A P R I L 1 3 , 2 0 1 0

BLOCKS PROPOSAL, N1451

BLAINE GARST

APPLE INC.

blaine@apple.com

Introduction

Apple Inc. first shipped a *blocks* extension to C in its Mac OS X 10.6 SnowLeopard product. It is implemented in both gcc 4.2 as well as Clang/LLVM 1.6 compilers, both of which are open-source under varying license conditions (the Clang/LLVM license being generally considered far less restrictive). The blocks runtime has been available since June 2009 as part of LLVM and has been back-ported to the Windows, FreeBSD, linux, and previous versions of Mac OS X by third parties. It was critical to the infrastructure work that comprised Mac OS X 10.6 SnowLeopard in that many new APIs using blocks were introduced and incorporated in system level software.

In addition to C, blocks are also available in Objective-C (with suitable extensions for garbage collection and direct messaging) and C++ (with extensions for const copied on-stack objects), and the hybrid language Objective-C++.

Blocks are a form of closures which were introduced in the Scheme dialect of LISP circa 1973. They take their name in Apple's product from Smalltalk, where the C language *block* was turned into a Smalltalk *object* and many flow control ideas were expressed as Smalltalk messages. In Objective-C Blocks are also Objective-C objects, and since this is for Apple the primary language for developers the Smalltalk name was chosen in homage to many other Smalltalk styled features of the Objective-C language.

In C, however, the term *block* is already well-defined. For the purposes of this document it is intended that Apple's *block* idea be described as *closures* to avoid the confusion which would otherwise ensue. This document does honor Apple's existing naming practice by describing the feature externally as Blocks and by using "block" labeled keywords as is currently practiced, since current practice is the standard by which the C Standard adopts proposals, and it is the syntax that would be used to examine existing code written to this practice. Naming is ultimately the choice of the entire C Standard committee of course, and whereas this document uses **__block**, **Block_copy**, and **Block_release**, it may well be the case that **_Closure**, **closure_copy**, and **closure_release** or other names will ultimately be approved.

WG14 documents may be found at http://www.open-std.org/JTC1/sc22/wg14/www/documents. Document N1270 "Apple's Extensions to C" provides an overview of the construct and is considered a **prerequisite** to this document.

Overview

More formally, there are four concepts that need to be introduced to the C language to accommodate closures.

First, there is a new form of compound type exactly paralleling that of function types, and that is of course closure types. Closures are function expression objects of these closure types. Closures objects are, essentially, a new form of scope providing equivalent access to *automatic* duration identifiers under highly constrained conditions. One primarily declares pointers to closure objects and uses the syntactic form of pointers to functions exactly with the one substitution of ^ for * when describing the pointer to closure aspect.

Second, there is a new closure literal construct that creates a pointer to a closure object. The object is of indeterminate size. It captures automatic variables in a const form, references a specialized function that knows how to access these captured variables, and also captures references to mutable variables declared directly in a new storage class known as __block. The most minimal closure literal is ^ { }.

Third is the new storage class __block representing essentially closure object "scope" duration. For efficiency closure objects and hence closure scoped identifiers are specified such that they may start as being implemented in automatic storage and, upon being preserved by explicit programmer action via the Block_copy primitive, move into allocated storage. Because they may move the & operator is forbidden upon these, much like it is forbidden for *register* duration objects.

Forth are the language primitives Block_copy and Block_release which manage the preservation of closure objects beyond their scope of declaration.

Rationale

Closures are an ideal form of a succinct description of units of work that are easily and trivially farmed off to anonymous threads of execution once copied. The predominant form of a closure rarely uses __block storage and as such is an immutable object that can be accessed by multiple threads safely in a trivial fashion in essentially an asynchronous form. Synchronous uses, say for sorting or searching, are cheap because they need never leave the stack, and the __block form is generally safe because there is no parallelism implied.

Modifications

The following are the more formal descriptions of changes necessary to the **N1425** proposed draft standard to reflect the preceding high level descriptions of blocks. Due to the parallel type system to functions there is a significant burden to describe these succinctly without simply copying and pasting the descriptions for *function* with *closure* substituted. The following changes are minimal but hardly sufficient, and it is recognized that significant additional "wordsmithing" will be required. It is, however, intended to be complete.

1. Section **6.2.1**

An identifier can denote an object; a function; a closure; a tag..

Add closure to paragraph 2: "five kinds of scopes, function, file, block, closure, and function or closure prototype.

At end of paragraph 2: "Similarly for closure prototypes"

Para 4: in a function definition or closure literal,...

...function or closure prototype...

Add paragraph:

As a special rule, the block associated with a closure imports identifiers from outer scopes of storage duration automatic as an implicitly declared const identifiers initialized with their value at the point of closure expression evaluation.. These implicit const

identifiers are of __block storage duration, e.g. can outlive the defining scope of the closure literal upon Block_copy of that closure.

2. Section 6.2.4

(p37) 1. There are five storage durations: static, thread, automatic, __block, and allocated.

The lifetime of __block storage identifiers is minimally that of the function, closure, or block of origin, or as extended by a Block_copy of a closure that references said __block storage identifier. Conforming implementations may choose to always allocate storage for __block identifiers from allocated storage, or start them in automatic storage and migrate them to allocated storage upon the first Block_copy. Like register, __block storage identifiers are forbidden the use of the & (address) operator. (see forward reference).

3. Section 6.2.5, add closure types

object types (types that fully describe objects), function types (types that describe functions), closure types (types that describe closures), and incomplete types

...

20. Any number of derived types can be constructed from the object, function, **closure**, and incomplete types

- Function and **closure** types describe functions and closures with specified return type....

- A pointer type may be derived from a function type, **a closure type**, an object type

- A closure type describes an opaque *object* of indeterminate size that references a *function*, and it is the function type that characterizes the closure type. The opaque object captures constant (const) copies of all identifiers of automatic storage as identifers of __block storage duration, and references to all identifiers of __block storage duration

referenced recursively in all closure literals defined within the closure literal. A closure type is considered incomplete from the perspective of size determination.

Para 24: Array, function. **closure**, and pointer...

Para 26. Any type so far mentioned...[[[this should move to become para 20 I think]]] there is no const/volatile/restrict function (or closure) type.

4. Section 6.3.2.3

A pointer to a function **or closure** of one type...

Pointers to closures are not equivalent to pointers to functions, even if their types are apparently the same, and undefined behavior will result if a function or closure call is performed upon an incorrect type.

5. Section 6.4.1,

Add __block, Block_copy, Block_release to the list of keywords

6. Section 6.5

or that designates an object or a **closure** or a function

7. Section 6.5.1

An identifier is a... or a function (...) or a closure reference.

8. Section 6.5.2

postfix-expression:

^ closure-decl-***opt*** compound-statement-body

closure-decl:

(argument-expression-list)

type-expression

[Closure literals are introduced by a unary use of the ^ operator, and result in a *pointer to a closure object*. The return type of the closure is generally inferred by the presence

and return type of any return statements, or void if no return statements are made. Closures that have a nullary argument list may specify it as such by either declaring it as (void) or by omitting the (void) entirely. Closures may explicitly provide a full return type and argument lists using a primary expression, which, with explict return type specified, will force normal return "conversions" to that type]

[^ char (int x) { return x; } is a closure that returns a char given an int argument]

[^ (int x) { return x; } is a closure that returns an integer given an integer argument]

[^ { printf("hello closure\n"); } is a closure returning void given no arguments]

[^(void) { printf("legal\n"); } is an closure taking no arguments returning void]

[^() { printf("illegal\n"); } is an illegal closure; empty argument list is not allowed]

[A closure's compound statement block does not share control flow with enclosing scope. Only a return statement is allowed.]

9. Section 6.5.2.2

Function and **closure** calls

The expression that denotes the called function (or closure) shall have type pointer to function (or closure) [generically, need to replace references to "function" with "function (or closure)". This is a presentation choice.]

Semantics

A closure call upon a closure object has the meaning of calling the function referenced within the closure object, and that the contents of the closure object be provided to the function such that uses of identifiers from an enclosing scope used within the compound-statement-body of the closure are to the captured const copy of that identifier held within the closure object. References to __block storage duration identifiers must be implemented in such a fashion that all such references across the declaring scope and those from closures are to the same object.

[Note: if the closure object is a structure, a pointer to that structure could be passed as a hidden argument to the function otherwise specified by the closure, and uses of captured identifiers are replaced in the closure function body to elements of a closure *structure* object.]

10. Section 6.5.3 Unary operators

unary-expression:

Block_copy (*closure-expression* **)**

Block_release (*closure-expression* **)**

Semantics

Block_copy is used to assure the lifetime of a closure object beyond its possible initial automatic storage lifetime. Block_copy "returns" an equivalent (possibly the same) closure object of the same type. Block_release is used to recover allocated storage for a closure object. They are intended to be used in pairs. An unmatched use of Block_release results in undefined behavior.

11. Section 6.5.3.2

... not declared with the register or __block storage-class specifier.

12. Section 6.7.1

Add __block to the list of storage-class-specifier

Constraint

__block can only occur within a compound statement body, e.g. only where auto is allowed. It is implementation defined as to whether such storage is implemented using *allocated* duration memory, or *automatic* duration memory and promoted to *allocated* upon Block_copy.

One may not designate __block storage duration class for arrays. [this is because array backing storage are not first class objects].

13. Section 6.7.6

pointer:

^ type-qualifier-listopt

^ type-qualifier-listopt pointer

14. Section 6.7.6.3 “Function and Closure declarators”

The ^ token is used to distinguish pointer to closure and is allowed only where pointer to function would be.