On Secure First-Class Control

A Thesis Presented to
The Faculty of the Computer Science Department

by

Eric Demko

In Partial Fulfillment
of Requirements for the Degree
Masters of Computer Science

Southern Polytechnic State University
August 2015

TODO another title page

In presenting this thesis as a partial fulfillment of the requirements for an advanced degree from Southern Polytechnic State University, I agree that the university library shall make it available for inspection and circulation in accordance with its regulations governing materials of this type. I agree that permission to copy from, or to publish, this thesis may be granted by the professor under whose direction it was written, or, in his absence, by the dean of the appropriate school when such copying or publication is solely for scholarly purposes and does not involve potential financial gain. It is understood that any copying from or publication of, this thesis which involves potential financial gain will not be allowed without written permission.

Eric Demko

Notice To Borrowers

Unpublished theses deposited in the Library of Southern Polytechnic State University must be used only in accordance with the stipulations prescribed by the author in the preceding statement.

The author of this thesis is:

Eric Demko

TODO A permanent address

The director of this thesis is:

TODO Thesis director's name

TODO Thesis director's address

Users of this thesis not regularly enrolled as students at Southern Polytechnic State University are required to attest acceptance of the preceding stipulations by signing below. Libraries borrowing this thesis for the use of their patrons are required to see that each user records here the information requested.

Name of user	Address	Date	Type of use (examination only or copying)

TODO yet another title page

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.

TODO yet another title page again

Contents

List of Figures

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 use our system to implement several constructs and 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*, operating on the entire stack. The section of stack manipulated by a delimited control operator is determined using special contexts called *delimiters* or *prompts*. Operators that place a delimiter on the stack are called *delimiting operators*.

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, module, 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 §9.3 after we have developed our system of first-class control.

[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.

```
(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.

```
(define (withFile filepath mode body) ... (body the_handle) ...)
(define (read handle) ...)
(define (write handle bytes) ...)
```

Figure 3: File API with FCC

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 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 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" (Felleisen (1990)). 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 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 (Felleisen (1990)). 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 (Flatt et al. (2007a)). Furthermore, simulation may increase the expected size-complexity of the simulated operators (Gasbichler and Sperber (2002)). 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

[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 clever, well-known technique takes advantage of a type checker to do this (Launchbury and Peyton Jones (1995)).

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

In our thesis, we develop λ_{SFCC}, a lambda calculus augmented with operators for both first-class control and the secure handling of resources. The key complaint we address is that there are too many variations of FCC systems, and the relationships between them are only defined pairwise. Further, these systems vary in their capabilities; most systems of FCC in the literature are unable to intercept non-local control flow.

Our system enjoys various advantages, depending on the systems it is compared to. The ultimate benefit of λ_{SFCC} is that it provides for deterministic resource initialization and cleanup, even in the presence of arbitrarily complex non-local control flow.

Formal models of first-class control in the literature are unconcerned with external resources, and so offer no operators for their management. By contrast, λ_{SFCC} not only provides non-local control flow operators, but also incorporates operators to intercept non-local control flow. The interception of control flow in the plain lambda calculus is so simple as to be taken for granted, so it is unsurprising that this capability was overlooked in the literature.

Most production languages do not offer any form of FCC. The advantage of λ_{SFCC} is, of course, that FCC is possible, and also that it is reasonably safe to use. Almost all languages which offer exception handling, a limited form of non-local control, do not formalize their exception handling mechanisms. Since λ_{SFCC} is capable of implementing various exception handling mechanisms, we immediately gain a formal specification of these control structures.

There are a few production languages which do offer a resource management operator, `dynamic-wind`, alongside FCC, though these languages are relatively little-used. Our advantage against these is that we have a formal specification[3], and our system is general enough to easily implement their breadth of operators, and more besides.

Thanks to our system's similarities to Gunter et al. (1995) and Dybvig et al. (2007), we expect a practical type system to be able to rule out several classes of error related to the use of FCC. Note that to achieve practicality, type systems of this sort 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. 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 and the preferences of language designers.

[3]Though there is a formalization of Scheme's `dynamic-wind` in Ramsdell (1992), it is *post-hoc*, not part of the language standards.

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. Statements in Python, C# and Java 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 examine a variety of 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, including advantages and disadvantages, is given in §5.

In Part Two, we apply the lessons learned in Part One to design λ_{SFCC}, a novel system of first-class control which is also capable of managing external resources with the same ease as in existing languages. We describe the system in §6; the whole system is fairly large, but can be split into a few small components, each of which we detail in a subsection there, along with the system as a whole in §6.6. Then, in §7, we analyze λ_{SFCC} in relation to its influences. We also examine its relationship to exception handling (§7.5) and a few other important FCC systems (§7.6). In §8, we give an alternate formulation of the calculus in terms of a transition system, which can be directly translated into an efficient interpreter (constant overheads aside). Finally, we examine a wide range of applications of FCC in §9, underscoring the effectiveness of FCC as an implementation technique. Our conclusions are reiterated in §10, along with related and further work.

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 method overriding and so 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 (Stroustrup (2000)). 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 (Weimer and Necula (2008)).

In Java, C#, and similar languages which use garbage collection, destructors[4] are called when an object is collected. Unfortunately, the time to detect, and therefore clean up unreachable objects is non-deterministic (Gosling et al. (2013), Wiltamuth and Hejlsberg (2003)). Whenever the resource is not in physically infinite supply, the garbage collector will race the program to maintain the illusion of automatic cleanup (Weimer and Necula (2008)). 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 (Weimer and Necula (2008)). 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 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

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

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 Weimer and Necula (2008).

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

    A...
}
finally {
    B...

    if (db_connection != null)
        db_connection.close();
}
```

Figure 4: Try-finally Pattern

Different programming languages have very different semantics in the event an exception attempts to propagate beyond a catch or finally block. Many languages, 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++ (Stroustrup (1994)). 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[5], 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 (Thomas et al. (2009)).

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. Even then, when the continuation needs to be passed to additional functions, the user must revert to the tedious and error-prone continuation-passing style. 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.

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.

3.3 Context Statements

The development of the "with statement" in Python, the "using statement" in C#, and the "try-with-resources statement" in Java 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

[5]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++, Javascript and Python do not meet these requirements even though they technically allow first-class functions.

the block. Further, the result of the expression is meant to be equipped with methods for 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 (Gosling et al. (2013), van Rossum and Coghlan (2011), Wiltamuth and Hejlsberg (2003)).

Figure 5 is an example of a with statement in Python (the corresponding C# and Java versions are 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 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" (Sitaram and Felleisen (1990a)).

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 (Alexandrescu (2010), Baugh (2010)). We will call these statements "defer statements" after Go terminology[6]. 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 scope 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 exception. This allows 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.

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

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 compared to try-finally or RAII (Mars (2015)).

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

Figure 8: D-style defer statement

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 Jones (2002). 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 primitives that affect or depend on the external environment 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 (Swierstra (2008), Wadler (1989)), 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`[7]), 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.

```
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

The advantages of this technique are clear if we consider Haskell's monads in relation to monads in other languages[8]. 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 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

[7]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 $\frac{1}{0}$.

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

`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 (Ploeg and Kiselyov (2014)). 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 ()
    (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.

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

Figure 10: Context-Stack Addition

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.

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

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)))))

;; 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))
```

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

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 λ_{SFCC}, then we will return to more significant examples of use.

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 (Filinski (1994)), though this is admittedly not a barrier in production languages, even those which already include exceptions (Herman (2007)).

Though powerful, undelimited continuations have been found to be a poor choice in comparison with delimited continuations (Dybvig et al. (2007), Haynes and Friedman (1987), Sitaram (1993), Sitaram and Felleisen (1990b)). 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[9]. 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 (Filinski (1994)), and as such is really only of interest to compiler writers. Indeed, as far as the user-level language is concerned, undelimited continuations are not able to fully abstract control flow (Sitaram and Felleisen (1990b)).

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.

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

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 (Flatt (2015)). In fact, every delimited control operator in Racket is either presented here directly, or else can be easily simulated.

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.

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 (Felleisen (1988)) 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 Dybvig et al. (2007), we will call the collection of these systems ${}^{\pm}\mathcal{F}^{\pm}$. The terminology derives from Felleisen (1988), 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 Felleisen (1988)—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 Felleisen (1988). It leaves the stack mark behind, and does not include it in the continuation.
- ${}^{+}\mathcal{F}^{+}$ is called *shift* from Danvy and Filinski (1990), 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 Hieb and Dybvig (1990), 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 Gunter et al. (1995), we will examine next. It removes the stack mark, and also includes it in the continuation.

[10] In Felleisen (1988), what we call stack mark is called a "prompt."

A more in-depth discussion of the properties of this system is available in Dybvig et al. (2007).

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 Hieb and Dybvig (1990): "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 Gunter et al. (1995). 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 Sitaram (1993), 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 as to be of questionable utility. Given the possibilities we have discussed, we see several design 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 the computation; in other systems, that information might only exist near the delimiting operator. Although the focus in FCC research has been into the former systems, exception handling certainly falls into the latter category.

There is one wrinkle in the choice of (3) to allow the user the ability to create stack marks on the fly With first-order FCC, we can implement finally, or any other control structures which intercept non-local control flow. However, when we have dynamically-generated prompts, all a code fragment has to do is abort to a prompt that is not recognized by a finalizer, e.g. any freshly-obtained prompt. The introduction of dynamic prompts thus requires an additional mechanism to implement finalizers and other control-flow interception mechanisms.

There is one additional design choice, related to the misuse of continuations. What should happen when a program attempts continuation capture, but does not encounter any matching delimiting stack mark? In Racket, a call to shift without a corresponding reset operates by capturing the entire continuation, rather than a subcontinuation. It is this author's opinion that, since undelimited control is almost never intended, undelimited continuation capture should always be an error. A similar error occurs when an exception is thrown but unhandled: the program crashes and a stack trace may be printed. Indeed, it would be simple when a continuation capture is undelimited, to terminate the program, possibly with a stack trace. The choice to terminate follows the "catch errors early" principle, and has proven effective in a vast number of exception implementations, whereas delimited control has only a few implementations.

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.

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.

5 Review

We have now examined a wide range of control flow operators from eight independent programming languages in several paradigms. In this section, we will summarize the properties which were found effective at resource management and those which were not.

The use of the stack to effect cleanup is effective and deterministic. RAII (§3.1), try-finally (§3.2) and defer statements (§3.4) all illustrate this. In contrast, we cannot tie cleanup to the garbage collector since it is non-deterministic, and manual, i.e. error-prone, cleanup is precisely what we are seeking to avoid. More subtly, an excessively lazy evaluation strategy eliminates linear nature of the stack, eroding the determinism of stack-based constructs.

The re-use of resource management code, as in RAII (§3.1) and context statements (§3.3) helps reduce code duplication. At the same time, however, some management code is only used once, in which case it is best to give the code in-line, rather than introduce additional boilerplate. Thus, we must be able to give resource management code both directly at the usage site as well as in a re-usable form.

Most control structures that intercept non-local control flow also intercept local control flow. Nevertheless, the scope statements in D (§3.4) and monads (§3.5) are able to discriminate between local and non-local exit[12]. Although we did not discuss it in the text,

[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.

[12] Since monads can only simulate non-local control flow, their ability to distinguish relies on encoding control flow into data.

Python is also able to detect local exit with the try-else construct (Foundation (2015)). The ability to take different actions depending on the nature of control flow is useful for software transactions.

The same way try-catch-finally (§3.2) can be straightforwardly audited for exception safety, we would like our language to be easily checked for the existence of non-local control flow. In the languages we have discussed so far, there has only been one operator which initiates non-local exit. With these constraints, it is simple to document any functions which transitively rely on control operator, or to encode this property with the type system, if possible. With the exception of call/cc (§4.2), non-local entrance to indistinguishable from a function call, so as long as we constrain ourselves to composable continuations, we need not specially note when captured continuations are restored.

We saw in §4.2 that we must be able to delimit control operators and in §4.3.2 we specifically saw the need for a supply of delimiters and operators, differing by prompts, so that we may implement independent control structures that can nest. This is accomplished by a new kind of value, a prompt, which attaches to a delimiter or operator; control operators are then only delimited if the delimiter has a matching prompt. With this feature, however, we also need specialized constructs, such as that in §4.4 to intercept non-local control flow, since we can no long rely on any delimiter to intercept arbitrary non-local control flow.

Now, with these advantages and disadvantages in mind, we are ready to turn to the development of λ_{SFCC}.

Part II

Thesis

Our thesis begins with the presentation of an extended lambda calculus designed for secure first-class control, which we call λ_{SFCC}. We then briefly examine its relationship to a few important existing control structures. Next, we present an operational semantics for λ_{SFCC} 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.

6 The SFCC Calculus

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 Dybvig et al. (2007), the `fcontrol` operator from Sitaram (1993), dynamic-wind from Scheme, and the ability to discriminate normal and abnormal exit from D.

As regards our delimiting and control operators, the key idea is to specify handlers on both the delimiting operator (as in fcontrol) and on the control operator (as in most other systems). 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. For the management of resources, the key insight is that there are only four forms of control flow, each of a distinct character: entrance vs. exit and local vs. non-local. Thus, all we need to do is to execute an additional expression (the setup or clean up code) whenever control moves past a guard.

We will present the concepts of λ_{SFCC} 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.

6.1 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[13] $\square$, 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	::=		
		$\mid$	$\square$	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

[13]Subcontinuations may nest, but we treat these as simple values rather than deconstructible syntax trees, so we don't count holes that appear underneath angle brackets.

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 Felleisen (1988) 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 15.

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

Figure 15: Extend with Control Operators

The last two productions for K now include a $\Box$ superscript in addition to the hole which appears in the K non-terminal of those productions. We recover the existence of a single salient hole by ignoring the superscript in these productions as we search for the hole. Essentially, the superscript is a bookkeeping device rather than a real part of an expression. We can do this because we now also restrict programs so that they not only

contain no free variables, as is normal in the lambda calculus, but they also may only have $\Box$ as the superscript of any $\mathcal{F}$ expression. Since the presence of the $\Box$ increases line noise without contributing much information, we will often elide it:

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

We will now turn to discussion of the three interlocking subsystems of this extension.

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 Felleisen (1988), 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:

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

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.

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

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.

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

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

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

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

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

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

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

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

Figure 16: Extend with Extent Guards

In the grammar, given in Figure 16, 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:

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

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

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

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

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

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

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

are a straightforward extension of the rule $\#_{p}^{e} v \mapsto v$ 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:

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

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:

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

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. We introduce these so that users and language designers have the option to insert them at any point where propagation of continuation capture would be in error, but we also generalize the ability to respond. Instead of simply terminating the process (as when an exception propagates out from within a destructor in C++), a handler may be installed which can respond as appropriate in the situation. The grammar is given in Figure 17.

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

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 Felleisen (1990), 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	$::=$	$\Box$
		$\mid$	$K\ e \mid v\ K$
		$\mid$	$\#^{e_h}_K e \mid \#^K_p e \mid \#^v_p K$
		$\mid$	$\mathcal{F}^{\Box}_K e \mid \mathcal{F}^{\Box}_p K$
		$\mid$	$\#^{e_g}_{\Uparrow} K \mid \#^{e_g}_{\Downarrow} K \mid \#^K_{\blacksquare} e$

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:

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

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

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

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

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

These together comprise the interesting core of the system.

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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:

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

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. We will also want a prompt $\mathbf{p_0}$ that is only meant to be installed at the top of the program, usually by the run-time system. Such a prompt will allow us to implement control structures that do capture or discard the entire continuation. We will introduce a few more prompts 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.

These constructs, and those we will develop in the rest of this section will be useful again in 9.

7.2 MFDC

Since we have already mentioned the major inspirations for λ_{SFCC}, it only makes sense to begin our survey of relationships with those inspirations. The system ${}^{\pm}\mathcal{F}^{\pm}$ given in Dybvig et al. (2007), 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 stack before evaluating its body.

Two of these operators are directly reflected in λ_{SFCC}, and a third is easily translated:

$$\begin{aligned} \mathsf{newPrompt} &\equiv \nu \\ \mathsf{pushSubCont}\ e_k\ e &\equiv e_k\ e \\ \mathsf{withSubCont} &\equiv \lambda p, f.\mathcal{F}_p \lambda \_, k.f\ k \end{aligned}$$

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 pushPrompt e_p $e \equiv \#_{e_p}^{v_?} e$, but the choice of $v_?$ is not fixed by the semantics of Dybvig et al. (2007). 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

$$\mathsf{pushPrompt}\ e_p\ e \equiv \mathsf{let}\ p = e_p\ \mathsf{in}\ \#_p^{\lambda k,x.\mathcal{F}_p \lambda h', k'.h'\ (k' \circ k)\ x} e$$

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

We will now examine an example from Dybvig et al. (2007).

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

This translates into λ_{SFCC} 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 $\mathbf{a}$—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.

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

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:

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

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

expression and a handler; the control operator requires only an abort value. Both operators may be equipped with a prompt, or allowed to use the default prompt, **p**.

Implementations of these operators are given below. Since the semantics in Sitaram (1993) applies the handler to arguments in reverse order, we will make use here of the well-known function **flip** $= \lambda f, x, y.f\ y\ x$. The following are faithful implementations of the operators defined in Sitaram (1993):

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

7.4 Scope Statements and Dynamic Wind

We will cover our last two inspirations together. The implementations here justify our descriptions of extent guards in 6.4.

We mentioned that extent guards are a generalization of scope statements, where now execution on entrance is made available alongside execution on exit. Here, we see that the three scope statements correspond to our three extent exit operators.

$$\begin{aligned}
\mathsf{scope(exit)}\ e_g; e &\equiv \#_{\downarrow\downarrow}^{e_g} e \\
\mathsf{scope(success)}\ e_g; e &\equiv \#_{\downarrow}^{e_g} e \\
\mathsf{scope(failure)}\ e_g; e &\equiv \#_{\Downarrow}^{e_g} e
\end{aligned}$$

Of course, λ_{SFCC} also has three extent entrance operators which operate symmetrically.

We also mentioned that our extent guards split the dynamic wind operator into its component parts. In Scheme, dynamic-wind is a function which takes three zero-argument functions, rather than being a special form taking three subexpressions; this is the reason for the enclosing lambda in our implementation. We see that all six extent guards are involved in dynamic wind: the $\#_{\downarrow\downarrow}$ and $\#_{\uparrow\uparrow}$ provide the most concise implementation, but recall that these can be expressed in terms of $\#_{\downarrow}, \#_{\Downarrow}$ and $\#_{\uparrow}, \#_{\Uparrow}$ respectively.

$$\begin{aligned}
\textbf{dynamic-wind} &\equiv \lambda pre, body, post.\#_{\uparrow\uparrow}^{pre\ ()} \#_{\downarrow\downarrow}^{post\ ()} body\ () \\
&\equiv \lambda pre, body, post.\#_{\uparrow}^{pre\ ()} \#_{\Uparrow}^{pre\ ()} \#_{\downarrow}^{post\ ()} \#_{\Downarrow}^{post\ ()} body\ ()
\end{aligned}$$

7.5 Try-catch-finally

We now move on to a demonstration that exception handling can be easily implemented within λ_{SFCC}. 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 $\#_{\Downarrow}$.

$$\begin{aligned} \mathsf{throw}\ e &\equiv \mathcal{A}_{\mathbf{exn}} e \\ \mathsf{try}\ e\ \mathsf{finally}\ e_f &\equiv \#_{\Downarrow}^{e_f} e \end{aligned}$$

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 the exception value 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.

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

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

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

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 definitions related to other FCC constructs. Nevertheless, each translation we give is very simple. 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.

7.6 Miscellaneous

We will now rapidly review a number of control operators we mentioned in §4.3. We there noted a resemblance between ${}^{-}\mathcal{F}^{-}$ and cupto, which we now implement:

$$\begin{aligned}
\#e &\equiv \#^{()}_{\mathbf{p}} e \\
{}^{-}\mathcal{F}^{-} e &\equiv \mathsf{let}\ f = e\ \mathsf{in}\ \mathcal{F}_{\mathbf{p}} \lambda_, k.f\ k \\
\mathsf{set}\ e_p\ \mathsf{in}\ e &\equiv \#^{()}_{e_p} e \\
\mathsf{cupto}\ e_p\ \mathsf{as}\ k\ \mathsf{in}\ e &\equiv \mathsf{let}\ p = e_p, f = e\ \mathsf{in}\ \mathcal{F}_p \lambda_, k.f\ k
\end{aligned}$$

In light of this similarity, we can simply rename cupto to ${}^{-}\mathcal{F}^{-}$ with a subscript to specify the prompt. We can also do the same for set.

$$\begin{aligned}
\#_{e_p} e &\equiv \#^{()}_{e_p} e \\
&\equiv \mathsf{set}\ e_p\ \mathsf{in}\ e \\
{}^{-}\mathcal{F}^{-}_{e_p} e &\equiv \mathsf{let}\ p = e_p, f = e\ \mathsf{in}\ \mathcal{F}_p \lambda_, k.f\ k \\
&\equiv \mathsf{let}\ p = e_p, f = e\ \mathsf{in}\ \mathsf{cupto}\ p\ \mathsf{as}\ k\ \mathsf{in}\ f\ k
\end{aligned}$$

We now have three more variations on $\mathcal{F}$ with subscript, where each varies based on whether they reinstall the prompt and when.

$$\begin{aligned}
{}^{+}\mathcal{F}^{-}_{e_p} e &\equiv \mathsf{let}\ p = e_p, f = e\ \mathsf{in}\ {}^{-}\mathcal{F}^{-}_{p} \lambda h, k.\#^{h}_{p} f\ k \\
{}^{-}\mathcal{F}^{+}_{e_p} e &\equiv \mathsf{let}\ p = e_p, f = e\ \mathsf{in}\ {}^{-}\mathcal{F}^{-}_{p} \lambda h, k.f\ \langle \#^{h}_{p} k\ \Box \rangle \\
{}^{+}\mathcal{F}^{+}_{e_p} e &\equiv \mathsf{let}\ p = e_p, f = e\ \mathsf{in}\ {}^{-}\mathcal{F}^{-}_{p} \lambda h, k.\#^{h}_{p} f\ \langle \#_{p} k\ \Box \rangle
\end{aligned}$$

With these, we can give the implementation of the entire ${}^{\pm}\mathcal{F}^{\pm}$ system quite easily by noting

$$\begin{aligned}
\#e &\equiv \#_{\mathbf{p}} e \\
{}^{\pm}\mathcal{F}^{\pm} e &\equiv {}^{\pm}\mathcal{F}^{\pm}_{\mathbf{p}} e
\end{aligned}$$

With just these few identifications, we have implemented every control operator discussed in §4.3 short of fcontrol, which was discussed earlier.

It was noted in Dybvig et al. (2007) that the control operator could be factored into two operators: one that captured the continuation and one which performed abort. It was also claimed that two systems differing only on this choice of factoring were equivalent. We will briefly show this here. As with our implementation of the MFDC operators, we will not expend much effort on the handlers installed with prompts, since these do not appear in Dybvig et al. (2007).

In the definition of capture, we arrange for the prompt and continuation to be re-installed before moving to use the continuation in the body of the operator. Aborting is even simpler: we simply discard the continuation and move directly to the body. To get back to MFDC's withSubCont operator, we simply chain these two new control structures together.

$$\begin{aligned}
\textbf{capture} &\equiv \lambda p, f.\mathcal{F}_p \lambda h, k.\#_p^h k\ (f\ k) \\
\textbf{abort} &\equiv \lambda p.\langle \mathcal{F}_p \lambda _, _.\Box \rangle \\
\textbf{withSubCont} &\equiv \lambda p, f.\textbf{capture}\ p\ (\lambda k. \\
&\qquad\qquad \textbf{abort}\ p\ (f\ k))
\end{aligned}$$

8 Efficient Implementation

Now that we have developed our semantics, we turn to efficiency concerns. We will show that λ_{SFCC} need not impose inherent algorithmic complexity penalties. In particular, if FCC techniques are not used, then the interpreter will be no slower than a CESK machine: all the cost of FCC is limited to capturing and restoring continuations. To demonstrate this, we develop a simple interpreter for the language.

8.1 Supporting Definitions

We will use largely the same grammar as in our semantics, with the exception of the representation of continuations. We apply a zipper transformation to the data structure; here continuations are represented as a list of size 1 continuations, which we call contexts. We continue our upward-growing stack convention in text, and also adopt a leftward-growing convention. Further, a continuation begin captured will not be stored by the control operator, but in a separate control stack. We therefore eliminate the superscript from the $\mathcal{F}$ operator.

Values	v	::=	$\lambda x.e \mid K \mid p$
Expressions	e	::=	$v \mid x \mid e\ e$
		$\mid$	$\nu \mid \#_{e_p}^{e_h} e \mid \mathcal{F}_{e_p} e$
		$\mid$	$\#_{\Uparrow}^{e_g} e \mid \#_{\Downarrow}^{e_g} e \mid \#_{\blacksquare}^{e_h} e$
Continuations	K	::=	$[k_n, \ldots, k_1]$
Contexts	k	::=	$\Box\ e \mid v\ \Box$
		$\mid$	$\#_{\Box}^{e_h} e \mid \#_p^{\Box} e \mid \#_p^v \Box$
		$\mid$	$\mathcal{F}_{\Box} e \mid \mathcal{F}_p \Box$
		$\mid$	$\#_{\Uparrow}^{e_g} \Box \mid \#_{\Downarrow}^{e_g} \Box \mid \#_{\blacksquare}^{\Box} e \mid \#_{\blacksquare}^{e} \Box$

Figure 19: Interpreter Grammar

We can adjust the top of the stack through concatenation; we write $K'K$ to mean the extension of the continuation K with an the contexts of K' on top.

We will need two functions, *enterGuard* and *exitGuard*, of two parameters: a continuation and a base-case expression. Intuitively, what these guards do is to accumulate a sequence of expressions from the extent guards, $\#_{\Uparrow}$ and $\#_{\Downarrow}$ respectively, in the continuation. The precise sequence is different for the two functions: the result of *enterGuard* will evaluate the guards in FIFO order, but *exitGuard* will evaluate them in LIFO order with respect to when the corresponding extent guards were pushed to the stack. Either way, the last expression in the sequence is the base-case expression. Thus, these functions formalize the process of collecting and evaluating control guards that were registered by a guard expression before continuing on. Formally, we have

- $enterGuard(K, e) = e_g; enterGuard(K^{\uparrow}, e)$ when $K = K^{\uparrow}[\#_{\Uparrow}^{e_g}\Box]K^{\downarrow}$ and $enterGuard(K^{\downarrow}, e) = e$,
- or $enterGuard(K, e) = e$ otherwise.
- $exitGuard(K, e) = e_g; exitGuard(K^{\downarrow}, e)$ when $K = K^{\uparrow}[\#_{\Downarrow}^{e_g}\Box]K^{\downarrow}$ and $exitGuard(K^{\uparrow}, e) = e$,
- or $exitGuard(K, e) = e$ otherwise.

We also require the partial function *splitStack* taking a continuation and a prompt. It finds in the passed continuation the location of the most-recently install control delimiter either of the passed prompt or the barrier. It then returns the stack portions strictly above and below the delimiter, as well as the handler attached to that delimiter. The result is tagged with whether control was delimited by a delimiter for the prompt or by a barrier. Formally, we have $splitStack(K, p)$ defined as

- either $\#\langle K^{\uparrow}, v_h, K^{\downarrow}\rangle$ when $K = K^{\uparrow}[\#_p^{v_h}\Box]K^{\downarrow}$
- or else $\blacksquare\langle K^{\uparrow}, v_h, K^{\downarrow}\rangle$ when $K = K^{\uparrow}[\#_{\blacksquare}^{v_h}\Box]K^{\downarrow}$,
- provided both $K^{\uparrow} \neq \hat{K}^{\uparrow}[\#_p^{\hat{v}_h}\Box]\hat{K}^{\downarrow}$
- and $K^{\uparrow} \neq \hat{K}^{\uparrow}[\#_{\blacksquare}^{\hat{v}_h}\Box]\hat{K}^{\downarrow}$

In all other cases, the function is undefined.

8.2 Evaluation

Our interpreter is similar to a SESK machine, except that we have no need for an environment (we will be performing substitution directly), and the only store necessary is a pool of fresh prompts. We define evaluation of an expression using a transition system operating on triples of the form $\langle e, K, P\rangle$, where P is a set of prompts. The translation relation is written $\longmapsto$, and its reflexive transitive closure as $\longmapsto^*$. We say that $eval(e) = v$ when $\langle e, [], P\rangle \longmapsto^* \langle v, [], P'\rangle$. In all other cases, $eval(e)$ is undefined.

Though there appear to be many transitions, there are really only a few ideas present which determine the pattern. We've split our reduction rules into three groups based on common theme. The reductions in Figure 20 simply evaluate subexpressions according to the order of evaluation. Those in Figure 21 unwind values past the various stack marks which can appear in the continuation. The third group, in Figure 22, performs the interesting work of the interpreter; we will discuss each of these rules individually.

$$
\begin{array}{rrcll}
\textsc{app}_1\text{:} & \langle e\ e', K, P\rangle & \longmapsto & \langle e, [\Box\ e']K, P\rangle & \\
\textsc{app}_2\text{:} & \langle v, [\Box\ e]K, P\rangle & \longmapsto & \langle e, [v\ \Box]K, P\rangle & \text{where } v \neq K' \\
\#_1\text{:} & \langle \#^{e_h}_{e_p} e, K, P\rangle & \longmapsto & \langle e_p, [\#^{e_h}_{\Box} e]K, P\rangle & \\
\#_2\text{:} & \langle v_p, [\#^{e_h}_{\Box} e]K, P\rangle & \longmapsto & \langle e_h, [\#^{\Box}_{v_p} e]K, P\rangle & \\
\#_3\text{:} & \langle v_h, [\#^{\Box}_{v_p} e]K, P\rangle & \longmapsto & \langle e, [\#^{v_h}_{v_p} \Box]K, P\rangle & \\
\textsc{c}_1\text{:} & \langle \mathcal{F}_{e_p} e, K, P\rangle & \longmapsto & \langle e_p, [\mathcal{F}_{\Box} e]K, P\rangle & \\
\textsc{c}_2\text{:} & \langle v, [\mathcal{F}_{\Box} e]K, P\rangle & \longmapsto & \langle e, [\mathcal{F}_{v} \Box]K, P\rangle & \\
\Uparrow_1\text{:} & \langle \#^{e_g}_{\Uparrow} e, K, P\rangle & \longmapsto & \langle e, [\#^{e_g}_{\Uparrow} \Box]K, P\rangle & \\
\Downarrow_1\text{:} & \langle \#^{e_g}_{\Downarrow} e, K, P\rangle & \longmapsto & \langle e, [\#^{e_g}_{\Downarrow} \Box]K, P\rangle & \\
\blacksquare_1\text{:} & \langle \#^{e_h}_{\blacksquare} e, K, P\rangle & \longmapsto & \langle e_h, [\#^{\Box}_{\blacksquare} e]K, P\rangle & \\
\blacksquare_2\text{:} & \langle v_h, [\#^{\Box}_{\blacksquare} e]K, P\rangle & \longmapsto & \langle e, [\#^{v_h}_{\blacksquare} \Box]K, P\rangle &
\end{array}
$$

Figure 20: Searching for a Redex

$$
\begin{array}{rrcl}
\#\text{:} & \langle v, [\#^{v_h}_{p} \Box]K, P\rangle & \longmapsto & \langle v, K, P\rangle \\
\Uparrow\text{:} & \langle v, [\#^{v_h}_{\Uparrow} \Box]K, P\rangle & \longmapsto & \langle v, K, P\rangle \\
\Downarrow\text{:} & \langle v, [\#^{v_h}_{\Downarrow} \Box]K, P\rangle & \longmapsto & \langle v, K, P\rangle \\
\blacksquare\text{:} & \langle v, [\#^{v_h}_{\blacksquare} \Box]K, P\rangle & \longmapsto & \langle v, K, P\rangle
\end{array}
$$

Figure 21: Removing Stack Marks

$$
\begin{array}{rrcll}
\textsc{app}_\lambda\text{:} & \langle v, [(\lambda x.e)\ \Box]K, P\rangle & \longmapsto & \langle e[x\backslash v], K, P\rangle & \\
\textsc{app}_K\text{:} & \langle K', [\Box\ e]K, P\rangle & \longmapsto & \langle e_g, K'K, P\rangle & \\
 & & \text{where} & e_g = \mathit{enterGuard}(K', e) & \\
\textsc{new}\text{:} & \langle \nu, K, \{p\} \cup P\rangle & \longmapsto & \langle p, K, P\rangle & \\
\textsc{c}_\#\text{:} & \langle v, [\mathcal{F}_p \Box]K, P\rangle & \longmapsto & \langle e_g, K^{\downarrow}, P\rangle & \\
 & & \text{where} & \#\langle K^{\uparrow}, v_h, K^{\downarrow}\rangle = \mathit{splitStack}(K, p) & \\
 & & \text{and} & e_g = \mathit{exitGuard}(K^{\uparrow}, v\ v_h\ K^{\uparrow}) & \\
\textsc{c}_\blacksquare\text{:} & \langle v, [\mathcal{F}_p \Box]K, P\rangle & \longmapsto & \langle e_g, K^{\downarrow}, P\rangle & \\
 & & \text{where} & \blacksquare\langle K^{\uparrow}, v_h, K^{\downarrow}\rangle = \mathit{splitStack}(K, p) & \\
 & & \text{and} & e_g = \mathit{exitGuard}(K^{\uparrow}, v_h\ p\ K^{\uparrow}\ v) &
\end{array}
$$

Figure 22: Core Reductions

The introduction of new prompts is handled by the NEW rule. This is the only rule which modifies the supply of prompts. In it, a prompt is removed from the supply and placed into the machine's active expression. Once the programmer has a prompt value on the stack, it will propagate through the continuation, being manipulated, passed it to functions, or directly to other control operators.

Application is handled by the two rules APP_λ, which comes directly from the lambda calculus, and APP_K, which is responsible for applying continuations. The later rule is the only place where the continuation might be extended with more than one context, and so this is the only place we see the use of the *enterGuard* function. After invoking APP_K, the next expression to be evaluated includes not only the argument expression, but also the sequence of all $\#_\Uparrow$ extent guards in the applied continuation.

Finally, control capture is handled by the two rules $\text{C}_\#$ and $\text{C}_\blacksquare$; they differ only in that the former is applied when the continuation is delimited by the appropriate prompt, and the later only when the continuation is delimited by a barrier. These are the only two places where the continuation stack can be reduced in size by more than one context, so these are the only places we use the *exitGuard* function. Similar to the APP_K rule, the next expression to be evaluated includes a sequence of expressions drawn from extent guards, this time from the $\#_\Downarrow$ guards of the captured continuation.

There are a few places where the interpreter could get stuck. In an application, if the first expression does not evaluate to a function or continuation, then evaluation cannot continue; this is as normal in the lambda calculus. Further, if *splitStack* is undefined in either $\text{C}_\#$ or $\text{C}_\blacksquare$, the interpreter will get stuck. This situation represents an attempt to capture a continuation which is not delimited anywhere in the continuation. This decision prevents unintentional undelimited control capture.

This particular interpreter was presented with simplicity in mind, so that the *splitStack* function will take linear time on the size of the stack. However, we can use metacontinuation technique as in Dybvig et al. (2007) (FIXME I'm sure they got it from somewhere else) to reduce the cost to linear time in the number of delimiters in the stack. This requires a slight transformation only on the internal representation of the control stack, which is why we did not involve those complications in our formalism.

9 Applications

We have previously given a few examples of control structures that can be implemented in λ_{SFCC}, but with the exception of try-catch-finally, these were implementations only of other low-level control constructs. In this section, we will examine a wide range of higher-level control constructs which could see immediate use in programming. We will also examine those control structures found in our literature review which have not already been implemented.

9.1 Context Statement

We already discussed context statements in §3.3, which is a technique in a few object-oriented languages. An object, the context manager is equipped with methods for setup and clean up, which are executed whenever control enters and exits, respectively, the body of the statement. The implementation presents itself immediately from this

description. We write $e.x(e' \ldots)$ to call the x method of e with the arguments $e' \ldots$, as usual.

$$\mathsf{with}\ e\ \mathsf{as}\ x\ \mathsf{in}\ e_{body} \equiv \mathsf{let}\ x = e\ \mathsf{in}\ \#_{\uparrow\uparrow}^{x.enter()} \#_{\downarrow\downarrow}^{x.exit()} e_{body}$$

9.2 Fluid Variables

Fluid variables are variables whose stored values can temporarily take on alternate values. That is, if e_f is a fluid variable, then in $\mathsf{fluid}\ e_f := e_v; e$, that variable takes the value of e_v within e, but outside takes its original value.

The simplest way to implement this behavior requires mutable reference cells. If we have a mutable cell, c, then we use the notation ${}^*c$ to dereference the cell, obtaining its contents, and we use the notation $c := v'$ to store a new value, v' into the cell.

Our implementation simply saves the old value of the fluid variable locally, then places appropriate extent guards around the body expression, e. Whenever control enters the body expression, the new value is stored, and whenever control exits, the old value is reinstated.

$$\begin{aligned}\mathsf{fluid}\ e_f := e_v; e \equiv \mathsf{let}\ & c = e_f, \\ & x_0 = {}^*c, x = e_v \\ \mathsf{in}\ & \#_{\downarrow\downarrow}^{c := x_0} \#_{\uparrow\uparrow}^{c := x} e\end{aligned}$$

9.3 Early Exit

One difference between functions in imperative and functional languages is that there is no early return from a fucntion in a functional language. Those functions which allow early return we will call subroutines. We can quite easily implement a "return statement" in a functional language with FCC. For each subroutine, we create a prompt and install it around the subroutine with a handler that simply returns the value aborted with. Finally, we make the name `return` available inside the body, which will abort up to the installed prompt with whatever value it is passed.

$$\begin{aligned}\mathsf{subroutine}\ e \equiv \mathsf{let}\ & p = \nu, \\ & \texttt{return} = \lambda x.\mathcal{A}_p x \\ \mathsf{in}\ & \#_p^{\lambda\_, x.x} e\end{aligned}$$

After implementing early return from subroutine, we can also implement more general early exit, similar to early exit from labeled loop constructs. In this implementation, the label's name becomes available in the scope of the body expression as an abort function.

The abort is delimited by an installed handler which, just as with subroutines, which will simply arrange for the abort value to be returned unaltered.

$$l\colon \{e\} \equiv \mathsf{let}\ p = \nu, l = \lambda x.\mathcal{A}_p x\ \mathsf{in}\ \#_p^{\lambda\_,x.x} e$$

The advantage of this construct over the labeled loop constructs of existing languages is that any expression may now be labeled and exited early from, rather than only those expressions and statements the language designer was able to predict the use of and encode into the language. For example, with a labeled exit construct, it is simple to implement subroutines as above:

$$\mathsf{subroutine}\ e \equiv \texttt{return}\colon \{e\}$$

An additional feature of these implementations is that the ability to perform early exit can be delegated to further functions. We used this technique in §4.1 to implement foldl/ee. We will now give an implementation and reduction sketch of that example in λ_{SFCC}.

$$\begin{aligned}\textbf{foldl/ee} &\equiv \lambda f, xs, z.\, break\colon \{foldl\ (f\ break)\ xs\ z\}\\ \textbf{all} &\equiv \lambda pred, xs.\, \mathsf{let}\ test = \lambda break, x, z.\, \mathsf{if}\ pred\ x\ \mathsf{then}\ z\ \mathsf{else}\ break\ \textbf{false}\\ &\qquad \mathsf{in}\ \textbf{foldl/ee}\ test\ xs\ \textbf{true}\end{aligned}$$

Notice in the reduction sketch how the *break* function is passed to another function. In particular, it is passed on to the reducing function, so that the *test* function in **all** can manipulate control flow as it needs. We have chosen this example precisely because it shows that the ability to abort can be delegated to a function which was not even anticipated when we implemented **foldl/ee**. This demonstrates the great potential for reuse of first-class control structures.

We now give a short reduction sketch of foldl/ee in λ_{SFCC}:

$$
\begin{aligned}
&\textbf{all}\ (\lambda x.x > 0)\ [0,1]\\
&\mathsf{let}\ test = \lambda break, x, z.\,\mathsf{if}\ (\lambda x.x > 0)\ x\ \mathsf{then}\ z\ \mathsf{else}\ break\ \textbf{false}\\
&\qquad \mathsf{in}\ \textbf{foldl/ee}\ test\ [0,1]\ \textbf{true}\\
&break\colon \{foldl\ (\textbf{test}\ break)\ [0,1]\ \textbf{true}\}\\
&\#^{\lambda_-,x.x}_{\mathbf{p}} foldl\ (\textbf{test}\ (\lambda x.\mathcal{A}_{\mathbf{p}}x))\ [0,1]\ \textbf{true}\\
&\#^{\lambda_-,x.x}_{\mathbf{p}} foldl\ (\lambda x, z.\,\mathsf{if}\ (\lambda x.x > 0)\ x\ \mathsf{then}\ z\ \mathsf{else}\ (\lambda x.\mathcal{A}_{\mathbf{p}}x)\ \textbf{false})\ [0,1]\ \textbf{true}\\
&\#^{\lambda_-,x.x}_{\mathbf{p}} foldl\ (\lambda x, z.\,\mathsf{if}\ (\lambda x.x > 0)\ x\ \mathsf{then}\ z\ \mathsf{else}\ (\lambda x.\mathcal{A}_{\mathbf{p}}x)\ \textbf{false})\ [1]\\
&\qquad (\mathsf{if}\ 0 > 0\ \mathsf{then}\ \textbf{true}\ \mathsf{else}\ (\lambda x.\mathcal{A}_{\mathbf{p}}x)\ \textbf{false})\\
&\#^{\lambda_-,x.x}_{\mathbf{p}} foldl\ (\lambda x, z.\,\mathsf{if}\ (\lambda x.x > 0)\ x\ \mathsf{then}\ z\ \mathsf{else}\ (\lambda x.\mathcal{A}_{\mathbf{p}}x)\ \textbf{false})\ [1]\\
&\qquad (\mathcal{F}_{\mathbf{p}}\lambda h, k.\,h\ k\ \textbf{false})\\
&\#^{\lambda_-,x.x}_{\mathbf{p}} \mathcal{F}^K_{\mathbf{p}}\lambda h, k.\,h\ k\ \textbf{false}\\
&(\lambda h, k.\,h\ k\ \textbf{false})\ (\lambda\_, x.x)\ \langle K\rangle\\
&(\lambda\_, x.x)\ \langle K\rangle\ \textbf{false}\\
&\textbf{false}
\end{aligned}
$$

9.4 Streams and Nondeterministic Choice

A stream is a list structure whose values are computed on demand (Abelson and Sussman (1996)). We will think of our streams here primarily as functions which can return multiple times. Our introduction form for streams takes an expression and make `yield` available within the expression. A call to `yield` produces the next item in the stream. We access the elements of a stream by applying **uncons**, which results either in a cons cell, $(head : tail)$, where $head$ is the next element in the stream, and $tail$ is the rest of the stream, or else the special value **nil**, when the end of the stream is reached.

In our implementation, we create a new prompt, just as in our subroutines. The yield operator takes the role of the return operator. This time, the handler is used to return the yielded value along with the rest of the stream as a continuation. The rest of the stream must be delimited again, with the same handler, so we use the Y-combinator to allow the hander to recursively install itself.

$$
\begin{aligned}
\mathsf{stream}\ e \equiv\ &\mathsf{let}\ p = \nu,\\
&\qquad h = \mathbf{Y}\ \lambda h, k, x.(x : \#^h_p k)\\
&\quad \mathsf{in}\ \langle \#^h_p \Box; \mathsf{let}\ \texttt{yield} = \lambda x.\mathcal{A}_p x\ \mathsf{in}\ e; \textbf{nil}\rangle\\
\textbf{uncons} \equiv\ &\lambda s.\,s\ ()
\end{aligned}
$$

Now that we have streams, it is simple to implement the *amb* operator, which performs non-deterministic choice. Essentially, the **amb** operator accepts a list of options, then chooses just the correct elements to ensure the following computation does not fail.

Of course, the technique cannot be magic, but it does rely on the backtracking ability of streams. First, **amb** turns a list into a stream by yielding each element insuccession. Then, we have two functions, **succeed** and **fail** which signal the corresponding situations to the implementation. Finally, we use the >>= operator to combine computations using **amb**. What it does is to accumulate all the selections that succeeded (in general, a single **amb** computation might itself produce multiple possible results).

$$\begin{aligned}\textbf{amb} &\equiv \lambda xs.\textbf{map}\ \texttt{yield}\ xs\\ \textbf{succeed} &\equiv \lambda x.[x]\\ \textbf{fail} &\equiv []\\ >>= &\equiv \lambda x, k.\textbf{fold concat}\ (\textbf{map}\ k\ x)\ []\end{aligned}$$

As an example, we can use this technique to generate a list of all pairs of factors for a given number. We simply select a number up to the input and another up to the first, and fail whenever the pair does not multiply to produce the input.

$$\textbf{factors} = \lambda a.[1..a] >>= \lambda x.[1..x] >>= \lambda y.\mathsf{if}\ x * y == a\ \mathsf{then}\ \textbf{succeed}\ (x, y)\ \mathsf{else}\ \textbf{fail}$$

9.5 Subinterpreters

In some cases it is useful to isolate the execution of some piece of code from the main process, e.g. to run third-party code, as in a REPL. It is possible to call on the operating system to start a new process which runs the code, but this is usually overkill if the untrusted code does not run concurrently. Instead, we could execute it in the same process by invoking a subinterpreter.

In the presence of FCC, care must be taken to ensure the code in the subinterpreter cannot capture a continuation which includes some of the main interpreter. To do this, we can implement *eval* using the $\#_\blacksquare$ delimiter, since it is able to intercept control operators of any prompt. Indeed, the main purpose of the $\#_\blacksquare$ is to isolate any control operators within its dynamic extent from affecting the computational context above it. The following implements a simple isolated subinterpreter which treats undelimited control within its argument as an error.

$$\mathsf{subinterpreter}\ e = \#_\blacksquare^{\lambda\_,\_,\_.\mathsf{throw}\ \texttt{"undelimited control flow"}} e$$

9.6 Natural Language Constructs

As our last examples in this section, we turn to a pair of constructs which, although they are quite natural in human language, have largely resisted integration into programming languages. It turns out that both constructs are quite simple to encode with continuations.

The first is the "anaphoric if". This corresponds to the constructs of the form "if the chair next to the table is green, then repaint it". The salient features are that some value is obtained by a complex expression, and this same value is both passed to a predicate, and used in one or both branches of the if statement. In most languages, this has to be laboriously encoded:

```
{
  let it = X;
  if (P) {A} else {B}
}
```

where `it` is free in `V`, `A` and/or `B`.

With λ_{SFCC}, we can implement this pattern by temporarily suspending a continuation while we perform binding. We create a new prompt and matching abort function, `break` as in our early exit operators. We will obtain a value, bound to `it`, and a continuation by evaluating the predicate, e_p. If e_p evaluates to a call to `break`, then the tuple will be the aborted value and its continuation, but if no such call occurs, the value of the entire e_p expression will be bound to a default, the empty continuation. Once we have the `it`-value and a (possibly empty) continuation, we simply evaluate the rest of the continuation and branch on the result. Scope rules ensure that `it` is available inside both the consequent, e_c, and alternate, e_a, expressions.

$$\begin{aligned}\mathsf{ifitis}\ e_p\ \mathsf{then}\ e_c\ \mathsf{else}\ e_a \equiv \mathsf{let}\ (\texttt{it}, k) = \mathsf{let}\ p = \nu, \texttt{break} = \lambda x.\mathcal{A}_p x \\ \mathsf{in}\ \#_p^{\lambda k, x.(x,k)}(e_p, \langle\Box\rangle) \\ \mathsf{in}\ \mathsf{if}\ k\ \texttt{it}\ \mathsf{then}\ e_c\ \mathsf{else}\ e_a\end{aligned}$$

Another common natural language construct is the distribution of a predicate over several objects, as in "is x equal to 3 or 4?" and "are both x and y positive?". This construction can become useful when programming; indeed, a common Python idiom uses set inclusion to implement the former: `if x in {3, 4}`. Unfortunately, set membership is only one instance of the pattern we find here.

In Barker (2004) however, the author suggests that a natural interpretation of these constructs is in terms of continuations. In this implementation, we will use a preselected prompt, which we will write as **c** here. In addition to the two infix conjunctions **and** and **or**, we will also require a suitable bracketing operator, **conj**, which delimits the predicate being tested.

$$\begin{aligned}\mathbf{conj} &\equiv \langle \#_{\mathbf{c}} \Box \rangle \\ \mathbf{and} &\equiv \lambda x, y.\mathcal{F}_{\mathbf{c}} \lambda\_, k.\, (\#_{\mathbf{c}} k\ x)\ \&\&\ (\#_{\mathbf{c}} k\ y) \\ \mathbf{or} &\equiv \lambda x, y.\mathcal{F}_{\mathbf{c}} \lambda\_, k.\, (\#_{\mathbf{c}} k\ x)\ ||\ (\#_{\mathbf{c}} k\ y)\end{aligned}$$

Of course, additional conjunctions might be implemented; we have limited ourselves to these two for brevity.

Now that we have the definitions, we will give an example reduction sketch.

$$
\begin{aligned}
& \textbf{conj}\ (0 == 1\ \textbf{or}\ 2\ \textbf{or}\ 0) \\
\equiv\ & \#_{\mathbf{c}}0 == (\lambda x, y.\mathcal{F}_{\mathbf{c}}\lambda\_, k.\,(\#_{\mathbf{c}}k\ x)\ ||\ (\#_{\mathbf{c}}k\ y))\ ((\lambda x, y.\mathcal{F}_{\mathbf{c}}\lambda\_, k.\,(\#_{\mathbf{c}}k\ x)\ ||\ (\#_{\mathbf{c}}k\ y))\ 1\ 2)\ 0 \\
\mapsto^* & \#_{\mathbf{c}}0 == \mathcal{F}_{\mathbf{c}}\lambda\_, k.\,(\#_{\mathbf{c}}k\ (\lambda x, y.\mathcal{F}_{\mathbf{c}}\lambda\_, k.\,(\#_{\mathbf{c}}k\ x)\ ||\ (\#_{\mathbf{c}}k\ y))\ 1\ 2)\ ||\ (\#_{\mathbf{c}}k\ 0) \\
\mapsto^* & (\#_{\mathbf{c}}\langle 0 == \square\rangle\ \ (\lambda x, y.\mathcal{F}_{\mathbf{c}}\lambda\_, k.\,(\#_{\mathbf{c}}k\ x)\ ||\ (\#_{\mathbf{c}}k\ y))\ 1\ 2)\ ||\ (\#_{\mathbf{c}}\langle 0 == \square\rangle\ 0) \\
\mapsto^* & (\#_{\mathbf{c}}0 == \mathcal{F}_{\mathbf{c}}\lambda\_, k.\,(\#_{\mathbf{c}}k\ 1)\ ||\ (\#_{\mathbf{c}}k\ 2))\ ||\ (\#_{\mathbf{c}}\langle 0 == \square\rangle\ 0) \\
\mapsto^* & (\#_{\mathbf{c}}\langle 0 == \square\rangle\ 1)\ ||\ (\#_{\mathbf{c}}\langle 0 == \square\rangle\ 2)\ ||\ (\#_{\mathbf{c}}\langle 0 == \square\rangle\ 0) \\
\equiv\ & 0 == 1\ ||\ 0 == 2\ ||\ 0 == 0 \\
\mapsto^* & \textbf{true}
\end{aligned}
$$

10 Conclusion

10.1 Results

We have developed a calculus, λ_{SFCC} which includes first-class control along with additional operators for managing the scope of external resources. (TODO FCC is not macro-expressible in the lambda calculus) We have also been able to show by explicit construction that many control structures, both common and rare, are macro-expressible in λ_{SFCC}, sometimes with mutable state. Although mutable state is not macro-expressible in λ_{SFCC}, this is an orthogonal issue. In this way, we have shown that the inclusion of a system of FCC like the λ_{SFCC} into a language without FCC will strictly increase its expressive power in the sense of Felleisen (1990).

We may ask what the risks are regarding the management of external resources after the inclusion of non-local control, and FCC in particular. In software engineering, it is widely accepted that any of a few control constructs are sufficient to reduce the risk of resource mis-management to an acceptable level when exceptions are involved. When FCC is involved, the Scheme community has settled on the dynamic-wind construct for resource management. Through a comparative analysis, we have identified a symmetry between these two cases, and have systematically extracted the primitive components of all these techniques. We have implemented all of these constructs in λ_{SFCC}. From the route of its construction, we think it is very likely that our generalization will provide an acceptable level of risk mitigation in the presence of unrestricted FCC.

More subtly, we think the techniques of λ_{SFCC} will lead to strictly more reliable systems than are currently in the mainstream. While resources are handled equally reliably, the strictly more expressive system will allow the abstraction—and thereby encapsulation—of strictly more code patterns. This abstractive property will reduce the amount of tedious and error-prone boilerplate code. In particular, we can eliminate boilerplate resource-management code.

Even with these benefits, an observer might still believe that existing languages are "good enough" in that they sufficiently cover the range of programs that need to be expressed. To dispel this idea, we have implemented a wide range of additional control structures not universally present–a few universally not present—using the FCC features of λ_{SFCC}. Each of these control structures have practical applications; even during his relatively short programming experience, this author has wished for a few of these structures. The range we presented was selected in no systematic way: essentially off the top of the author's head. It is therefore unlikely that the range of applications presented here is in any way comprehensive. There can be little doubt then that there are applications best expressed with first-class control, and so any language without the feature is not "good enough".

A critic might wonder why we have not examined existing programs for the presence of FCC techniques. If FCC is not commonly used, then why should we increase the complexity of our interpreters and compilers to cope with the additional features? Such an exercise is analogous to examining Cobol programs for uses of higher-order functions. The continued success of the Lisp and functional programming communities can only be attributed to the power of higher-order functions, even though Cobol gives no motivation for them. Current practice is not an indicator of what might be beneficial. Our argument relies on theoretical utility, with the expectation that what is useful in theory will find a use in practice, as has happened so often in the past.

TODO macro-expressibility is not really that great, we'd like functional expressiblity. The trick in many cases is passing arguments unevaluated. If we can do this, there's no problem, and even if we can't, we can protect the argument from evaluation by wrapping it in a thunk before passing.

10.2 Related Work

A history of continuations as a means of investigating the formal properties of languages is given in Reynolds (1993). Since they appear in several areas of research, they were independently discovered on several occasions.

First-class control operators as we might recognize them begin with the **J** operator in Landin (1966). This had the distinction of capturing a continuation which could be used multiple times. Though it was an ad-hoc introduction meant to model goto statements, it is the ancestor of all of today's FCC operators, including exception handling and call/cc.

After this, variations on FCC operators abounded, especially delimited control operators. The clearest treatment of first-order delimited control is given in Felleisen (1988), on which we have based much of our formalism. Higher-order delimited control is examined by Gunter et al. (1995) in the context of ML; their system allows the introduction of new prompts at runtime, and provides a type system for this technique. Throw and catch are given in Sussman and Steele (1975), and their generalization to fcontrol in Sitaram (1993). Since this explosion of operators, work has been done examining the relationships between these operators (Sitaram and Felleisen (1990a), Filinski (1994), Kiselyov et al. (2006), Herman (2007)). In this area, Dybvig et al. (2007) is

most important for our paper, as it unifies nearly all these control operators short of try-catch and fcontrol.

Work in bringing security to call/cc was done in Haynes and Friedman (1987), which led to the development of dynamic-wind. A detailed analysis of the shortcomings of exception handling as regards performing cleanup is given in Weimer and Necula (2008). As far as this author can find, we are the first to apply these same security considerations to delimited control. Recently, work has been done in the efficient implementation of FCC. Efficient implementations are given for the dynamically-typed Racket is given in Flatt et al. (2007b), and for the statically-typed Scala in Rompf et al. (2009).

10.3 Further Work

Although this paper presents a strong theoretical case for the possibility of reliable first-class control, there is still much work left to be done bridging these results into production programming. In particular, it would be best to gain extensive experience with λ_{SFCC} in real programming tasks. We also expect a serious education campaign would be required to acquaint programmers with these relatively alien techniques. Beyond these, there are also interesting theoretical questions we did not have time here to treat.

The system we presented here allowed for only one reduction strategy, but we believe this is not an inherent limitation for the system. Instead, it would be interesting to reformulate our dynamics so that we can take full advantage of compatible closure (Barendregt (1992)).

Since our system was derived from typed systems (Dybvig et al. (2007), Gunter et al. (1995)), we believe it would be possible to assign meaningful types to the terms in λ_{SFCC}. The difference for our system is that we allow for a handler to be installed at the delimiting operator. Whereas in the aforementioned type systems, the type of prompts involved only one type variable, for the type expected by the continuation at the point the delimiter is installed, λ_{SFCC} will likely require a second type variable parameterizing the type expected by the handler then installed.

We also find it likely that a type system may need to impose restrictions on the features in our operators. We mentioned above that the $\#_{\blacksquare}$ delimiter may be difficult to assign a type, but if there is a single, set handler installed by the runtime system with $\#_{\blacksquare}$, then the question of its type may be made trivial. In all likelihood, the operators we have presented are too low-level to be of direct use to users, but will form a framework for implementors to provide slightly higher-level concepts. Thus, we expect reasonable restrictions on the available features of λ_{SFCC} to be prescribed anyway.

Finally, there are clear questions concerning optimizing compilers, given these new techniques. Some uses of first-class control, e.g. our subroutines in §9.3, are functionally equivalent to a program without FCC, but which instead uses simple jumps. Further, since subcontinuations are essentially functions, albeit created at runtime, it may be profitable to compile these into real functions at runtime, making use of just0in-time compilation techniques.

References

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition.

Alexandrescu, A. (2010). *The D Programming Language*. Addison-Wesley Professional, 1st edition.

Barendregt, H. P. (1992). Lambda calculi with types. In Abramsky, S., Gabbay, D. M., and Maibaum, S. E., editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., New York, NY, USA.

Barker, C. (2004). Continuations in Natural Language. In Thielecke, H., editor, *Proceedings of the Fourth ACM SIGPLAN Continuations Workshop (CW'04).*, Birmingham, UK.

Baugh, J. P. (2010). *Go Programming*. CreateSpace, Paramount, CA.

Danvy, O. and Filinski, A. (1990). Abstracting control. In *In Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160. ACM Press.

Dybvig, R. K., Jones, S. L. P., and Sabry, A. (2007). A monadic framework for delimited continuations. *J. Funct. Program.*, 17(6):687–730.

Felleisen, M. (1988). 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. ACM.

Felleisen, M. (1990). On the expressive power of programming languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag.

Filinski, A. (1994). Representing monads. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457. ACM Press.

Flatt, M. (2015). The Racket Reference. http://docs.racket-lang.org/reference/index.html. Accessed: 2015-6-12.

Flatt, M., Yu, G., Findler, R. B., and Felleisen, M. (2007a). Adding delimited and composable control to a production programming environment. *SIGPLAN Not.*, 42(9):165–176.

Flatt, M., Yu, G., Findler, R. B., and Felleisen, M. (2007b). Adding delimited and composable control to a production programming environment. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 165–176, New York, NY, USA. ACM.

Foundation, P. S. (2015). Python language reference, version 2.7. http://www.python.org/.

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

Gosling, J., Joy, B., Steele, Jr., G. L., Bracha, G., and Buckley, A. (2013). *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition.

Gunter, C. A., Rémy, D., and Riecke, J. G. (1995). 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. ACM.

Haynes, C. T. and Friedman, D. P. (1987). Embedding continuations in procedural objects. *ACM Trans. Program. Lang. Syst.*, 9(4):582–598.

Herman, D. (2007). Functional pearl: The great escape: Or, how to jump the border without getting caught.

Hieb, R. and Dybvig, R. K. (1990). Continuations and concurrency. *SIGPLAN Not.*, 25(3):128–136.

Jones, S. P. (2002). 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.

Kiselyov, O., Shan, C.-c., and Sabry, A. (2006). Delimited dynamic binding. *SIGPLAN Not.*, 41(9):26–37.

Landin, P. J. (1966). The next 700 programming languages. *Commun. ACM*, 9(3):157–166.

Launchbury, J. and Peyton Jones, S. (1995). State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341.

Mars, D. (2015). Exception safety. http://dlang.org/exception-safe.html. Accessed: 2015-07-19.

Ploeg, A. v. d. and Kiselyov, O. (2014). 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. ACM.

Ramsdell, J. D. (1992). An operational semantics for scheme. *Lisp Symb. Comput.*, 5(2):6–10.

Reynolds, J. C. (1993). The Discoveries of Continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248.

Rompf, T., Maier, I., and Odersky, M. (2009). Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. *SIGPLAN Not.*, 44(9):317–328.

Sitaram, D. (1993). Handling control. In *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 147–155. ACM Press.

Sitaram, D. and Felleisen, M. (1990a). Control delimiters and their hierarchies. *LISP and Symbolic Computation*, 3(1):67–99.

Sitaram, D. and Felleisen, M. (1990b). 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.

Stroustrup, B. (1994). *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Stroustrup, B. (2000). *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition.

Sussman, G. J. and Steele, Jr., G. L. (1975). An interpreter for extended lambda calculus. Technical report, Massachusetts Institute of Technology Articifical Intelligence Laboratory, Cambridge, MA, USA.

Swierstra, W. (2008). Data types à la carte. *J. Funct. Program.*, 18(4):423–436.

Thomas, D., Fowler, C., and Hunt, A. (2009). *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 3rd edition.

van Rossum, G. and Coghlan, N. (2011). PEP 343: The "with" Statement. http://legacy.python.org/dev/peps/pep-0343/.

Wadler, P. (1989). Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA. ACM.

Weimer, W. and Necula, G. C. (2008). Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, 30(2).

Wiltamuth, S. and Hejlsberg, A. (2003). C# Language Specification. Technical report, Microsoft.