r_i\}) \cap variables) =$$

$$Type(\{d : v \in Type(\{d_1 : (s_1, h_1, b_1) \in x,\ d_1 = s_1(r_i)\}) \cap variables,\ (s, h, b) \in x,$$

$$d = s(v)\})$$

$$\supseteq Type(\{d : v \in \{d_1 : (s_1, h_1, b_1) \in x,\ d_1 = s_1(r_i)\}) \cap variables,\ (s, h, b) \in x,\ d = s(v)\})$$

$$= Type(\{d : (s, h, b) \in x,\ (s_1, h_1, b_1) \in x,\ s_1(r_i) \in variables,\ d = s(s_1(r_i))\})$$

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

$$Deref_3(\hat{s}, r_i) \supseteq Type(\{d : (s, h, b) \in x,\ d = s(r_i) \notin variables\}) \cup$$

$$Type(\{d : (s, h, b) \in x,\ s(r_i) \in variables,\ d = s(s(r_i))\})$$

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

$$= Type(\{d : (s, h, b) \in x,\ d = Deref_1(s, r_i)\})$$

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.

$$assign_1(x) = \{(s_1, h, b) : (s, h, b) \in x, s(r_1) \in variables, \forall\, v \in variables$$

$$s_1(v) = \begin{cases} s(v) & \text{if } v \neq s(r_1) \text{ and } v \neq r_0 \\ s(r_1) & \text{if } v = r_0 \\ Deref_1(s, r_2) & \text{if } v = s(r_1) \end{cases} \}$$

(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 assignments treated as strong updates (those with only one variable on the left hand side) and those treated as weak updates (those where 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

$$assign_3(\hat{s}) = \hat{s}_1$$

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

$$\hat{s}_1(t) = \begin{cases} \hat{s}(t) & \text{if } t \neq \{r_0\} \text{ and } t \not\subseteq t_1 \\ t_1 & \text{if } t = \{r_0\} \\ Deref_3(\hat{s}, r_2) & \text{if } t = t_1 = \{v\}, \text{ for some variable } v \neq r_0 \\ Deref_3(\hat{s}, r_2) \cup \hat{s}(t) & \text{if } t \neq \{r_0\}, t \subseteq t_1, \text{ and } \forall\, v, t_1 \neq \{v\} \end{cases}$$

for $t_1 = t_2 \cup t_3$ where $t_2, t_3 \in$ variable types

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

Condition 5 for assignment is

$$\forall\, x \in 2^{\text{environments}},\ \alpha(\text{assign}_1(x)) \leq \text{assign}_3(\alpha(x))$$

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

$$t \in \text{variable basic types}$$

are presented here. By the definition of α and assign_1

$$\hat{s}_1(t) = \text{Type}(\{d : v \in t, (s_1, h_1, b_1) \in \text{assign}_1(x), d = s_1(v)\})$$

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

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

Let

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

and let t_1 be the variable type in Model 3 on the left-hand-side of the assignment for the given set of environments, x, from Model 1:

$$t_1 = \hat{s}(\{r_1\}) \cap \text{variables} = \text{Type}(\{d : (s, h, b) \in x, d = s(r_1)\}) \cap \text{variables}$$

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

$$\hat{s}_2(t) = \begin{cases} \hat{s}(t) & \text{if } t \neq \{r_0\} \text{ and } t \not\subseteq t_1 \\ t_1 & \text{if } t = \{r_0\} \\ Deref_3(\hat{s}, r_2) & \text{if } t = t_1 = \{v\}, v \neq r_0 \\ Deref_3(\hat{s}, r_2) \cup \hat{s}(t) & \text{if } t \neq \{r_0\}, t \subseteq t_1, \text{ and } \forall\, v, t_1 \neq \{v\} \end{cases}$$

The proof of condition 5 is driven by the cases in the preceding formula.

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

t does not appear on the left-hand-side of the assignment in Model 3. First, show by contradiction that it does not appear on the left-hand-side in any store in Model 1.

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

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

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

$$= t_1$$

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

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

Applying this to the formula for $\hat{s}_1$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

t is the variable type on the left-hand-side of the assignment in Model 3 and it contains a single variable. First, show by contradiction that this variable is the only one that appears on the left-hand-side in any store in Model 1.

Suppose $\exists (s, h, b) \in x$, 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

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

Applying this to the formula for $\hat{s}_1$

$$\hat{s}_1(t) = \text{Type}(\{d : (s, h, b) \in x, s(r_1) \in \text{variables}, d = \text{Deref}_1(s, r_2)\})$$

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

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

Therefore, by Lemma 1

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

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

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

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

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

$$\text{Type}(\{d : v \in t, (s, h, b) \in x, s(r_1) \in \text{variables}, v = s(r_1), d = \text{Deref}_1(s,r_2)\})$$

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

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

$$= \text{Deref}_3(\hat{s}, r_2) \cup \text{Type}(\{d : v \in t, (s, h, b) \in x, d = s(v)\})$$

Therefore, by Lemma 1

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

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.

$$\text{list}_1(x) = \{(s_1, h_1, b) : (s, h, b) \in x,$$

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

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

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

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

for $t \in$ basic types

$$\hat{s}_1(t) = \begin{cases} \hat{s}(t) & \text{if } t \neq \{r_0\} \text{ and } t \neq V_n \\ P_n & \text{if } t = \{r_0\} \\ \hat{s}(V_n) \cup \bigcup_{i=1}^{m} \text{Deref}_3(\hat{s}, r_i) & \text{if } t = V_n \end{cases}$$

for $t_1 = t_2 \cup t_3$ where $t_2, t_3 \in$ variable types

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

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

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

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

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

$$\hat{s}_1(t) = \text{Type}(\{d : v \in t, (s_1, h_1, b_1) \in \text{list}_1(x), d = s_1(v)\})$$

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

$$d = \begin{cases} s(v) & \text{if } v \neq r_0 \text{ and } \forall\, i,\ 1 \leq i \leq m,\ v \neq v_{n,s(A_v)+i-1} \\ p_{n,s(A_p)} & \text{if } v = r_0 \\ \text{Deref}_1(s, r_i) & \text{if } 1 \leq i \leq m \text{ and } v = v_{n,s(A_v)+i-1} \end{cases} \})$$

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 $t \neq V_n$

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

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

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

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

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

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

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

$$= Type(P_n)$$

$$= P_n$$

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

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

Case 3: $t = V_n$

$$\hat{s}_1(t) = \text{Type}(\{d : (s, h, b) \in x, v_{n,s(A_\_)+i-1} \in V \text{ sub } n ,$$

$$d = \begin{cases} s(v_{n,s(A_\_)+i-1}) & \text{if } i < 1 \text{ or } i > m \\ \text{Deref}_1(s, r_i) & \text{if } 1 \le i \le m \end{cases} \})$$

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

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

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

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

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

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

Applying Lemma 1

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

This completes the proof of correctness for the framework of the abstract interpretation and for two operations in the language. Proofs for most other operations are similar.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1990.
2. K. Walker and R. E. Griswold, *Building and Installing the Icon Compiler*, The Univ. of Arizona Icon Project Document IPD158, 1991.
3. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.
4. R. Milner, "A Theory of Type Polymorphism in Programming", *Journal of Computer and System Sciences 17*, 3 (Dec. 1978), 348-375.
5. N. Suzuki, "Inferring Types in Smalltalk", *Eighth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1981, 187-199.
6. J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", *J. ACM 12*, 1 (Jan. 1965), 23-41.
7. 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.
8. 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.
9. M. S. Hecht, *Flow Analysis of Computer Programs*, North-Holland, New York, NY, 1977.
10. S. S. Muchnick and N. D. Jones, *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
11. 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.
12. P. Mishra, "Towards a Theory of Types in Prolog", *Proceedings 1984 Symposium on Logic Programming*, Montvale, NJ, 1984, 289-298.
13. 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.
14. 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.
15. K. Walker, *A Type Inference System for Icon*, The Univ. of Arizona Tech. Rep. 88-25, 1988.
16. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1985.
17. T. W. Christopher, *Efficient Evaluation of Expressions in Icon*, Unpublished Draft, Illinois Institute of Technology, 1985.

18. J. O'Bagy, *The Implementation of Generators and Goal-Directed Evaluation in Icon*, The Univ. of Arizona Tech. Rep. 88-31, 1988.

19. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

20. *American National Standard for Information Systems — Programming Language - C, ANSI X3.159-1989*, American National Standards Institute, New York, 1990.

21. G. R. Andrews and R. A. Olsson, et al.,*"An Overview of the SR Language and Implementation"*, *ACM Trans. Prog. Lang. and Systems 10*, 1 (Jan. 1988), 51-86.

22. J. L. Weiner and S. Ramakrishnan, "A Piggy-back Compiler for Prolog", *Proceeding of the 1988 Conference on Programming Language Design and Implementation, SIGPLAN Notices 23*, 7 (July 1988), 288-295.

23. J. Bartlett, "SCHEME->C a Portable Scheme-to-C Compiler", Research Report 89/1, Dec. Western Research Laboratory, Jan. 1989.

24. T. Yuasa and M. Hagiya, *"Kyoto Common Lisp Report"*, Research Institute for Mathematical Sciences, Kyoto University.

25. B. Stroustrup, *The C++ Programming Language*, Addison Wesley, Reading, MA, 1986.

26. S. C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, New Jersey, 1978.

27. M. E. Lesk and E. Schmidt, *Lex — A Lexical Analyzer Generator*, Bell Laboratories, Murray Hill, New Jersey, 1979.

28. D. H. D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, SRI International, Menlo Park, CA, Oct. 1983.

29. M. J. C. Gordon, *The Denotational Description of Programming Languages, An Introduction*, Springer Verlag , 1979.

30. J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, 1977.

31. D. Gudeman, "Denotational Semantics of a Goal-Directed Language", *ACM Trans. Prog. Lang. and Systems*, to appear.

32. H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

33. J. Rees and W. Clinger, et al.,*"Revised^3 Report on the Algorithmic Language Scheme"*, *SIGPLAN Notices 21, 12 (Dec. 1986),*.

34. J. F. Nilsson, "On the Compilation of a Domain-Based Prolog", *Information Processing*, 1983, 293-299.

35. 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.

36. J. Martinek and K. Nilsen, *Code Generation for the Temporary-Variable Icon Virtual Machine*, Technical Report 89-9, Department of Computer Science, Iowa State Universty, Dec. 1989.

37. K. Walker, *An Implementation Language for Icon Run-Time Routines*, The Univ. of Arizona Icon Project Document IPD79, 1989.

38. B. Prabhala and R. Sethi, "Efficient Computation of Expressions with Common Subexpressions", *Fifth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1978, 222-230.

39. S. K. Debray, Private Communication.

40. W. M. McKeeman, "Peephole Optimization", *Comm. ACM 8*, 7 (July 1965), 443-444.

41. W. A. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier Pub. Co., New York, 1975.

42. A. S. Tanenbaum, H. Staveren and J. W. Stevenson, "Using Peephole Optimization on Intermediate Code", *ACM Trans. Prog. Lang. and Systems 4*, 1 (Jan. 1982), .

43. R. E. Griswold and M. T. Griswold, *The Icon Analyst*, No. 1, Aug. 1990.

44. R. E. Griswold, *The Icon Program Library*, The Univ. of Arizona Tech. Rep. 90-7, 1990.

45. J. Kececioglu, Private Communication.

46. K. Walker, *A Stand-Alone C Preprocessor*, The Univ. of Arizona Icon Project Document IPD65a, 1989.