A Survey of Topics in Closure Conversion

Daniel Rosenwasser

The Background, Motivation, and Crux of Closures

Reynolds's *Definitional Interpreters for Higher-Order Programming Languages* is surely the most fundamental paper to begin reading prior to delving into the topic of closure conversion, as it was one of the foundational papers in the field. The paper begins with a general introduction to terminology and techniques utilized in different languages while establishing several loose dichotomies. These include applicative and imperative languages, call-by-value and call-by-name order of evaluation, and first-order and higher-order languages. The first and last dichotomies are the most important in the context of closure conversion.

An applicative language is one in which programs are heavily focused on the definition and application of functions throughout, whereas imperative languages use jumps, assignments, and other side-effects. Purely applicative languages manage to achieve the same results of variable assignments through the use of an *environment*, which effectively acts as a lookup structure which need only be extended through successive function calls. Functions defined at any point in the program inherit the environment in which they are defined. Function applications at any point in the program extend their inherited environment with their formal parameter bound to the given argument.

In higher-order languages such as variants and descendants of Lisp and ML, one can freely treat functions as data (e.g. storing functions in variables, lists, and passed to other functions). On the other hand, first-order languages such as Ada, Fortran, and ALGOL 60 enforce certain restrictions that disallow direct and simple use of functions as values, and are considered first-order languages. A benefit of first-order languages is that function identification is explicit at all call-sites, which may be easier to reason about. Generally speaking, applicative languages tend to also be higher-order.

Reynolds then introduces a basic untyped higher-order applicative call-by-value language, where functions take only one argument at a time (multiple-parameter functions can be achieved through currying, thanks to the presence of an environment). The paper demonstrates an elegant "meta-circular" interpreter; however, the interpreter is used to motivate solutions for several shortcomings, one of which is that the existing interpreter is written in a higher-order language. In other words, it is not immediately obvious how to port the interpreter to a language where functions cannot be generated and passed around freely.

The crucial idea is that in a higher-order applicative language, the behavior of a function is effectively based on (1) its code and (2) the environment which it inherited under its point of definition. Thus, all function values can be represented as a pair of a function (or something with which an entire function can be identified) and an environment. These structures are known as *closures*.

With this, the solution is to introduce a new function in the defining language named *apply* which takes two arguments: a closure and an argument. *apply* will effectively deconstruct the closure into its function/lambda portion and its environment portion. It then evaluates the body of the function with the extended environment (note that Reynolds's *apply* function actually also handles special runtime function values which are not actually closures, as they need no environment).

This procedure is known as *defunctionalization*. While Reynolds does not explicitly discuss it in his paper, defunctionalization is easily performable in a different flavor past the realm of interpreters, and in compilers themselves as part of a transformation pass. When translating into a first-order language with the ability to pass functions around, common practice seems to hold that functions are mapped to some symbolic representation and an *apply* function is generated to interpret that symbol and dispatch the appropriate function with a given argument. Since environments are not commonplace either, functions are also transformed to take environments as part of their parameter list.

Note that defunctionalization is *not* a necessary procedure in closure conversion. In a language with function pointers, like C, a direct translation can be made by representing the function portions of closures as function pointers themselves, and then calling those functions with the appropriate environment and argument. Still, defunctionalization has its uses in intermediate forms. Since many optimization techniques are developed for first-order languages, higher-order languages may take advantage of such optimizations following defunctionalization.

It should also be noted that defunctionalization is most easily implemented in the context of a whole-program compiler, where all call sites and functions are known. When programs are compiled as separate portion it becomes rather difficult to account for functions passed outside their compilation units. These functions are called *escaping functions*.

Defunctionalization with Types

Reynolds's defunctionalization approach to lambda lifting works fine in a simple untyped language. What happens when we shift our discourse to compilers of a language with a basic type system, or even a more sophisticated type system such as the Hindley-Milner type system? Furthermore, how can one perform the same defunctionalization translation while keeping the target-language representation typed as well? Bell, Bellegarde, and Hook seek to answer this in the paper *Type-driven Defunctionalization*.

One may demand a motivating factor before proceeding. The motivating factor is the very same as the motivating factor for a front-end type system: just as front-end type systems exist to reject semantically nonsensical programs, enforcing a type system for both the source and target representations ensures soundness and a gives loose validation that the transformation pass has preserved the semantic description.

The primary issue with Reynolds's notion of *apply* here is that it is able to accept arguments of all possible argument types and dispatch to all possible functions, meaning that it can have all possible return values as well. This is clearly not type-safe. Rather, the immediate solution is to create an *apply* function for each function type (e.g. $apply_{\mathrm{int}\to\mathrm{bool}}$). This is certainly plausible in a language where types are known at compile-time.

In a language with parametric polymorphism, things are slightly more complex. What version of *apply* should be utilized at a call site where the polymorphic identity function is utilized? The most obvious approach is to create a concretely-typed clone for each function's call-site type. For instance, in the Standard ML program

$$\textbf{let fun } id\ x = x \textbf{ in } (id\ id)(id\ 10)$$

we can see that id is defined polymorphically; however, both instances where id is used would be calls to different functions, as the rightmost identity function has the type $\textbf{int} \to \textbf{int}$, whereas

the leftmost call to *id* has the type $(\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to \mathbf{int})$, so the program could be transformed into something such as

$$
\begin{aligned}
&\mathbf{let} \\
&\quad \mathbf{fun}\ id_{int \to int}\ x = x \\
&\quad \mathbf{fun}\ id_{(int \to int) \to (int \to int)} x = x \\
&\mathbf{in} \\
&\quad (id_{(int \to int) \to (int \to int)} id_{int \to int})\ (id_{int \to int}\ 10)
\end{aligned}
$$

This conversion is called *monomorphisation.* One could imagine three types of approaches to monomorphisation: (1) generating a clone for every type, (2) generating a clone for every type that occurs within the program itself, and (3) demand-driven clone generation based on function application. The first is impossible because one can produce arrow-types of arbitrarily high order, and the second is simply infeasible due to code bloat. The third is actually common practice, and to relate it to a more familiar process, it is the same type of pass that occurs in most C++ template engines. As with C++ templates, monomorphisation generally requires that the relevant polymorphic definitions be known during compilation, since concretely-typed clones are generated at call-sites. Thus, a whole-program compiler is well-suited to monomorphisation.

The given approach in the paper is to first reduce polymorphism by performing three transformation passes. Following this, two transformation passes help to eliminate higher-ordered nature of the program so that we may finally deem it defunctionalized.

Control Flow Analysis

Until this point, we have viewed the monolithic *apply* function to be just as good as the collective type-based *apply* functions. To some extent, this is true, however, in breaking down *apply*, we managed to narrow down the functions that might be dispatched at any call site. Why might this be useful? There are several optimizations that can be put in place, such as inlining the call to a version of *apply*, or if a given version of *apply* only accounts for one function, inlining the call to that function itself.

This implies that there is an added benefit to narrowing down the functions which variables may be bound to, and identifying the paths which our programs may choose. This is known as *control-flow analysis*. Control flow analyses are often tied to *data-flow analyses* which bring the idea of *abstract values* into discourse. An abstract value is a generalized approximation of the values that a variable can take on. For instance, if we have a variable x of an integral type, then we may say that x is approximated by negative, positive, and zero values. We have already seen a very simple example of abstract values: types themselves. Having just this value abstraction lends itself to many different optimizations.

We can, of course, be more specific in granularity of the types of values, as well as be more exact in the abstract values that a variable takes on. For example, given that x is approximated by negative, positive, and zero values, a compiler could theoretically determine from the binding $y = x + x + 1$ that y is approximated by only positive and negative abstract values, but not 0. If desired, one could imagine the ability to add or completely change the granularity of the abstract value, such as by specifying that y is now also an odd number; however, such analysis is beyond our scope.

Cejtin, Jagannathan, and Weeks's *Flow-Directed Closure Conversion for Typed Languages* serves as a good transitioning point between type-driven and flow-driven analysis, since both are heavily used. Rather importantly, in addition to defining abstract values, the paper defines a *flow* as what is essentially a function from values to abstract values for each program. Further of note is the concept of flow *safety*. In essence, a flow is safe if and only if is always approximates an abstract value conservatively. In other words, in our example of the variable y which is never zero (and always odd), a safe flow can report that y can be positive, negative, and zero, but it can never report that y is only positive or zero. While this means that an analysis can over-approximate and be less exact, it is still better than the analysis being wrong.

Cejtin, Jagannathan, and Weeks give a source language with user-definable algebraic data types, arrow-types, and tuples. The abstract values used for values of algebraic data types are defined by the set of all type constructors. The abstract values of arrow-type variables are defined by the power set of functions that may flow to those variables. Finally, the abstract values of tuples are tuples of abstract values that correspond to each respective value in the original tuple.

The class of control-flow analyses suggested is called 0-CFA, and the analysis of 0-CFA is performed through a process where abstract values flow to variables in a somewhat transitive manner. As an example of a minimal system under which 0-CFA can be analyzed, consider the untyped lambda calculus, where all we have are functions and function applications. Might's article *k-CFA: Determining types and/or control-flow in languages like Python, Java and Scheme* somewhat informally describes the process of finding flows as follows:

1) For each lambda term $\lambda x.e$,
   $\lambda x.e$ flows to $\lambda x.e$.
2) For each function application $(f\ e)$, if $\lambda x.e'$ flows to f, and the value v flows to e,
   v flows to x.
3) For each function application $(f\ e)$, if $\lambda x.e'$ flows to f, and the value v' flows to e',
   v' flows to $(f\ e)$.

k-CFA is a more general control flow analysis (that encompasses 0-CFA as the names would imply). Higher values of *k* imply more accurate approximations; however, this comes with an increase in runtime of the analysis.

Lattice Structure of Abstract Values

In our discussion of programs, we take the number of variables in a program to be finite. Following monomorphisation, the same goes for the number of functions defined within a program. From this, we have the fact that a finite number of functions can flow to variables, and that a finite number of variables can transitively flow to other variables. When one considers the set of functions that may flow to a given variable, it is clear that with a finite number of functions, this set may take on any combination of defined functions – that is, a member of the power set of all functions. Power sets hold the property that they form a lattice, which gives us a way to reason about the relations between exact approximations, over-approximations, and unacceptable approximations.

As a light background, lattices are partially ordered sets, with some ordering operation. Here that operation is denoted $\leq$. $\perp$ (called "bottom") is shorthand for the element of a given

lattice L where $\forall l \in L, \bot \leq l$. Similarly, $\top$ (called "top") represents the element where $\forall l \in L, \bot \leq l \leq \top$.

As mentioned, Cejtin, Jagannathan, and Weeks defined abstract values of arrow-types as members of the power set of all functions. Note, however, that because of their work with a statically typed language, all programs are type-checked, and so a flow analysis need only approximate using a subset of the lattice where functions of the expected type are contained within the set. In fact, so long as a flow stays safe, then the goal of that flow is always to try to descend to the lowest point of the lattice, giving a least solution. Eliminating incompatible elements can certainly ensure that an analysis will be more accurate.

Representations of Environments

Until this point we have only discussed the representation of the code portion of closures. To reiterate, the code portion of a closure is typically an address to code (a function pointer, for instance), or a symbolic representation which will be used for dispatching. However, we have mostly glazed over environment representation. We mentioned that an environment was a lookup structure, which means that the most straightforward way to implement this in an interpreter is through any kind of associative container/map (e.g. hash maps, search trees, and association lists) which gets extended and reduced as necessary.

Interpreters written in higher-level languages have the luxury of using lookup structures in a fairly "care-free" manner. In lower-level languages, this is typically more difficult to achieve as these languages lack certain runtime and language abstractions. Furthermore, the point of representing a program in a lower-level language is typically for the purpose of performance. Thus, it is not exactly ideal to perform a map lookup every time we require the use of a variable in our programs.

Worse still, we have acted under the assumption that at every function definition, that function inherits the existing environment to produce a closure; but doing this may be overkill, since the code may only utilize a subset of the variables within an environment. This on its own should call for a new representation immediately.

Flat closures are by far the simplest representation that is in modern use[1]. The basis of flat-closures is that functions in applicative languages have a finite number of free variables – variables that are neither parameters nor local bindings. In such a case, aggregating the free variables of a function will allow us to know which variables are required within the environment at a function definition site. If the environment variables required at a definition site are non-local to that definition site, then it is considered a free variable for that scope as well, and so these requirements back-propagate to a point where the binding of that variable actually occurs.

For flat closures, assigning a unique ordinality to each free variable in the respective scope is the next step. Once this ordinality is established, a "contract" can be established between the definition point of a function and the code itself. This contract means that the defining function will pack the free variables into an environment, ordered by the specified ordinality. The code will be perfectly aware of this ordering and will be able to select from this structure appropriately. In essence, the environment becomes a contiguous block of memory containing the necessary free variables.

[1] It should be noted that contrary to the name, "flat closures" have less to do with the closure itself and more to do with the environment; this is true for the term "linked closures" as well.

One may note that flat closures have some redundancy, because every single time a closure is created, free-variables that are common to the function definition scope and the defined function's scope need to be unpacked and repacked. This is exacerbated when there are multiple internally defined functions, where the same free variables may have to be packed multiple times. *Linked closures* are a less simple, but still approachable, technique that manages to get around this problem. Linked closures essentially represent environments as linked lists of free variables record blocks, where the "nodes" each correspond to the scope at which the respective free variables were defined, and the "tail" or "next" element of each node corresponds to the outer scope. By noting the scope at which free variables are defined, we are able to determine how deep we will have to traverse down a chain of environment nodes. This way, while lookup time is no longer constant (it is now proportional to the distance in scope from where the free variable is defined) we avoid packing and unpacking the data. One may argue that variables defined farther out from a scope are less likely to be used, although this is not something we can say conclusively.

The Safe for Space Complexity Rule

Linked closures unfortunately have one other flaw in addition to non-constant environment variable lookup: retained pointers to unnecessary environments. Even though functions several levels deep may not need variables from farther-out scopes, their parent scopes may have at some point, and those functions may need to retain the environments of their parent scopes. Thus, a garbage collector will not be able to safely reclaim the memory of the unnecessary environments and the objects pointed to within them.

The *safe for space complexity* rule states that "any local variable binding must be unreachable after its last use within its scope." Clearly, if closures can be passed around, and we continue to point to "dead" values as with linked closures, the SSC rule is violated. On the other hand, with flat closures, our environments are strictly defined by internal need of a function, so flat closures are compatible with the SSC rule.

Appel and Shao, both strong proponents of the SSC rule, but unsatisfied with the amount of copying that flat closures necessitated, proposed *safely linked closures* in both *Space Efficient Closure Representations* and *Efficient and Safe-for-Space Closure Conversion*. Safely linked closures are a bit of a cross between flat and linked closures. Like linked closures, some unpacking and repacking is avoided by pointing to another record that already contains the necessary free variables. However, unlike linked closures, the number of links is bounded to one (meaning that the depth of environment records is at most two), and the linked environment is certain to contain no more than what the "head" environment does not account for in the free variables.

This representation is achieved by performing a liveness analysis, in which the lifetime of a variable's usage is analyzed to determine when it will no longer be in use. Variables with similar lifetimes are grouped together, and shared in a single environment record. Then closures that utilize these variables share them while maintaining copies of variables that have lifetimes that are tied to their own scope.

Of course, there is nothing to prevent the mixed use of closure representations. Before implementing safely linked closures, the SML/NJ compiler used linked closures when the SSC rule wouldn't be violated and flat closures otherwise. A compiler would simply have to note the different environment contracts associated with the respective definition points.

Personal Work

Over the first week of the term, Dr. Fluet and I worked on the intended portions of the curriculum.

Over the next two weeks, I spent time reading up and understanding Appel and Shao's closure conversion algorithm proposed in *Space Efficient Closure Representations* and *Efficient and Safe-for-Space Closure Conversion*. I also spent that time trying to familiarize myself with MLton's source code and infrastructure.

Following this, my focus shifted to *Flow-Directed Closure Conversion for Typed Languages,* in an effort to understand the concept of control- and data-flow analyses, especially within the context of MLton's current approach.

Reynolds's foundational *Definitional Interpreters for Higher-Order Programming Languages* paper was the next target, which strongly clarified certain intents of defunctionalization, and the history of its initial motivation.

With that understanding, *Type-Driven Defunctionalization* was the natural progression, which demonstrated the need for monomorphisation to properly defunctionalize programs. At this point, I also began to grow frustrated with my own inability to understand the source code which aggregated the free variables of every function term, so Dr. Fluet and I spent a good amount of time that week trying to pin down the general direction that the code took

After this, to get a sense of the types of control-flow analyses that exist, and a very quick history of their culmination, I read Midtgaard's *Control-flow analysis of functional programs* survey paper, which briefly touched on each analysis so as to point one in the correct direction if a certain analysis struck their interest.

The Principles of Program Analysis book came next. Since the content is quite heavy in its mathematical approach, the intent was mainly to get a "feel" for the relevant sections on control-flow analysis. In addition to this, I managed to fully trace through and understand the free variable aggregation pass in MLton and fully comment the procedure so as to guide both myself and future maintainers who may have a difficult time understanding exactly what is taking place.

Finally the most recent paper that I have read is Matthew Might's *Improving Flow Analyses via ΓCFA*. It mostly served to give me a better idea of an even broader category of approaches to control-flow analysis.

Overall, it would be safe to say that I dedicated around 3 hours in meeting time a week as well as 6 hours a week outside of the class.

Further Work

Finishing the implementation of Stephen Weeks's flatter closure conversion algorithm would be the next logical step from this point on.

Works Used

Bell, J., Bellegarde, F., Hook, J.: Type-driven defunctionalization. In ICFP'97: Proceedings of the second ACM SIGPLAN International Conference on Functional Programming. pp. 25-37. ACM (August 1997).

Cejtin, H., Jagannathan, S., Weeks, S.: Flow-directed closure conversion for typed languages. In: Smolka, G. (ed.) ESOP'00: Proceedings of the Ninth European Symposium on Programming. Lecture Notes in Computer Science, vol. 1782, pp. 56-71. Springer-Verlag, Berlin, Germany (Mar 2000)

Danvy, O., Nielsen, R. Defunctionalization at work. In PPDP'01: In Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. pp. 162-174. ACM (September 2001).

Matthew Might. k-CFA: Determining types and/or control-flow in languages like Python, Java and Scheme. Matthew Might. Blog. May 15 2012.

Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. 11th ACM International Conference on Functional Programming (ICFP 2006). Portland, Oregon. September, 2006. pages 13--25.

Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys 44(3), 10:1-10:33 (Jun 2012).

Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999).

P.J. Landin. The mechanical evaluation of expressions. *Computer Journal* 6(4):308-20 (1964).

Reynolds, J.C.: Definitional interpreters revisited. Higher-Order and Symbolic Computation 11(4), 355-361 (1998).

Reynolds, J.C.: Definitional interpreters for higher-order programming languages. Higher-Order and Symbolic Computation 11(4), 363-397 (1998) [reprint].

Shao, Z., Appel A.W.: Efficient and safe-for-space closure conversion. ACM Transactions on Programming Langugaes and Systems. 22(1), 129-161.

Shao, Z., Appel A.W.: Space-efficient closure representations. In LFP'94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming. pp. 150-161. ACM (July 1994).