On secure first-class control

Eric Demko

July 2, 2015

Contents

Abstract

First-class control (FCC) is a generic name for a variety of techniques for dynamically manipulating program control flow. The general claim is that FCC allows any control structure to be efficiently implemented within an existing language. While there has been considerable research into how the different FCC techniques relate to each other, how these techniques relate to software architecture is less well-developed. As such, FCC is commonly viewed as complex and error-prone: only a few languages incorporate it. Our claim is that native FCC can be integrated with existing control techniques, and that doing so reduces the likelihood of programmer error.

In this paper, we develop the semantics of an extended lambda calculus incorporating secure, expressive and convenient first-class control. Our formulation unifies continuations with other non-local control constructs, esp. including exceptions and transactions. We demonstrate the utility of our system with examples of advanced constructs that may be efficiently implemented by users. Ideally, this would pave the way for major production programming systems to incorporate first-class control and thereby increase at once their expressivity and security.

1 Introduction

Exceptions and first-class control (FCC) are deeply interconnected, but are perceived as having wildly different efficacy. Though exceptions are widely implemented and used, FCC is viewed as dangerously complex – a tool only for gurus – and therefore shunned by production programming systems. Nevertheless, exceptions admittedly come with their own dangers. Careless use of exceptions renders systems vulnerable to slowdowns, crashes, and data corruption. These problems manifest themselves because of poor maintenance of resources such as file handles, network connections, locks, and even simple mutable state. To mitigate this, programmers have developed a notion of "exception safety"—a usually informal analysis meant to reveal exception-related programming errors. It is reasonably easy to check for exception safety, even without compiler support, so many developers have judged the advantages of exceptions to outweigh the risks. Given the deep connection between exceptions and FCC, we can reasonably expect the perceived danger of FCC to be only a product of formulation.

The main contribution of this paper is a dynamic semantics which:

- incorporates both higher order first-class control along with non-local jump protection, which
- easily simulates all major forms of non-local control flow (including exceptions and various delimited control operators),
- is no more likely to cause crashes or mis-manage resources than existing exception systems, and
- offers a route to more expressive languages in that they would be capable of encapsulating arbitrarily complex control flow structures behind simple interfaces.

In the process, we review the non-local control flow techniques already implemented in production programming languages We also implement our system to illustrate several applications of our system of FCC. We further point to several additional variations, laying the foundation for discussion to determine the ideal primitives for FCC.

2 Overview

2.1 Fundamental Concepts

When we—as humans—evaluate a complex expression, we engage in a two-fold process: 1) focus on and evaluate a particular subexpression, and 2) remember that once we have suitably simplified the subexpression, we need to substitute the result back into to our original expression in place of the subexpression. As we iterate this process, we build up a stack of *contexts* through which will need unwind a value. A contiguous section of the context stack is called a *continuation*, and the stack as a whole is often simply referred to as *the continuation*. First-class control (FCC) simply means being able to access continuations and use them as first-class values. This definition is of course somewhat informal, but a more rigorous definition is not widely available, perhaps due to the breadth of variations and relative rarity of its implementation.

Continuations are a rich source of concepts and terminology, so we will now address the terminology relevant in this thesis.

We say evaluation occurs within the *dynamic extent* of an expression e when we have a context of derived from e onto the stack. When the evaluator places a context on the stack in response to e, we say we have *entered its dynamic context*. Similarly, when that context is removed from the context, we say evaluation has *exited its dynamic extent*.

Altering the continuation only in the topmost context of the stack is the essence of *local control flow*, whereas *non-local control flow* involves altering the stack by adding or removing whole continuations at a time.

Beyond local control flow, FCC involves at most a few operations on the

continuation.

A top-most section of the stack can be selected and saved elsewhere in memory, which we call *capture*. This stack section is then *reified*—made available as a value. Often a captured continuation is also removed from the stack, which is called an *abort*, although it is not necessary for these to occur together. An operator which performs any combination of capture, abort and reification is called a *control operator*. Any reified continuation may later be *restored*: copied back onto the stack on top of the current continuation. Continuations may even even be restored multiple times. In contrast, exceptions remove a section of the control stack, but do not reify it and therefore cannot restore it[1].

When a control operator affects a proper subsection of the stack, we say that it is *delimited*, as opposed to *undelimited*, or operating on the entire stack. The section of stack operated by a delimited control operator is determined using special contexts called *delimiters*, *prompts* or *stack marks*. Operators that place a delimiter on the stack are called *delimiting operators*.

[1]Some exception mechanisms may capture the stack behind the scenes for use in generating stack traces, but do not allow programmer access to the continuation itself. Generating a stack trace is a distinct operation from continuation capture.

2.2 Motivation

While the above definition is accessible to implementors, it leaves the advantages of first-class control opaque to the programmer. The advantage of FCC is that *it allows for the abstraction and reuse of any pattern of control flow.* That is, any control flow structure, no matter how complex, can always be centralized to reduce maintenance costs. First-class control is thus a natural extension of existing high-level languages, which have already brought the power of abstraction to patterns of algorithm, data, and syntax.

As a simple example, a common pattern is an accumulation loop with early-exit. We can use FCC to efficiently implement this pattern and reuse it without difficulty, as in Figure 1. We will give further analysis in §14 after we have developed our system of first-class control.

```
(define (foldl/ee zero xs body)
   (define mark newCue)
   (define (break val) (abort mark val))
   (handle mark id
      (foldl zero xs (body break))))


(define (all? xs p)
   (foldl/ee true xs (lambda (break acc x)
      (if (p x)
          true
          (break false)))))
```

Figure 1: Define the accumulation loop with early exit

Expressivity—the reduction of manual pattern-based programming—is not

however a goal in itself. Rather, expressive power should be used to increase the quality of the software, both in terms of efficiency and reliability. Native first-class control enables just such an improvement. Consider: library writers often create interfaces—esp. for working with external resources—that require the exposed procedures to be used in rigidly constrained orders. With first-class control, library writers can codify these patterns in executable code rather than mere documentation.

```
(define (open filepath mode) ...) ;; returns a file handle


;; once you're done with the file, call this or the resource will leak
(define (close handle) ...)


;; don't call read or write on a file that's been closed,
;; or Bad Things(TM) will happen!
(define (read handle) ...)
(define (write handle bytes) ...)
```

Figure 2: File API without FCC

As a simple example, consider the hypothetical API for working with files given in Figure 2. Improper ordering to any of these four procedures at runtime can cause a supposedly reliable system to fail. Although the requirements on the structure of control-flow are documented, they are not enforced by the language. Thus, if the programmer has incomplete knowledge of the documentation or of his program, it is likely that one of these requirements will be violated.

With FCC, we can present a different interface by combining the effects of the `open` and `close` procedures into a single higher-order procedure `withFile`. The `withFile` procedure opens the file, exposes it only to the `body` procedure, and ensures

```
(define (withFile filepath mode body) ... (body the_handle) ...)

(define (read handle) ...)

(define (write handle bytes) ...)
```

Figure 3: File API with FCC

that the file will be closed immediately after the body is finished. Such an interface is not only smaller, and therefore easier to comprehend, but also guarantees that the file handle is used in accordance with the control-flow structure requirements. Simply by interface design, we can ensure[2] that read/write cannot be performed on a closed file and file handles are closed quickly. In this way, any possibility of the aforementioned programmer errors is eliminated. Even when control-flow dependencies become complex, FCC is still able to encapsulate the complexity, providing APIs that are resilient even in the face of incomplete programmer knowledge. Without first-class control, such guarantees are generally impossible to obtain practically.

There are downsides to first-class control. After all, "an increase in expressive power comes at the expense of less 'intuitive' semantic equivalence relations."[4] For FCC in particular, we lose the ability to easily map from the static source code to the dynamic control flow of a program. The implications for program reliability and security are poorly-understood, and are assumed to be significant. On the one hand, this means that managers of sensitive production systems should be skeptical of FCC, but on the other, it means that no-one is capable of systematically evaluating the relative advantage of incorporating it.

It is relatively easy for the designer of a new language to include whatever system of FCC they see as most advantageous. Implementing a desired system of FCC in an

[2]For reasons of space, we have not shown how to ensure that a file handle does not escape from the body of `withFile`. A technique to do this is well-known[16], which we will exploit in §**??** to make a different encapsulation guarantee.

existing language can be rather more difficult. Nevertheless, it doesn't make much difference from an academic viewpoint whether FCC is implemented natively or not. There are two broad alternatives to native first-class control.

First, a programmer may fall back on pattern-based programming. This is an insidious alternative, being both very easy to accomplish when first writing the software, but very difficult to untangle and maintain later. In general, many properties of pattern-based programs cannot be verified with anything short of whole-program analysis. Furthermore, the structure and intent of such a system are entirely opaque from the source code.[4] Both of these drawbacks significantly raise the both the cost of maintenance and the risk of damage due to the program. We shall see concrete examples of this in our review.

Second, it is possible to simulate first-class control with varying ease depending on the host language. It is our opinion—after Greenspun's Tenth Rule—that these implementations will, with overwhelming probability, be incomplete, ad-hoc, buggy and/or slow. In particular, it is difficult to arrange that simulated FCC will not interfere with existing control operators.[8] Furthermore, simulation may increase the expected size-complexity of the simulated operators.[9] In this case as well, the drawbacks are not likely worth the costs and risks in a production system.

It is clear then, that if we are to maximize the reliability and security of computer systems without sacrificing efficiency, we must develop a system of first-class control wherein programmers may easily develop intuitions about program equivalence. This is exactly the task we solve in this paper. We will not argue that we have the unique best possible solution, but our system is clearly the beginning of the end for the confusion that has previously surrounded first-class control in production languages.

2.3 Our System

TODO I won't elaborate this section until I'm sure I've got the right formulation.

What are the advantages of our system? It is formally specified. It allows for static typing (for which we give natural deduction axioms). It unifies a wide range of control operators (all major ones) already deployed. It provides for deterministic resource initialization and cleanup, even in the presence of arbitrarily complex non-local control. Efficient implementation techniques are known for similar systems or subsystems. It is somewhat modular, allowing language designers to pick and choose subsets appropriate for their goals rather than start from scratch (esp. without theorems).

How does it relate to other systems? We split the primitives up roughly along the same lines as MFDC. However, we extend MFDC along the lines of fcontrol: delimiters include a handler, rather than force the control operator to provide one. The control operator is intimately connected with throw, since it only takes a value; thus, exception handling can be implemented entirely in terms of a stock delimiter. So far as we know, there is no formal system concerning the control guards we develop, though we are inspired by D's scope statemetns and Scheme's dynamic-wind.

FIXME: these next two paragraphs are highly related; they need to be reviewed against each other.

Note however, that to achieve practicality, type systems applicable to this system likely must make a small compromise in their soundness: this is the same compromise that exists in unchecked exception systems. We do not believe this to be a cause for concern, since unchecked exceptions are already widely used in production. Further, it is this author's opinion that without a focus on lexical scoping and immutable values, our system – perhaps any FCC system – has the potential to introduce incidental

complexity which can hinder maintenance and performance. We refer the reader to Stroustrop's discussion[21] of the complex exception system in Mesa.

Thanks to our system's similarities to (TODO generalizing exceptions in ML) and [3], we expect a practical type system to be able to rule out several classes of error related to the use of FCC. Nevertheless, alternate type systems are certainly possible, each making different trade-offs regarding soundness vs. ease of use. In this paper, we focus only on the dynamic semantics of FCC, leaving a static semantics to further research.

2.4 Structure of this Paper

In Part One, we examine a number of control flow constructs that already exist in production programming systems, first looking at exception handling, then at existing first-class control operators. We consider resource-acquisition-is-initialization in §3.1 and try-catch-finally in §3.2. We also look at some less-widely implemented structures. Python's with statement and C#'s using statement share properties detailed in §3.3. Go's defer-expressions and D's scope statements are described in §3.4. Monads are examined in §3.5 specifically in the context of Haskell, though they can also be used in other statically-typed functional languages. in §4 we turn to first-class control, beginning with a small tutorial. In §4.2 and §4.3 we turn to first-class control operators. In §4.4 we explain Scheme's dynamic-wind construct for managing resources under FCC. A broad analysis of these constructs' capabilities is given in §5.

In Part Two, we apply the lessons learned in Part One to analyze a novel system of first-class control. The system is fairly large, but can be split into a few small components. We give an informal introduction to the system in §12 and point the reader towards its historical influences. In §13, we give the dynamic semantics of the system using a transition system. In §**??**, we show how our system can macro-express several of the most important non-local control operators already being used. To exemplify the control operators, we have implemented a toy language which we describe in §15 along with several code examples implementing useful sizable flow abstractions.

Part I

Literature Review

If we are to unify existing systems of control flow, we must examine a wide range of existing control structures. To maintain our focus on reliability, we restrict our examination to those control features available in production languages. First, we will focus on non-local control flow without continuation capture: exceptions and related concepts. Then, we will examine first-class control as it exists in Lisp dialects, to date the only production languages which support a variety of first-class control operators. Our goal is to identify successful patterns of control flow for use both as inspiration for our control calculus, and as a sanity check to ensure that the control calculus developed is expressive enough for general use.

3 Exception Handling

3.1 RAII

One of the very first methods developed for automatic resource cleanup is called "resource acquisition is initialization" or RAII. The technique fundamentally relies on ad-hoc polymorphism and appears only in object-oriented languages. It ties an object's initialization and deinitialization to its lifetime using a constructor(s) and destructor.

In C++, when an automatic variable becomes unavailable, such as when it goes out of scope or its stack frame is removed, its destructor (if any) is implicitly called. RAII takes advantage of this feature by placing any relevant cleanup code into the destructor.[22] Since the lifetime of automatic variables is easily apparent, RAII performs deterministic cleanup. However, C++ cannot detect lifetimes of heap-allocated data, and so for these objects, destruction must be performed manually. Manual memory management is not only expensive to develop, but also contributes to serious bugs which can easily compromise secure systems.[25]

In Java, C#, and similar languages which use garbage collection, destructors[3] are called when an object is collected. Unfortunately, the time to detect, and therefore clean up unreachable objects is non-deterministic.[10][26] Whenever the resource is not in physically infinite supply, the garbage collector will race the program to maintain the illusion of automatic cleanup.[25] Further, the order in which objects are finalized during garbage collection cannot be predicted, so the technique is unsuitable when there are ordering dependencies between several objects' finalizers.[25] This should not be surprising, since garbage collectors are meant mainly to free up memory during program execution rather than as a means of registering callbacks. In short, the use of

[3]Some languages call these "finalizers", but the result is essentially the same.

finalizers tied to the the garbage collection cycle is not a generally suitable technique for timely and correct resource management.

RAII succeeds well when faced with managing external resources such as database connections, but when faced with internal logical state management, it offers no advantage. For example, in languages with garbage collection, the lack of determinism can mean that RAII is even technically unsuitable. Beyond the technical limitations of RAII, there is also a psychological one. Every definition of cleanup code needs its own class definition, which is quite large considering the few lines that are really important. With these pressures, programmers will, with good reason, shun RAII and resort to ad-hoc techniques that are likely to introduce other flaws.

3.2 Try-finally

Try-finally is a fairly simple idea: whenever and however control exits from the `try` block, the code in the `finally` block will be executed. The ubiquity of try-finally can be traced to several advantages. In this approach, cleanup is deterministic. Try-finally and throw constructs are also easy to detect and reason about both programmatically and visually, making auditing for exception safety straightforward. Finally, control flow during exception handling is particularly transparent compared to other non-local control techniques.

Unfortunately, the transparency of exception handling is largely due to the fact that exception handling code cannot be abstracted; it is transparent in the sense that it is always exposed. Mainstream OO languages effectively force developers to resort to pattern-based programming, writing out common exception handling code for every use. As mentioned before, pattern-based programming is costly: it requires developers to take time writing more code, and opens the code to regressions during maintenance.

Consider the code pattern in Figure 4, for example. In the pattern, A is to be replaced by some business logic and B by any additional cleanup. If any code within the B slot could throw an exception, then `connection` will not be closed as intended. If, for example, some logging were to be added in B, an I/O error could occur – in a distributed system, a simple network outage is sufficient – causing an additional exception before the rest of the cleanup and preventing the connection from being closed. This example is very simple not only for reasons of space, but also because it is at the core of the problem. For a thorough examination of examples found in the wild, we refer the interested reader to [25].

Different programming languages have very different semantics in the event an exception attempts to propagate beyond a catch or finally block. Many languages,

```
try {
    var db_connection = new DBConnection(...);


    A...
}
finally {
    B...


    if (db_connection != null)
        db_connection.close();
}
```
Figure 4: Try-finally Pattern

including Java and Python, allow the new exception to propagate, sometimes with additional information in the stack trace. Perhaps the most reliable solution is to instead terminate the program, as in C++.[21] This decision reflects the notion that an exception handler must be exception-safe. That is, an exception handler should only do trivial cleanup tasks, not respond with exception-prone actions. Guaranteeing this level of exception-safety is the first of two major solutions to mitigate the problems of try-finally.

Exception safety may be automatically determined by the compiler using checked exceptions. However, checked exceptions have largely been rejected in production designs. Although Java does have a checked exception system, safety is only checked at the method level, not as relates to the example here. Further, the system co-exists with the more commonly used unchecked exception system. Support for checked exceptions is lacking largely because checked exceptions in Java have evidenced their uselessness and inconvenience in common languages (such as Java, C++, and C#), and of course

because they are quite impossible in dynamic languages.

The second solution, in a language with good support for first-class functions[4], to the problems of Example 4 is to use continuation-passing style (CPS). The best example of this technique is in Ruby, where it is unobtrusive to pass a single continuation during function call. For example, Ruby users can simply write `File.open("example.txt", "r") do |fp| ... end`, where cleanup is centralized in the `.open` method, and the user can supply their own file manipulation code in the block.[23]

Unfortunately, most languages are not able to do this so cleanly, either because they have not (yet) adopted first-class functions, or because the CPS transform must be manually performed. Indeed, even in Ruby only a single continuation can be passed naturally; as soon as a second is needed, manual transformation is required. We have yet not mentioned lazy languages, but it turns out that these are unsuitable as well. Although they can fully overcome the syntactic issues, lazy languages erode determinism, which is the key benefit of try-finally. Additional techniques can restore the determinism, but we will return to this issue in Section 3.5.

The combination of unchecked exceptions and the inability to abstract is detrimental to reliable programming. Careful code review may be able to mitigate these exception-safety bugs, but from both accuracy and cost-efficiency standpoints, it would be much better to have compiler support.

[4]By good support, I mean that the language should make it easy to use functions in a first-class manner. In particular, the syntax should not be overly verbose, static scoping should be default, and tail call optimization should be performed. Many common languages, such as Java, C, C++ and Python do not meet these requirements even though they technically allow first-class functions.

3.3 Context Statements

TODO Java 1.7 now has the "try-with-resources statement"

The development of the "with statement" in Python and the "using statement" in C# bear a remarkable resemblance to the development of the for-each loop. The for loop was so commonly used as `for (x = start(xs); notAtEnd(x); x = getNext(x)) {...}` that language designers decided to codify this pattern as the syntax `foreach x in xs {...}`. Similarly, the pattern in Figure 4 is so common that some language designers have chosen to codify the pattern roughly as `with x = some_resource {...}`. Although Python, C# and Java each use different names and syntaxes for largely identical concepts, for simplicity we will simply refer to these constructs as "context statements".

The key ingredients of a context statement are an expression and a block. The statement may also bind the result of evaluating the expression to a variable for use inside the block. Further, the result of the expression is meant to be equipped with subtype-polymorphic setup and teardown code. Before the block is entered, the setup code is executed, and just before the block is exited, by any means, the cleanup code is executed.[24][26]

Figure 5 is an example of a with statement in Python (the corresponding C# version is only superficially different). In it, the expression `open(filePath, 'r')` opens a file in read-mode. In this case, no additional setup is given, so the resulting file handle is bound to `fp`. The body of the statement is then executed. If no exceptions are raised, then immediately after the block, the cleanup code is called on the file handle; in Python, this is the `__exit__` method. If an exception is raised in the body, then the cleanup method is called with the exception details before propagating the exception. In the specific case of file handles, the exit method simply closes the file.

```
with open(filePath, 'r') as fp:
    do_something(fp)
```

Figure 5: With statement

Figure 5 maps neatly to the pattern of Figure 4. As such it retains advantages of the more general exception-handling approach, in particular determinism. It saves only a very little in source code size, but importantly, its use does not offer any ability to compromise the exception safety of the handler. Because the try-finally pattern is so ubiquitous, preferring context statements eliminates many opportunities for error. In this author's experience, the benefits for program maintenance outweigh the cost of a slightly larger language. However, there are inherent limitations with strategy of naïvely codifying patterns into syntax as has been done with context statements and for-each loops.

```
acc = start_value
foreach x in xs:
    acc.add_element(x)
```

Figure 6: Accumulation loop

Consider for the moment the standard accumulation pattern give in Figure 6. This is a very common pattern, but it has not been codified into the syntax of any language, even though there are multiple locations where errors can be introduced. For example, the loop body may contain an early exit, a conditional path may accidentally omit an addition to the accumulator, and so forth. In contrast, consider the higher-order functions `map` and `fold`. The `map` function corresponds to for-each, and `fold` solves accumulation problems, but importantly, *they are not built into the language*. As such, higher-order functions are able to codify many looping constructs without extending the language.

Similarly, context managers solve one particular problem very well, but cannot be

generalized. Context managers are unable to encode patterns of stopping error propagation, logging errors, and so forth. No matter how many specialized control constructs we add to the language, there will always be patterns that have not been codified; such a language is fundamentally limited in its abstractive power. Perhaps the most accessible expression of this idea is that "it is impossible to anticipate all possible needs for control abstractions during the design of a programming language."[19]

Finally, we mention some contact with the RAII approach. In particular, both solutions externalize the setup-cleanup code pairing to another class. One may think the strategies share the problems of one-off cleanup logic, but it turns out that for single-use scenarios we can instead use try-finally. Switching to try-finally does not increase the likelihood of programmer error, because in either case there is one definition of the logic: in-line with try-finally, versus in an external class with context managers. It is only when a definition is duplicated that the possibility for error increases.

3.4 Defer Statements

The languages D and Go share a similar language feature whereby expressions may be registered to be evaluated before the call stack unwinds (CITE D and Go standards). We will call these statements "defer statements" after Go terminology[5]. This particular feature is quite new, so there are several differences between the two languages, though they share the same idea.

In the case of Go, each stack frame maintains an additional stack of unevaluated expressions. When control reaches a defer statement, `defer` e, the expression e is pushed to the stack, but not evaluated. When the stack frame is popped (whether from normal return or due to an exception), the expressions it had built up are evaluated in the reverse order they were pushed.

```
fp, err := os.Open(filepath)
if err != nil {
    return
}
defer fp.Close()
do_something(fp)
```

Figure 7: Go-style defer statement

A simple defer statement in D is written `scope(exit)` e. D can also discriminate between normal and abnormal exits with `scope(success)` e and `scope(failure)` e respectively. As in Go, the expression is unevaluated when control passes over the statement. When control exits from a block, all the `scope(exit)` statements are executed, as well as all the `scope(success)` statements if the block was exited normally, or all the `scope(failure)` statements, if the block was exited because of an

[5]D calls them "scope guard statements".

```
void processFile(string filename) {
    scope(exit) logger.info("Attempting file... ");
    auto fp = File(filepath, "r");
    scope(success) logger.info("done.\n");
    scope(failure) logger.info("FAILURE:" + filepath + "\n");
    do_something(fp);
}
```

Figure 8: D-style defer statement

exception. D's defer statements are registered at compiletime, rather than at runtime as in Go. Nonetheless, the effect is very similar, as the registered expressions will be evaluated in reverse order.

The ability for D to discriminate between normal and abnormal exit is very useful for performing atomic operations. For example, Figure 8 shows a program which performs an operation on a file and logs the result. When multiple error-prone operations must be performed together atomically, this technique can save significant development time (CITE that D article).

Unfortunately, the defer statements of both languages share with try-finally the same the inability to be abstracted. Although patterns of defer statements are generally smaller than patterns of try-finally, they are nevertheless error-prone and cannot be recommended as a general solution for secure software.

3.5 Purity and Monads

The lazy, pure functional programming language Haskell takes an interesting approach to exceptions in idiomatic code. Although Haskell incorporates unchecked exceptions as a language feature, native exceptions and exception handling are not prevalent in Haskell code.

Although Haskell is a pure functional programming language, is is meant for use in real-world systems. To this end, Haskell adopted an idea from category theory called "monads." A detailed discussion of monads in Haskell and their applications is beyond the scope of this paper, but we point the interested reader to [15]. For our purposes, we can simply say that monads are able to use the type system to make and track very fine-grained distinctions concerning computational effects. Furthermore, monad values can be easily composed, so large, effectful computations, called "actions," can be easily built from smaller actions, the same way we might build a large procedure by calling several smaller procedures in an imperative language.

In Haskell, the choice was made early that any effectful primitives produce values in the IO monad. For example, the action of reading a byte from a file is a value of type `IO Word8`; the byte cannot be taken out of the enclosing monad, but values in the IO monad can be flexibly combined together. In this way, Haskell retains the ability to perform side-effects as in imperative languages, but everywhere a side-effect might occur is noted in the (strong) type system. Since exceptions are impure, any function that might cause an exception (directly or indirectly) must be an action in the IO monad.

As other authors have noted [17](FIXME I know I have more direct citations for this), the advantage of monads in Haskell is that we can be guaranteed that only a narrow range of effects can occur within a monadic computation. Unfortunately, the number of primitives available in the IO monad in particular is so broad that its

presence is not informative in this way. Although a value may be in the IO monad only so that it can take advantage of exceptions, we cannot deduce from the type alone that the value will not, for example, delete files from disk. Instead, new monads are regularly implemented which give stronger guarantees as to their range of possible behavior.

One such monad is the *maybe monad*, on which we will heavily focus here, but note that there are many more exception-like monads available, including the general-purpose *error* monad, which is capable of reporting any additional information required. A value of type `Maybe a` can either be a unit value, `Nothing`, or a value of type `a`, wrapped by the `Just` constructor (e.g. `Just 5` is of type `Maybe Int`). Since the type `Maybe a` is distinct from the type `a`, values of one type cannot be used in a context where the other is expected. Instead, we must use case analysis to "unwrap" the underlying `a` value, but this leads us naturally to consider the case where there is no such value! Figure 9 shows examples where we define an action (`divide`), compose it into a larger action (`calculate`[6]), and safely unwrap the monad (`main` or `main'`), naturally taking into account both the successful and unsuccessful cases. These patterns are heavily preferred in Haskell over native exceptions.

The advantages of this technique are clear if we consider Haskell's monads in relation to monads in other languages[7]. To be concrete, let us consider the null pointer in Java, `null`. The null pointer can be used in any context where an object is expected, even though it cannot satisfy any requests for data or any method calls. Under these circumstances, the compiler cannot guarantee that values stored in a variable of class A

[6]The `calculate` function attempts to compute $\Sigma_{i=-10}^{10} 1/i$ with a series of `divide` actions interspersed with pure additions, which is clearly undefined due to the term $\frac{1}{0}$.

[7]Monads are an abstract concept appearing everywhere, whether it is realized or not. Although elementry schoolers can perform addition and multiplication, they are not likely to know that they are operating within the ring algebraic structure. Likewise, every `if (obj != null) f(obj);` is a pattern-based implementation of a monad.

```
divide :: Float -> Float -> Maybe Float
divide _ 0 = Nothing
divide x y = Just (x / y)


calculate :: Maybe Float
calculate = foldM (\acc x -> (acc +) <$> divide 1 x) 0 [-10 .. 10]


main = case calculate of
    Nothing -> putStrLn "Divide by zero error"
    Just result -> print result
main' = fromMaybe (putStrLn "Divide by zero error") print calculate
```

Figure 9: Maybe monad in action

can actually be used as a value of class A; all non-primitive-type variables are vulnerable to holding a null pointer instead of an instance of A. As such, it is not uncommon to obtain a `NullPointerException` at runtime, which is almost always the result of programmer error rather than system error. To avoid this possibility, we need to add code checking some objects against `null`, but the compiler offers no help in identifying where there is danger.

In contrast, Haskell has no need for null pointers, since we can always use the maybe type. Monads such as maybe allow us to distinguish between an object which really is an object versus an "object or nothing", which could be an object or a null, and also to conveniently manipulate both. Because Haskell is strongly-typed, everywhere a null (a `Nothing`) might occur can be audited at compile time, and the programmer forced to consider the often neglected edge-case. The use of monads, in this author's direct experience and in communications with other programmers, has reduced the incidence of bugs at runtime dramatically and has led to significantly more reliable

programs. Indeed, a common phrase in the Haskell community is that "if it compiles, it works", which speaks very well of the reliability of Haskell programs, even if it is not strictly true.

The downside of monads is that any control structure they make available is simulated, rather than native. In particular, while action composition may be efficient enough when right-associated, many monads perform asymptotically worse when left-associated.[17] For simple exception-like monads, improper association can lead to the exception being needlessly propagated through every following action when we would prefer simply to short-circuit the entire computation at the point where the problem occurred. In more complex monads, such as those incorporating additional effects, improper association can cause asymptotically worse performance. Authors of new monads or composition functions must be very careful to consider the performance tradeoffs of their implementation, which can be quite subtle, especially in a lazy language such as Haskell. Monads certainly shine in their static semantics, but their dynamic semantics shows little concern for their efficiency, especially in light of the ubiquity of monadic constructs.

4 First-class Control

4.1 Introduction to First-class Control

As we have mentioned, first-class control is not a widely-implemented language feature, and so many readers will correspondingly have little or no experience with or intuition for FCC. This section gives several short examples of a shift-reset—a particular flavor of FCC—in the programming language Racket, a Scheme dialect. Readers already familiar with FCC can skip to the next section. If you are following along in a Racket interpreter, then you will need to import the control operators library with `(require racket/control)`.

A shift-reset system adds two new special forms to the language, `(reset` *`expression`*`)` and `(shift` *`identifier expression`*`)`. In `shift`, the identifier is bound within the expression, but being able to determine precisely what value is bound is exactly the intuition for FCC we are developing now.

Our first examples below appear to show that `reset` has no effect on a value passed to it. We may as well remove the calls to reset from the examples, and everything would work out the same. When there are no calls to `shift`, indeed `reset` has no effect on the process.

```
(reset 10)
  => 10
(+ 1 (reset 10))
  => 11
```

In our next examples, we add `shift`, first just inside the `reset`. Again, there seems to be no effect. However in the second example, we move the reset out further, so

that `(+ 1 ...)` is inside the `reset`, but outside the `shift`. Here, it seems like the interpreter has lost track of the `+ 1` call, so we get 10, not 11. Where did it go?

```
(+ 1 (reset (shift k 10)))
  => 11
(reset (+ 1 (shift k 10)))
  => 10
```

So far, we have not made use of `shift`'s identifier, so we will do that now. In the first expression, we see that `k` acts like a function, and that introducing it mysteriously makes up for the missing `+ 1` operation. In the second expression, we see that we can call `k` multiple times, and that it appears to act like a successor function. However, the third example shows that `k` is bound in a very strange way: by changing the `+ 1` between `reset` and `shift` to `* 2`, we see that `k` is then bound to a doubling function. So, the identifier in `shift` is bound to a very interesting value, which is something like a function whose definition depends on the program *outside* of the `shift` form.

```
(reset (+ 1 (shift k (k 10))))
  => 11
(reset (+ 1 (shift k (k (k 10)))))
  => 12
(reset (* 2 (shift k (k (k 10)))))
  => 40
```

The next examples develop systematically. They consist of a core `(shift k 100)` expression, which should create some mysterious function-like object, but ignore it. There are also calls to add one, add ten, and a reset form. We move the reset form progressively inwards, which has the effect of including more operations in the value of `k`. We see then, that the effect of this function-like object is determined not by

anything outside the `shift` form, but only by what is between the `shift` and `reset`. Although we will not show it, it turns out that the value of `k` depends on what is between the `shift` that creates it and only the nearest enclosing `reset`; additional `reset` forms make no difference.

```
(reset (+ 1 (+ 10 (shift k 100))))
  => 100
(+ 1 (reset (+ 10 (shift k 100))))
  => 110
(+ 1 (+ 10 (reset (shift k 100))))
  => 111
```

However, note that when we say "between" the shift and nearest reset, we do not mean the program text between the special forms. Instead, we mean the continuation is defined by the dynamic process that evolved between the time when `reset` was encountered and when `shift` was encountered during evaluation. This is an essential feature of FCC which directly leads to its expressive power. The next examples split `shift` and `reset` into separate parts of the program.

```
(define (push-inc-continuation thunk)
    (reset + 1 (thunk)))
(define (with-last-continuation thunk)
    (shift k (thunk k)))

(push-inc-continuation (lambda ()
    (with-last-continuation (lambda (k) 10))))
  => 10
(push-inc-continuation (lambda ()
```

Current Subexpression	**Context Stack**
`(+ 1 (* 2 3))`	
`(* 2 3)`	`(+ 1 □)`
`(+ 1 6)`	
`=> 7`	

Figure 10: Context-Stack Addition

```
   (with-last-continuation (lambda (k) (k 10)))))
=> 11
```

Before moving on to a much less trivial example, we would like to briefly examine an informal operational semantics for evaluating expressions under FCC. As we evaluate an expression, we also maintain a control stack. The control stack is a stack mostly of contexts, each of which is an expression with exactly one subexpression replaced with a hole, written □. During evaluation, complex expressions under consideration are recursively split into two parts: a context and a simpler sub-expression. The context is pushed onto the stack, and evaluation continues with the subexpression. When we reach a suitably simple expression, we reduce it to a value, then pop the topmost context. The hole in this context is replaced with the value, forming an expression on which evaluation continues. When we have a value but no remaining contexts in the stack, evaluation is complete. A small example is given in Figure 10 for concreteness.

To extend this language with first-class control, we add a stack mark, written #, which may also be pushed to the stack. Whenever we have a value, but a mark is on top of the stack, we pop the topmost marks and continue from there. There are then two special forms: the `reset` form pushed a stack mark, and the `shift` form aborts and reifies a portion of the control stack, then binds the result in the body and continues evaluation. The portion of the stack captured extends from the top of the stack up to

(but not including) the nearest stack mark. If there is no such mark, then the entire stack is captured. During reification, we form an expression by iteratively substituting contexts from this captured stack segment into the hole of the next context down, but replace the hole in the bottommost context with a fresh variable, x. When this is done, we wrap the resulting expression in a `reset` form and also a lambda which binds x, thus obtaining the reified continuation. So for example, if the stack looks like [`(+ 1` □`)`, `(+ 2` □`)`, #, `(+ 3` □`)`], then evaluation of a shift form would leave [#, `(+ 3` □`)`] on the stack, and the captured continuation would be `(lambda (hole) (reset (+ 1 (+ 2 hole))))`.

As an example, Figure 11 evaluates `(reset (+ 1 (shift k (k 10))))` to normal form. In the first step, we encounter a reset, so we push a stack mark. In the second, we see a primitive operator applied to a number and a complex expression, so we push a context and move to the complex expression. In the third step, we encounter a shift, so we create a lambda form by popping the topmost section of stack. At that point, we essentially have a standard lambda calculus term left, so evaluation proceeds without note. That the captured continuation includes a reset form is not essential to first-class control in general, but it is part of the definition of the shift-reset system.

We will now examine a more interesting example, which implements a loop with early-exit. First, recall the definition of `foldl`, which reduces a list from left-to-right according to the passed function. Two example uses of `foldl` are given as well.

```
;; loop over a list, accumulate a value
(define (foldl f xs acc)
    (if (= (length xs) 0)
        acc
        (foldl f (cdr xs) (f acc (car xs)))))
```

Current Subexpression	**Context Stack**
`(reset (+ 1 (shift k (k 10))))`	
`(+ 1 (shift k (k 10)))`	#
`(shift k (k 10))`	`(+ 1 □)`, #
`((λ (hole) (reset (+ 1 hole))) 10)`	#
`(reset (+ 1 10))`	#
`(+ 1 10)`	#, #
`=> 11`	

Figure 11: Simple FCC Reduction

```
;; foldl can be used like this:
(define (sum numbers) (foldl + numbers 0))
(define (product numbers) (foldl * numbers 1))
```

We can very cleanly define a function that returns true only when every element in a list satisfies some predicate. However, doing so can be wildly inefficient, because `foldl` always consumes every element in the list.

```
(define (slow-all predicate items)
  (define (test x acc)
    (if (predicate x)
        acc
        false))
  (foldl test items true))

;; we already know the answer by the second element
;; but foldl will just keep testing all the remaining numbers
```

```
(slow-all (lambda (x) (> x 0)) '(1 -5 2 3 ...))
```

Instead, we would like is to perform early-exit. We can do this using shift-reset, and even re-use our old `foldl` function. Here, `foldl/ee` wraps the call to `foldl` in a `reset` form, but there is no `shift` form immediately present. Instead, our definition of `all` will arrange for `break` to be called if we should find an item that does not satisfy the predicate. This implementation of `all` will not check any further items once a counter-example has been found. Figure 12 shows an outline of the reduction steps during evaluation. Alternately, the reader may confirm this with a Racket debugger.

```
(define (foldl/ee f xs acc)
  (reset (foldl f xs acc)))
(define (break ans) (shift k ans))

(define (all predicate items)
  (define (test x acc)
    (if (predicate x)
        acc
        (break false)))
  (foldl/ee test items true))
```

Although our purpose in this section was only to build some intuition, the reader may now have several questions regarding FCC. In particular, shift-reset does not lend itself well to modularity, nor have we developed a significant example demonstrating the power of FCC beyond exceptions. The FCC introduced here was particularly simple flavor, though it has seen implementation in a production language. In the following sections, we will examine several flavors of FCC in more depth. Once we have developed our system of FCC, then we will return to more significant examples of use.

Current Subexpression	**Context Stack**
`(all (λ (x) (> x 0)) '(0 1))`	
`(foldl/ee test '(0 1) true)`	
`(reset (foldl test '(0 1) true))`	
`(foldl test '(0 1) true)`	#
`(foldl test '(1)` `  (test true (car '(0 1))))`	#
`(test true 0)`	`(foldl test '(1) □)`, #

`(if ((λ (x) (> x 0)) 0)`
`  true`
`  (break false))`
	`(foldl test '(1) □)`, #

`((λ (x) (> x 0)) 0)`	`(if □ true (break false))`, `  (foldl test '(1) □)`, #
`(> 0 0)`	`(if □ true (break false))`, `  (foldl test '(1) □)`, #
`(if false true (break false))`	`(foldl test '(1) □)`, #
`(break false)`	`(foldl test '(1) □)`, #
`(shift k false)`	`(foldl test '(1) □)`, #
`false` ` => false`	#

Figure 12: foldl/ee Reduction Sketch

4.2 Undelimited Continuations

We earlier defined a continuation as the stack of computations that we will need to perform after simplifying a subexpression. In fact, this stack in its entirety is called an *undelimited continuation*. In contrast, a *delimited* continuation, which we will examine next, is only a portion of the undelimited continuation. However, undelimited continuations alone cannot implement arbitrary control structures. Instead, a lambda calculus must also be extended with mutable state [6], though this is admittedly not a barrier in production languages, even those which already include exceptions [13].

Though powerful, undelimited continuations have been found to be a poor choice in comparison with delimited continuations [3][12][18][20]. Understanding undelimited continuations is only necessary so that we can avoid the flaws in their design, and to set the stage for what follows. We will therefore end this section by summarizing a few of the arguments against undelimited control most relevant to our purpose.

Note that any subprogram with the ability to manipulate undelimited continuations can trivially take total control over and manipulate the entirety of the remaining process[8]. This is in fact the definition of undelimited control, but it is also a prescription for arbitrary code injection. Undelimited continuations are thus a very easy pathway to hijack a program, whether by accident or malicious intent. As such, they cannot serve as a basis for secure or reliable programming systems.

Furthermore, although undelimited continuations are capable of simulating arbitrary control flows, the process is difficult and subtle. In general, such simulation requires whole program transformation [6], and as such is really only of interest to compiler writers. Indeed, as far as the user-level language is concerned, undelimited

[8] In fact, a buffer overrun attack that leads to arbitrary code execution can be seen as a form of `call/cc`.

continuations are not able to fully abstract control flow [20].

More generally, it is important to note that delimited continuations are often called "composable" to distinguish them from undelimited continuations, which are not directly composable. Non-composable operations are expensive to maintain, when they can be maintained at all. From a software engineering perspective, then, we must avoid undelimited control.

4.3 Delimited Continuations

One difficulty of describing FCC is that there are so many different variants in the literature. We will consider here three broad groups of delimited control operators. In the following, we will often make reference to academic literature; lest the reader think we have abandoned our focus on production programming languages, note that all of these operators have been implemented in Racket [7]. (TODO how much are these implemented in SCheme and ML?) In fact, every delimited control operator in Racket is either presented here directly, or else can be easily simulated[9].

In what follows, we will see that Racket implements a veritable zoo of control operators. There are of course additional delimited control operators available in the literature, often with unique features of questionable utility. It would be preferable of course to agree on only a few operators, and this is perhaps the largest single issue with Racket's system of control: there is no unifying principle motivating the inclusion or exclusion of these operators, beyond what happens to be easily implementable.

[9]TODO: I should find out if they are in fact simulated

4.3.1 $^{\pm}\mathcal{F}^{\pm}$

The form of delimited control introduced in 4.1 is called *shift-reset*, and is distinguished by the fact that the reified continuation re-installs a stack mark, but the abort does *not* remove the delimiting stack mark. Another form, called *prompt-application*, is defined in [5] which is the same system, except that the reified continuation does *not* re-install the stack mark[10].

In fact, prompt-application was the first of four variants which differ only in whether the continuation-capture operator would remove the stack mark on abort and whether a reified continuation would re-install it. After [3], we will call the collection of these systems $^{\pm}\mathcal{F}^{\pm}$. The terminology derives from [5], which wrote the capture operator as $\mathcal{F}$. In each system, the operator to install a stack mark—called `reset` in 4.1 and written # in [5]—is the same. The systems differ in the behavior of there control operator; these differences are represented by the choices between plus and minus:

- $^{+}\mathcal{F}^{-}$ is the delimited $\mathcal{F}$ operator in [5]. It leaves the stack mark behind, and does not include it in the continuation.
- $^{+}\mathcal{F}^{+}$ is called *shift* from [2], an in our Sec. 4.1. It leaves the stack mark behind, but includes it in the continuation.
- $^{-}\mathcal{F}^{+}$ is similar to the *spawn* operator in [14], though we will see an exact implementation in ???. It removes the stack mark, but includes it in the continuation.
- $^{-}\mathcal{F}^{-}$ is similar to *cupto* from [11], we will examine next. It removes the stack mark, and also includes it in the continuation.

[10] In [5], what we call stack mark is called a "prompt."

A more in-depth discussion of the properties of this system is available in [3].

4.3.2 cupto

So far, we have considered control operators that can capture and abort only up to the nearest stack mark. In fact, systems based only on these operators suffer from a lack of composability. Consider the implementation of `foldl/ee` implemented in Sec. 4.1; if we were to nest calls to `foldl/ee`, the inner call would be unable to break out of the outer call. In major imperative languages, by contrast, it is simple to break out from nested loops. The depth of the situation we face here is most readily seen if we consider the analogy from [14]: "It is as if we were programming in a block-structured language that restricts us to one variable name."

To create fully abstract control structures, we need a supply of distinct stack marks, and some way to obtain a fresh mark. Our control operators will then take an additional parameter determining which mark they work with. Just such a system is developed in [11]. Importantly, it introduces the primitive **newPrompt**, which obtains a fresh stack mark. Instead of `reset`, we use **set** p **in** e, where p should evaluate to a stack mark, to push p before evaluating e. Instead of the ${}^{\pm}\mathcal{F}^{\pm}$ operators, we use a **cupto** p **as** k **in** e operator, which captures and aborts up to the nearest p mark (bypassing other marks), and binds k to the captured continuation in e. With **cupto**, the stack mark is neither left behind, nor is it reinstalled, as in ${}^{-}\mathcal{F}^{-}$. However, the authors do not express a strong opinion on the flavors presented by ${}^{\pm}\mathcal{F}^{\pm}$, and in fact note that it is easy to simulate other flavors starting from ${}^{-}\mathcal{F}^{-}$.

4.3.3 fcontrol

In the operators so far, the control operator has not only been responsible for effecting continuation capture and abort, but is also responsible for combining the reified continuation with any other required values which must be available locally. In contrast, exception mechanisms bundle up any local values into an exception, but the control flow after the abort is specified by the delimiter—a catch block. In [18], Sitaram generalized try-catch to include continuation capture, introducing the control operator `fcontrol` and the corresponding delimiting operator `run`. This system, apart from specifying the handler on a delimiter and therefore requiring any required local data to be packaged with the control operator, is essentially the same as those discussed above, and so is also subject to the same range of variation.

4.3.4 Analysis

The reader might surely wonder at this point if there are further variants. Indeed, there are, but these are either easily simulated by the above systems, or else are so unique to be of questionable utility. Given the possibilities we have discussed, we see several choices:

1. whether to remove the delimiting stack mark on abort,
2. whether to include the stack mark in the captured continuation,
3. whether to provide a supply of fresh stack marks, or else allow only a single mark, and
4. whether to specify the handler with the control operator or with the delimiting operator.

While these are orthogonal, certain choices are more expressive than others. As already mentioned, choosing to remove the delimiting mark but not include it in the captured continuation allows simple simulation of the other three possibilities in this space. Further, providing a supply of fresh stack marks allows us to use multiple control structures at once without being concerned with adverse interactions.

The choice in (4) has no immediately clear answer. It seems in some programs there is sufficient information near the control operator to proceed with computation; in other systems, that information might only exist near the delimiting operator. Although the focus in research has been into the former systems, exception handling certainly falls into the latter category.

TODO

Since exceptions cause a crash when they are not delimited, we should choose the same when a capture is not delimited.

Single-mark control can implement finally behavior, but many-mark cannot.

There are expressive type systems for first-class control. There may even be one which can statically rule out a failure to delimit.

NOTES

What happens when a matching stack mark is not found during capture? Either we capture the whole stack, or we crash. The choice to capture the whole stack is problematic because a capture will sometimes be delimited and sometimes undelimited, depending on the context. This is highly unintuitive behavior, and not likely useful. More likely, a programmer has made an error, and we should stop execution immediately, following the "catch errors early" principle.

An attempt to capture a delimited continuation leads to a program crash when a matching stack mark cannot be found on the control stack. Type systems that can statically rule out these crashes are known (CITE really? there's got to be a citation from mfdc that says so), but these systems are complex. Therefore, the type system for languages incorporating delimited continuations is often left unsound in this respect. This is the tactic used in Racket.

When we have a fixed prompt, we can easily trap all attempts to abort past a certain point. With first-order fcontrol, we can implement finally. However, when we have dynamically-generated prompts, all a subroutine has to do is abort to a prompt that is not caught by a finalizer of the same prompt. The introduction of dynamic prompts then allows for composable nesting of control constructs, but requires an additional mechanism to implement finalizers. In either case, there is no "initializer" system available to run whenever control enters a portion of code.

4.4 Dynamic Wind

Dynamic-wind is another simple concept, which is broadly a generalization of try-finally. Dynamic-wind is a form which takes three blocks[11] of code: a before-, body-, and after-block. Control then attempts to enter the body-block, but whenever the body is entered, whether initially or because it is placed back onto the stack, the before-block is executed first. Furthermore, no matter how control exits the body, the after-block will be evaluated.

TODO where multiple prompts alone cannot guarantee setup or cleanup when a dynamic extent is entered/exited, the addition of dynamic-wind gives us this capability

It should be clear then, that try-finally is a limit case of dynamic-wind, where the before-block is empty. Conversely, dynamic-wind generalized try-finally by not only allowing code to be registered for execution on exit, but also on entrance. Try-finally is sufficient to manage cleanup of resources in the presence of non-local exit, but can manage acquisition of resources under only local entry. Dynamic-wind allows the programmer to ensure that both setup and cleanup code are properly executed even under non-local control flow such as that introduced by FCC. Since the two mechanisms are so similar, they share essentially the same advantages and disadvantages. In any case, we will not repeat our analysis here.

[11] Technically, it takes three thunks, but there's really no good reason to introduce the boilerplate `(lambda () ...)` all over the place. So, we will speak as if the arguments were passed unevaluated, as if to a special form.

5 Review

RAII: abstractible, unsafe or non-deterministic, verbose, directly applies only to physical resources try-finally: non-abstractible, determistic, applies generally with/using: abstractible, deterministic, verbose, directly applies only to physical resources scope statements: non-abstractible, concise, deterministic, applies generally monads: abstractible, deterministic, concise, poor performance characteristics, applies generally

the question of number of use-cases is difficult.

although user-implemented monads offer poor performance characteristics, the type system is very good at checking exceptions in a non-annoying way. before control exits from blocks, some code must be executed: usually implemented by marking the stack. allow for transactions by distinguishing between normal/abnormal exit.

from exceptions: determinism, mark the stack with handlers, mark the stack with finalizers, abstraction

from fcc: reify/replace parts of the stack, mark stack fragments with initializers, abstraction

use a rich type system to check exception safety

find a place for this: it is commonly claimed that exceptions can be implemented with continuations + mutable state. Unfortunately, this strategy is not very pure

Good features found: 1) stack-allocated memory can be cleaned deterministically, 2) directly (i.e. inline) install deterministic handlers (e.g. catch-clauses), 3) reuse setup/cleanup pairs (well, reuse more generally, but...) 4) straightforward or (better)

automatic auditing for exception safety, 5) cleanup must be executed no matter how control exits, 6) corollary: setup must be executed no matter how control enters, 7) discriminate between normal and abnormal exit Features to avoid: 1) manual management of heap-alloc'd memory, 2) destruction/cleanup tied to garbage collector, 3) boilerplate per method of setup/cleanup, 4) manual translation to continuation-passing style, 5) indiscriminate laziness, 6) codifying any specific patterns (other patterns are always left un-codified, but if we let the user codify patterns himself, no pattern need be codified), Initial goals were to promote efficient code re-use in favor of pattern-based programming. We won't worry about syntactic abstraction, as that's independent and has several possible solutions, though our demonstration langauge does have it.

MEMOS:

- 'syntactic theory of fcc' could not give higher praise to fcc as a implementation technique

Part II

Thesis

Our thesis begins with the presentation of an extended lambda calculus designed for secure first-class control, which we call the SFCC calculus or λ_{SFCC}. We then briefly examine its relationship to a few important existing control structures. Next, we present an operational semantics for our system in the form of an interpreter, and note additional efficiencies that may be garnered with an implementation. Finally, we examine a wide range of control structures, both mundane and advanced, and show implementations of these in our calculus.

As in most discussions of formal semantics, we will be working simultaneously with syntax, semantics and metasyntax. To be unambiguous, we will typeset each level differently. Metasyntax will be in *italics*, semantic objects will be written in **boldface**, and syntax will be written either in sans-serif for keywords or in `typewriter` for variable names.

In the design of SFCC, our guiding principle has been in relation to the four parameters mentioned in 4.3.4: at each step, we have chosen notions which are able to implement either choice. The techniques we use are drawn primarily from four existing control concepts: the system $\pm\mathcal{F}\pm$ as presented in [3], the `fcontrol` operator from [18], dynamic-wind from Scheme, and the ability to discriminate normal and abnormal exit from D. Our system was developed independently of the requirements summarized in §5, but we will show that it does meet those standards. This sort of demonstration is interesting because it shows that the power of the system extends beyond its direct influences.

We will present the concepts of the SFCC calculus in stages, each building on the last. We have taken care so that the presentation of each system will remain valid we extend it with additional concepts. When we have developed SFCC in full, we give the system as a whole, so that readers may skip over the intermediate stages of development.

6 The SFCC Calculus

6.1 Base Calculus

We begin our work with λ_v, the call-by-value lambda calculus. We have chosen call-by-value specifically so that we can easily control the order of evaluation, esp. to obtain determinism. To briefly review, we give the grammar in Figure 13. The notation $[x\backslash e']e$ is used for capture-avoiding substitution. The only reduction rule is β-reduction:

$$(\lambda x.e)\ v \mapsto [x\backslash v]e$$

Values	v	::=	$(\lambda x.e)$	abstraction
Expressions	e	::=		
		|	v	
		|	x	variable
		|	$(e\ e)$	application

Figure 13: Grammar of λ_v Calculus

As in most other presentations, we freely drop parenthesis where doing so would be unambiguous. We also include a few common syntactic niceties.

- Adjacent lambdas may be written as a single lambda.

$$\lambda x_1, \ldots, x_n.e \equiv \lambda x_1.\lambda x_2 \ldots \lambda x_n.e$$

- Lambdas may discard their argument.

$$\lambda\_.e \equiv \lambda x.e \quad \text{where } x \notin fv(e)$$

- Expressions may be sequenced; semicolon is right-associative.

$$e_1; e_2 \equiv (\lambda\_.e_2)\ e_1$$

- Let expressions increase readability,

$$\mathsf{let}\ x = e'\ \mathsf{in}\ e \equiv (\lambda x.e)\ e'$$

- and may make multiple bindings in sequence.

$$\mathsf{let}\ x_1 = e_1; \ldots; x_n = e_n\ \mathsf{in}\ e \equiv \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = e_n\ \mathsf{in}\ e$$

- We will occasionally have need of "zero-argument" lambdas. These are really no different than lambdas which discard their argument, which should not have required any work to compute. We nevertheless include syntax for them, so as to better express our intentions.

$$\lambda().e \equiv \lambda\_.e$$

$$e\ () \equiv e\ v$$

6.2 Continuation Values

In our first extension, we add the notion of a subcontinuation—a reified continuation. The grammar is given in Figure 14. It is clear from structural induction that every subcontinuation has exactly one $\Box$, also called a hole. Subcontinuations are always defined such that replacing this hole by an expression yields an expression. We denote the replacement of the hole in a subcontinuation K by an expression e as $K[e]$.

Values	v	::=	$\ldots$	
		$\mid$	$\langle K \rangle$	subcontinuation
Continuations	K	::=		
		$\mid$	$\Box$	hole
		$\mid$	$K\ e$	
		$\mid$	$v\ K$	

Figure 14: Extend with Subcontinuations

Subcontinuations purposely bear a remarkable resemblance to functions. They essentially are functions of one variable, where that variable is only used once in the function body. Unlike lambda-introduced functions which take their argument by value, subcontinuations take their argument by-name. To this end, we add the following reduction rule so that application is productive for both functions and subcontinuations:

$$\langle K \rangle\ e \mapsto K[e]$$

A dedicated notion of subcontinuation—as distinct from function—is important so that the calculus is able to manipulate the continuation before proceeding with other work. This will become particularly important for several control structures we

implement later, esp. a control operator for continuation capture without abort.

Readers might note that Felleisen in [5] models continuations using lambdas in continuation passing style. This reduces the size of his grammar, but at the cost of the presence of difficult to untangle values, such as $\lambda z.(\lambda y.(\lambda x.x)\ (1 + y))\ (f\ z)$, which is simply $\langle 1 + f\ \Box \rangle$ in our notation. As we will be regularly working with continuations, we have opted for a calculus which is easy to read, even at the cost of additional nonterminals and productions in the grammar. Nevertheless, Felleisen clearly shows that this distinction is merely cosmetic.

6.3 First-class Control

We now move to the core of the system—the control and delimiting operators—which will require several interlocking notions: those of prompt creation, and delimiting and control operators. The grammar is given in Figure 22. The last two productions for K now include two K non-terminals; we recover the existence of a single salient $\square$ by ignoring the K' in these productions as we search for $\square$.

Values	v	::=	$\ldots$	
		$\mid$	p	prompt
Expressions	e	::=	$\ldots$	
		$\mid$	ν	new prompt
		$\mid$	$\#_{e_p}^{e_h} e$	delimiting operator
		$\mid$	$\mathcal{F}_{e_p}^{K} e$	control operator
Continuations	K	::=		
		$\mid$	$\#_{K}^{e_h} e$	
		$\mid$	$\#_{p}^{K} e$	
		$\mid$	$\#_{p}^{v} K$	
		$\mid$	$\mathcal{F}_{K}^{K'} e$	
		$\mid$	$\mathcal{F}_{p}^{K'} K$	

Figure 15: Extend with Control Operators

First, we have decided to allow for both control and delimiting operators to be tagged as in §4.3.2. To do this, we will need a new category of values—prompts—which we write with the metavariable p. Further, we need an operator with which we can create a new prompt value on demand, which we write ν. This will require a crosscutting change to our evaluation procedure. Where before we simply reduced one

expression to another, evaluation will now need to track which prompts are in use. To do this, we use relations of the form $e/P \mapsto e'/P'$, where P is a set of prompts. We need not restate our previous reduction rules, however, as we can simply make the following identification:

$$e \mapsto e' \equiv e/P \mapsto e'/P$$

which is to say, we can leave out the specification of the set of prompts when it is neither inspected nor altered.

With this in mind, the reduction rule for ν is

$$\nu/P \mapsto p/P \cup \{p\} \quad \text{where } p \notin P$$

Unlike other reduction rules, this one cannot be used underneath an abstraction. Further, a if e contains a ν subexpression, multiple copies of e in the same program will produce different results. Thus, each application of $\lambda().\nu$ produces a distinct value.

After [5], the delimiting operator is written # and the control operator written as $\mathcal{F}$. Both enclose an expression called the body, which is written to the right of the operator. For the delimiting operator, the body is within the dynamic extent of the operator, and for the control operator, the body should evaluate to a handler. Both also take an expression, written as a subscript, which should reduce to a prompt. Through that, a control operator is able to identify a matching delimiter.

The reduction rule for delimiters simply allows a value to pass through it without modification:

$$\#_p^e v \mapsto v$$

Reducing a control operator is much more intricate, however, as it involves

delimited continuation capture. Control operators begin reduction when their body is reduced to a value. They recurse up the expression until they match an appropriate delimiter.

The base rule matches a control operator tagged with a prompt to an immediately enclosing delimiter tagged with the same prompt. At that point, we apply the handler and the captured continuation to the value passed to the control operator.

$$\#_p^{v_h} \mathcal{F}_p^K v \mapsto v\ v_h\ \langle K \rangle$$

We now need reductions which allow a control operator to walk up an expression, capturing a continuation. We will need one reduction rule for each place an expression occurs in each production. While this means there will need to be many rules, they all follow the same pattern: we keep the control operator, but move the surrounding expression into the control operator's captured continuation, replacing where the control operator was with the previously captured continuation.

$$(\mathcal{F}_p^K v)\ e_2 \mapsto \mathcal{F}_p^{K\ e_2} v$$

$$v_1\ \mathcal{F}_p^K v \mapsto \mathcal{F}_p^{v_1\ K} v$$

$$\#_{\mathcal{F}_p^K v}^{e_h} e \mapsto \mathcal{F}_p^{\#_K^{e_h} e} v$$

$$\#_{p'}^{\mathcal{F}_p^K v} e \mapsto \mathcal{F}_p^{\#_{p'}^K e} v$$

$$\#_{p'}^{v_h} \mathcal{F}_p^K v \mapsto \mathcal{F}_p^{\#_{p'}^{v_h} K} v \quad \text{where } p \neq p'$$

$$\mathcal{F}_{\mathcal{F}_p^K v}^{K'} e \mapsto \mathcal{F}_p^{\mathcal{F}_K^{K'} e} v$$

$$\mathcal{F}_{p_1}^{K'} \mathcal{F}_{p_2}^K v \mapsto \mathcal{F}_{p_2}^{\mathcal{F}_{p_1}^{K'} K} v$$

It is not often necessary to specify an initial continuation other than $\Box$ for control operators in the source language. Thus, we will often simply elide it:

$$\mathcal{F}_{e_p} e \equiv \mathcal{F}_{e_p}^{\Box} e$$

6.4 Extent Guards

Having now introduced control operators, we turn to the other main part of the system—the operators which can ensure correct setup and cleanup of resources. These we call extent guards, because they register expressions which are then executed when control enters/exits the dynamic extent of the guard. We supply enough guards to recognize local and non-local control flow, or either. The extent guard concept can be approached in any of several ways, two of which we will now give.

First, they are an extension of D's scope statements. Where scope statements can be triggered on normal, abnormal, or any exit, extent guards can also be triggered on normal, abnormal or any entrance.

Second, extent guards may be seen as an atomization of the aspects of dynamic-wind. In dynamic-wind, code for both entrance and exit are specified at one, and these codes are executed regardless of whether the entrace/exit was normal or abnormal. The entrance and exit extent guards can be specified independently of the other, and can also discriminate between normal and abnormal jumps.

To help keep track of these six extent guards, we use mnemonics in our notation. Those guards which are executed on entrance are represented with upwards arrows, which those executed on exit with downwards arrows; this is in keeping with an upward-growing stack convention. Extent guards executed on local control flow are represented with a simple arrow ($\uparrow$), those executed on non-local control flow with a broad arrow ($\Uparrow$), and those executed on any control flow with double arrows ($\uparrow\uparrow$).

Expressions	e	::=	$\ldots$
		$\mid$	$\#^{e_g}_{\Uparrow} e$ non-local entrance guard
		$\mid$	$\#^{e_g}_{\Downarrow} e$ non-local exit guard
Continuations	K	::=	
		$\mid$	$\#^{e}_{\Uparrow} K$
		$\mid$	$\#^{e}_{\Downarrow} K$

Figure 16: Extend with Extent Guards

In the grammar, given in Figure 23, we only add the forms for guards triggered on non-local control flow. We do this to reduce the size of our calculus, recognizing that the others are easily simulated:

$$\#^{e_g}_{\uparrow} e \equiv e_g; e$$

$$\#^{e_g}_{\downarrow} e \equiv \mathsf{let}\ x = e\ \mathsf{in}\ e_g; x$$

$$\#^{e_g}_{\Uparrow} e \equiv \#^{e_g}_{\uparrow} \#^{e_g}_{\Uparrow} e$$

$$\#^{e_g}_{\Downarrow} e \equiv \#^{e_g}_{\downarrow} \#^{e_g}_{\Downarrow} e$$

For each new expression, we need two reduction rules. Note that these only apply to the non-local extent guards specified in the grammar, not to the simulations. First, the regular reduction rules

$$\#^{e}_{\Uparrow} v \mapsto v$$

$$\#^{e}_{\Downarrow} v \mapsto v$$

are a straightforward extension of the rule $\#^{e}_{p} v \mapsto v$ from the last section.

The next pair relate to continuation capture, and they each slightly break the pattern for continuation capture established in the last section. First, examine the

capture rule for non-local exit:

$$\#^{e}_{\Downarrow}\mathcal{F}^{K}_{p}v \mapsto e; \mathcal{F}^{\#^{e}_{\Downarrow}K}_{p}v$$

In addition to floating the control operator up the expression tree, this rule also inserts a copy of the guard expression to be evaluated before continuation capture can proceed. The case for non-local entrance is somewhat more subtle:

$$\#^{e}_{\Uparrow}\mathcal{F}^{K}_{p}v \mapsto \mathcal{F}^{e;\#^{e}_{\Uparrow}K}_{p}v$$

Here, capture proceeds as normal, but the copied guard expression is inserted into the captured continuation. The idea is that when the continuation is reinstated, the guard expression will be evaluated before another redex is found.

6.5 Barriers

Finally, we introduce one additional stack mark, which we call a barrier. Barriers are introduced in order to delimit all control operators, regardless of their prompt. The grammar is given in Figure 17.

Marks	e	::=	$\ldots$	
		$\mid$	$\#_{\blacksquare}^{e_h} e$	barrier
Continuations	K	::=		
		$\mid$	$\#_{\blacksquare}^{K} e$	

Figure 17: Extend with Barriers

There are two normal reduction rules. The first should look familiar; it simply allows values to unwind past the barrier.

$$\#_{\blacksquare}^{v_h} v \mapsto v$$

The other reduction concerns an attempt to bypass the barrier with a control operator:

$$\#_{\blacksquare}^{v_h} \mathcal{F}_p^K v \mapsto v_h \; p \; \langle K \rangle \; v$$

In this case, the various values involved are diverted to the handler specified by the barrier. Usually, the handler should simply initiate an error-handling mode, perhaps using its arguments to produce diagnostic messages.

Finally, there is a standard continuation capture rule:

$$\#_{\blacksquare}^{\mathcal{F}_p^K v} e \mapsto \mathcal{F}_p^{\#_{\blacksquare}^{K} e} v$$

Barriers are likely the most difficult expressions of this calculus to give a type. Indeed, it may well be impossible to do so in any practical manner. The formulation of barriers here is nevertheless the most general possibility for dynamically-typed languages. If we were to adapt this concept to statically-typed languages, we need only recognize the original purpose for barriers: to prevent control from escaping the extent the barrier sets. In statically-typed languages, we can restrict the barrier to either raise an appropriate exception or simply terminate the program immediately. In this way, the handler is not dependent on the type of value and captured continuation obtained. We will say more on this as we develop the applications of this system.

6.6 Summary of the SFCC Calculus

This section collects the formal system of SFCC described above. In addition, it applies somewhat more formalism in defining what reductions may be applied in what cases, thus giving a formal definition of equivalence. We then collect the various syntactic abbreviations we have so far defined. These do not extend the expressive power of the calculus in the sense of [4], and so we may regard these abbreviations as inessential to the theory; nevertheless, their use greatly improves readability, which will be important when we apply our calculus.

We begin with the grammar, given in Figure 18.

Values	v	$::=$	$\lambda x.e \mid \langle K \rangle \mid p$
Expressions	e	$::=$	$v \mid x \mid e\ e$
		$\mid$	$\nu \mid \#^{e_h}_{e_p} e \mid \mathcal{F}^K_{e_p} e$
		$\mid$	$\#^{e_g}_{\Uparrow} e \mid \#^{e_g}_{\Downarrow} e \mid \#^{e_h}_{\blacksquare} e$
Continuations	K	$::=$	$\Box$
		$\mid$	$K\ e \mid v\ K$
		$\mid$	$\#^{e_h}_{K} e \mid \#^{K}_{p} e \mid \#^{v}_{p} K$
		$\mid$	$\mathcal{F}^{\Box}_{K} e \mid \mathcal{F}^{\Box}_{p} K$
		$\mid$	$\#^{e}_{\Uparrow} K \mid \#^{e}_{\Downarrow} K \mid \#^{K}_{\blacksquare} e$

Figure 18: Grammar of SFCC Calculus

The equational theory is specified by a set of reduction relations. A reduction relation takes the form

$$e/P \mapsto e'/P'$$

where P, P' are sets of prompts and R is a reduction rule, or set of such rules. In the

case where $P = P'$, we will elide mention of the prompt set:

$$e \mapsto e' \equiv e/P \mapsto e'/P$$

We now turn to the individual reduction rules of SFCC. The primary reductions are those dealing with application:

$$(\lambda x.e)\ v \mapsto [x \backslash v]e \qquad \langle K \rangle\ e \mapsto K[e]$$

There are then three rules dealing with the creation of prompts and the delimiting of continuation capture:

$$\nu/P \mapsto p/P \cup \{p\} \quad \text{where } p \notin P$$

$$\#_p^{v_h} \mathcal{F}_p^K v \mapsto v\ v_h\ \langle K \rangle$$

$$\#_{\blacksquare}^{v_h} \mathcal{F}_p^K v \mapsto v_h\ p\ \langle K \rangle\ v$$

Next, two rules implement extent guards, which offer the ability to trigger expressions based on no-local control flow:

$$\#_{\Downarrow}^e \mathcal{F}_p^K v \mapsto e; \mathcal{F}_p^{\#_{\Downarrow}^e K} v \qquad \#_{\Uparrow}^e \mathcal{F}_p^K v \mapsto \mathcal{F}_p^{e;\#_{\Uparrow}^e K} v$$

These together comprise the interesting core of the system.

Four more rules are required to allow values to exit locally past various stack marks.

$$\#_p^{v_h} v \mapsto v \qquad \#_{\Uparrow}^e v \mapsto v \qquad \#_{\Downarrow}^e v \mapsto v \qquad \#_{\blacksquare}^{v_h} v \mapsto v$$

Finally, a number of rules are required to implement continuation capture. Each of these rules follows a standard pattern, allowing the control operator to float to the top

of an expression by moving the surrounding expression to surround the control operator's K slot.

$$(\mathcal{F}_p^K v)\ e_2 \mapsto \mathcal{F}_p^{K\ e_2} v \qquad v_1\ \mathcal{F}_p^K v \mapsto \mathcal{F}_p^{v_1\ K} v$$

$$\#_{\mathcal{F}_p^K v}^{e_h} e \mapsto \mathcal{F}_p^{\#_K^{e_h} e} v \qquad \#_{p'}^{\mathcal{F}_p^K v} e \mapsto \mathcal{F}_p^{\#_{p'}^K e} v$$

$$\#_{p'}^{v_h} \mathcal{F}_p^K v \mapsto \mathcal{F}_p^{\#_{p'}^{v_h} K} v \quad \text{where } p \neq p'$$

$$\mathcal{F}_{\mathcal{F}_p^K v}^{K'} e \mapsto \mathcal{F}_p^{\mathcal{F}_K^{K'} e} v \qquad \mathcal{F}_{p_1}^{K'} \mathcal{F}_{p_2}^K v \mapsto \mathcal{F}_{p_2}^{\mathcal{F}_{p_1}^{K'} K} v$$

$$\#_{\blacksquare}^{\mathcal{F}_p^K v} e \mapsto \mathcal{F}_p^{\#_{\blacksquare}^K e} v$$

Although most of the calculus is confluent, the presence of ν and its associated reduction rule present some difficulties. The difficulty occurs because some of the reduction rules make a copy of an expression. Since ν results in a distinct prompt on each evaluation, we will note a difference whether the ν expression is copied or its result is copied. Since most language make use of a left-to-right applicative order evaluation strategy, we will simply generalize this strategy to our calculus. The system is then trivially confluent, as there is at most one reduction possible. No doubt conservative extensions of this strategy may be adopted, and we leave this open to later authors, who may wish to make additional extensions (such as the inclusion of mutable state) which would impact the extension.

When we have $e_1/P_1 \mapsto_R e_2/P_2$, then we also have

$$e_1\ e/P_1 \mapsto_R e_2\ e/P_2 \qquad v\ e_1/P_1 \mapsto_R v\ e_2/P_2$$

$$\#_{e_1}^{e_h} e/P_1 \mapsto_R \#_{e_2}^{e_h} e/P_2 \qquad \#_p^{e_1} e/P_1 \mapsto_R \#_p^{e_2} e/P_2 \qquad \#_p^{v_h} e_1/P_1 \mapsto_R \#_p^{v_h} e_2/P_2$$

$$\mathcal{F}_{e_1}^K e/P_1 \mapsto \mathcal{F}_{e_2}^K e/P_2 \qquad \mathcal{F}_p^K e_1/P_1 \mapsto \mathcal{F}_p^K e_2/P_2$$

$$\#^{e_g}_{\Uparrow} e_1/P_1 \mapsto \#^{e_g}_{\Uparrow} e_2/P_2 \qquad \#^{e_g}_{\Downarrow} e_1/P_1 \mapsto \#^{e_g}_{\Downarrow} e_2/P_2$$

$$\#^{e_1}_{\blacksquare} e/P_1 \mapsto \#^{e_2}_{\blacksquare} e/P_2 \qquad \#^{v_h}_{\blacksquare} e_1/P_1 \mapsto \#^{v_h}_{\blacksquare} e_2/P_2$$

We are finally ready to define program equivalence. We say that $e_1 \equiv e_2$ if and only if there is an e' for which $e_1 \mapsto^* e'$ and $e_2 \mapsto^* e'$, where $\mapsto^*$ is the reflexive-transitive closure of $\mapsto$.

Finally in this section, we will just gather the syntactic abbreviations we had defined in our stepwise development of λ_{SFCC}. First, there are a few additional ways of writing functions.

$$\lambda x_1, \ldots, x_n.e \equiv \lambda x_1.\lambda x_2 \ldots \lambda x_n.e$$

$$\lambda\_.e \equiv \lambda x.e \quad \text{where } x \notin fv(e)$$

$$\lambda().e \equiv \lambda\_.e \qquad e\ () \equiv e\ v$$

Then, we have sequencing and let-expressions.

$$e_1; e_2 \equiv (\lambda\_.e_2)\ e_1$$

$$\mathsf{let}\ x = e'\ \mathsf{in}\ e \equiv (\lambda x.e)\ e'$$

$$\mathsf{let}\ x_1 = e_1; \ldots; x_n = e_n\ \mathsf{in}\ e \equiv \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = e_n\ \mathsf{in}\ e$$

Since we will almost always write the $\mathcal{F}$ operator with an initially empty continuation, we have:

$$\mathcal{F}_{e_p} e \equiv \mathcal{F}^{\square}_{e_p} e$$

Finally, the extent guards for local control flow and for either local or non-local control

flow area are not in the core grammar, since they are easily simulated.

$$\#_{\uparrow}^{e_g} e \equiv e_g; e \qquad \#_{\downarrow}^{e_g} e \equiv \mathsf{let}\ x = e\ \mathsf{in}\ e_g; x$$

$$\#_{\Uparrow}^{e_g} e \equiv \#_{\uparrow}^{e_g} \#_{\Uparrow}^{e_g} e \qquad \#_{\Downarrow}^{e_g} e \equiv \#_{\downarrow}^{e_g} \#_{\Downarrow}^{e_g} e$$

7 Relationship to Existing Operators

7.1 Preliminaries

Before delving into various operators, we will first want to define a few control operators of our own to help us along the way.

Note that once the control operator $\mathcal{F}$ reaches its delimiter, values are passed to the control operator's body value. Often, the body will be ill-equipped to continue computation, and will simply pass control directly to the delimiter's handler. This we call *abort* and write with the $\mathcal{A}$ operator:

$$\begin{aligned}\mathcal{A}_{e_p} e \equiv \mathsf{let}\ & x_p = e_p, x_v = e \\ \mathsf{in}\ & \mathcal{F}_{x_p} \lambda h, k.h\ k\ x_v\end{aligned}$$

We will also want to agree on a few prompts beforehand. In several FCC structures, there is no notion of prompt, so for these we will introduce a default prompt, $\mathbf{p}$ for these structures to use. TODO $\mathbf{p_0}$ We will introduce a few more as we go along, but each of them will be smaller in scope. Since we have written these in boldface, their appearance represents the prompt itself, rather than a name to which the prompt is bound; thus, we are able to ignore questions of variable capture.

7.2 MFDC

Since we have already mentioned the major inspirations for the SFCC calculus, it only makes sense to begin our survey of relationships with those inspirations. The system ${}^{\pm}\mathcal{F}^{\pm}$ given in *A Monadic Framework for Delimited Continuations*[3], which we abbreviate MFDC, relies on an extension of the lambda calculus with a system of four operators.

- Their operator newPrompt creates a fresh prompt.
- Their delimiting operator is pushPrompt $e_p\ e$. It takes a prompt an an expression, but it does not install a handler.
- Their control operator is withSubCont $e_p\ e_h$. It evaluates its two arguments, a prompt and a function. It captures the appropriate continuation and passes it to the function.
- Finally, their pushSubCont $e_k\ e$ takes a subcontinuation value and adds it to the control context before evaluating its body.

Two of these operators are directly reflected in our system, and a third is easily translated:

$$\mathsf{newPrompt} \equiv \nu$$

$$\mathsf{pushSubCont}\ e_k\ e \equiv e_k\ e$$

$$\mathsf{withSubCont} \equiv \lambda x_p, x_f.\mathcal{F}_{x_p}\lambda\_, k.x_f\ k$$

The implementation of withSubCont does little more than arrange for the prompt-specified handler to be discarded during the call to the function.

The fourth operator requires a design choice. We will implement an equivalence of the form $\mathsf{pushPrompt}\ e_p\ e \equiv \#_{e_p}^{v_?} e$, but the choice of $v_?$ is not fixed by the semantics of [3]. Two major options present themselves: either provide a simple value, not necessarily usable as a handler, or provide a handler which will continue to propagate the process of continuation capture when it is called.

$$\mathsf{pushPrompt}\ e_p\ e \equiv \#_{e_p}^{()} e$$

or

$$\mathsf{pushPrompt}\ e_p\ e \equiv \mathsf{let}\ p = e_p\ \mathsf{in}\ \#_p^{\lambda k,x.\mathcal{F}_p^k\ \Box x} e$$

Neither option is sufficient to rule out crashes and unexpected behavior when the operators of MFDC are used in the SFCC calculus, but for the purposes of illustrating MFDC, the issue will never become important. In the interests of brevity we will simply choose the former.

We will now examine an example from [3].

$(\lambda p.2 + \mathsf{pushPrompt}\ p$
  $\mathsf{if}\ (\mathsf{withSubCont}\ p$
    $(\lambda k.(\mathsf{pushSubCont}\ k\ \mathbf{False}) + (\mathsf{pushSubCont}\ k\ \mathbf{True})))$
  $\mathsf{then}\ 3\ \mathsf{else}\ 4)$
$\mathsf{newPrompt}$

This translates into the SFCC calculus as the following (we have taken the liberty of applying an η-conversion in the body of the $\mathcal{F}$):

$$
\begin{array}{l}
(\lambda p.2 + \#_p^{()} \\
\quad \mathsf{if}\ ((\lambda x_p, x_f.\mathcal{F}_{x_p}\lambda\_.x_f)\ p \\
\quad\quad \lambda k.(k\ \mathbf{False}) + (k\ \mathbf{True})) \\
\quad \mathsf{then}\ 3\ \mathsf{else}\ 4) \\
\nu
\end{array}
$$

First, the ν is evaluated to a fresh prompt—let's represent that with **a**—which is then substituted into the body of the lambda. Another pair of β-reductions bring the expression to the point where we will need to begin continuation capture.

$$
\begin{array}{l}
2 + \#_{\mathbf{a}}^{()}\mathsf{if}\ (\mathcal{F}_{\mathbf{a}}\lambda\_, k. \\
\quad\quad (k\ \mathbf{False}) + (k\ \mathbf{True})) \\
\quad \mathsf{then}\ 3\ \mathsf{else}\ 4
\end{array}
$$

Continuation capture terminates quickly, since there is a delimiter just one node away. The base-case reduction for continuation capture leads to the following, which

reduces to 9 by a few simple reductions:

$$
\begin{aligned}
& 2 + (\lambda\_, k.(k\ \mathbf{False}) + (k\ \mathbf{True}))\ ()\ \langle \mathsf{if}\ \square\ \mathsf{then}\ 3\ \mathsf{else}\ 4 \rangle \\
\mapsto\ & 2 + (\langle \mathsf{if}\ \square\ \mathsf{then}\ 3\ \mathsf{else}\ 4 \rangle\ \mathbf{False}) + (\langle \mathsf{if}\ \square\ \mathsf{then}\ 3\ \mathsf{else}\ 4 \rangle\ \mathbf{True}) \\
\mapsto\ & 2 + (\mathsf{if}\ \mathbf{False}\ \mathsf{then}\ 3\ \mathsf{else}\ 4) + (\mathsf{if}\ \mathbf{True}\ \mathsf{then}\ 3\ \mathsf{else}\ 4) \\
\mapsto\ & 2 + 4 + 3 \\
\mapsto\ & 9
\end{aligned}
$$

7.3 fcontrol

The next inspiration we mention is fcontrol. Recall that fcontrol consists of the delimiting operator %, and the control operator fcontrol. The delimiting operator requires a body expressionand a handler, and the control operator requires only an abort value. Both operators may be equipped with a prompt, or allowed to use the default prompt.

Implementations of these operators are given below. Since the semantics in [18] applies the handler to arguments in reverse order, we will make use here of the well-known function $\mathbf{flip} = \lambda f, x, y.f\ y\ x$. The following are faithful implementations of the operators defined in [18]:

$$\begin{aligned}
\textbf{run-tagged} &\equiv \lambda p, f, h.\#_p^h f\ () \\
\textbf{fcontrol-tagged} &\equiv \lambda p, x.\mathcal{A}_p x \\
\textbf{run} &\equiv \textbf{run-tagged p} \\
\textbf{fcontrol} &\equiv \textbf{fcontrol-tagged p} \\
\% \ e\ e_h &\equiv \textbf{run}\ (\lambda_.e)\ e_h
\end{aligned}$$

7.4 Scope Statements and Dynamic Wind

$$\textsf{scope(exit)}\ e_g; e \equiv \#_{\Downarrow\Downarrow}^{e_g} e$$

$$\textsf{scope(success)}\ e_g; e \equiv \#_{\downarrow}^{e_g} e$$

$$\textsf{scope(failure)}\ e_g; e \equiv \#_{\Downarrow}^{e_g} e$$

$$\textbf{dynamic-wind} \equiv \lambda pre, body, post.\#_{\uparrow\uparrow}^{pre\ ()} \#_{\Downarrow\Downarrow}^{post\ ()} body\ ()$$

7.5 Try-catch-finally

We now move on to a demonstration that exception handling can be easily implemented within the SFCC calculus. This is the essence of our claim of the unification of exception handling and FCC. Since each language has their own idiosyncratic take on exception handling, we will illustrate a few systems, at which point the reader should be prepared to implement their favorite exception system. Although most languages offer try-catch constructs as statements, we will offer them as an expression so as not to complicate our grammar.

We begin with the components of the system which remain constant for all the systems we will examine. We will need a prompt, which we will write **exn**, specifically set aside for exceptions. The throw operator simply needs to abort up to the nearest exception handler, which we will implement with $\mathcal{A}$. The try-finally construct need only ensure that some code is executed no matter how control exits from a block, which is clearly implementable with $\#_{\Downarrow\Downarrow}$.

$$\mathsf{throw}\ e \equiv \mathcal{A}_{\mathbf{exn}} e$$

$$\mathsf{try}\ e\ \mathsf{finally}\ e_f \equiv \#_{\Downarrow\Downarrow}^{e_f} e$$

The first definition of try-catch is a particularly simple system, an untyped try-catch-finally, similar to that which appears in JavaScript. Here, the try-catch implementation need only install an exception handler, which in turn need worry only

about the value aborted with (the exception itself), and may discard the continuation.

$$\mathsf{try}\ e_{body}\ \mathsf{catch}\ x : e_h \equiv \#_{\mathbf{exn}}^{\lambda\_,x.e_h} e_{body}$$

We can also include stack traces in our simulation, provided that the language implementation has some facility, **addTrace** that will add a stack trace to an exception value. We only need to change the handler to first add the trace before moving to the handler's body.

$$\mathsf{try}\ e_{body}\ \mathsf{catch}\ x : e_h \equiv \#_{\mathbf{exn}}^{\lambda k,a.\mathsf{let}\ x=\mathbf{addTrace}\ k\ a\ \mathsf{in}\ e_h} e_{body}$$

We need only make the restriction that $k, a \notin fv(e_f)$ so as to avoid inadvertent variable capture. Here again, the continuation value is not accessible to the user, so the implementation does not allow the continuation to be resumed.

Our next refinement allows a catch block to trigger only on certain subtypes of an exception type. To do this, we will use the metavariable τ to represent an arbitrary type, and the expression $e\ \mathsf{instanceof}\ \tau$ to check at runtime whether the type of e is a subtype of τ. We can implement this using either of our prior untyped try-catch definitions. Now however, the handler first checks the type of before moving to the user's handling code. When the type is not suitable, then the exception is thrown

again, so that the exception has another chance to be caught by an acceptable handler.

$$\begin{aligned}\mathsf{try}\ e_{body}\ \mathsf{except}\ \tau\ x_1 : e_1 \equiv \mathsf{try}\ e_{body}\ \ \mathsf{except}\ x :\\ \mathsf{if}\ x\ \mathsf{instaceof}\ \tau\\ \mathsf{then}\ e_h\\ \mathsf{else\ throw}\ x\end{aligned}$$

Finally, as a syntactic nicety, we allow for sequences of $\mathsf{try}\ldots\mathsf{try}$ to be compressed into a single try keyword. Thus, instead of writing

$$\mathsf{try\ try\ try}\ e_a\ \mathsf{catch}\ \tau_1\ x : e_b\ \mathsf{catch}\ \tau_2\ x : e_c\ \mathsf{finally}\ e_d$$

we can simply write the more familiar

$$\mathsf{try}\ e_a\ \mathsf{catch}\ \tau_1\ x : e_b\ \mathsf{catch}\ \tau_2\ x : e_c\ \mathsf{finally}\ e_d$$

which we now recognize as being equivalent to

$$\#^{e_d}_{\Downarrow}\ \#^{\lambda\_,x.\mathsf{if}\ x\ \mathsf{isinstance}\ \tau_2\ \mathsf{then}\ e_c\ \mathsf{else}\ \mathcal{A}_{\mathbf{exn}}x}_{\mathbf{exn}}\ \#^{\lambda\_,x.\mathsf{if}\ x\ \mathsf{isinstance}\ \tau_1\ \mathsf{then}\ e_b\ \mathsf{else}\ \mathcal{A}_{\mathbf{exn}}x}_{\mathbf{exn}}\ e_a$$

The apparatus we have now developed includes all the intuitions for how exception handling is supposed to work. If no exception is thrown, the catch blocks are not executed, but the finally block is. On the other hand, if an exception is thrown in a try block, the exception is matched against the catch clauses one-by-one in order. Once the

appropriate catch block has executed, the finally clause then executes. If no catch block is suitable, then the exception continues to propagate to the next dynamically-enclosing try-catch block, but not before the finally block is executed.

Exception handling was developed independently of both delimited control and formal methods, and so we did not expect the translation to be as small and straightforward as our prior definitions. Nevertheless, each translation we give is very small. More importantly, the translations operate on the source program in local sections rather performing a transformation on the program as a whole. This demonstration makes clear two important points. First, it shows that our system is capable of handling concepts broader than its own historical influences, and is therefore quite general. Second, as exception handling is present in nearly every production programing language, it shows that our system is capable of solving real-world issues that commonly appear in production programs.

8 Design of a Source Language

9 Efficient Implementation

10 Applications

Inventory of control structures:

- [DONE] abort
- [DONE] fcontrol
- [DONE] MFDC
- [-] scope statements
- [-] dynamic-wind
- [.] try/catch/else/finally: typed and untyped, with/without stack trace; with/without rethrow
- [ ] capture (without abort)
- [ ] cupto
- [ ] $^{\pm}\mathcal{F}^{\pm}$
- [ ] marker
- [ ] spawn
- [ ] call/cc

- [ ] context manager
- [ ] panic
- [ ] labeled constructs
- [ ] conjunction
- [ ] anaphoric if
- [ ] fluid variables
- [ ] subroutines
- [ ] coroutines
- [ ] backtracking algorithms
- [ ] software transactions
- [ ] concurrent algorithms (cooperating threads)

NOTE: everything below is in an intermediate state, subject to dramatic change, and probably deletion.

Outline:

- Implementation of throw/catch/finally, in various states of refinement.
- Design choices for a source language: disallow anything other than $\Box$ as K in $\mathcal{F}$; always raise a stock exception when a barrier is tripped; a default handler for operators that do not normally specify handlers; only allow $\#_{\uparrow\uparrow}$ instead of independent $\#_{\uparrow}$ and $\#_{\Uparrow}$.
- Efficient implementation (the operational semantics).
- Library of control structures: everything in the review and plenty more besides.

11 Reformulation

Key idea is to specify handlers on both prompt and control operators. We automatically invoke the operator's handler, but control may from there be given to the delimiter's handler. Essentially, either the control operator has enough information to move the computation along on its own, or else it has enough information to know that it must seek outside help from the delimiter's handler. This is our answer to choice (4) from the analysis of delimited continuations; its power comes from its ability to model both cases in a unified way.

Dynamic extent protection:

$$e_{body} \text{ finally } e \equiv$$

$$\text{onExit } e; e_{body} \equiv \#^{e}_{\downarrow}\#^{e}_{\Downarrow}e_{body}$$

$$\text{with } e_{cm} \text{ as } x \text{ in } e \equiv \text{let } x_{cm} = e_{cm}, x = x_{cm}.\_\_enter\_\_() \text{ in onExit } x_{cm}.\_\_exit\_\_(); e$$

Exceptions:

$$\text{panic } e \equiv \text{let } x_{report} = e \text{ in } \mathcal{F}_{\mathbf{p_0}}\lambda\_, \_.x_{report}$$

cupto:

$$\text{set } e_p \text{ in } e \equiv$$

$$\#_{e_p}e \equiv \#^{()}_{e_p}e$$

$$\text{cupto } e_p \text{ as } k \text{ in } e \equiv {}^{-}\mathcal{F}^{-}_{e_p}\lambda k.e$$

$${}^{-}\mathcal{F}^{-}_{e_p}e \equiv \text{let } x_p = e_p, f = e \text{ in } \mathcal{F}_{x_p}\lambda h, k.f\ k$$

$${}^{+}\mathcal{F}^{-}_{e_p}e \equiv \text{let } x_p = e_p, f = e \text{ in } {}^{-}\mathcal{F}^{-}_{x_p}\lambda h, k.\#^{h}_{x_p}f\ k$$

$${}^{-}\mathcal{F}^{+}_{e_p}e \equiv \text{let } x_p = e_p, f = e \text{ in } {}^{-}\mathcal{F}^{-}_{x_p}\lambda h, k.f\ \langle\#^{h}_{x_p}k\ \Box\rangle$$

$${}^{+}\mathcal{F}^{+}_{e_p}e \equiv \text{let } x_p = e_p, f = e \text{ in } {}^{-}\mathcal{F}^{-}_{x_p}\lambda h, k.\#^{h}_{x_p}f\ \langle\#_{x_p}k\ \Box\rangle$$

$^{\pm}\mathcal{F}^{\pm}$:

$$\begin{aligned}
\mathsf{prompt}\ e &\equiv \\
\mathsf{reset}\ e &\equiv \\
\#e &\equiv \#_{\mathbf{p}} e \\
{}^{\pm}\mathcal{F}^{\pm} e &\equiv {}^{\pm}\mathcal{F}^{\pm}_{\mathbf{p}} e \\
\mathsf{control}\ e &\equiv \\
\mathcal{F} e &\equiv {}^{+}\mathcal{F}^{-} e \\
\mathsf{shift}\ x\ e &\equiv {}^{+}\mathcal{F}^{+} \lambda x.e
\end{aligned}$$

Undelimited control:

$$\begin{aligned}
\mathsf{withCont}\ e &\equiv {}^{+}\mathcal{F}^{-}_{\mathbf{p_0}} e \\
\mathsf{abort}\ e &\equiv \mathsf{withCont}\ \lambda_.e \\
\mathsf{callcc}\ e &\equiv \mathsf{let}\ f = e\ \mathsf{in}\ \mathsf{withCont}\ \lambda k.f\ \langle \mathsf{abort}\ (k\ \Box)\rangle
\end{aligned}$$

Misc:

$$\begin{aligned}
\mathsf{marker}\ e &\equiv \mathsf{let}\ f = e, x_p = \nu\ \mathsf{in}\ \#_{x_p} f\ x_p \\
\mathsf{spawn}\ e &\equiv \mathsf{let}\ f = e\ \mathsf{in}\ \mathsf{marker}\ \lambda x_p.f\ \langle {}^{-}\mathcal{F}^{+}_{x_p} \Box\rangle \\
\mathcal{C}_{e_p} e &\equiv \mathsf{let}\ x_p = e_p, f = e\ \mathsf{in}\ \mathcal{F}_{x_p} \lambda h, k.\#^{h}_{x_p} k\ (f\ k)
\end{aligned}$$

If we get something like $\#^{e}_{\Downarrow} \mathcal{F}_p^{\mathcal{F}_{p'}\Box} v$, which leads to $\mathcal{F}_p^{e;\#^{e}_{\Uparrow}\mathcal{F}_{p'}\Box} v$, one might be afraid

that that continuation, when called—$\langle e; \#^e_{\Uparrow} \mathcal{F}_{p'} \Box \rangle\ v'$—will lead to a buildup of e sequenced in a row—perhaps $e; \#^e_{\Uparrow} \mathcal{F}_{p'} v' \mapsto^* \mathcal{F}_{p'}^{e;(e;\#^e_{\Uparrow}\Box)} v'$. This cannot happen, however, since e will need to be forced to a value before the control operator can capture it. Preferring to capture continuations wherever possible, the correct evaluation sequence is:

$$e; \#^e_{\Uparrow} \mathcal{F}_{p'} v' \mapsto e; \mathcal{F}_{p'}^{e;\#^e_{\Uparrow}\Box} v' \mapsto^* v; \mathcal{F}_{p'}^{e;\#^e_{\Uparrow}\Box} v' \mapsto \mathcal{F}_{p'}^{e;\#^e_{\Uparrow}\Box} v'$$

12 Overview

12.1 Introduction

The system we have developed is inspired by three exiting control concepts: the system $\pm\mathcal{F}\pm$ from [3], dynamic-wind from Scheme, and the ability to discriminate normal and abnormal exit from D. Our system was developed independently of the requirements summarized in §5, but we will show that it does meet those standards. This sort of demonstration is interesting because it shows that the power of the system extends beyond its direct influences.

TODO I'm removing type system stuff, so this may need a re-write to flow properly

We begin our work with a call-by-value lambda calculus. Finally, with call-by-value, we can easily control the order of evaluation, esp. to obtain determinism. To aid comprehension by humans, we also added expression sequencing, denoted by the semicolon infix. The most basic form is $e; e$, but our extensions will include other productions involving semicolon. When semicolon separates two nodes of the syntax, we assume they are right-associated wherever explicit grouping is not given.

We give a brief review of the grammar in Figure 19. We will not review here the

dynamic semantics (which will be fully specified in §13) or the static semantics (which is intuitively clear, though formally large; we point the interested reader to CITE practical higher-rank type inference).

As in most discussions of formal semantics, we will be working simultaneously with syntax, semantics and metasyntax. To be unambiguous, we will typeset each level differently. Metasyntax will be in *italics*, semantic objects will be written in **boldface**, and syntax will be written either in sans-serif for keywords or in `typewriter` everywhere else.

Values	v	::=	$\lambda x.e$	Term Abstraction
Expressions	e	::=	v	
		|	x	Term variable
		|	$e\ e$	Term application
		|	$e; e$	Sequencing

Figure 19: Base Grammar

As we stated earlier, FCC is usually considered in systems using a control stack during evaluation. We will also use a control stack in our definition of a small-step semantics. We will adopt an upward-growing convention. Figure 20 defines the basic structure of a control stack suitable for our base calculus. We will also wish to speak about when evaluation proceeds above some point on the control stack. FIXME: dynamic extent should be defined in the section on dynamic wind. In such cases, we will say that evaluation is within that point's "dynamic extent," or "extent" for short. When the stack shrinks beyond some point, we say that control has exited that point's dynamic extent.

Continuations	K	::=	$\blacksquare$	Empty continuation
		|	$k : K$	Continuation step
Contexts	k	::=	$\square\ e$	
		|	$v\ \square$	
		|	$\square; e$	

Figure 20: Control Stack

12.2 Stack Marking

The base calculus only contains continuations which are stacks of contexts. To this, we will add five *stack marks*, which are simply non-context elements which may be placed on the control stack. The first we will consider we call *delimiters* and are very similar to the pushed prompts of [3]. Delimiters consist of an identifier, which we call a *cue*, along with a function, called the *handler*. The grammar is defined in Figure 21. Marking the stack with a delimiter associates that use of the cue with the handler, and each delimiter may have different combinations of cue and handler. The user can create new cues at runtime with the newCue expression. This is our only way of delimiting abort and capture, and by having only this one, we greatly simplify reasoning about the interactions between different control flow constructs.

A cue is a primitive concept which need only obey the loose property: given c, its successor $succ(c)$ is defined, and distinct from all of its direct and indirect predecessors[12]. That is, each new cue is unique. The natural numbers are the obvious structure that satisfies these properties, but implementors are free to decide on an alternate structure. For example, when implementing a distributed system, assuring synchronous access to a global counter is likely too expensive.

[12]In fact, a cue may be re-used after it is no longer reachable, but this would require garbage collection, which we will not define here.

We also add special kind of stack mark, which we call *boundary*, and write it ■. FIXME racket calls this a "continuation barrier", and I may as well stick with the terminology. Note that this use of the ■ symbol is different from its use representing an empty control stack, though they do share a common property. Just as non-local control cannot capture or abort past the end of the control stack, non-local control cannot capture or abort past a boundary either. For this reason, we will speak of boundaries as marking the boundary between a sub-interpreter and a main interpreter. We will in a moment see control guards, which are meant to execute before/after changes in control flow. Boundaries are designed to prevent non-local control flow from inadvertently bypassing these guards. In the interests of simplicity, we have elected to simply terminate the program when control attempts to capture or abort past a boundary – escape from within a sub-interpreter .

Values	v	::=	...	
		|	c	Cue identifier
Expressions	e	::=	...	
		|	newCue	
Continuations	K	::=	...	
		|	**cue** $c\ v : K$	Cue
		|	■ : K	Boundary

Figure 21: Stack Marks

We should note an interesting possible feature of boundaries, which is to attach an alternate expression to a boundary. When control attempts to escape from a sub-interpreter bounded by one of these boundaries, the stack is truncated through the boundary and evaluation resumes with alternate expression. For example, a language with primitive **eval** might insert a boundary before interpreting the passed expression such that an attempt to escape from the sub-interpreter instead raises a predictable

exception in the main interpreter. This strategy is a necessary part of sandboxing under FCC; with it we can ensure that sandboxed code cannot take control over trusted code.[8] Though important, this feature is a straightforward extension and tangential to our purposes, so we will only consider boundaries without alternatives.

12.3 Control Operators

We now move on to discussing new operators for manipulating the stack using cues. Our operators are directly inspired by [3] and correspond closely to theirs. Mostly, we simply extend their operators to work with handlers, but we do refactor one of their operators into two of ours as we will soon see. The syntax of these operators is given in Figure 22.

Perhaps the simplest operator is handle[*cue*] *handler* in *body*. This corresponds to an extension of pushPrompt to take an additional handler argument. In brief, handle is the mechanism by which we associate a cue with a handler and place a delimiter on the stack.

Next we have restore *cont body*, which corresponds exactly to pushSubcont from [3]. The restore operator appends the passed (sub)continuation before evaluating the body. Thus, once the body has reduced to a value, it will be further operated on by the new continuation before the original continuation performs its transformations. The semantics of this operator will roughly follow those of function application. In fact, the author had debated overloading the existing application syntax against introducing a new operator. In the end, it was considered that an explicit operator aids comprehension by reminding the reader of the interesting effects.

Next, we examine abort[*cue*] *value*, which removes a portion of a control stack up

Values	v	::=	...
		|	K First-class continuation
Expressions	e	::=	...
		|	handle$[e]$ e in e
		|	abort$[e]$ e
		|	capture$[e]$ e
		|	restore e e
Contexts	k	::=	...
		|	handle$[\square]$ e in e
		|	handle$[v]$ $\square$ in e
		|	abort$[\square]$ e
		|	abort$[v]$ $\square$
		|	capture$[\square]$ e
		|	capture$[v]$ $\square$
		|	restore $\square$ e

Figure 22: Non-local Control Operators

through a delimiter, then applies its value to that delimiter's handler. This shares much in common with throw operators in languages with exceptions. Just like throw, we supply a value to send to a handler higher up in the stack, and this construct will be essential in implementing traditional exception handling later on. The English grammar should be slightly different, however, so whereas we would say "throw a value," we will instead say "abort *with* a value."

Finally, we have capture$[cue]$ *thunk*, which captures and reifies a portion of the control stack up to a delimiter. The reified continuation is then passed, along with the delimiter's handler, to the thunk, which should be a function accepting just those

arguments. Also passing the handler along allows the thunk to control whether the cue should be re-installed should the need arise[13]. An important relationship between capture and abort is that if the operators are given the same cue in the same control context, they would respectively capture and abort precisely the same portion of the control stack.

The two operators abort and capture together correspond to the two "halves" of the withSubcont operator in [3]. In fact, those authors mention that withSubcont could just as well be factored into two – as we have done here – without affecting the expressivity of their system. Their decision to combine capture and reify functionality with abort is appropriate for their purposes, but because our system has a richer system of stack marks, splitting up the two operations will be the simpler route.

12.4 Control-flow Guards

Finally we add three more operators, onWind, onUnwind, and onAbort, which we collectively call control-flow guards, or control guards for short. Each of these operators introduces a corresponding stack mark, which we call a *control guard*, before evaluating their body. In turn, the control guards ensure that the passed expressions are always appropriately evaluated when control flow enters or exits a section of the stack.

There is a strong correspondence of onUnwind and onAbort with D's scope statements scope(success) and scope(success). On the other hand, there is also a strong correspondence of onWind with dynamic-wind's ability to register a thunk before evaluating the body as well as between dynamic-wind's after-thunk registration and a combination of onUnwind and onAbort. As such, we can think of these three operators

[13]Of course, we already have the *cue* available, so reconstructing the precise stack mark which delimited the capture is simply a matter of re-associating the cue with the found handler.

in two ways. They are an extension of D's scope statements to handle environments where control can re-enter an expression. They also split dynamic-wind into its components and allows it to discriminate between normal and abnormal – non-local – exit. The syntax is given in Figure 23.

With these guard expressions, we see the other syntax productions using semicolon. As an example of how semicolon groups expressions, consider the term e_1; (onAbort e_g; e_2; e_3); e_4. Here, e_1 is clearly not guarded by e_g, but *both* e_2 and e_3 are. Finally, e_4 is not guarded. In general, unless explicit grouping is given, guard expressions protect all the following expressions in a semicolon sequence. This is very similar to – in fact inspired by – the syntax of D's scope statements, which we believe aids comprehension.

Expressions	e	::=	...
		|	onWind e; e
		|	onUnwind e; e
		|	onAbort e; e
Continuations	K	::=	...
		|	**onWind** $e : K$
		|	**onUnwind** $e : K$
		|	**onAbort** $e : K$

Figure 23: Stack Guards

13 Dynamic Semantics

Here we develop a small-step dynamic semantics using a transition system. We will first need to develop a few additional concepts.

13.1 Supporting Definitions

To evaluate function application, we will use hygienic substitutions, written $e[x\backslash e']$. For abstractions, cues and non-value expressions, this is defined as usual. Importantly however, substitution has no effect on continuations: $K[x\backslash e'] = K$.

Continuations can be constructed from other continuations. In particular, we will make use of continuation catenation, which is essentially list catenation. We write the catenations of two continuations simply as adjacency: the catenation of K onto K' is written KK'. Formally, $(t : K)K' = t : (KK')$ and $\blacksquare K' = K'$. Note that although the empty continuation is written similarly to a boundary, they are treated differently.

We will need the total functions $guardWind$ and $guardAbort$ of two parameters: a continuation and a base-case expression. They generate a continuation by accumulating the guard expressions that have been pushed onto the passed continuation with onWind and onAbort respectively. Formally, we have

- $guardWind(K, e) = e_g : \blacksquare : \square; guardWind(K^{\uparrow}, e)$ when
  $K = K^{\uparrow}(\textbf{onWind}\ e_g : K^{\downarrow})$ and $guardWind(K^{\downarrow}, e) = e : \blacksquare$,
- or $guardWind(K, e) = e : \blacksquare$ otherwise.
- $guardAbort(K, e) = e_g : \blacksquare : \square; guardAbort(K^{\downarrow}, e)$ when
  $K = K^{\uparrow}(\textbf{onAbort}\ e_g : K^{\downarrow})$ and $guardAbort(K^{\uparrow}, e) = e : \blacksquare$,
- or $guardWind(K, e) = e : \blacksquare$ otherwise.

Intuitively, what these guards do is to accumulate a number of expressions. Each of the expressions is to be evaluated in its own sub-interpreter. The precise sequence is different for the two functions: the result of $guardWind$ will evaluate the guard expressions in FIFO order, but $guardAbort$ will evaluate them in LIFO order with

respect to when the corresponding control guards were pushed to the stack. Either way, the last expression in the sequence is the base-case expression, and is evaluated in the main interpreter. Thus, these functions formalize the process of collecting and evaluating control guards that were registered by a guard expression before continuing on.

We also require the partial function *splitStack* taking a continuation and a cue. It finds in the passed continuation the location of the most-recently pushed instance of the passed cue and returns the stack portions strictly above and below the cue, as well as the handler attached to that cue. However, *splitStack* will not allow the upper portion to contain a boundary; this protects the rest of the process from non-local control-flow bugs in any ostensibly control-pure expression. Formally, we have $splitStack(K, c) = \langle K^{\uparrow}, v_h, K^{\downarrow} \rangle$ defined when

- $K = K^{\uparrow}(\mathbf{cue}\ c\ v_h : K^{\downarrow})$,
- $K^{\uparrow} \neq \hat{K}^{\uparrow}(\mathbf{cue}\ c\ \hat{v}_h : \hat{K}^{\downarrow})$,
- and $K^{\uparrow} \neq \hat{K}^{\uparrow}(\blacksquare : \hat{K}^{\downarrow})$

In all other cases, the relation is undefined.

13.2 Evaluation

We define evaluation of an expression using the transition system in Figure 24. The system acts on pairs, the first of which is the control stack, and the second element of which simply provides a supply of fresh cues. Each transition is of the form $\langle K, c \rangle \longmapsto \langle K', c' \rangle$. We write $\longmapsto^*$ for the reflexive-transitive closure of $\longmapsto$. We say that $eval(e) = v$ when $\langle e : \blacksquare, c \rangle \longmapsto^* \langle v : \blacksquare, c' \rangle$. In all other cases, $eval(e)$ is undefined.

Since there is only one applicable transition for each pair, the system is trivially confluent.

$$\boxed{\langle K, c\rangle \longmapsto \langle K, c\rangle}$$

$$\begin{array}{rrcl}
\textsc{app}_1\text{:} & \langle e\ e' : K, c\rangle & \longmapsto & \langle e : \square\ e' : K, c\rangle \\
\textsc{seq}_1\text{:} & \langle e; e' : K, c\rangle & \longmapsto & \langle \square;\ e : \square\ e' : K, c\rangle \\
\textsc{cue}\text{:} & \langle \mathsf{newCue} : K, c\rangle & \longmapsto & \langle c : K, succ(c)\rangle \\
\textsc{h}_1\text{:} & \langle \mathsf{handle}[e_c]\ e_h\ \mathsf{in}\ e : K, c\rangle & \longmapsto & \langle e_c : \mathsf{handle}[\square]\ e_h\ \mathsf{in}\ e : K, c\rangle \\
\textsc{a}_1\text{:} & \langle \mathsf{abort}[e_c]\ e : K, c\rangle & \longmapsto & \langle e_c : \mathsf{abort}[\square]\ e : K, c\rangle \\
\textsc{c}_1\text{:} & \langle \mathsf{capture}[e_c]\ e : K, c\rangle & \longmapsto & \langle e_c : \mathsf{capture}[\square]\ e : K, c\rangle \\
\textsc{r}_1\text{:} & \langle \mathsf{restore}\ e\ e' : K, c\rangle & \longmapsto & \langle e : \mathsf{restore}\ \square\ e' : K, c\rangle \\
\textsc{g}_\uparrow\text{:} & \langle \mathsf{onWind}\ e_g;\ e : K, c\rangle & \longmapsto & \langle e_g : \blacksquare : \square; e : \mathbf{onWind}\ e_g : K, c\rangle \\
\textsc{g}_\downarrow\text{:} & \langle \mathsf{onUnwind}\ e_g;\ e : K, c\rangle & \longmapsto & \langle e : \mathbf{onUnwind}\ e_g : K, c\rangle \\
\textsc{g}_\Downarrow\text{:} & \langle \mathsf{onAbort}\ e_g;\ e : K, c\rangle & \longmapsto & \langle e : \mathbf{onAbort}\ e_g : K, c\rangle \\
 & & & \\
\textsc{app}_2\text{:} & \langle v : \square\ e : K, c\rangle & \longmapsto & \langle e : v\ \square : K, c\rangle \\
\textsc{app}\text{:} & \langle v : \lambda x.e\ \square : K, c\rangle & \longmapsto & \langle e[x\backslash v] : K, c\rangle \\
\textsc{seq}\text{:} & \langle v : \square; e : K, c\rangle & \longmapsto & \langle e : K, c\rangle \\
\textsc{h}_2\text{:} & \langle v : \mathsf{handle}[\square]\ e_h\ \mathsf{in}\ e : K, c\rangle & \longmapsto & \langle e_h : \mathsf{handle}[v]\ \square\ \mathsf{in}\ e : K, c\rangle \\
\textsc{h}\text{:} & \langle v_h : \mathsf{handle}[c]\ \square\ \mathsf{in}\ e : K, c\rangle & \longmapsto & \langle e : \mathbf{cue}\ c\ v_h : K, c\rangle \\
\textsc{a}\text{:} & \langle v : \mathsf{abort}[\square]\ e : K, c\rangle & \longmapsto & \langle K_g K^{\downarrow}, c\rangle \\
 & & \text{where} & \langle K^{\uparrow}, v_h, K^{\downarrow}\rangle = splitStack(K, c) \\
 & & \text{and} & K_g = guardAbort(K^{\uparrow}, v_h\ e) \\
\textsc{c}_2\text{:} & \langle v : \mathsf{capture}[\square]\ e : K, c\rangle & \longmapsto & \langle e : \mathsf{capture}[v]\ \square : K, c\rangle \\
\textsc{c}\text{:} & \langle v : \mathsf{capture}[c]\ \square : K, c\rangle & \longmapsto & \langle v\ v_h\ K^{\uparrow} : K, c\rangle \\
 & & \text{where} & \langle K^{\uparrow}, v_h, K^{\downarrow}\rangle = splitStack(K, c) \\
\textsc{r}\text{:} & \langle K' : \mathsf{restore}\ \square\ e : K, c\rangle & \longmapsto & \langle K_g K' K, c\rangle \\
 & & \text{where} & K_g = guardWind(K', e) \\
\mathrm{M}_c\text{:} & \langle v : \mathbf{cue}\ c\ v_h : K, c\rangle & \longmapsto & \langle v : K, c\rangle \\
\mathrm{M}_\blacksquare\text{:} & \langle v : \blacksquare : e : K, c\rangle & \longmapsto & \langle v : K, c\rangle \\
\mathrm{M}_\uparrow\text{:} & \langle v : \mathbf{onWind}\ e_g : K, c\rangle & \longmapsto & \langle v : K, c\rangle \\
\mathrm{M}_\downarrow\text{:} & \langle v : \mathbf{onUnwind}\ e_g : K, c\rangle & \longmapsto & \langle e_g : \blacksquare : \square; v : K, c\rangle \\
\mathrm{M}_\Downarrow\text{:} & \langle v : \mathbf{onAbort}\ e_g : K, c\rangle & \longmapsto & \langle v : K, c\rangle
\end{array}$$

Figure 24: Dynamic Semantics

Let us now examine the structure of the rules, which we have grouped into two sections. The first set we call "winding rules" because the transitions increase the size of the control stack, but take the top of the control stack from a non-value expression to a proper subexpression. We call the second set "unwinding rules". Here, the transitions always start with a value on top of the control stack and usually[14] reduce the size of the stack. In this way, evaluation starts in the winding rules and recurses until a value is found. Then, the unwinding rules iteratively shrink the control stack until either a new complex expression is found, such as the body of a function, or else the empty continuation is reached.

Though there appear to be many transitions, there are really only a few ideas present which determine the pattern. Several rules perform only minor transformations, ensuring that a subexpression is evaluated down to a value before proceeding to the next part of the expression. Where we have this, we have grouped sets of rules by expression and given them a common base name. The uninteresting order-of-evaluation rules are given numeral subscripts.

We have also included rules from our base calculus, which further increases the apparent size of the system. The three APP rules implement call-by-value lambda calculus, and the two SEQ rules implement sequencing.

It should be noted that only one rule, CUE, inspects or alters the supply of cues in any way. Every other rule ignores the supply. So, even though writing the supply explicitly everywhere increases the size of the representation, its appearance in almost every rule is simply an uninteresting bookkeeping device.

Finally, there are several rules for guard expressions (the G rules) and stack marks

[14]Rule R has the explicit purpose of adding onto the stack, and rule A may increase the control stack if many control guards are aborted past.

(the M rules). There are commonalities between each of these groups. The G rules merely install control guards, except for one which also executes the guard. Then, the M rules merely allow values to unwind past a stack mark, except for one which also executes a control guard.

Now that we have the basic structure, let us examine the more novel cases in detail. We begin with the simplest, CUE, which is the rule for the newCue expression. This is essentially a cue constructor; with it, a cue is obtained from the supply and placed onto the stack, and the supply is incremented to maintain the supply's freshness. Once the programmer has a cue value on the stack, they can manipulate it, passing it to functions, or directly to other control operators.

Next, the set of rules H deal with the handle expression, and are also quite simple. It marks the stack with a delimiter of the given cue and handler, and the body is evaluated above this mark. Thus, if an abort or escape is performed with the same cue, this mark will be found – assuming it is still the uppermost with the cue.

We now turn to stack marks and control guards. Note that there are only two ways by which a stack mark may be removed from the control stack, either by the M rules or by rule A. The first case we call "normal" exit, and the second we call "abnormal." For normal exit, we see that unwinding past a stack mark simply removes the mark, except for rule $M_{\downarrow}$. In $M_{\downarrow}$, we "trip" the registered **onUnwind** control guard: we evaluate the guard in a sub-interpreter before continuing to unwind. We will examine rule A in more detail in a moment.

Conversely, there are only two ways in which a control guard may be added to the stack, either by the G rules or by rule R. It would again possible to distinguish between these two cases, but no production languages do so, so we will follow that tradition. As in the M rules, the G rules all simply push the control guard and evaluate the body,

except for one rule. In G$_\uparrow$, the control guard is pushed, but before the body is evaluated, the guard expression is tripped. We will examine rule R in more detail in a moment.

We now turn to consider rule A, which is probably the most complex rule. In it, we use *splitStack* to remove the control stack down to and including the nearest matching delimiter. This is one of the two ways we may remove stack marks. Thus, in addition to removing a topmost section of the stack, we also accumulate using *guardAbort* all of the **onAbort** control guards which were removed. These accumulated guards are then set to be evaluated in LIFO order. Finally, the base case of *guardAbort* applies the handler function to the value aborted with, and this is where the main interpreter's evaluation resumes.

Rule C applies when we unwind a value v into a capture context. Here, the continuation beneath the capture context is not altered. Instead, we use *splitStack* to obtain the section of the stack above the nearest delimiting mark and the mark's handler function. We then apply the handler function and the continuation found from *splitStack* to v, the value we are unwinding. Therefore, v must be a function of two (curried) arguments, the first of which accepts a handler function and the second a continuation.

The rule R only allows a captured continuation to be unwound. The captured continuation is placed on the stack but, because this may add control guards to the stack, we also collect all of the captured continuation's wind-guards with *guardWind*. The body of the restore is used as the base-case expression. Thus, if no wind-guards are in the captured continuation, the effect of the rule is to insert the captured continuation just beneath the body expression so that once the expression reduces to a value, it will enter the captured continuation before control passes to the rest of the continuation.

This concludes our discussion of the individual rules. We now turn to consider the

implications of the system.

TODO note unhandled cases: apply where first value is not a function, abort/capture where splitStack undefined, restore where first value is not a subcont

TODO make note that some cases are not handled because the system is set up so that they do not appear: e.g. there is never an e in K except at the top of the stack, and an e is always at the top.

TODO in our system, boundaries are always over sequence contexts. We've added this complication so that it is easier to extend the system with more general sub-interpreters.

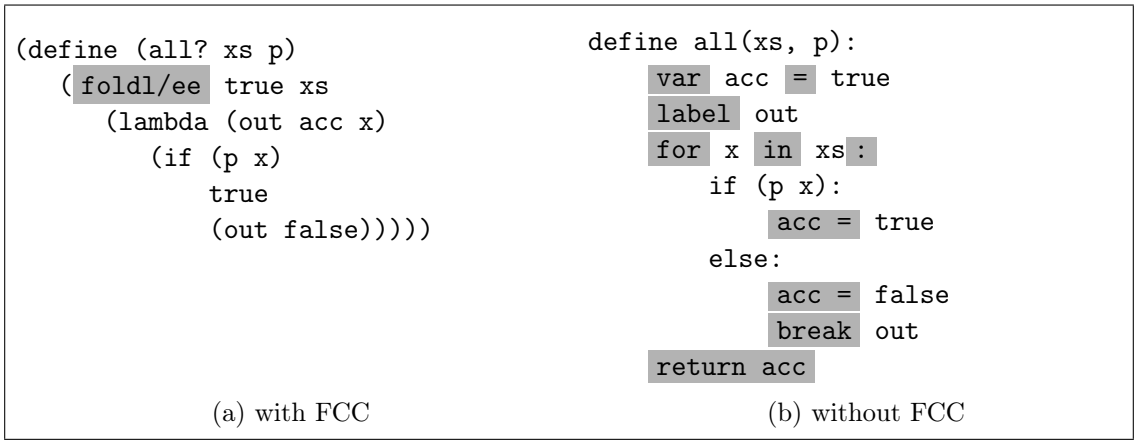
TODO: Since we are using a transition system, it should be clear how to incorporate mutable state into the semantics without much difficulty.

13.3 Efficient Implementation

TODO: meta-continuatinos as in MFDC. I know there's some research in implementing continuations effectively in Racket (which I may have around here somewhere).

14 Extended Example

TODO pick an example that actually uses capture, though I want to keep some of this analysis



```
(define (all? xs p)
   (foldl/ee true xs
      (lambda (out acc x)
         (if (p x)
             true
             (out false)))))
```

(a) with FCC

```
define all(xs, p):
    var acc = true
    label out
    for x in xs:
        if (p x):
            acc = true
        else:
            acc = false
            break out
    return acc
```

(b) without FCC

Figure 25: Accumulation-loop pattern codified

Our example is particularly simple, so several simplifying transformations might be automatically made by the programmer. We have specifically not made these transformations because we would like to examine the accumulation-loop pattern, regardless of the complexity of the problem to which it is applied.

The lines-of-code savings here are not particularly interesting. For one, this is purely due to surface syntax. More importantly, as the body becomes larger, the comparison will show less difference in line count. The important thing to note is that there are fewer "moving parts" in the example with FCC – fewer fixed things that could be misplaced during further maintenance. Furthermore, even if the test imperative language had foldl already implemented, we cannot re-use it here, so we are forced to repeat – manually or by copy-paste – the body of foldl.

note the possibility of nesting calls to foldl/ee. if Contoy were typed, we show that this is a pure function.

15 Applications

15.1 Conjunction

In ammonite, I might write:

```
conjCue is newCue
obtain_ is prompt conjCue
_and_ is lambda (x; y) +F- conjCue $ lambda (h; k) (k x && k y)
_or_ is lambda (x; y) +F- conjCue $ lambda (h; k) (k x || k y)


assert $ obtain 4 and 3 = 3 or 4
```

This application was taken from [1]. In that paper, it is noted that in J, sometimes `x (f g h) y` evaluates as `(x f y) g (x h y)`, but doesn't say much more than that.

15.2 Generalized Anaphoric If

```
{
  let x = X;
  if (V) {A} else {B}
}
```

where `x` is free in `V`, `A` and/or `B`. Also, `x` is a temporary variable and so should only be in scope for the branch.

In Ammonite:

```
cue is newCue
ifitis is vau e (p; c; a) :
```

```
  handle cue
   lambda (k; it) :
      eval
        (extend-env e {it: it})
        (if k it then eval c else a)
eval e p
itis is lambda it :
  abort cue it
```

In languages where if statements are able to test many data types—Lisp, C and Python, among others—a common pattern is to

15.3 Subroutines

let **call-frame** be a cue

subroutine $x\ e \equiv \lambda x.\mathsf{handle}[\textbf{call-frame}]\ \textbf{id}\ \mathsf{in}\ e\ x$

$\mathsf{return}\ e \equiv \mathsf{abort}[\textbf{call-frame}]\ e$

15.4 Software Transactions

transactions recur, apparently: they're in pep 343 as well as D

16 Related Work

17 Further Directions

type system: probably using predicative System F_ω. The type of cues would be a two-place type constructor, enforcing purity with monads would rule out control-impure handlers and guards. Cues and subcontunations may need to account for co-/contravariance during type inference.

sandboxing: restrict the environment in which a piece of code executes. as long as there is no way to generate new prompts and you can determine (at least dynamically before you call the untrusted code) which prompts are available to the untrusted code, then you can prevent the untrusted code from gaining control over the rest of the continuation. These are easy things to verify. Of course, the client (source of untrusted code) can still make use of advanced control constructs.

compilation: JIT subcontinuations into functions

Variations:

- Should we distinguish normal and abnormal entrance? Should we distinguish a true abort (without possibility of re-instatement) from a capturing abort? If for example we only closed a file on true abort, how would we manage closing the file when the captured continuation is no longer needed?

References

[1] Chris Barker. Continuations in Natural Language. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Continuations Workshop (CW'04).*, Birmingham, UK, 2004.

[2] Olivier Danvy and Andrzej Filinski. Abstracting control. In *In Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160. ACM Press, 1990.

[3] R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *J. Funct. Program.*, 17(6):687–730, 2007.

[4] Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag, 1990.

[5] Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 180–190, New York, NY, USA, 1988. ACM.

[6] Andrzej Filinski. Representing monads. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457. ACM Press, 1994.

[7] Matthew Flatt. The Racket Reference. Technical report, PLT, 2015.

[8] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. *SIGPLAN Not.*, 42(9):165–176, October 2007.

[9] Martin Gasbichler and Michael Sperber. Final shift for call/cc:: Direct implementation of shift and reset. *SIGPLAN Not.*, 37(9):271–282, September 2002.

[10] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition.* Addison-Wesley Professional, 1st edition, 2013.

[11] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ml-like languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 12–23, New York, NY, USA, 1995. ACM.

[12] Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Trans. Program. Lang. Syst.*, 9(4):582–598, October 1987.

[13] David Herman. Functional pearl: The great escape: Or, how to jump the border without getting caught, 2007.

[14] R. Hieb and R. Kent Dybvig. Continuations and concurrency. *SIGPLAN Not.*, 25(3):128–136, February 1990.

[15] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pages 47–96. Press, 2002.

[16] John Launchbury and Simon Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[17] Atze van der Ploeg and Oleg Kiselyov. Reflection without remorse: Revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the 2014 ACM*

SIGPLAN Symposium on Haskell, Haskell '14, pages 133–144, New York, NY, USA, 2014. ACM.

[18] Dorai Sitaram. Handling control. In *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 147–155. ACM Press, 1993.

[19] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *LISP and Symbolic Computation*, 3(1):67–99, 1990.

[20] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations ii: Full abstraction for models of control. In *In Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 161–175. ACM, 1990.

[21] Bjarne Stroustrup. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[22] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[23] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 3rd edition, 2009.

[24] Guido van Rossum and Nick Coghlan. PEP 343: The "with" Statement. Technical report, Python Software Foundation, June 2011.

[25] Westley Weimer and George C. Necula. Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, 30(2), 2008.

[26] Scott Wiltamuth and Anders Hejlsberg. C# Language Specification. Technical report, Microsoft, 2003.