gafter.blogspot.com

A Definition of Closures

13–16 minutes

There has been some confusion over our proposal to add closures to the Java Programming Language. After all, doesn't Java already have closures in the form of anonymous inner classes? What is the point of adding something to the language that it already has? To some there appears to be a lot in the proposal that has nothing to do with closures, including the control invocation syntax, `null` as a type, `Unreachable`, `throws` type parameters, function interface types, and "nonlocal" returns. In my Javapolis talk I tried to give an explanation for why these features are in the proposal from the practical point of view of what kinds of things would be possible that were not formerly possible. But that begs the question: why do we call it "Closures" for Java? In this blog post I'll try to show how the definition of closures relates to the features of the proposal, and identify which features of the proposal do (and which do not) result from the definition.

Before discussing the definition of closures, it helps to understand the historical context in which the term was introduced.

Lisp was created in the late 1950's by John McCarthy and others at M.I.T. One feature of the language was function-valued expressions, signified by *lambda*. The name "lambda" was borrowed from a mathematical formalism known as the lambda calculus. Although Lisp was not based on an effort to model that formalism, lambda plays approximately the same role in Lisp as it does in the lambda calculus: lambda is the syntax for a function-valued expression. McCarthy's intent was that Lisp should be designed to be implemented very efficiently, ideally compiled. That desire for efficiency influenced the design of the language.

Lisp used something called *dynamic scoping*. Logically, in a dynamically scoped language, when a variable reference is evaluated the runtime looks up the call stack until it finds a scope in which a variable of that name is defined. But as a practical matter variable references in a dynamically scoped language can be resolved in constant time simply by maintaining a value cell for each variable name; that value cell caches the variable's current definition. Dynamic scoping is easy to implement in an interpreter or compiler. Some very clever people had found ways to not only take advantage of dynamic scoping, but had developed what would now be thought of as programming patterns that depended deeply on it. But it was soon discovered that dynamic scoping suffered subtle problems, something the Lisp community called the *FUNARG problem*.

Now we fast-forward to the mid 1970's. On the radio you would hear[1] Elton John, Emerson Lake & Palmer, Joni Mitchell, The Captain and Tennille, John Denver, Paul Simon, Paul McCartney and Wings, ABBA, David Bowie, Janis Ian, Aerosmith, Fleetwood Mac, Heart, and Queen. A number of popular Lisp dialects were in use including InterLisp, MacLisp, UCI-Lisp, Stanford Lisp 1.6, and U. Utah's Standard Lisp. All of them were dynamically scoped. It was in this context that Guy Steele and Gerald Jay Sussman developed *Scheme*, a very simple Lisp dialect.

One thing about Scheme was different[2]. Scheme was *lexically scoped*, like the lambda calculus and most mathematical notations, which means that a variable reference *binds* to the *lexically* enclosing definition for that name that was active at the time the enclosing lambda form was evaluated. To explain the semantics in terms of the implementation, evaluating a lambda expression was said to produce a ***closure***. This is a function value represented as an object that contains references to the current bindings for all the variables used inside the lambda expression but defined outside it. These are called the *free variables*. When this closure object, or *function*, is applied to arguments later, the variable bindings that had been captured in the closure are used to give meaning to the free variables appearing in the code. The term closure describes more than just the abstract language construct, it also describes its implementation.

To many in the Lisp community at the time, it didn't make sense to adopt a Lisp dialect with closures. Not only would it undermine common programming techniques but it would obviously be much less efficient. For a short time these issues were debated, and Guy Steele wrote a series of papers entitled *Lambda the Ultimate* ______ (where ______ is *Imperative*, *Declarative*, *GOTO*, or *Opcode*) to help explain the power of lexically scoped lambda (closures). Fast forward only a few years and the debate was largely settled: lexical scoping is Right and dynamic scoping is Wrong and we've all learned our lesson. Since that time the word *closure* is used to mean lexically scoped anonymous function, but the connotation is that it is possible to get the semantics wrong for any number of reasons, including bugs and concerns about implementation efficiency. It also hints that we should let the language design drive the implementation, not the other way around. Virtually every programming language, whether or not it has something like lambda and anonymous function values, uses lexical rather than dynamic scoping. The basic definition of a closure, however, shows its Lisp roots:

> A *closure* is a *function* that captures the *bindings* of *free variables* in its *lexical context*.

Around this time, Smalltalk was introduced. Smalltalk is the most pure and simple of the object-oriented languages: everything is an object. Object-oriented languages add a twist to lexical scoping. Rather than binding all names in the lexical scope, free variables appearing in *methods* are bound in the scope of the object that the method is a member of. In other words, names in a method are bound to members of the "current" object. The current object is accessible by the name "self". Another small but interesting detail is that you can return early from a method in Smalltalk using the syntax "^*expression*". We'll return (no pun intended) to the significance of this fact later.

Methods aren't the only kind of code abstraction in Smalltalk. There is also an expression form for writing a *block* expression, which is essentially a lambda. Early dialects had limitations on them, but most modern Smalltalks do not. They are a true analog to Scheme's lambda. Free variables in a Smalltalk block are bound in the enclosing scope, which is typically the scope of some enclosing method. The result of evaluating a block expression is a closure, and like everything else it is an object. In this case the object has a method that you use to invoke the code of the block.

Anonymous functions (closures) were not blindly introduced into Smalltalk just because it seemed like a neat idea, or because they had worked out well in another language. Rather they were integrated fully and carefully into the language. Anonymous functions can properly be integrated into even an existing language, but there is an advantage when adding them early. As Guy Steele's papers demonstrated, they are so powerful that they subsume other language features. If you add them early, you might save yourself the trouble of adding language features that can instead be added as libraries. Smalltalk provides few control constructs directly in the language. Even the conditional "if" is provided as a library method and invoked using blocks.

Two things distinguish blocks in Smalltalk from Scheme's lambda. First, the meaning of "self" within a block refers to whatever meaning it had in the enclosing context. Specifically, it doesn't refer to the closure object itself. Second, the syntax for returning from a method, "^*expression*", returns from the enclosing method; it doesn't return from the method representing the closure invocation. These two details are a natural consequence of the fact that, while Scheme has only one lexically scoped language construct (variable bindings), Smalltalk has three lexically scoped language constructs: name bindings (like Scheme), the referent of the return syntax, and the meaning of "self". The definition of closures above mentioned only "the bindings of free variables", but that is because the definition was written for the language Scheme, and name (variable) binding is the only lexically scoped construct in Scheme. Common Lisp also has "return" and "goto", and these too are captured lexically in a closure. In order to realize the full power of closures, described in Guy Steele's lambda papers, they must capture all lexically scoped language constructs. Generalizing the definition of closure to cover other languages would require using more language-neutral terminology: instead of "bindings of free variables" we would have something like "lexically scoped semantic language constructs." However, that obscures the origins of the term.

Fast forward more than 25 years, and we're once again listening to some of the same music we listened to in the late 1970's. We are now considering adding closures to Java, a significantly more complex language than either Scheme or Smalltalk. We're not considering them because they seem like a neat idea, or because they worked out well in other languages, or because we're bored. Rather we're considering them: because of the power and flexibility they will add to the programmer's arsenal; because of the improved readability we expect from programs that use closures instead of the existing alternatives; and because of a number of other recently proposed language extensions that will be unnecessary if closures are added. In order to get the full power of closures, they should capture all lexically scoped semantic language constructs. What are the lexically scoped language constructs in Java?

- The meaning of *variable names*.
- The meaning of *method names*.
- The meaning of *type names*.
- The meaning of `this`.
- The meaning of names defined as statement labels.
- The referent of an unlabelled `break` statement.

- The referent of an unlabelled `continue` statement.
- The set of *checked exceptions* declared or caught.
- The referent of a `return` statement.
- The *definite assignment* state of variables.
- The *definite unassignment* state of variables.
- The *reachability state* of the code[3].

  In addition, Java has one other significant difference from either Scheme or Smalltalk: Java is statically typed. That means that each expression has a type at compile-time. So if we add closures, we need to have some appropriate type for a closure. Since a closure is an anonymous function, it is natural to consider adding function types to the language. But this is not a mandate. As you can see by the two variations of our closures proposal (the nominal and the functional versions) we believe it is possible to add closures without adding function types with a limited loss of functionality (higher-order programming becomes impractical). Our proposal for closures addresses every item on this checklist. There are additional features of our proposal (the control invocation syntax and the closure conversion) that don't relate directly to the definition of closures, but which make them integrate very nicely with existing language features. And there are additional features not mentioned in the spec (such as proper tail recursion) that would be helpful to realize the full potential of closures.

  What about anonymous inner classes? It turns out that they don't pass muster on any item on this checklist. Let's set aside the fact that local variables from enclosing scopes must be `final` to be used inside an anonymous class. The problem is that variable names are simply not resolved in the correct scope. They are resolved in the scope of the anonymous class that you're creating, not the enclosing scope. If you're creating an instance of an interface then it's probably not too much of a problem because most interfaces don't have any (constant) variable definitions. But anonymous inner classes fail every other item on this checklist as well, most of them fatally. Most alternative proposals don't actually address any of the items on this list, and so fail to provide the power of closures any more than existing language constructs.

  Setting aside all the programming language theory, don't anonymous inner classes provide, in practice, all of the advantages of closures? I believe I've already shown that the answer is no. It is certainly true that for any program you can write using closures, you can write a roughly equivalent program using anonymous inner classes. That's because the Java programming language is Turing-complete. But you will probably find yourself resorting to a significant and awkward refactoring of the code that has nothing to do with the purpose of the code. In fact, you can write a roughly equivalent program using assembly language if you have the stomach for such an effort. On the other hand, true closures increase the power of a language by adding to the kinds of abstractions you can express.

Acknowledgments: my thanks to Gilad Bracha, John Rose, and Guy Steele for filling me in on and fact-checking the terminology and relevant history. Any remaining historical fantasies are my own.

[1] This is Guy Steele's impression of the late 1970's music era [personal communication].

[2] Scheme was the first lexically scoped Lisp, but certainly not the first lexically scoped programming language. Algol60, for example, was lexically scoped. See also Landin's *The Next 700 Programming Languages*.

[3] Arguably part of the lexical semantics or not, reachability state is valuable to capture in practice.