Tutorial on Adding Functions to the Icon Run-Time System

Kenneth Walker

Department of Computer Science, The University of Arizona

1. Introduction

Operations in the Icon run-time system are written in a special language called RTL [1]. This language is based on C but contains features to help the Icon compiler produce better executable code. While RTL was designed for the compiler, the Icon interpreter is now also built from the same run-time system coded in RTL. Anyone adding functions to either the compiler or interpreter writes them in RTL.

RTL code is translated into C by the program rtt. The actions performed by rtt vary depending on whether it is performing a translation for the compiler or for the interpreter. A translation for the interpreter is triggered by rtt's –x option and is reasonably straightforward.

RTL source files have a .r suffix (or .ri for included files). The translated C files for the interpreter have a suffix of .c with an x prepended to the file names to distinguish them from C files produced for the compiler. For example, the output of translating fmisc.r is xfmisc.c. These files are then compiled with a C compiler. See the Makefile that builds the interpreter for details of the entire translation process.

This tutorial introduces the features of RTL that are most often used in writing built-in functions. The same features are also used to create new keywords and operators, but adding these operations to Icon require changes to the translator, icont, and is beyond the scope of this tutorial.

2. Making Existing C Functions Available to Icon

One way to add functionality to the run-time system is to make C functions from existing libraries available to Icon as built-in functions. If the arguments and result types of these C functions consist only of integers, doubles (or floats), and strings, then adding them to Icon's run-time system is simple. A number of the Icon functions in the standard run-time system are created from corresponding C functions. Several of these are used as examples in this section.

In order to use existing C functions in Icon built-in functions, it is usually necessary to convert values between Icon representations and C representations. The simplest such built-in functions perform three actions:

- Convert arguments from Icon representations to C representations.
- Call the C function with the converted arguments.
- Convert the function result value from a C representation to an Icon representation.

Four parts of RTL are used when adding a C function to the Icon run-time system: a function header, type conversions, an abstract type computation, and embedded C code. Consider the existing built-in function sin(). It simply makes the C sin() function available to Icon.

A function header introduces the definition of the built-in function. The function header starts with function. This is followed by a specification of the length of the function's possible result sequences (this is only used by the compiler, but is required by rtt even when translating RTL for the interpreter). The sin() function produces exactly one result. A result sequence of length one is indicated by {1}.

As explained below, the Icon version of sin() terminates with an error message if the argument is of the wrong type. It might seem that error termination should be considered a result sequence of length zero. However, a zero-length result sequence is used only to indicate failure (in the Icon sense). Possible error termination must be ignored when determining the length of a result sequence.

The next item in the header is the name of the Icon function. It may be the same as the C function or it may be different. The final item is a comma-separated list of parameters enclosed in parentheses. The only form of parameter needed here is a simple identifier. The header for the sin() function is

```
function{1} sin(x)
```

The parameter x can be any Icon value. Even if it is of type real, it has a different representation than the corresponding C type, double. Type checking and conversion must be done before the C function is called with the value. RTL has several features that perform type checking and conversions. The conversion that is needed here has the form

cnv:*C–type*(*parameter*)

where *C-type* is C_integer, C_double, or C_string. C_integer indicates a C integer type (it is long rather than int on 16-bit machines), C_double indicates the C double type, and C_string indicates a null-terminated character string. This form of cnv performs the standard Icon conversions between strings, integers, reals, and csets, but the result is a C value. The conversion establishes a new scope for the parameter's identifier. Within this scope the identifier refers to the C value. The conversion does not appear by itself; it is used in an RTL if-then statement to check the success of the conversion. Typically, the success is checked with a negated conversion, as it is in the conversion for sin():

```
if !cnv:C_double(x) then
  runerr(102, x)
```

runerr() is an executable statement that results in a run-time error message; it automatically handles error conversion (that is, the conversion of an error condition to failure when &error is nonzero). The message is determined by looking up the error number in a table (see the file src/runtime/data.r). If a value is supplied, it is printed. For example, if sin() is called with a null value, the program terminates with the message

```
Run–time error 102
numeric expected
offending value: &null
```

Because runerr() is on the execution path for the failure of the conversion, it sees the original parameter x, which is an Icon value.

Abstract type computations are used to inform the compiler's type inferencing system of the effects of the function. While these effects can be complicated, particularly for functions that update Icon data structures, for the functions addressed here, they are very simple. They indicate the type of value returned by the built-in function and are of the form

abstract {
   return *icon–type*
   }

Where the *icon-type*s likely to be needed with pre-existing C functions are null, string, integer, and real. sin() uses real. Abstract type computations are not needed for the interpreter, but are required for the compiler and should be supplied for functions that might be used in the compiler.

The last feature of RTL needed for sin() is embedded C code. It takes the form

```
inline {
   C code with extensions
   }
```

inline indicates that the C code is reasonable for the compiler to put in-line in generated code. The interpreter makes no distinction between this and body that is used in examples below.

In RTL, the C return statement is replaced by statements that perform an Icon-style return, suspend, or fail. Some forms of these statements also do C-to-Icon type conversions. For many C functions all that is needed is a return statement that does such a conversion. These return statments have the form

```
return C–type expr;
```

The *expr* must produce a C value of the indicated type. In this example, *expr* is simply the call to the C sin() function:

```
return C_double sin(x);
```

Note that this use of x is in the scope of the type conversion and refers to a C value.

Putting these pieces together and preceding them by an optional comment in the form of a string literal results in the following declaration. Note that while semicolons are needed in inline clause because it is essentially C, in the pure RTL portion of the code, they are optional. When in doubt, it is always safe to use semicolons in places they would be used in C.

```
"sin(x), x in radians."
function{1} sin(x)
   if !cnv:C_double(x) then
      runerr(102, x)
   abstract {
      return real
      }
   inline {
      return C_double sin(x);
      }
end
```

Sometimes a straightforward implementation of a C function as an Icon built-in function is rather un-Iconish. For example, the C getenv function returns NULL if the requested environment variable does not exist. It makes sense for an Icon version of the function to fail under these circumstances rather than return the null value. The actual implementation of the getenv function is shown below. This function can either produce a result or fail, so it has a maximum result sequence length of 1 and a minimum result sequence length of 0. This is represented by {0,1} in the function header. Note the use of the extented C feature fail in the inline code.

```
"getenv(s) – return contents of environment variable s."

function{0,1} getenv(s)

   if !cnv:C_string(s) then

      runerr(103,s)

   abstract {

      return string

      }

   inline {

      register char *p;

      if ((p = getenv(s)) != NULL)        /* get environment variable */

         return C_string p;

      else                                /* fail if not in environment */

         fail;

      }
```

Another useful form of type conversion is def. This is used for supplying a default value. It has the form

def:*C–type*(*parameter*, *value*)

def is similar to cnv. However, if *parameter* is null, the conversion does not fail. Instead, *value* is used for the converted value. The following built-in function makes the Unix sleep() function available to Icon. It uses a default interval of 1 second. This function uses a return statement with no type conversion. The return statement's expression must evaluate to an Icon value, which is represented in C as a *descriptor* (see [2]). nulldesc is descriptor that always contains the null value.

```
function{1} sleep(n)

   if !def:C_integer(n, 1) then

      runerr(101, n)
   abstract {
      return null
      }
   inline {
      sleep((unsigned)n);
      return nulldesc;
      }
end
```

It is always a good idea to cast a C_integer to the desired C integer type in a C function call as demonstrated here.

The C sleep() function returns an integer. This could be used as the result of the Icon built-in function, but the following will not work in the compiler (it will work in the interpreter but is a bad programming practice):

```
inline {
   return C_integer sleep((unsigned)n); /* *** WRONG *** */
   }
```

This is because of a coding convention required by the compiler. This requirement is that expressions on return statements do not have side effects. If an Icon program ignores the result of the built-in function, the compiler may eliminate the entire return expression from in-line code, removing the side effect along with the result. An auxiliary variable must be used in cases like this.

The Icon built-in function atan() provides an example of a more complex function. It calls either the C function atan() or atan2() depending on whether a second argument is given. It uses a type check of the form

is:*icon–type*(*parameter*)

This type check performs no conversions. It simply checks to see if *parameter* is of the desired type. Like the type conversions, it is used in an if statement. The code for atan() is

```
"atan(x,y) — x,y  in radians; if y is present, produces atan2(x,y)."
function{1} atan(x,y)
   if !cnv:C_double(x) then
      runerr(102, x)
   abstract {
      return real
      }
```

```
    if is:null(y) then
        inline {
            return C_double atan(x);
            }
    if !cnv:C_double(y) then
        runerr(102, y)
    inline {
        return C_double atan2(x,y);
        }
end
```

3. Writing Operations From Scratch

Sometimes new Icon functions are needed that have no counterparts in a C library, or existing C library functions are too low-level to make reasonable Icon built-in functions. In these cases, more substantial built-in functions must be written.

Some facilities are best implemented as generators. Even if there are C functions that provide such facilities, they are, of course, not generators. As explained in Section 2, RTL includes an Icon-style suspend statement that is used to create generators. Like the Icon suspend expression, execution continues after the statement if the operation is resumed. The suspend statement comes in the same forms as the return statement. A result sequence of arbitrary length is indicated by {∗} in the operation header. The following code implements the seq() function.

```
"seq(i, j) – generate i, i+j, i+2∗j, ... ."

function{∗} seq(from, by)
    if !def:C_integer(from, 1) then
        runerr(101, from)
    if !def:C_integer(by, 1) then
        runerr(101, by)

    abstract {
        return integer
        }
```

```
      body {
         /*
          * Produce error if by is 0, i.e., an infinite sequence of from′s.
          */
         if (by == 0) {
            irunerr(211, by);
            errorfail;
            }

         /*
          * Suspend sequence, stopping when largest or smallest integer
          *  is reached.
          */
         while ((from <= MaxLong && by > 0) || (from >= MinLong && by < 0)) {
            suspend C_integer from;
            from += by;
            }
         fail;
         }
   end
```

Note the use of body instead of inline. When it is used, the compiler does not try to put the code in-line, but instead calls a function that contains the body code. Whether inline or body is used is a value judgement made by the programmer who writes the operation. The choice only affects the compiler, not the interpreter.

irunerr() is a function that acts like runerr(), except that its second argument is a C integer rather than a descriptor. In addition, it does not automatically convert errors into Icon failure when &error is nonzero. All it does is return to the code following its call. The Icon failure is accomplished by errorfail. errorfail acts like fail, but tells the compiler that failure only occurs when error conversion is enabled. errorfail is built into runerr() because runerr() is used extensively, but errorfail supplied as an separate feature of RTL for flexibility when runerr() is not appropriate. Note that there is also a drunerr() function, similar to irunerr(), that takes a C double as a value argument.

Some operations are polymorphous and take different actions based on the type of an argument. The type_case statement of RTL can be used for this. It similar to a C switch statement (or an Icon case expression), but selection is based on the type of an argument rather than a value. The built-in function type() makes use of it:

```
"type(x) – return type of x as a string."

function{1} type(x)

   abstract {

      return string

      }

   type_case x of {

      string:          inline { return C_string "string"; }

      null:            inline { return C_string "null"; }

      integer:         inline { return C_string "integer"; }

      real:            inline { return C_string "real"; }

      cset:            inline { return C_string "cset"; }

      file:            inline { return C_string "file"; }

      procedure:       inline { return C_string "procedure"; }

      list:            inline { return C_string "list"; }

      table:           inline { return C_string "table"; }

      set:             inline { return C_string "set"; }

      record:          inline { return BlkLoc(x)–>record.recdesc–>proc.recname; }

      co_expression: inline { return C_string "co–expression"; }

      default:

         runerr(123,x);

      }

end
```

See [2] for an explanation of the record entry.

Icon’s storage management system includes a garbage collector. This places constraints on other parts of the run-time system. When a garbage collection occurs, it must be able to locate all references to objects in the string and block regions, and to co-expressions. Operations must be careful to keep all such references *visible* at times when storage allocations (including malloc()s on some systems) might occur. It is important to note that storage allocations may occur while an operation is suspended.

RTL has a tended declaration that insures that a variable is visible to garbage collection. For example, a block pointer is tended by the declaration

```
tended union block *variable;
```

This declaration is used by the key() function while generating the keys of a table. The details of this code that deal with data structure manipulations are beyond the scope of this tutorial. See [2] for an explanation of the code that loops through the table. See [1] for an explanation of the abstract type computation. The implementation of key() function is

```
"key(t) – generate successive keys (entry values) from table t."

function{*} key(t)

   if !is:table(t) then

      runerr(124, t)


   abstract {

      return store[type(t).key]

      }


   inline {

      tended union block *ep;   /* tended since function suspends */

      register int i;


      for (i = 0; i < TSlots; i++) {

         for (ep = BlkLoc(t)–>table.buckets[i];ep != NULL;ep = ep–>telem.clink)

            suspend ep–>telem.tref;

         }

      fail;

      }

end
```

Other types of pointers, including pointers into the string region, can be tended. See [1] for a complete list along with declaration syntax to accomplish the tending.

4. Additional Notes

RTL files may contain definitions of ordinary C functions to use as support routines for operations. Several RTL features may be used in these otherwise ordinary C functions. These features include tended declarations, and type checking and conversions. Note that the usual convention is to put C functions in separate files from the operation definitions even when they use RTL features.

rtt contains a C-style preprocessor [3] and automatically includes several header files. src/h/grttin.h is the toplevel header file that gets included. The C compilation step also includes some header files; see src/h/rt.h. Because rtt can handle only standard C syntax and system header files on some platforms make use of C extensions, includes for these header files should not be put in .r files. They should be put in src/h/sys.h, which is not included until the C compilation step.

Information needed to write functions more complicated than those presented here can be found by studying [1], [2], and the source code for the run-time system. In particular, for writing many functions, it is necessary to have some understanding of Icon's data structures and its storage management system. Note, however, that malloc() and free() may be used as they are in ordinary C functions.

See Appendix F of [1] for adding new types to Icon.

5. Building Icon with New Functions

A new function is put in a file in the directory src/runtime. If a new file is created for the function, the Makefile must be updated. Note that the Makefile may be copied from elsewhere during the Icon configuration process; be sure to update the original if another configuration will be done.

The interpreter requires an entry for the new function in src/h/fdefs.h. The entry consists of a call to the macro FncDef() with the function name as the first argument and the number of parameters to the function as the second argument.

When the new function is in place and fdefs.h is updated, recompile icont and the runtime system. The runtime system consists of the interpreter iconx and, if Icon includes the compiler, rt.db and a link library. For the compiler, iconc need not be recompiled; this is because iconc dynamically determines from rt.db what functions are available in the link library.

References

1. K. Walker, *The Run-Time Implementation Language for Version 8.7 of Icon*, The Univ. of Arizona Tech. Rep. 92-18. July 1992.
2. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.
3. K. Walker, *A Stand-Alone C Preprocessor*, The Univ. of Arizona Icon Project Document IPD65a, 1989.