Adding New Types to Version 8.7 of Icon

Kenneth Walker

Department of Computer Science, The University of Arizona

1. Introduction

This document describes how to add new types to Icon. It deals with simple types, aggregate types whose values contain other Icon values, and keywords that are variables. These are the kinds of types that are most likely to be added to the language. Executable types such as procedures and co-expressions are beyond the scope of this document as are types that require special representations and variable types that have special dereferencing semantics.

When referring to files in the Icon implementation, this document gives path names relative to the Icon source directory, src. For example, if Icon is installed in /usr/icon/v8 then the file common/filespec.txt has a full path name of /usr/icon/v8/src/common/filespec.txt

2. The Implementation of Icon Types

An Icon value is implemented as a two-word *descriptor* containing type information and value information [1]. The first word of a descriptor is the *dword*. For the types discussed here, the dword contains a unique code that identifies the type (see [1] for types that have additional information in the dword). The second word of the descriptor is the *vword*; it contains the actual value or a reference to the value.

Actual values that are too large to fit in one word are usually put in the block region. This region is controlled by a storage management system that includes a garbage collector. The garbage collector is driven by information in arrays indexed using the type codes associated with the blocks. The block region contains values for both simple and aggregate types.

There are several other parts of the run-time system besides the garbage collector that need information about Icon types. Some are Icon operations such as the type() function, while others are automaticly invoked features such as error trace back. These are described in more detail below. Types, of course, typically have operations associated with them that create and use values of the type; see [2] and [3] for instructions on adding new operations to Icon.

3. The Type Specification System

The Icon run-time system is written in RTL; this language is a superset of C with extensions specifically designed for implementing Icon [2]. Icon types are used in several places in RTL and new types must be added to this language. These uses include type checking constructs, return/suspend statements, and abstract type computations. In addition, the Icon compiler needs information about types in order to perform type inferencing. These requirements are satisfied with a type specification system.

This system is a simple declarative language for naming types and describing some of their properties. Information from the type specification system is incorporated in rtt (the translator for RTL) and in iconc when they are built.

All types specified by the system may be used in the RTL is and type_case constructs. They may also be used in abstract type computations that describe the type behavior of an operation to iconc's type inferencing system.

Aggregate types may be used in a new type expression in an abstract type computation. A type specification may optionally indicate that RTL supports a special form of return/suspend statement that constructs a return value, in the form of a full descriptor, from a C value for the vword of the descriptor.

Type specifications are in the file common/typespec.txt. Comments in the file start with # and continue to the end of the line. This file is translated into a C header file by the program typespec. This is not part of the normal Icon build process; entries at the end of common/Makefile must be uncommented if typespec.txt is updated.

A type definition in the specification system has the form:

type-def ::= *identifier opt-abrv* : *kind opt-return*

where *identifier* is the name of the type and *opt-abrv* is an optional abbreviation for the type name. The abbreviation has the form

opt-abrv ::= *nil* |
{ *identifier* }

The abbreviation is used in tracing type inferencing and other places where a compact notation is desired. If no abbreviation is given, the full type name is used.

There are three *kind*s of types: simple, aggregate, and variable. Their syntax and usage are described in separate sections below. *opt-return* indicates optional RTL return/suspend support for the type. The four types of vwords supported by this construct are introduced below as needed. Appendix A contains a complete grammar for the specification language.

4. Simple Value Types

Types with a *kind* clause of simple are simple in the sense that values of the type do not have components that contain other Icon values. These types may otherwise have sophisticated semantics.

There are three ways to implement the values of a type: encode them as C integers (these are guaranteed to be at least 32 bits long), implement them as blocks in the block region, or implement them in storage allocated using malloc() (in theory values can also be put in the string region, but it is only well suited for Icon strings; descriptors pointing into this region must have a special form). The choice of implementation determines the type of C value stored in the vword of the descriptor representing the Icon value.

The dword of a descriptor for one of these types contains a fixed code. It consists of a small integer type code along with flags describing characteristics of the descriptor. The small integer is represented by a preprocessor constant defined in h/rmacros.h. Its name is created by capitalizing the first character of the type name and prepending T_. For example, the definition of the type code for the cset type is

```
#define T_Cset 4
```

These definitions must be manually added to h/rmacros.h and the constant MaxType must be updated if the integer exceeds the current value of MaxType.

There are corresponding constants with names beginning with D_ that include both the T_ type code and the flags required for the type. For the types dealt with here, the flags in the constant D_Typecode must always be included, and if the vword points to storage under control of the garbage collector, the flag F_Ptr must be included. The definition of the D_ constant for the cset type is

```
#define D_Cset (T_Cset | D_Typecode | F_Ptr)
```

These must also be manually added to h/rmacros.h.

Three of the *opt-return* type specification clauses are useful for implementing value types (the fourth is used for variable types; see below). These clauses add return/suspend statements to RTL of the form

return *type–name*(*expr*)
suspend *type–name*(*expr*)

type-name is the identifier naming the type. It determines the D_ constant used for the dword of the operation's result descriptor. *expr* is a C expression whose value is placed in the vword of the result. The particular *opt-return*

clause chosen determines how the C value is stored in the vword. The clauses are

```
return C_integer
return block_pointer
return char_pointer
```

C_integer indicates that the value is cast to a C integer; see the definition of word in h/typedefs.h for the exact C type used. block_pointer indicates that the value is cast to (union block *); this is usually used for pointers to blocks in the block region. char_pointer indicates that the value is cast to (char *). Note, only descriptors of a special form may point into the string region; the storage used with return char_pointer must reside elsewhere.

As an example, the type specification for the cset type is

```
cset(c): simple
        return block_pointer
```

Suppose a variable cp within an Icon operation written in RTL points to a cset block. Then the statement

```
return cset(cp);
```

constructs a result descriptor for the cset and returns it.

For a type with an associated block, a declaration for the block structure must be added to h/rstructs.h. By convention, the structure name is created by prepending b_ to the type name. The first word of a block must contain its T_ type code. If different instances of the block may vary in size, the second word of the block must contain this size in bytes. The structure name of the new block must be added to the union block declaration in h/rstructs.h. An allocation routine for the block must be added to runtime/ralc.r. The macros AlcFixBlk() and AlcVarBlk() are useful in such routines; see other allocation routines for guidelines.

There are five arrays in runtime/rmemmgt.r that must be updated for all types. These are used by garbage collection and diagnostic routines. The array bsizes contains the sizes of the blocks for corresponding T_ type codes. An entry of −1 indicates a type for which there is no block. An entry of 0 indicates a block whose second word contains the size.

It is assumed that values of simple types implemented in the block region consist of single blocks containing no descriptors and no pointers to other blocks. Therefore for simple types, the arrays firstd, firstp, and ptrno, contain 0 for types with blocks and −1 for types with no blocks. More complicated implementations are discussed in the next section.

The array blkname contains strings used to identify types for use by debugging routines.

Storage for the values of a type usually should be allocated in the block region. However, for interfaces to packages written in C, it may be necessary to use storage that is not relocated by garbage collection. While it is possible to place types allocated with malloc() under control of garbage collection, this is complicated and beyond the scope of this document. See the implementation of co-expressions for an example of how this can be done. Alternatives are to ignore the storage leakage caused by unfreed storage or provide an Icon function, along the lines of close(), that explicitly frees storage associated with a value.

Three built-in functions must be updated to handle any new type. copy() and type() are in the file runtime/rmisc.r. image() is updated by changing the support routine getimage() in the file runtime/rmisc.r. If a type has a logical notion of size, then the size operator in runtime/omisc.r must be updated.

Several other support routines must also be updated. outimage() in the file runtime/rmisc.r produces the images of values for diagnostics and tracing. order() determines the collating sequence between types, anycmp() determines the collating sequence between any two values, and equiv() determines whether two values are equivalent (it is only updated for types, such as cset, for which a simple descriptor comparison is not adequate to determine equivalence). These routines are in the file runtime/rcomp.r.

Appendix B contains a check list of files that must be updated when a type is added to Icon.

5. Aggregate Types

Aggregate types have values with components that are other Icon values. The aggregate type specification provides more sophisticated RTL abstract type computations for the type. These in turn allow iconc to produce code that is better optimized.

For interpreter-only implementations, abstract type computations are not used and are optional in RTL code; the simple type specification may be used in that case. However, the discussion later in this section on block layout and on the storage management arrays still applies.

The *kind* clause of an aggregate type specification establishes and names abstract components for the type. The clause is of the form

kind ::= aggregate(*component*, ...)

component ::= *identifier* |
var *identifier opt–abrv*

Note, the *opt-return* clauses discussed in the previous section may be also used with aggregate types.

The aggregate specification can be thought of as establishing a sort of "abstract type record" whose fields, the abstract components, summarize the type information of the actual components of values in the type. Most types are give one abstract component. For example, the set type has the specification

```
set(S): aggregate(set_elem)
        return block_pointer
```

where set_elem represents all the elements of a set.

Abstract components can be accessed using dot notation, and the new abstract type computation can be used to establish a new subtype of the type (subtypes only exist internally in the compiler and have no existence at the Icon language level). A subtype can be returned by the operation and has its own component types independent of subtypes created elsewhere. The abstract type computation for set intersection uses both dot notation and a new expression. It performs intersection in the abstract type realm. x and y are the parameters of the operation and may contain different subtypes of the set type:

```
return new set(store[type(x).set_elem] ** store[type(y).set_elem])
```

(Note that the components can be thought of as references to information contained in a *type store* – thus the indexing notation.)

Components that represent Icon variables are preceded by var and may be given abbreviations for use in tracing type inferencing. For example, the list type has the specification

```
list(L): aggregate(var lst_elem(LE))
         return block_pointer
```

These components may be returned from operations and represent the component as a variable. For example, the abstract type computation for element generation operator when applied to a list is

```
return type(dx).lst_elem
```

where dx is the parameter of the operation. When a value rather than a variable is returned, the component must be "dereferenced" by indexing into the store, as in the abstract type computations of get():

```
return store[type(x).lst_elem]
```

Non-variable components must always be dereferenced.

For types, such as tables, that contain Icon values serving different purposes, it may be effective to establish several abstract components.

Aggregate types are implemented using blocks that contain descriptors, and they may be implemented using several kinds of blocks, with some blocks having pointers to others. When there are multiple blocks, there is always a *header* block that uses the T_ code of the type. Other blocks are given internal type codes; these codes must be

added to h/rmacros.h and entries must be made in the storage management arrays.

Any descriptors in a block must be at the end. The type's entry in the firstd array is the location of the first descriptor. Any block pointers in the block must be contiguous. The type's entry in the firstp array is the location of the first pointer and its entry in the ptrno array is the number of pointers.

6. Keyword Variable Types

Keyword variable types have a type specification with a *kind* clause of the form

kind ::= variable *var-type-spec*

var-type-spec ::= initially *type* |
always *type*

type ::= *type-name* |
type ++ *type-name*

type-name ::= *identifier*

The compiler must be able to infer the types of values stored in a keyword variable. The initially option causes the keyword variable type to be treated as a set of global variables, each initialized to the given type specified by the *type* clause. The always option indicates that the keyword always contains values of the given type and the compiler does no actual inference on it. *type* may be the union of several types; this indicates that the type is uncertain and may be any of the ones specified. A special *type-name*, any_value, indicates complete uncertainty. The clause

```
always any_value
```

is a correct, although entirely imprecise, description of any keyword variable.

It is assumed that keyword variables are implemented by global descriptors (though other techniques are possible). The *opt-return* clause of the form

```
return descriptor_pointer
```

is useful for implementing keyword variables. The vword of a result descriptor from a corresponding return/suspend expression is of type struct descrip *.

Some of the same files must be updated for variable types as for value types. Type codes must be added to h/rmacros.h. The D_ code must have the F_Var flag set, for example:

```
#define D_Kywdint (T_Kywdint | D_Typecode | F_Ptr | F_Var)
```

The storage management tables and the outimage() routine also must be updated.

Other updates are unique to variable types. The global descriptor must be established. runtime/data.r contains its declaration, icon_init() in runtime/init.r initializes the descriptor, and h/rexterns.h contains an extern for it. The keyword itself goes in runtime/keyword.r. Dereferencing must be updated; it is performed by deref() in runtime/cnv.r. Assignment must be updated; it is handled by the macro GeneralAsgn() in runtime/oasgn.r. The name() function is updated by changing the support routine get_name() in runtime/rdebug.r. The variable() function is updated by changing the support routine getvar() in runtime/rmisc.r.

If the descriptor may contain a value under control of garbage collection, the support routine collect() in runtime/rmemmgt.r must be updated. postqual() preserves references to the string region; the macro Qual() is used to check for such references. markblock() preserves references to blocks; the macro Pointer() is used to check for such references.

References

1. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.
2. K. Walker, *An Implementation Language for Icon Run-Time Routines*, The Univ. of Arizona Icon Project Document IPD79, 1992.
3. K. Walker, *A Tutorial on Creating Run-Time Operations for the Icon Compiler*, The Univ. of Arizona Icon Project Document IPD164, 1991.

Appendix A: Grammar for the Type Specification System

type-def	::=	*identifier opt-abrv* : *kind opt-return*
kind	::=	simple |
		aggregate(*component*, ...) |
		variable *var-type-spec*
component	::=	*identifier* |
		var *identifier opt-abrv*
var-type-spec	::=	initially *type* |
		always *type*
type	::=	*type-name* |
		type ++ *type-name*
type-name	::=	*identifier*
opt-abrv	::=	*nil* |
		{ *identifier* }
opt-return	::=	*nil* |
		return block_pointer |
		return descriptor_pointer |
		return char_pointer |
		return C_integer

Appendix B: Check List

All Types

- ☐ common/typespec.txt – add type specification
- ☐ common/Makefile – uncomment entries near the end of the file
- ☐ h/rmacros.h – add T_*Type* macro
- ☐ h/rmacros.h – add D_*Type* macro
- ☐ runtime/rmemmgt.r – bsizes table
- ☐ runtime/rmemmgt.r – firstd table
- ☐ runtime/rmemmgt.r – firstp table
- ☐ runtime/rmemmgt.r – ptrno table
- ☐ runtime/rmemmgt.r – blkname table
- ☐ runtime/rmisc.r – update outimage()

All Value Types

- ☐ runtime/fmisc.r – update copy()
- ☐ runtime/fmisc.r – update type()
- ☐ runtime/rcomp.r – update anycmp()
- ☐ runtime/rcomp.r – update order()
- ☐ runtime/rcomp.r – update equiv()
- ☐ runtime/rmisc.r – update getimage()

Types Implemented In The Block Region

- ☐ h/rstructs.h – add declaration for the block structure
- ☐ h/rstructs.h – update the union block declaration
- ☐ runtime/ralc.r – add an allocation routine

Types With Sizes

- ☐ runtime/omisc.r – update size operator

All Keyword Variable Types

- ☐ h/rexterns.h – extern for keyword descriptor
- ☐ runtime/cnv.r – update deref()
- ☐ runtime/data.r – declaration for keyword descriptor
- ☐ runtime/init.r – initialize keyword descriptor
- ☐ runtime/keyword.r – add keyword declaration
- ☐ runtime/oasgn.r – update GeneralAsgn() macro
- ☐ runtime/rdebug.r – update get_name() (two versions)
- ☐ runtime/rmisc.r – update getvar() (two versions)

Keyword Variables That Must Be Garbage Collected

- ☐ runtime/rmemmgt.r – update collect()