PROPER TAIL CALLS FROM FIRST-CLASS CONTINUATIONS IN JAVASCRIPT

by

Caner Derici

Undergraduate Program in Computer Science Department

Istanbul Bilgi University

2010

PROPER TAIL CALLS FROM FIRST-CLASS CONTINUATIONS IN JAVASCRIPT

APPROVED BY:

Chris Stephenson .....
(Thesis Supervisor)

Assoc. Prof. Shriram Krishnamurthi .....

Assoc Prof. Alpaslan Parlakçı .....

Matthew Edwards .....

DATE OF APPROVAL: Day.Month.Year

ACKNOWLEDGEMENTS

My original dept is to Chris Stephenson, who first taught me Scheme when I came here at Bilgi. However, along the way, Scheme was very little contribution to those marvelous tons of knowledge that I learned from him. Chris has been wise and hippy, understanding and difficult, and has consistently challenged me over the years. He made me the computer scientist that I am today, for which no thanks are enough.

In between, I have been assisted by numerous people. Remzi Emre Başar listened me for all the time except mornings, noons, afternoons, nights... He linearized my (originally not) train of thoughts and consistently expected and obtained better from me than I myself did. Since from the start, Shriram Krishnamurthi helped me with his insightful thoughts and advices that made my thinking clear and precise. Despite a 11.000 kilometers, he helped me a lot to improve myself a one step further in my professional career. Danny Yoo and Ethan Cecchetti have helped improve this study by their implementations. Danny has developed Moby itself, and helped me all the time by checking all codes that I produced and gave me a lot of insights and advices about implementation issues. Ethan studied on and implemented compiler-side transformation. Once there, my friends and instructors have been wonderful resources and supports: Ayşe Karaca, Elif Pınar Hacıbeyoğlu, Boran Puhalıoğlu, Bahar Beyaznar, Olcay Demirkesen, Ercan Muşkara, Seda Albayrak, Melis Yüksel, Yiğit Yiğitbaşı, Ceyhun Aytekin, Ruhan Alpaydın, Dicle Öztürk, Elena Battini Sönmez helped give me a wider perspective and fresh ideas as well as a valuable support whenever I got stuck with a professional or personal issue. My instructors Alpaslan Parlakçı and Matthew Edwards have helped improve this dissertation.

Finally, my greatest dept is to my family, Muharrem Derici, Ayşe Derici, Özlem Derici and Rüştü Derici. They have supported me all my life in any way that they can, and encouraged me at every step as I have pursued my indefinite wonders.

ABSTRACT

PROPER TAIL CALLS FROM FIRST-CLASS CONTINUATIONS IN JAVASCRIPT

Implementing proper tail calls can pose a challenge if a target machine makes no provisions for accessing and re-installing a run-time stack. Classical techniques proposed over the years often have little significance, mainly due to limited characteristics, such as optimizing only the recursive tail calls. As a result of this restrictions, programmers are often forced to write unnatural programs in which they can access and manipulate data only with imperative constructs. This leads to critically unnatural program structures which are difficult to read, debug or maintain over long time periods.

This study provides a proper tail calling model from most functional programming languages without previous short comings by using the exception handling mechanism of Javascript to achieve the tail calling behavior with first-class continuations, while refraining from modification of the interpreter. For practical support, we embedded our model into Moby Compiler, which is designed as a compiler from Advanced Student Language of PLT-Racket with the *world* library to Javascript for both web browsers and mobile platforms.

ÖZET

JAVASCRIPT DİLİNDE BİRİNCİ-SINIF UZANTILARDAN DÜZGÜN SON ÇAĞRILAR

Düzgün son çağrıların gerçeklenmesi, hedef makinanın uygulama çalışma zamanındaki yığının programlarca erişimine ve yenilenmesine dair hiçbir yardımının olmadığı hallerde büyük zorluklar içerir. Yıllar içerisinde sunulan çözümler, yalnızca son çağrı tekrarlamalarının optimizasyonunu yapmak gibi sınırlı karakteristiklere sahip olduklarından, problem ancak sınırlı alanlarda çözümlenebilmiştir. Bunun sonucu olarak programcılar verileri işlemede sıklıkla sadece emir yapılarını kullanarak, verilere göre doğal olmayan programlar üretmek zorunda kalmışlardır. Bu da uzun vadede geniş yazılımların idare edilmesi ve geliştirilmesini hayli zor kılmıştır.

Bu çalışma ile, dilin işleyicisinin değişikliğinden kaçınarak, Javascript dilinin *exception handling* mekanizması ve birinci-sınıf *uzantılar* kullanılarak büyük bir yetersizlik olmadan son çağrı davranışını elde etmek için bir düzgün son çağrı modeli sunuyoruz. Bunu desteklemek için bu modelimizi PLT-Racket dilinin bir alt kümesi olan Advanced Student Dili (ASL) ve *world* kütüphanesindan, ağ tarayıcıları ve taşınabilir platformlar için Javascript kodu üretmek üzere tasarlanmış olan Moby Derleyicisine oturttuk.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

κ_i	The i^{th} continuation
λ	Binding literal from Scheme language: **lambda**
!	The Factorial function

HTDP	How To Design Programs
CPS	Continuation Passing Style
call/cc	call-with-current-continuation
ASL	Advanced Student Language of PLT-Racket
IDE	Integrated Development Environment
GC	Garbage Collection
ANF	Administrative Normal Form

1. INTRODUCTION

Program structuring and reusable modularity gains importance as software becomes more complex. A well-structured program is typically straightforward to write and debug, in addition to being an elegant reusable abstraction, as it provides a set of reutilization modules. In that sense, many programmers agree that conventional programming languages bring conceptual limits to the modularization of programs, while functional languages push those limits back. As being a mathematical oriented paradigm, functional programming principally provides exactly the same equational reasoning with algebra, which is a compulsory aspect of mathematics for most students in high-school. Due to its less (even completely not) "side-effected" perspective, functional programming tends to increase referential transparency relation as much as possible, which enables a much greater use of equational reasoning, therefore it results in more understanding, is easier to write and maintain complex softwares (Hughes 1989) (Krishnamurthi 2007).

Among other important features of functional programming languages, such as higher-order functions and lazy evaluation, one of the most eligible and powerful notions of functional languages is proper tail calls. As Chapter 3 explains, proper tail calling is a notion of the semantics of space usage of a particular programming language. In other words, tail calls in a program written in a programming language with proper tail calls do not place any burden on the run-time stack. Thus, for example, iterations achieved by recursion via tail calls (i.e. tail recursion) runs on a constant run-time stack space, as does conventional looping constructs such as 'for' and 'while'. Examples can be extended to many different perspectives such as mutual recursion, Continuation Passing Style (CPS), etc. As an insightful effect, proper tail calls equip programmers with the capability of matching the organization of functions to that of the data (Krishnamurthi 2007) (Felleisen, Findler, and Flatt 2009).

Since the idea of tail calls was first introduced by Steele and Sussman in the mid 1970s, implementation of the Scheme interpreter for lambda calculus, several currently

well-known techniques were developed in an attempt to implement proper tail calling for conventional programming languages such as C, Java or Javascript (Sussman and Steele 1975).

In 1990, Tarditi et al. tried to compile Standard ML to C without compromising on proper tail calls. Instead of a direct compilation, a continuation-passing style λ-calculus intermediate language has been used to generate C codes. Since the runtime stacks of CPS formed programs are explicit, tail calls can be easily captured from a CPS formed program, and turned into jump instructions using **setjmp** and **longjmp** functions. The problematic side of this technique however, is that the execution speed of generated code is slower than the original code about a factor of two. Besides, not all languages have the unconditional jump functions of **setjmp** or **longjmp**. Most importantly, this study proposes a well-known technique called *trampolines*. While trampolines ensure the constant stack space consumption automatically, the general problem of them is that they are expensive, in the sense that they can cause a two-three orders-of-magnitude slowdown. Additionally, it is not limited only to code that contains tail calls, trampolines impact all codes (Tarditi, Lee, and Acharya 1990).

An improved version of trampolining technique was used in 2001, by Schinz and Odersky, to eliminate tail calls on Java Virtual Machine. In addition to the generic trampoline technique, this technique used a numeric value to keep track of tail calls that had been made. A trampoline is used only after a certain number of tail calls have been made. However, this improved version of trampolining technique is not effective. Firstly, this technique achieves constant space consumption by ceasing both code size and execution time. Secondly, choosing the right TCL intuitively, or with trial and error for each system are not acceptable solutions for practical use. Thirdly, Schinz and Odersky use visitors to simulate the algebraic data types. A possible heavy usage of visitors would obviously pressurize the stack. To summarize, this technique apparently requires a lot of optimizations (e.g. inlining for visitors) to be useful in practice (Gamma, Helm, Johnson, and Vlissides 1994) (Schinz and Odersky 2001).

In 1994, Baker developed another famous technique called "Cheney on M.T.A.", which has been used to compile Scheme to C language. Baker's technique requires programs to be converted into CPS form. By this way, internal anonymous functions can be transformed into individual functions. The problem is however, since none of the functions actually **return** in CPS form, the stack eventually becomes full. Baker turned all stack allocations to use C's stack allocation mechanism. Consequently, passing a latest continuation closure to a copying garbage collector effectively makes the stack the youngest generation of a generational garbage collector. The main problem of this technique is that it requires an entire program transformation. The entire program (including the required modules) has to be transformed into CPS form even if it does not contain any tail call (Baker 1994).

Another confrontation of the problem of implementing proper tail calls on conventional programming languages was engaged in 2006, by Loitsch and Serrano. In this study, a compiler from Scheme to Javascript had been developed. Surprisingly, the resulting compiler had no support for tail calls, instead, it only transformed very simple and common recursive tail calls to the looping constructs of the target language, such as 'for' and 'while'. The reason for leaving proper tail calls unimplemented, is that the performance penalties which well-known techniques generate are completely unacceptable (Loitsch and Serrano 2006).

In the meantime, the functional programming community made an advance for HTTP (i.e. statelessness), which could lead to a continuation-based understanding of web applications. When a traditional server receives a request from a client, it prepares a response, dispatches and closes the communication; this means the program has to be terminated. This further implies that any subsequent computation has to be continued by another program. Functional programmers claim that this is a *continuation*. Thus, given the existence of first-class continuations, any continuous web application could be executed on HTTP or any other stateless protocol, by capturing and storing a continuation on the server and then applying it to subsequent requests from its client (McCarthy 2009) (Pettyjohn, Clements, Marshall, Krishnamurthi, and Felleisen 2005).

This brings me to my thesis:

> *First-class continuation models in imperative programming languages allow for proper tail calling behavior without any support from an underlying machine.*

In support of this thesis, I specify a program transformation based on one from Pettjohn et al. which presents a model for first-class continuations through stack inspection mechanisms. The idea is to translate Scheme programs with call/cc into a language with a generalized stack inspection mechanism. In one of the practical applications, they show how exception handlers and exception throws can collaborate to simulate continuation marks, which is MzScheme's novel abstraction of all mechanisms for manipulating the stack in some form or another (Pettyjohn, Clements, Marshall, Krishnamurthi, and Felleisen 2005) (Flatt 2005).

In addition, neither Pettyjohn et al. nor McCarthy describe how to apply a continuation, even though they provide a novel solution on how to model them. The transformation described in this thesis overcomes this problem by annotating the continuation semantics with additional capabilities, such as replacing a continuation with a previously captured one.

1.0.1. The Structure of this Thesis

In this thesis, a program transformation is described, which will achieve proper tail calling behavior. Chapter 2 explains what it is like to be a programmer from the HTDP perspective, and then describes from a mathematical point of view, how functional programs are similar to the mathematical structure of data along with example data and function definitions as well as their imperative counterparts. Chapter 3 describes the notion of tail calls, tail recursion and proper tail calling with examples in detail. In chapter 4, the function of compilers and the rationale behind them are explained, followed by a discussion of the main problems of compiling functional languages into imperative languages. This includes both first-class functions and lexical closures, as well as scoping and the free variable capture problem. Chapter 5 explains

the continuations, their usage on web and the *continuation-passing style*, as well as a call/cc example describing the transfer of control both in and out of a computation in detail. In the subsequent four chapters, solution, implementation and theory and practice are discussed:

- Chapter 6 explains the inner structure, the idea and the main characteristics of Moby Compiler comparative to previous and ongoing works.
- Chapter 7 presents the program transformation in order to achieve the desired behavior.
- Chapter 8 describes the principles behind support infrastructure on which the transformed program works, along with some implementation problems and their solutions.

This study, along with quantitative measurements of the time and space consumption of generated code, as well as the advantages and drawbacks compared to the other methods presented in chapter 9. A conclusion is made in chapter 10.

2. PHILOSOPHY OF HTDP

To some, programmers are very much like artists, and thus programming itself may be considered an act of art. A composer starts with an idea, a feeling. Then he carefully investigates and structures his idea, tries to understand it in a very fundamental level. When the composer fully conceives his feelings, he starts to express himself employing the basic tools and techniques he possesses, such as a musical instrument. In the end, small portions of melodies constitute a large and detailed symphony, which may perfectly express the mental image of the musician.

Just like an artist, a programmer tries to investigate, understand and structure his/her own idea, then express his mental image with a computer program, which is very much like a symphony, consisting of small programs running in harmony with each other. Therefore, computer programs are like carefully designed and structured compositions, as well as the act of programming is an art of designing, composing and organizing computational solutions.

In order to program a computer, a programmer needs to interact with it. Since computers cannot understand complex expressions in English or in any other natural language, programmers use *programming languages* to interact with a computer. Learning programming languages means understanding how computers represent, express and manipulate data. This leads to the main idea of programming, as well as the philosophy of HTDP (Felleisen, Findler, Flatt, and Krishnamurthi 2001):

Programs follow data

According to HTDP there are two concepts in the essence of programming at the lowest level:

- relating one quantity to another quantity, and
- evaluating a relationship by substituting values for names.

Provided this philosophy, HTDP proposes the idea that programming is basically another form of algebra, which we already know from high school. In order to program a computer, a programmer only needs to understand its language, its way of describing and operating data. Once the understanding of a computer has been achieved, there is only one step left for a programmer to program: modeling, the act of denoting real objects to a computer. After carefully representing data, a programmer follows the design recipe, also proposed by HTDP, to build the program.

The idea of conceiving a machine's way of expressing itself and modeling data to create a computer program leads to an essential understanding that a computer program is basically a set of instructions for a computer to manipulate data. Therefore a programmer needs to understand the nature of data, which he/she uses in his/her model, in order to build a computer program that manipulates data. Creating a computer program without understanding the natural structure of data is like trying to paint without knowing which type of paint to use to express your mental image, an unnatural approach which will create an absurdity at all levels.

"In order to use a computer properly, we need to understand the structural relationships present within data, as well as the basic techniques for representing and manipulating such structure within a computer." Donald E. Knuth (Knuth 1997)

Computers operate on binary arithmetic. Since it is extremely inefficient to write the set of instructions (i.e. programs) in binary, there should be an abstraction. All of the data types we use in computing (numbers, strings, programs etc.) have some form of abstraction for the binary system and all of them can be considered as data at appropriate abstraction levels, which depend on the programmer's imagination. (Mano and Kime 2001) The example in Figure 2.1 points out that there can be different abstractions of similar objects. In one case, a programmer represents a circle with a number in his/her model, while in the other case, programmer models the circle with a collection of informations related to a circular object; radius and speed.

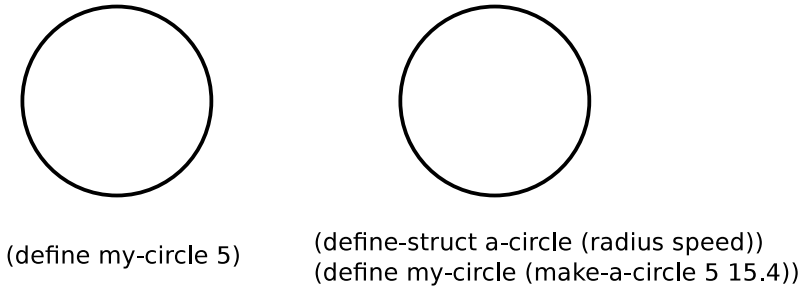

Figure 2.1. Two Different Modeling of a Circular Object

The above example in Figure 2.1 basically demonstrates the essence of data representation, however we should employ a more detailed approach in order to achieve a deeper understanding about the nature of data, by investigating how to define them.

2.1. Recursive Data

There is a particular kind of data, namely *recursive data*, which is a data type that may contain other values of the same type. This section provides some examples of recursively defined data to reach a better understanding of the "programs follow data" principle (Aczel 1977).

2.1.1. Natural Numbers

The general definition of natural numbers is:

$a_i = a_{i-1} + 1$ where $a_1 = 0$ (Bancerek 2003)

The essential structure of the definition of natural numbers demonstrates that we need to know only the initial value and the successor function in order to model them with our computational tools, such as the Haskell data type:

```
data Nat = Zero | Succ Nat
```

Which indicates a natural number is either:

1. zero, or
2. a successor of another natural number.

Using this definition, one can easily construct a conditional recursive program in Scheme which has the purpose of finding the value of a particular natural number, as follows:

```
(define (nthNatural n)
   (if
     (= n 1) 0
     (+ 1 (nthNatural (- n 1)))))
```

2.1.2. Structural Recursion : Factorial

Let us show another kind of recursively defined data. The formal definition of a factorial of a positive integer is:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! * n & \text{if } n > 0, \end{cases}$$

By knowing the mathematical principle of factorials, one can easily build a recursive Scheme program to find a factorial of a particular natural number:

```
(define (factorial n)
  (cond
   ((= n 0) 1)
   (else (* n (factorial (- n 1))))))
```

Observing the similarity between the algebraic mathematical definition and a recursively defined computer program, it is obvious that this kind of approach is definitely natural and clear in the sense of both writing and reading a computer program.

2.1.3. List Processing: Map

Our final example is a classical form of a recursive data type, a list. The definition of a list is:

```
data List a = Nil | Cons a (List a)
```

Which says a list of a is either:

1. null, or
2. a pair of an a's and a list of a's

Again, from the definition, one can easily construct a recursive computer program which can manipulate a list given to it:

```
;; map: ( a -> b ) list-of-a -> list-of-b
(define (map aFunction aList)
 (if
   (null? aList) null
   (cons (aFunction (first aList))
         (map aFunction (rest aList))))))
```

The examples in this section demonstrate the effectiveness of building computer programs considering the nature of data, which the program will express and manipulate. Constructing recursive programs to process naturally recursive data is easy, understandable and clear, as well as intuitive and normal.

However, there is an obvious problem about recursion, considering the space growth of the run-time stack. Although run-time stacks are multi-purpose stack data structures, they are generally used to keep track of memory locations for each active subroutine to transfer control to an appropriate function after finishing their execution. For any executed function, a *stack frame* (or an *activation record*) will be created to store data (such as its actual parameters, local variables, return address, etc.) for it

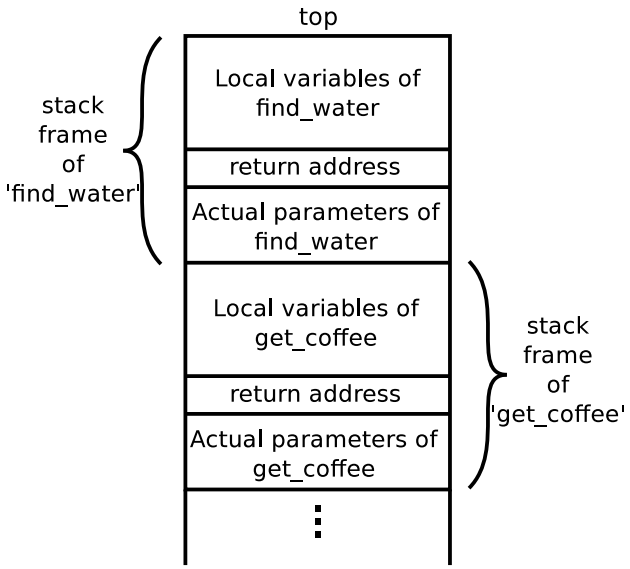

Figure 2.2. General Layout of a Run-time Stack

to use while and after execution. Since a recursive computation based on successive function calls, any recursively defined program goes into danger of overflowing the run-time stack. (Aho, Sethi, and Ullman 1986)

2.2. Natural Approach To Data: To Recurse or Not To Recurse

While recursion is a natural approach for processing inductively defined data types, programmers often choose to use classical iteration to preserve the program from the inefficiency in terms of space consumption which the recursion would naturally produce.

```
function fact(n) {
    var retValue = 1;

    for(i=1 ; i<=n ; i++) {
      retValue *= i;
    }

    return retValue;
}
```

The program above is an iterative factorial function written in Javascript language. In contrast to a recursively defined factorial function, iterative approach does not follow the mathematical definition of factorials. Building a program in this way requires extra effort for re-organizing the solution to use the looping constructs (such as 'for' and 'while'). As this re-organization is easy for a factorial function, it can be extremely difficult to use in some different cases (e.g. mutual recursion). In addition, by using an iterative approach on recursively defined data, a program loses its position as a natural, clear and intuitive program. Thus it becomes an unclear, hard to read or understand function (Felleisen, Findler, and Flatt 2009).

The iteratively defined program above runs in a constant stack space[1] , although it does not make any additional subroutine calls, and accumulates the knowledge in a local variable (retValue) while running. Therefore the iterative approach becomes the perfect choice for practical purposes (such as web or database programming).

This problem of the unnatural being chosen for efficiency has been solved by a marvelous idea, namely *Proper Tail Calls*, which utilizes the natural aspects of recursion with the run-time efficiency of iteration in terms of space consumption. In the next chapter, we shall continue with examining the notion of "calling a tail position function properly" in detail.

[1]Stack space is not proportional to input.

3. PROPER TAIL CALLING

Previous chapters investigated the advantages of recursive programs working on naturally recursive data. This chapter explains an essentially mathematical notion, an evaluation strategy, namely, *Proper Tail Calls*, which has been fundamentally forged from the reasoning of space consumption of a program evaluation. The main focus of proper tail calls is providing necessary infrastructure where programmers easily match data and organization of functions. Additionally, proper tail calls, by its nature, exhibits the *constant stack space behavior*, which exalts programs to even beyond being natural, to also being efficient.

3.1. Tail Calls

In order to understand *Proper Tail Calls*, it would be convenient to first introduce the notion of tail calls. The idea of tail calls was introduced by Steele and Sussman in the mid 1970s, via implementation of the Scheme interpreter for lambda calculus. Among other improvements brought to the field, in our scope, includes the most impressive focus of this study has been a huge development in the path to understand the true nature of function calls (Sussman and Steele 1975).

Shortly, a tail call is the last procedure call of an enclosing procedure. There is no computational work done between the termination of a tail call and a termination of a procedure that calls it. Thus, the resulting return value will be identical to the return value of its caller. This overview illustrates that a tail call itself is not a language property, but rather a notion that all useful[2] programming languages already have.

As section 2.2 mentions, without proper tail calls, programmers are forced to organize iterations over data to use only classical looping constructs such as 'for' and 'while'. However, in functional programming languages, programmers are not restricted

[2]By saying useful, we mean the programming languages in which one can build any kind of procedural abstraction.

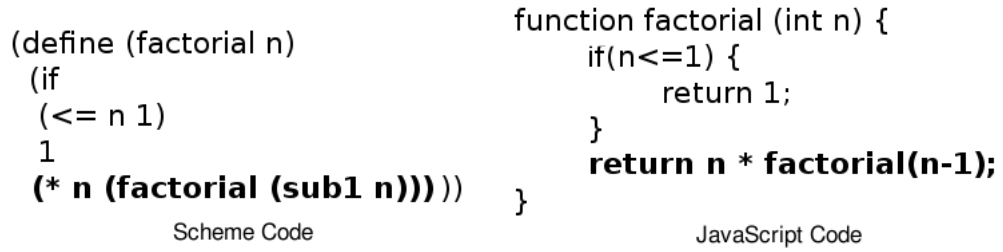

Figure 3.1. Tail Calls of Two Same Programs in Different Languages

in this way. The same computation can be achieved by recursive procedures. The philosophy, "programs follow data", provides simplicity and elegance to the manipulation of data which already has the same semantic structure with implementation. This is achieved by proper tail calls. More precisely, taking tail calls into account provides the necessary framework, on which programmers can approach data pure[3] functionally. Therefore, proper tail calls give the natural power to catch reasoning of data (even non-linear ones) directly over the organization of functions (Felleisen, Findler, and Flatt 2009).

3.2. Tail Recursion

The most common usage of proper tail calls is obviously in tail recursive programs.

For a procedure, to recurse means to make a procedure call to itself. In other words, all recursive programs make at least one procedure call to itself. Tail recursion is used to make these recursive calls via a tail calls.

Two programs in figure 3.2 illustrate the difference between normal recursion and tail recursion. The one in 'a' is properly recursive, while the one in 'b' is tail-recursive. The recursive calls in 'a' are not tail calls, because the '+' (addition) operation waits for results of an addition. In other words, there is more computation left between the termination of the fibonacci program and recursive calls. The one which is a tail call is actually addition, the one which decides the result of the main fibonnaci call. On the

[3]Without side effects.

```
(define (fibonacci n)
 (if
  (<= n 2)
  1
  (+ (fibonacci (- n 1))
     (fibonacci (- n 2)))))
```

-a-

```
(define (fibo n f s i)
 (if
  (<= n i)
  f
  (fibo n (+ f s) f (add1 i))))

(define (fibonacci n)
 (fibo n 1 1 2))
```

-b-

Figure 3.2. Fibonacci Program in 'a' Normal Recursive and 'b' Tail Recursive Forms

other hand, the program 'b' illustrates how a programmer might use proper tail calls for his/her own benefit. Obviously the tail call in 'b' (fibo function) is:

```
(fibo n (+ f s) f (add1 i))
```

At first glance, this call seems to help by eliminating the need for an additional stack frame for the addition operation, but the actual effect is even greater. The tail recursive Fibonacci is computationally equivalent[4] to a possible Fibonacci program written in Java using conventional looping constructs such as 'for' or 'while'. The actual power comes from the *Proper Tail Calls*, which will be studied detailedly in the next section.

3.3. Calling Tail-Position Functions Properly

The fact that there is no computational work done between the termination of a tail call and the termination of a procedure that calls it (as explained in section 3.1) indicates the redundancy to build two different stack frames for both the tail call and its enclosing procedure. Since both will have the same result, the only job the upper stack frame (which belongs to the enclosing procedure) has to do is pass the result coming from the lower frame to an upper frame belonging to whichever function starts this particular computation. This constitutes the main idea of *Proper Tail Calls*, which is basically to overwrite the stack frame of the caller function with the stack frame of

[4]In terms of space consumption.

the tail call.

Consider a small portion of a properly recursive Factorial program evaluation:

```
⋮
A
(factorial 5)
(* 5 (factorial 4))
(factorial 4)
⋮
```

Somewhere in the program flow, a procedure A makes the call *(factorial 5)* and started to wait for return value. Then *(factorial 5)* call *(* 5 (factorial 4))* and so on. When one of these continuing calls ever returns a value, the program will roll upwards and return results (as well as the control) one-by-one to each other. Here we may observe that the *(factorial 5)* only passes the return value of *(* 5 (factorial 4))* to A. It is basically an identity function.

Making tail calling proper, eliminates redundancy by overwriting the stack frame of the callee by the tail call. Therefore the evaluation of a properly recursive factorial program seems like the following, in an environment in which the tail calls are proper tail calls.

```
⋮
(* 5 (factorial 4))
(* 4 (factorial 3))
(* 3 (factorial 2))
⋮
```

In addition to its naturalness on (even non-linear) data, tail recursion also becomes the most efficient way of looping technique with the existence of the stack frame elimination mechanism. The idea is that saving the control context regarding an argument evaluation, rather than function calls (Felleisen, Findler, and Flatt 2009).

While evaluating a properly recursive function, control context is saved in run-time stack frames during the evaluation of the sub-expressions. That is to say, if we run **(factorial 3)** on the Scheme code in Figure 3.1[5] , the run-time evaluation of it will be:

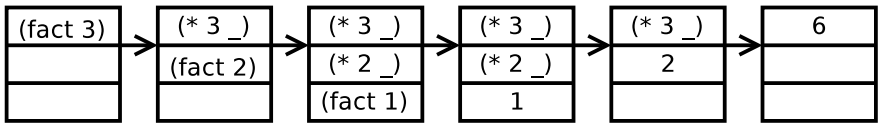

Figure 3.3. Run-time Behavior of Properly Recursive Factorial Program

```
(define (factorial n result)
 (if
  (<= n 1)
  result
  (factorial (sub1 n) (* n result))))
```

Figure 3.4. Tail Recursive Factorial Program

In contrast, the tail recursive factorial function in Figure 3.3 saves its control context on argument evaluation (i.e. the result is accumulated in the parameter 'result'). Therefore, the run-time evaluation will be:

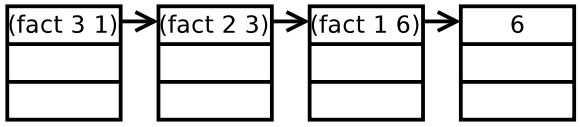

Figure 3.5. Run-time Behavior of a Tail Recursive Factorial Program

This illustrates that proper tail calls provide a constant stack space behavior, by saving the control context on argument evaluation, as in classical looping constructs, such as 'for' and 'while'. Using tail recursion instead of imperative looping provides an elegant and natural approach to recursive data, without sacrificing efficiency.

Section 3.1 explains that the notion that proper tail calls is not a language-specific property per se, but rather a semantical concept that all programming languages have. However, as it is mostly said to be (tail call optimization), proper tail calls are not an optimization, but an aspect of the semantics of the space usage of the language. Aside

[5]In an environment which has proper tail calls.

from this fact, proper tail calling can be achieved by optimizing the compiler to make language space-safe in the sense of tail calls. Some languages have this optimization (i.e. Scheme), therefore, proper tail calls, while others decide to compromise from this artifact (i.e. Java), forcing the programs to be unnatural and hard to read, which leads to a lot of problems regarding maintenance, testing and further development of the software in the long term.

4. COMPILERS

This chapter examines compilers and discusses difficulties and advantages of functional compilers which target imperative programming languages.

4.1. Compilation: Stair Is the Stairs

Considering theoretical computing, there is a method for proving problems are computationally unsolvable, namely *reducibility*. Reduction is a model of transforming a problem into another, such that the second problem can be used to solve the first one. A particular kind of reduction, namely *mapping reduction*, is a transformation such that the solution of the second problem is the solution of the first. Computer operation strongly relies on the mapping reducibility principle (Sipser 2006).

Chapter 2 explains that a computer program is basically a set of instructions for a computer to execute. In order for a computer to perform the operations expressed by a set of instructions, there are a lot of middle stages for applying the necessary arrangements on the program, such as lexical or semantic analysis and intermediate code generation. In other words, a program has to be translated from a high-level programming language into a low-level one, such that a computer can execute it. Translation is carried out by a couple middle stage programs, such as preprocessor, parser etc. This leads to an interesting idea that all programs are input data for other programs, except the lowest-level programs consisting of actual instructions represented as a memory word [6] . Even very low level assembly programs require an assembler to interpret program codes (Aho, Sethi, and Ullman 1986).

In order to prove that there is a mapping reducibility between two languages (or problems/questions), one has to show that a mapping function exists, which can map a solution of one language into the solution of another [7] . Thereby; compilers act as

[6] Chris Stephenson, March 2010

[7] Such a construction is obvious due to the *Turing-decidability* of computers.

a mapping function between a source and target language. Therefore; a theoretical or practical machine which can execute the programs (solve the problems) generated in a target language becomes capable of executing programs (solving problems) generated in source language as well.

4.2. Functional To Imperative Compilers

There are several kinds of compilers designed for functional programming languages which target imperative programming languages. Usually, these compilers are used to compile a program generated for a functional programming language to be executed by a computer. Since computers operate on a set of instructions executed in a sequential manner, the lowest-level step must be a program in an imperative programming language, no matter which type the intermediate languages are.

Another area of the functional-to-imperative compilers' usage is to run the programs written in one of the functional programming languages on a non-supportive environment, such as browsers (i.e. Mozilla Firefox).

4.2.1. Difficulties

Designers of compilers for functional programming languages which target imperative programming languages are often confronted with problems about modeling novel features of functional programming languages on an imperative target language. In that sense, the most common problematic features of functional programming languages are proper tail calling and first-class continuations, which will be elaborated on in chapter 5.

A classical difficulty arises when compiling/implementing functions as first-class values in imperative programming languages. Being a first-class value means that a particular kind of value (i.e. numbers) which can be supplied for a function as an actual parameter, returned by functions or stored in a data structure. Therefore, implementing functions as first-class values requires assigning all privileges of ordinarily first-class

values (such as numbers, strings, etc.) to functions. This can be very problematic for programming languages which maintain their lexical and local variable environments in a run-time stack, concisely in *stack-based programming languages*, such as C++. The problematic side of implementing first-class functions in stack-based programming languages can easily be described by the "Funarg Problem" (abbreviated for functional argument), which has an exact correspondence with the *free variable capture problem* in lambda calculus. In lambda calculus, the variable capture problem arises when plain substitution is performed rather than proper beta-reduction with renaming (Krishnamurthi 2007) (Moses 1970) (Barendregt 1985).

The funarg problem occurs when the body of a function makes a direct reference (e.g. not passing arguments) to identifiers not defined in the environment of the function call. There are two kinds of funarg problem: *upwards funarg problem* and *downwards funarg problem.*

4.2.1.1. Upwards Funarg Problem. The upwards funarg problem occurs when the calling function refers to a function's state after that function has returned. Consider a Scheme definition of a functional composition:

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

The 'compose' function produces a one argument function, which returns the result of applying f to the result of applying g to its argument x. If the 'compose' function stores the parameter variables 'f' and 'g' on the stack, when it returns the one argument function, the stack frame designated for 'compose' would be deallocated, therefore the values for 'f' and 'g' become irrecoverably lost. Which means if we run[8] :

```
(define my-composed-func (compose sqrt sqr))

(my-composed-func 5)
```

[8]... in a stack-based language

the 'my-composed-func' program would produce an undefined reference error, because the stack frame containing the real values of 'f' and 'g' had been lost when the function 'compose' was returned.

4.2.1.2. Downwards Funarg Problem. The Downwards Funarg Problem takes place when a function is executed outside the scope in which it has been defined. Consider the code snippet[9] :

```
function() {
  var temp = 1;
  var func = function() { return temp++;};
  alienFunc(func);
}
```

The function 'func' has been passed downwards to the 'alienFunc' function from its declaration scope. After this function returns, the stack frame containing the local variables 'temp' and 'func' will be deallocated from the stack. However, another activation record will be generated for the 'alienFunc' function. Because the stack frame of 'alienFunc' contains only a reference to 'func' (i.e. not also to 'temp'), any execution of 'func' will produce an error indicating that the variable 'temp' is undefined.

One can resolve the funarg problem via either forbidding such references or creating closures (Sandewall 1971).

> *"Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary"* Revised[6] Report on the Algorithmic Language Scheme (Sperber, Dybvig, Flatt, van Straaten, Findler, and Matthews).

4.2.1.3. Lexical Closures. As P.J. Landin defined and used in his *SECD* machine for evaluating expressions, a closure has:

[9] 'foo' is a one-argument function declared somewhere.

1. an *environment part* which is a list whose two items are:
   (a) an environment
   (b) an identifier or list of identifiers,
2. a *control part* which consists of a list whose sole item is an expression (Landin 1964).

Joel Moses improved Landin's idea by providing that closures are lambda expressions whose free variables have been bound in the lexical environment, resulting in a closed expression; which were then adopted by Steele and Sussman in the implementation of Scheme language (Moses 1970) (Sussman and Steele 1975).

Therefore, in a lexically (statically) scoped environment:

```
(my-composed-func 5)
```

This will produce 5 as a result, because it has been defined as a closure, which means that when 'my-composed-func' is applied to 5, even the definitions of 'f' and 'g' are deallocated from the run-time stack, their declarations exist in the environment on which the generated closure has been closed.

5. CONTINUATIONS

In the early history of programming languages, the ability of Algol 60 to jump out of blocks and even procedure bodies, led to the realization that the representation of a label must include a reference to an environment (Backus, Bauer, Green, Katz, McCarthy, Perlis, Rutishauser, Samelson, Vauquois, Wegstein, van Wijngaarden, and Woodger 1963) (Aho, Sethi, and Ullman 1986).

> *"... in order to specify a transfer of control we must in general supply both the static description of the destination ... and a dynamic description of its environment, the stack reference. This set ... together define what we call a program point*[10] *" Peter Naur* (Naur 1963).

A more sophisticated understanding was that the return addresses of procedures could be processed as procedure parameters. Thus, Edsger W. Dijkstra stated:

> *"We use the name "parameters" for all the information that is presented to the subroutine when it is called in by the main program; function arguments, if any, are therefore parameters. The data grouped under the term "link" are also considered as parameters; the link comprises all the data necessary for the continuation of the main program when the subroutine has been completed"* (Dijkstra 1960).

Another occurrence of continuations was in Peter Landin's SECD machine, a state-transition interpreter for a language of applicative expressions that was syntactically similar to the untyped lambda calculus but used a call-by-value order of evaluation. A state in the SECD machine is a four-tuple of following components:

- **S** : a stack for tracking values $\hat{V}$ locally
- **E** : an environment, associating variables X with values $\hat{V}$
- **C** : a control string for directing the execution
- **D** : a "dump" representing a saved state: $\langle \hat{S}, \hat{E}, \hat{C}, \hat{D} \rangle$

[10] In that time, program point was a representation of a continuation

In terms of practical implementation, the S part corresponds to local data for a particular procedure while D encodes the remaining computation by informally corresponding the rest of the stack. Therefore, it was another representation of continuations (Felleisen, Findler, and Flatt 2009) (Reynolds 1993).

At present, beside the two kinds of denotational context, which are: *naming context (i.e. environment)* and *state context (i.e. store)*, there is a third general context which is also important to the evaluation of programming language expressions, namely *control context*. While environments lead to a better understanding about the naming and scoping problems and stores help to investigate issues involving mutation, control contexts provide the capability of observing, expressing and manipulating the normal flow of a program's execution (Krishnamurthi 2007) (Turbak and Gifford 2008).

Evaluation contexts can denote control contexts in an operational framework. Consider the following expression:

```
(+ (* (factorial 5) 2) (factorial 5))
```

Since both *(factorial 5)* calls are evaluated in the same environment and store the only difference between them, it is how their results are used by the rest of the program (i.e. their results will always be the same). Informally, control contexts basically describe the rest of the computation that remains to be done after the expression has been evaluated.

The control context of the first occurrence of *(factorial 5)* is:

```
(+ (* □ 2) (factorial 5))
```

while the control context of the second occurrence of *(factorial 5)*:

```
(+ (* 120 2) □)
```

Note that the control context also shows in which order the expressions are evaluated (i.e. left-to-right in this case).

Denotational descriptions which do not have the notion of the rest of the computation, in other words the *denotational semantics* without an explicit control model (i.e. *direct semantics*) can not properly handle interruptions of the normal flow of a computer program. Such deficiencies can easily be manifested by the constructs of various languages that suspends a particular computation and transfers the control to another computation, such as: *return*, *break* and *continue* statements of JAVA or C; *exception handling mechanisms* of ML or COMMON LISP; *backtracing* of PROLOG and *goto* statements of various programming languages supporting *unrestricted jumps*. Mathematical entities which model such control transfers denotationally called *continuations*. They are denotational projections of evaluation contexts of operational semantics (Turbak and Gifford 2008).

5.1. Continuations As First-Class Procedures

In an operational framework, many sophisticated control behaviors can be achieved with continuations using only first-class functions, that is, procedural representations of continuations, namely functional continuations. A functional continuation of an expression E is basically a unary procedure that takes the result of E and performs the rest of the computation. For example, consider the first occurrence of the (factorial 5) expression in:

```
(+ (* (factorial 5) 2) (factorial 5))
```

The continuation of the first (factorial 5) can be represented with a procedure as follows:

```
(λ (v1) (+ (* v1 2) (factorial 5)))
```

as the continuation of the second (factorial 5) can be represented as:

```
(λ (v2) (+ 240 v2))
```

Given that the continuations represented explicitly, every computation can be expressed as a set of steps which are iterated by two state variables (Felleisen, Findler, and Flatt 2009) (Turbak and Gifford 2008).

- The expression currently being evaluated
- The continuation of the current expression

Therefore, we can represent the evaluation of:

```
(+ (* 4 3) (- 10 2))
```

as shown in Table 5.1:

current expression	**continuation**
(+ (* (- 10 2) 3) 5)	κ_{top}
(* (- 10 2) 3)	$\kappa_1 = (\lambda\ (v1)\ (\kappa_{top}\ (+\ v1\ 5)))$
(- 10 2)	$\kappa_2 = (\lambda\ (v2)\ (\kappa_1\ (*\ v2\ 3)))$
8	κ_2
(* 8 3)	κ_1
24	κ_1
(+ 24 5)	κ_{top}
29	κ_{top}

Table 5.1. Iteration table of continuation-based computation of an expression

The expression is being evaluated with respect to κ_{top}, the continuation of this whole computation, in other words, the rest of whichever computation started this one. In this case, we can assume that κ_{top} is an identity procedure.

As a classical example, a summary of the iteration of factorial function from Figure 3.1 is shown in Table 5.2.

current expression	**continuation**
(factorial 3)	κ_{top}
(factorial 2)	$\kappa_1 = (\lambda\ (v1)\ (\kappa_{top}\ (*\ 3\ v1)))$
(factorial 1)	$\kappa_2 = (\lambda\ (v2)\ (\kappa_1\ (*\ 2\ v2)))$
(* 2 1)	κ_1
(* 3 2)	κ_{top}
6	κ_{top}

Table 5.2. Iteration table for a properly recursive factorial evaluation

This representation displays the stack-like nature of continuations. It can be considered that calling a procedure creates a new continuation that corresponds to the frame that is pushed onto the top of the call stack. On the other hand, invoking a continuation corresponds to popping the top frame off the call stack and returning control to code in the calling procedure, whose frame becomes the new top-of-stack frame.

As a comparative example, a summary of the iteration of tail-recursive factorial function, from Figure 3.4 is shown in Table 5.3.

current expression	**continuation**
(factorial 3 1)	κ_{top}
(factorial 2 3)	κ_{top}
(factorial 1 6)	κ_{top}
6	κ_{top}

Table 5.3. Iteration table for a tail-recursive factorial evaluation

Note that any implementation of the tail-recursive factorial computation need not push a new invocation frame onto the function-call stack for the subsequent factorial calls in tail context, since it does not involve any new continuations. Thus, the run-time stack space can remain constant during an evaluation. In contrast, invocations of the recursive factorial do involve new continuations, therefore any implementation will push a new invocation frame onto the run-time stack, resulting the space required to

evaluate such a computation to be proportional to the growth of the input data.

5.2. CPS – Continuation-Passing Style

A program which takes its continuation explicitly as an extra parameter is said to be in *continuation-passing style (CPS)* form. Instead of a normal return, a program in CPS form invokes its continuation on its return value to resume the overall computation. For example, here is the CPS formed properly factorial function[11] :

```
(define (fact-cps n k)
  (if
    (<= n 1)
    (k 1)
    (fact-cps (sub1 n)
              (λ (v) (k (* n v))))))
```

With the existence of explicit continuations, in every step how the function will resume the computation can be hardcoded into the procedure call. As this example also clearly indicates, a CPS formed function is invoked with an extra parameter which is a functional continuation, some form of a representation of how its results will be processed after such results have been obtained by the function's computation. Informally, it says what to do next after the CPS formed function finishes its computation.

```
(fact-cps 4 (λ (v) v)) → 24


(fact-cps 4 (λ (v) (+ v v))) → 48
```

While this can be seem to have no benefits in practice, explicit continuations can be used to obtain very sophisticated control behaviors such as multiple-value-returning, nonlocal exits, error handling and backtracing etc. Detailed discussion of

[11]Observe that the use of explicit continuations turned a properly recursive program into a tail-recursive one, because any computation other than the recursive call has been automatically embedded into the continuation passed to the subsequent iteration.

such applications are obviously beyond the scope of this thesis. Another advantage of using explicit continuations is by turning all codes into CPS form, a compiler makes it unnecessary to use an explicit procedure-call stack since all the frames are implicit in the continuation (Krishnamurthi 2007) (Turbak and Gifford 2008).

5.3. A Pearl of Scheme: call/cc

Many programming languages adds operations such as **label**, **setjmp** or **longjmp** to reify some features to manipulate the control of the underlying language. Scheme has one: call/cc (call-with-current-continuation). Call/cc is a truly powerful reification which makes all the other control manipulation operators syntactic sugar, because all of them can be implemented using call/cc (Krishnamurthi 2007).

Call/cc is a unary function that takes a one argument function as its only parameter and gets the "snapshot" of the current control context as an object, then applies that object to the one argument function it has been given as its only parameter[12] (Friedman and Wand 1984) (Krishnamurthi 2007).

5.3.1. Get Out!

Early termination of a procedure and throwing an exception are both examples of escaping from a computation. This requires a computation to transfer the control to another computation, while abandoning the current context. Call/cc does the both. Consider the following expression:

```
(+ 2
   (call/cc
     (λ (k)
       (* 5
          (k 4)))))
```

[12] An unobvious reification here is Scheme not distinguishing a function application from continuation application by overloading the procedure application semantics.

As the call/cc captures the current continuation as a procedural object as follows:

```
k → (+ 2 □)
```

it immediately applies it to 4, resulting in 6.

Another example involving a model for exceptions using call/cc from PLAI:

```
(define (f n)
 (+ 10
   (* 5
     (call/cc
       (λ (esc)
        (/ 1 (if (zero? n)
                 (esc 1)
                 n)))))))
```

In this example, any occurrences of "division by zero" is handled by the continuation, such that, if the denominator is 0, captured continuation which is:

```
(λ (k)
   (+ 10 (* 5 k)))
```

applied immediately to 1, sneaked out from division by zero. Otherwise, the computation will be performed as if there is no call/cc at all, in other words, as if the body of the function is:

```
(define (f n)
 (+ 10
   (* 5
     (/ 1 n))))
```

Here, the continuation bound to *esc* can be considered as an *exception handler*, while invoking it considered as *throwing an exception*.

5.3.2. Get Back In!

Since the continuations captured by call/cc are first-class values (i.e. functions), one can easily store any captured continuation in a persistent data structure and jump from anywhere in the program (transfer the control) to the computation in which the continuation was originally captured.

Consider the following program:

```
(define return false)

(+ 1 (call/cc
       (λ (k)
         (set! return k)
         1)))
```

This program will capture the continuation:

```
(λ (v)
  (+ 1 v))
```

and binds it to an identifier called *return*. After this program terminates (i.e. after *return* has been modified), any invocation of *return* will get into the computation in which the continuation was captured [13] . Thus:

```
> (return 22)
23
```

[13]Invoking the continuation bound to *return* in the same context would result in an infinite loop since the captured continuation would also include that invocation.

5.3.3. A call/cc puzzle

Here is an example of how *call/cc* could be used to suspend and continue a particular computation. Consider the program:

```
;; generate-one-element-at-a-time :
;;                          (list-of-a) -> a OR 'fell-down
(define (generate-one-element-at-a-time lst)
   (define (control-state ret)
     (map
       (lambda (element)
         (set! ret (call/cc (lambda (resume-here)
                              (set! control-state resume-here)
                              (ret element)))))
       lst)
     (ret 'fell-down))

     (lambda ()
       (call/cc control-state)))

 (define aDigit
   (generate-one-element-at-a-time '(1 2 3 4 5)))
```

In this example, numbers from a list of numbers are returned one-by-one at every invocation. The key idea is, every time the loop is about to process another item from the list, two continuations are modified:

- While initially being the closure that iterates through all the elements of the list, after the first invocation, the *control-state* becomes a continuation which is captured at an intermediate step of *map*. This will be used to continue returning numbers from whichever number was on the line when the continuation has been captured.

- The continuation bound to *ret* is modified to return the element to whichever computation that has been applied to *control-state*.

Therefore, successive invocations of *aDigit* will return the elements (along with *fell-down*) one-by-one instead of a complete list.

```
> (aDigit)
1
> (aDigit)
2
> (aDigit)
3
⋮
> (aDigit)
fell-down
⋮
```

5.4. Continuations and The Web

Since the HTTP is a stateless protocol, a program terminates after processing a request received from a client. Thus, any successive computation must be continued by another program. This requires a request to carry enough information to resume a computation at the point it was suspended. Since continuations represent the rest of the computation, continuation-based programs allow continuous computations (such as list processing) to run on traditional web servers, such as Apache (TheApacheTeam) (McCarthy 2009).

6. MOBY COMPILER

This chapter discusses pedagogical ideas and technical background of Moby Compiler; a compiler that consumes Advanced Student Language (ASL) programs of PLT-Racket language that use World primitives, and produces applications for mobile platforms.

More precisely, Moby Compiler produces Javascript code from ASL programs, implements ASL primitives by runtime libraries written in Javascript included with the compiled application. For smartphones, Moby Compiler uses a bridge library, namely, *Phonegap*, which is an open source development infrastructure for building cross-platform mobile applications. Phonegap bridges HTML and Javascript with native facilities of Google Android, iPhone/iTouch/iPad, Palm, Blackberry and Symbian. Therefore, supporting multiple platforms by using such a bridge makes Moby a powerful abstraction regarding code reusability. Moby currently implements auxiliary features, such as tilt, location, sms and music only for Android platforms.

6.1. Pedagogical Idea

Most of computer science curriculum employ an HTDP approach on freshman programming courses, such as Istanbul Bilgi University Computer Science Department. Freshman curriculum includes building animation applications using DrScheme with World/Universe teachpack.

At the age of online interaction with computers and spread of smartphone platforms, such as Android, freshman students of programming can share applications with their friends, families or instructors using Moby. Compiling programs written in Advanced Student Language of PLT-Racket to Javascript or J2ME (for Android) using Moby provides an opportunity to embed applications on web-sites as well as on smartphones without knowing platform specific details.

6.1.1. Bootstrap Curriculum

There is a particular pedagogic program, namely *Bootstrap*, which involves Moby as a technical back end. In Bootstrap curriculum, middle-school students receives the 'first-dose of algebra' by creating images and animations through programming. Students learn that the algebra is not only a mathematical aspect; they learn to perform algebraic operations on a much richer set of data types, such as images (Schanzer and PLT-RacketTeam).

WeScheme is an online programming environment running Moby language. In other words, students are using WeScheme as an online (i.e. browser based) IDE to produce their formerly offline and stand alone animations created in DrScheme (currently DrRacket). Moby is the compiler underneath WeScheme. In addition to Scheme language, Moby provides additional features for using platform-dependent tools. For example, an application in WeScheme could communicate with a Google web service to receive a map into a program and process it to create a logistic application supported with additional location information from various GPS services (Yoo, Zhang, Cecchetti, Hickey, Krishnamurthi, Derici, and Zimmt).

Aside from a pedagogical perspective, ultimate goal of Moby is to provide a new programming environment for cell-phone application development. The 'world' is a starting point of a much greater objective which is to provide an easy development of new softwares for Android and J2ME. This has an implication that the 'world' teachpack should be extended toward cell-phone application development. Therefore, performance, efficiency and resource consumption are essential issues to the further development and expansion of Moby.

7. PROGRAM TRANSFORMATION

This chapter describes program code transformation for implementing proper tail calling behavior by modeling first-class continuations in a language that do not support run-time stack inspection and manipulation.

Presented transformation as well as support infrastructure are strongly based on one from Pettyjohn et al. However, in order to obtain proper tail calling, one needs to be able to actually apply the continuation, not only to capture. Our contribution is to make a transformed program be able to apply a previously captured and stored continuation. (Pettyjohn, Clements, Marshall, Krishnamurthi, and Felleisen 2005)

For most of the conventional programming languages, a function has no access to its continuation, which generally indicates parts of the program other than its own. However, a function does have access to its own dynamic state. Therefore, procedure bodies could be modified to cooperatively build a continuation object by appending its own part of a computation to an accumulated continuation model. This constitutes the main idea of our transformation.

7.1. ANF Transformation

The first step is to transform a code into *Administrative Normal Form (ANF)*, which is a canonical form where all function calls appear either as a right-hand side of an assignment or as an expression element of a return statement. In other words, each argument of a function must be named in ANF as a nested 'let' expressions having simple function calls in body. (Flanagan, Sabry, Duba, and Felleisen 1993) (McCarthy 2009)

To build and store a continuation object, one needs to have an access to all information about the control flow, which compound expressions abstract away within a computation such as temporary variables. However, transforming the code into

ANF linearizes a control flow by replacing compound expressions with corresponding primitive expression sequences and previously bounded identifiers.

Consider the fibonacci program written in Javascript:

```
function fibonacci(n) {
 if(n<=1){
   return n;
 }
 return fibonacci(n-1) + fibonacci(n-2);
}
```

The ANF-transformed version of the fibonacci program above:

```
 function fibonacci(n) {
   if(n<=1) {
     return n;
   }
   var temp1 = fibonacci(n-2);
   var temp2 = fibonacci(n-1);
   return temp1 + temp2;
 }
```

The function calls expressed by 'fibonacci(n-2)' and 'fibonacci(n-2)' are named as 'temp1' and 'temp2' at their parts, therefore the computation has been linearized (by subsequent assignments).

7.2. Fragmentation

A continuation resumes a computation at a point it was captured. Since functions has only a single entry point at the beginning, there should be a number of inter-procedural functions (i.e. fragments), each of which has an effect of resuming from the middle of original function. Because we previously transformed the fibonacci code into

ANF, all fragments begin with an assignment associated with a function call. Variables needed by that function call would be the formal parameters of corresponding fragment.

Fragmented code of previously ANF-converted fibonacci program:

```
function fibonacci_anf1(n) {
 if(n<=1) {
   return n;
 }
 var temp1 = fibonacci_anf1(n-2);
 return fibonacci_anf2(temp1, n);
}

function fibonacci_anf2(temp1, n) {
 var temp2 = fibonacci_anf1(n-1);
 return fibonacci_anf3(temp1, temp2);
}

function fibonacci_anf3(temp1, temp2) {
 return temp1 + temp2;
}
```

To resume the computation from a particular point, each fragment performs a tail-call to a subsequent fragment by passing the required parameters, which are originally the free-variables of that fragment.

7.3. Closure Conversion

At the beginning of this chapter, we explained that a continuation would be generated cooperatively by modified procedures appending their own part of the computation and returning that object to other procedures. This has an implication that a continuation would be a composition of subsequent frames, namely *continuation*

frames. Each frame should close over the values of live variables at the point where the continuation was captured and have a unary function which would invoke a correct fragment that resumes the computation from a desired point (Pettyjohn, Clements, Marshall, Krishnamurthi, and Felleisen 2005).

For our continuing fibonacci example, closures would be automatically generated because underlying language (Javascript in this case) supports anonymous functions as lexical closures. In languages that do not support anonymous functions such as C♯ or Java, class instances may be used to close over the live variables. In the last section of this chapter, we use the prototyping mechanism of Javascript to show that the anonymous functions are not the only solution to close over the live variables.

The prototype frame object for *fibonacci_anf2* shown below:

```
fib_frame2 = function(temp1){
    this.temp1 = temp1;
};

fib_frame2.prototype = new ContinuationFrame;
fib_frame2.prototype.Invoke = function(return_value){
    return fibonacci_anf3(this.temp1, return_value);
};
```

7.4. Annotation

To collaborate a continuation building process by adding an appropriate frame to an accumulated object, each function body should be annotated with an appropriate mechanism. In Javascript, using exception handling mechanism may be the most efficient solution for this task, while another strategy would be to return a special value instead of normal return value or to use a global flag to indicate that a continuation building is in progress.

Each function body are modified to collaborate both a computation and a continuation building process. It is a three-way mechanism which includes three functionality one procedure may perform:

- If a subsequent computation returned normally, return normally.
- If a subsequent computation throws an exception which is **not** an instance of a *SaveContinuationException*, throw that exception without any modification.
- If a subsequent computation throws an exception which **is** an instance of a *SaveContinuationException*, append an appropriate frame and throw the accumulated exception object (which is actually a continuation).

Annotated body of *fibonacci_anf2*:

```
function fibonacci_anf2(temp1, n) {
 var temp2;
 try {
   var temp2 = fibonacci_anf1(n-1);
 } catch(sce) {
   if(sce instanceof SaveContinuationException) {
     sce.Append( new fib_frame2(temp1) );
   }
   throw sce;
 }
 return fibonacci_anf3(temp1, temp2);
}
```

8. SUPPORT INFRASTRUCTURE

Although a transformed program has a capability of contributing the continuation building process by saving its part of computation, however, the code must be run within an additional framework which can build a continuation object from collected frames. Next chapter explains the fundamental issues about the support infrastructure which any transformed program would run in a continuation-based manner.

8.1. Modelling Continuations: An Exceptional Help

Continuations are composed of a series of *continuation frames*, which indicate the exact copy of the native procedure-call stack. Therefore, its constructor consumes the new and old frames and stores these appended bound to **frames** identifier.

```
var Continuation = function(new_frames, old_frames){
    this.frames = old_frames;
    while(new_frames !== null){
        // head_frame is a ContinuationFrame
        var head_frame = new_frames.first;
        new_frames = new_frames.rest;
        if(head_frame.continuation !== null)
            throw "Continuation not empty?";
        head_frame.continuation = this.frames;
        this.frames = new FrameList(head_frame, this.frames);
    }
};
```

More importantly, the frames in the continuation are collected in an exception object, namely *SaveContinuationException*, but this model (i.e. partial continuation) should be turned into a real continuation in order for it to be used. In other words, continuations are captured by subsequently throwing an accumulated *SaveContinuationException* by each fragment, reaching to the very first continuation which will be

the root continuation of the system. Establishing this initial continuation has to be the first thing the program do, because all other continuations have to be in the dynamic context of the root in order for them to keep the original meaning of the program. (Pettyjohn, Clements, Marshall, Krishnamurthi, and Felleisen 2005)

```
Continuation.EstablishInitialContinuation = function(thunk){

  while (true){
    try {
      return Continuation.InitialContinuationAux(thunk);
    } catch(wic) {
      if (! (wic instanceof WithinInitialContinuationException)) {
        throw wic;
      }
      thunk = wic.thunk;
    }
  }
};

// thunk -> object
Continuation.InitialContinuationAux = function(thunk) {
  try {
    return thunk();
  } catch(sce) {
    if (sce instanceof SaveContinuationException) {

     var k = sce.toContinuation();
     throw new WithinInitialContinuationException(makeWICThunk(k));

    } else if (sce instanceof ReplaceContinuationException) {

      throw new WithinInitialContinuationException(
                          makeReplaceContinuationThunk(sce));
```

```
    } else {
      throw sce;
    }
  }
};
```

8.1.1. Building The Continuation

While the 'EstablishInitialContinuation' keeps invoking the thunks, the 'InitialContinuationAux' function catches any 'SaveContinuationException' that is thrown and generate an actual continuation object through:

```
SaveContinuationException.prototype.toContinuation = function() {
    return new Continuation(this.new_frames, this.old_frames);
};
```

This saves all frames appended together in the Continuation object. The constructor of Continuation assigns all the 'continuation' fields of generated 'ContinuationFrames'. Here is the definition of 'ContinuationFrames':

```
ContinuationFrame = function(){
    this.continuation = null;
};

ContinuationFrame.prototype.Reload =
function(frames_above, restart_value) {

  var continue_value;

  if (frames_above === null) {
     continue_value = restart_value;
  } else {
     continue_value =
       frames_above.first.Reload(frames_above.rest, restart_value);
```

```
  }

  try {
      return this.Invoke(continue_value);
  } catch(sce) {
      if (! (sce instanceof SaveContinuationException))
            { throw sce; }

      sce.Append(this.continuation);
      throw sce;
  }
};
```

The 'Reload' function starts to invoke the 'Invoke' procedures of subsequent frames, starting from the last one, which would either be a 'CWCC_frame0' or 'ContinuationApplication_frame0'.

8.1.2. Invoking Continuations

After a certain number of procedure calls have been made, a continuation would be built by subsequent exception throws in order to deallocate current activation records (i.e. stack frames) from the run-time stack. To resume the computation from that point, accumulated continuation should be invoked. This is done by 'Continuation.CWCC' (abbreviated for call-with-current-continuation):

```
Continuation.CWCC = function(receiver) {
  try {
    Continuation.BeginUnwind();
  } catch(sce) {
    if (sce instanceof SaveContinuationException)) {
      sce.Extend(new CWCC_frame0(receiver));
      throw sce;
    }
```

```
    throw sce;
  }
  return null;
};
```

When a 'Continuation.CWCC' procedure invoked with a receiver, a 'SaveContinuationException' is thrown and immediately extended with a 'CWCC_frame0'. Therefore, the 'CWCC_frame0' would always be the first frame in a continuation model (i.e. SaveContinuationException). However, the invocation of 'Continuation.CWCC' starts a continuation building procedure, and all previous functions would extend the 'SaveContinuationException' by appropriate frames, which are already hardcoded into the function bodies by the annotation stage of the transformation.

The 'SaveContinuationException' would then finally caught by 'InitialContinuationAux' function. The 'InitialContinuationAux' would turn that exception object into an actual 'Continuation', by appending all its frames while arranging the 'continuation' fields and modify its first frame to be a 'ContinuationApplication_frame0'.

```
ContinuationApplication_frame0 = function(){
  ContinuationFrame.call(this);
};
ContinuationApplication_frame0.prototype = new ContinuationFrame();
ContinuationApplication_frame0.prototype.Invoke =
    function(return_value) { return return_value; };
```

The 'Invoke' function of a 'ContinuationApplication_frame0' is an identity function, thus, returns its only parameter as a result. That result is the final value of the computation which was suspended when the continuation building process is started. Therefore a suspended computation would be resumed by a 'ContinuationFrame' which is a subsequent frame of 'ContinuationApplication_frame0' that would invoke an appropriate function to continue an original execution.

8.2. Issues About Implementation

8.2.1. Live Variable Analysis

Our first implementation passes a thunk to 'WithinInitialContinuationException' as follows:

```
Continuation.InitialContinuationAux = function(thunk){
⋮

if (sce instanceof SaveContinuationException) {
  var k = sce.toContinuation();

  throw new WithinInitialContinuationException(
              function() {
                var escapingContinuation = k.adjustForEscape();
                return k.reload(escapingContinuation);
              });

⋮
}
```

The subsequent function 'EstablishInitialContinuation' would catch the 'WithinInitialContinuationException' and invoke the thunk hardcoded into it as an anonymous function.

However, running this version led to a realization that the variable 'thunk' in 'InitialContinuationAux' is still alive even after a 'WithinInitialContinuationException' is thrown and the 'InitialContinuationAux' function is terminated. The reason was a lexical closure generated by native Javascript interpreter closes over the 'thunk' as well as the other variables. This raise a problem of an unnecessary run-time stack growth proportional to the number of continuation captures.

The solution was to build thunks in a clean environment:

```
var makeWICThunk = function(k) {
  return function() {
    var escapingContinuation = k.adjustForEscape();
    return k.reload(escapingContinuation);
  };
};

var makeReplaceContinuationThunk = function(rce) {
  return function() {
    return rce.k.reload(rce.v);
  };
};
```

8.2.2. Continuation.apply

As mentioned before, no other continuation modeling technique involves any method to apply a previously captured and stored continuation. Our technique extends the model of Pettyjohn et al. with an actual continuation application method. With this way, our transformation and support infrastructure allows programs using extensive powers of continuations to implement some non-trivial control models, such as 'generate-one-element-at-a-time' example in Section 5.3.3, to be able to run on a stack-based imperative programming language, such as Javascript.

```
Continuation.apply = function(k, v) {
    throw new ReplaceContinuationException(k, v);
};
```

'Continuation.apply' function throws a 'ReplaceContinuationException' which includes a continuation and a continue value. As distinct from 'SaveContinuationException', the 'ReplaceContinuationException' can pass through all the try/catch blocks of a sequence of functions without swelling with various kinds of frames appended to it.

Additionally, a thunk prepared for a 'ReplaceContinuationException' does not modify implicit frames by replacing 'CWCC_frame0' with 'ContinuationApplication_frame0'. In other words, when a 'ReplaceContinuationException' is raised, the first frame of the 'FrameList' representing implicit parts of a continuation would always be the 'CWCC_frame0'.

Programming languages which support continuations usually provide two distinct operations for capturing and invoking a continuation, such as *callcc* and *throw* constructs in SML, for capturing and invoking continuations respectively. However, Scheme does not distinguish a function application from a continuation application. While it provides some constructs, such as *call/cc* and *let/cc*, they are binding operations. Thus, continuations treated as if they were ordinary functions. (Krishnamurthi 2007)

This kind of distinction appears in the 'Invoke' function of a 'CWCC_frame0'. In order to be able to run a program containing 'call/cc', one needs to provide a mechanism (or a distinction) for applying not only continuations represented by an actual Javascript function, but also continuations represented by a 'Continuation' instance. Therefore, our support infrastructure handles different representations with different mechanisms:

```
CWCC_frame0.prototype.Invoke = function(return_value) {
  if(this.receiver instanceof Continuation) {
    return Continuation.apply(this.receiver, return_value);
  }

  return this.receiver(return_value);
};
```

If the internal 'receiver' of a 'CWCC_frame0' is an instance of a 'Continuation', it immediately throws a 'ReplaceContinuationException' by invoking 'Continuation.apply' function, hereby causes a computation to be injected to an actual flow of a program. Otherwise (if the 'receiver' is an actual function), 'CWCC_frame0.Invoke'

applies its receiver to the 'return_value'.

Therefore, a program originally containing continuation capture or invocation also can be run on a stack-based programming language (Javascript in this case) in a proper tail calling environment using this method.

9. RESULTS

Using a first-class continuation model to obtain the proper tail calling behavior in Javascript assures a constant stack space consumption, because a continuation is built by using an exception object containing a list of frames added by each procedure fragment. Whenever a fragment adds its own part, it throws a continuation object to its caller as an exception, causing the deallocation of a corresponding activation record in native run-time stack. Since an invocation of a continuation would resume a computation from a point where it continuation is captured, computation would continue with an empty stack.

9.1. Time Consumptions

In this section, we discuss on run-time performance of transformed programs along with a concrete comparison with the most recent technique, namely *Cheney on M.T.A.* (Baker 1994).

A possible decrease in run-time performance of our technique would be expected from programs involving continuations, but currently there is no other technique which involves both continuations and proper tail calls. Thus we must make a comparison between two user programs without originally involving continuations (i.e. call/cc).

Fragments and closures (i.e. frames) would exhibit a performance decrease in low call depths[14] , because their effect is proportional to the frequency of continuation capture and invocation. With a frequent use of continuations, there would be a little performance penalty on run-time.

Averages of execution time, indicating performance test [15] results are shown in

[14]... indicates at how many procedure calls a continuation is built.

[15]Implementation and testing are done in Intel Pentium D i945P machine with Ubuntu 9.10 operating system on 1 GB memory. Cheney on M.T.A implementation is based on Danny Yoo's code on: http://hashcollision.org/tmp/cheney/cheney-test.html

Table 9.1:

Input	**Call Depth**	**Cheney on M.T.A** (ms)	**Continuation-Based** (ms)
10	100	0.1	0.1
100	100	21.0	1.4
1000	100	217.4	5.6
10000	100	2209.9	33.7
100000	100	22080.2	304.6
10	10	20.5	1.1
100	10	204.2	3.7
1000	10	2070.3	12.2
10000	10	20417.8	84.4
100000	10	204845.9	792.6

Table 9.1. Run-time Comparison of Cheney on M.T.A. and The Continuation-based Technique For Factorial program

Table 9.1 illustrates the run-time performance comparison between continuation-based proper tail calling and 'Cheney on M.T.A.", implementing a tail-recursive factorial program for various inputs and call (i.e. trampoline) depths. The results[16] indicate that using a first-class continuation model to implement proper tail calls has a notable advantage on classical trampolining and CPS with garbage collection.

9.2. Advantages and Drawbacks

The advantage of this technique is that it introduces the framework where tail calls do not need any additional stack space. Furthermore, it provides an infrastructure for an actual *call/cc* implementation on imperative stack-based languages, because the ability to abandon a context and inject a computation to some context are preserved.

The generated code after the transformation as well as the support codes do not

[16]Results are obtained by taking the average of ten successive execution. Since the difference between two techniques is obvious, no greater degree of test scale is needed.

exhibit a large performance decrease on run-time. Since a compiler would make similar transformations, the ANF-conversion does not make a big effect on the performance. However, a possible drawback would be the hygiene problem where a collusion occurs between compiler temporaries and local identifiers.

Another drawback is closure conversion and annotation greatly increases the code size as well as the fragmentation phase and the closure conversion after fragmentation introduces some overhead on run-time but they are only used when a continuation is invoked. Code annotation introduces some try/catch blocks but their effect on run-time performance strongly depends on how frequently the continuations are captured.

10. CONCLUSIONS

The lack of proper tail calls in conventional imperative programming languages forces programmers to employ a mathematically unnatural as well as a difficult approach to recursively defined data. For programs processing such data to become maintainable, understanding and expendable programs, implementing novel features from function programming languages for imperative ones is essential. We have proposed that first-class continuation models in imperative programming languages makes a proper tail calling not only possible, but efficient in terms of time complexity.

Furthermore, this approach creates the possibility that pedagogical tools, such as Moby-WeScheme, could be fully supported to teach programming to freshman students employing HTDP philosophy. Without such an improvement, schools with programming education are forced to use offline programming environments. Thus, such advancements in imperative programming languages help to use features from functional programming languages without any inefficiency which could lead to a possible discouragement in students.

Additionally, first-class continuation models on imperative programming languages, involving a continuation capture as well as invocation, allows programmers to construct many different computational models with a lot of possible control mechanisms due to the extensive power of continuations, which provides a capability of inspecting and modifying a native run-time stack even underlying language originally does not support such operations.

To support this thesis, we have extended a transformation based on one from Pettyjohn et al. Using stack inspection by first-class continuations to achieve a constant stack space behavior of tail calls as well as each stage of a necessary transformation have been stated and discussed. Implementation details and run-time support infrastructure are explained. Concrete run-time results are shown, and based on these experiments it is found that although our technique produces a much larger code, it is considerably

more efficient than the most recent technique, namely Cheney on M.T.A. method.

10.1. Future Work

Although the proposed transformation implements proper tail calling from first-class continuations, a sound proof using denotational semantics should be provided. This method implements an asymptotically safe-for-space behavior, but case by case semantical analysis should be studied in order to be sure that this method correctly implements proper tail calls in any case.

Moreover, we proposed an additional mechanism to also invoke a previously captured continuation. This will raise an interesting future work about a possible implementation of *call/cc* itself on conventional stack-based programming languages that do not support run-time stack manipulation.

APPENDIX A: TRANSFORMED CALL/CC EXAMPLES

A.1. try-to-escape.js

load("../support.js");

Translation of the function:

```
(define (f)
  (call/cc (lambda (k) (* 5 (k 4)))))
```

The expected value from (f) should be 4.

This should translate to the javascript code:

```
var f = function() {
    return callCC(function(k) {
              return 5 * Continuation.apply(k, 4);
            });
};
```

which should then be a-normalized to:

```
var f = function() {
    return callCC(function(k) {
       var t = Continuation.apply(k, 4);
       return 5 * t;
     });
};
```

which should then be annotated to the following code:

```
var f = function() {
  return Continuation.CWCC(aReceiver);
```

```
};

var aReceiver = function(k) {
  var t;
  try {
    t = Continuation.apply(k, 4);
  } catch (sce) {
    if (! (sce instanceof SaveContinuationException)) {
      throw sce;
    }
    sce.Extend(new aReciever0_frame());
    throw sce;
  }
  return 5 * t;
};


var aReceiver0 = function(t) {
  return 5 * t;
}

var aReceiver_frame = function() {
  ContinuationFrame.call(this);
}
aReceiver_frame.prototype = new ContinuationFrame();
aReceiver_frame.prototype.Invoke = function(v) {
  return aReceiver0(v);
}
aReceiver_frame.prototype.toString = function() {
  return "[f0_frame]";
}

// Let's try to test this code.
```

```
var test = function() {
    return Continuation.EstablishInitialContinuation(
            function() { return f(); }
           );
}
```

A.2. try-to-get-in.js

load("../support.js");

Translation of the function:

```
        (define return false)

        (+ 1 (call/cc
                        (lambda (cont)
                                        (set! return cont)
                                        1))))
```

```
> (return 22))
23
```

This should translate to the javascript code:

```
var ret = false;
var g = function() {
 1 + callCC(function(cont) {
                ret = cont;
                return 1;
              });
};

var f = function() {
 return Continuation.apply(ret, 22);
```

```
};
```

which should then be a-normalized to:

```
var ret = false;

var g = function(){
 temp1 = callCC(function(cont) {
                  ret = cont;
                  return 1;
                });
 return 1 + temp1;
};

var f = function(){
 return Continuation.apply(ret, 22);
};
```

which should then be annotated to the following code:

```
var ret = false;

var f = function() {
  return Continuation.apply(ret,22);
};

var g = function() {
  var temp1;
  try{
    temp1 = Continuation.CWCC(function(cont){
                                ret = cont;
                                return 1;
                              });
  }catch(sce) {
```

```
    if (! (sce instanceof SaveContinuationException)) {
      throw sce;
    }
    sce.Extend(new g_frame0());
    throw sce;
  }
  return 1 + temp1;
};

var g_frame0 = function() {
  ContinuationFrame.call(this);
};
g_frame0.prototype = new ContinuationFrame();
g_frame0.prototype.Invoke = function(return_value) {
  return 1 + return_value;
};
g_frame0.prototype.toString = function() {
  return "[g_frame0]";
};

// Let's try to test this code.

var test = function() {
    Continuation.EstablishInitialContinuation(
      function() { return g(); }
    );
    return Continuation.EstablishInitialContinuation(
              function() { return f(); }
            );
}
```

A.3. generateOneElement.js

// Switching Control / Jump out-Back in

load("../support.js");

Original scheme code:

```
(define (generate-one-element-at-a-time lst)
  (define (control-state return)
    (map
      (lambda (element)
        (set! return (call/cc
                       (lambda (resume-here)
                         (set! control-state resume-here)
                         (return element)))))
      lst)
    (return 'fell-down))

  (lambda ()
    (call/cc control-state)))

(define generate-digit
        (generate-one-element-at-a-time '(0 1 2 3 4 5 6 7 8)))
```

```
> (generate-digit)
0
> (generate-digit)
1
> (generate-digit)
2
```

— These should translate to JS code: —

```
function map(f, arglist){
  if(arglist.isEmpty()){
    return plt.types.Empty.EMPTY;
  }
  return plt.Kernel.cons(f(arglist.first()),
                         map(f, arglist.rest()));
}



function generateOneElementAtATime(lst){

  function controlState(ret){
    map(function(element){
        ret = callCC(function(resumeHere){
                  controlState = resumeHere;
                  return Continuation.apply(ret,element);});
        },lst);
    return Continuation.apply(ret,"fell-down");
        }

    return function(){return callCC(controlState);};
}

var generateDigit =
  generateOneElementAtATime(plt.Kernel.list([0,1,2,3,4,5,6,7,8,9]));
```

— After A-Normalization : —

```
function map(f, arglist){
        if(arglist.isEmpty()){
                return plt.types.Empty.EMPTY;
        }
                var temp1 = f(arglist.first());
                var temp2 = map(f, arglist.rest());
```

```
                return plt.Kernel.cons(temp1, temp2);
}

function generateOneElementAtATime(lst){

  function controlState(ret){
    map(function(element){
        ret = callCC(function(resumeHere){
                controlState = resumeHere;
                return Continuation.apply(ret,element);});
        },lst);
    return Continuation.apply(ret,"fell-down");
        }

    return function(){return callCC(controlState);};
}

var generateDigit =
  generateOneElementAtATime(plt.Kernel.list([0,1,2,3,4,5,6,7,8,9]));

function f() {
        generateDigit();
        generateDigit();
        generateDigit();
}
```

Which should then fragmented & annotated to the following code:

```
var f = function(){

  try{
    generateDigit();
  }catch(e){
    if(e instanceof SaveContinuationException){
```

```
      e.Extend(new f_frame0());

        throw e;
    }
    throw e;
  }
  return f_1();
};

var f_frame0 = function(){
};
f_frame0.prototype = new ContinuationFrame();
f_frame0.prototype.Invoke = function(return_value) {
  return f_1();
};
f_frame0.prototype.toString = function() {
  return "[f_frame0]";
};

var f_1 = function(){
        try{
                //alert("f_1");
                generateDigit();
        }catch(e){
                if(e instanceof SaveContinuationException){
                        e.Extend(new f_frame1());
                        throw e;
                }
                throw e;
        }
        return generateDigit();
};

var f_frame1 = function(){
```

```
};

f_frame1.prototype = new ContinuationFrame();
f_frame1.prototype.Invoke = function(return_value) {
  return generateDigit();
};
f_frame1.prototype.toString = function() {
  return "[f_frame1]";
};



var generateDigit =
  generateOneElementAtATime(plt.Kernel.list([0,1,2,3,4,5,6,7,8,9]));

function generateOneElementAtATime(lst){

  var controlState = function(ret){
    try{
      var func =
        function(element){
          try{
            ret = Continuation.CWCC(function(resumeHere){
                    controlState = resumeHere;
                    return Continuation.apply(ret,element);
            });
          }catch(sce){
            if(!(sce instanceof SaveContinuationException)) {
              throw sce; }
            sce.Extend(new func_frame0(element));
            throw sce;
          }
        };
      var func_frame0 = function(element){
```

```
      this.element = element;
    };

    func_frame0.prototype = new ContinuationFrame();
    func_frame0.prototype.Invoke = function(return_value) {
      ret = return_value;
    };
    func_frame0.prototype.toString = function() {
      return "[func_frame0]";
    };

    map(func , lst);
  } catch(e) {
      if(e instanceof SaveContinuationException) {
        e.Extend(new controlState_frame(ret));
        throw e;
      }
      throw e;
  }
  return Continuation.apply(ret,"fell-down");
};

var controlState_frame = function(ret){
  this.ret = ret;
};

controlState_frame.prototype = new ContinuationFrame();
controlState_frame.prototype.Invoke = function(return_value) {
  return Continuation.apply(this.ret,"fell-down");
};
controlState_frame.prototype.toString = function() {
  return "[controlState_frame]";
};
```

```
  return function(){
    return Continuation.CWCC(controlState);
  };

}

function map(f, arglist){
  if(arglist.isEmpty()){
    return plt.types.Empty.EMPTY;
  }
  var temp1;
  try{
    temp1 = f(arglist.first());
  }catch(sce){
    if(sce instanceof SaveContinuationException) {
      sce.Extend(new map_frame0(f,arglist.rest()));
      throw sce;
    }
    throw sce;
  }
  return map1(temp1, f, arglist);
}

function map1(temp1, f, arglistrest){
  var temp2;
  try{
    temp2 = map(f, arglistrest);
  }catch(sce){
    if(sce instanceof SaveContinuationException){
      sce.Extend(new map_frame1(temp1));
      throw sce;
    }
    throw sce;
  }
```

```
  return map2(temp1,temp2);
}

function map2(temp1,temp2){
  return plt.Kernel.cons(temp1,temp2);
}

var map_frame0 = function(f,arglist){
  this.f = f;
  this.arglist = arglist;
};

map_frame0.prototype = new ContinuationFrame();
map_frame0.prototype.Invoke = function(return_value) {
  return map1(return_value,this.f,this.arglist);
};
map_frame0.prototype.toString = function() {
  return "[map_frame0]";
};

var map_frame1 = function(temp1){
  this.temp1 = temp1;
};

map_frame1.prototype = new ContinuationFrame();
map_frame1.prototype.Invoke = function(return_value) {
  return map2(this.temp1,return_value);
};
map_frame1.prototype.toString = function() {
  return "[map_frame1]";
};

// testing
```

```
var test = function() {
    return Continuation.EstablishInitialContinuation(
               function() { return f(); }
           );
};
```

APPENDIX B: COMPLETE SUPPORT CODE

B.1. ContinuationFrame

```
ContinuationFrame = function(){
    this.continuation = null;
};

ContinuationFrame.prototype.Reload =
function(frames_above, restart_value) {

  var continue_value;

  if (frames_above === null) {
     continue_value = restart_value;
  } else {
     continue_value =
       frames_above.first.Reload(frames_above.rest, restart_value);
  }

  try {
      return this.Invoke(continue_value);
  } catch(sce) {
      if (! (sce instanceof SaveContinuationException))
            { throw sce; }

      sce.Append(this.continuation);
      throw sce;
  }
};

ContinuationFrame.prototype.Invoke = function(return_value) {
    throw new Error("Unimplemented!");
```

```
};

ContinuationFrame.prototype.toString = function() {
    return "[ContinuationFrame]";
};
```

B.2. FrameList

```
FrameList = function(_first, _rest){
    this.first = _first;
    this.rest = _rest;
};

FrameList.reverse = function(originalFrameList){

    var result = null;

    while(originalFrameList!=null){
        result = new FrameList(originalFrameList.first, result);
        originalFrameList = originalFrameList.rest;
    }

    return result;
};

FrameList.prototype.length = function() {
        temp1 = this;
        var ret = 0;

        while(temp1 != null){
                ret++;
                temp1 = temp1.rest();
        }
```

```
        return ret;
};

FrameList.prototype.toString = function() { return "[FrameList]"; }
```

B.3. SaveContinuationException

```
SaveContinuationException = function(){
    this.new_frames = null;
    this.old_frames = null;
};

SaveContinuationException.prototype.Extend = function(extension) {
    this.new_frames = new FrameList(extension, this.new_frames);
};

SaveContinuationException.prototype.Append = function(old_frames) {
    this.old_frames = old_frames;
};

SaveContinuationException.prototype.toContinuation = function() {
    return new Continuation(this.new_frames, this.old_frames);
};

SaveContinuationException.prototype.toString = function() {
    return "[SaveContinuationException]";
};

ReplaceContinuationException = function(k, v) {
    this.k = k;
    this.v = v;

};
```

B.4. Continuation

```
var Continuation = function(new_frames, old_frames){
    this.frames = old_frames;

    while(new_frames !== null){
       // head_frame is a ContinuationFrame
       var head_frame = new_frames.first;

       new_frames = new_frames.rest;

       if(head_frame.continuation !== null)
              throw "Continuation not empty?";

       head_frame.continuation = this.frames;

       this.frames = new FrameList(head_frame, this.frames);
    }
};

// Adjusts the continuation to be used to escape out of the context.
Continuation.prototype.adjustForEscape = function() {
    var newFrames =
          new FrameList(new ContinuationApplication_frame0(), null);

    return new Continuation(newFrames, this.frames.rest);
};



Continuation.prototype.reload = function(restart_value){
    var rev = FrameList.reverse(this.frames);
    return rev.first.Reload(rev.rest, restart_value);
};

Continuation.prototype.toString = function() {
```

```
        var retval = "[Continuation] :\n" + "frames;\n";
        var temp = this.frames;

        while(temp !== null){
                retval += temp.first + "\n";
                temp = temp.rest;
        }

        return retval;
};

Continuation.BeginUnwind = function(){
    throw new SaveContinuationException();
};

Continuation.CWCC = function(receiver){
    try {
        Continuation.BeginUnwind();
    } catch(sce) {
        if (! (sce instanceof SaveContinuationException)) {
            throw sce;
        }

        sce.Extend(new CWCC_frame0(receiver));

        throw sce;
    }
    return null;
};

Continuation.apply = function(k, v) {
    throw new ReplaceContinuationException(k, v);
};
```

```
Continuation.EstablishInitialContinuation = function(thunk){

  while (true){
    try {
      return Continuation.InitialContinuationAux(thunk);
    } catch(wic) {
      if (! (wic instanceof WithinInitialContinuationException)) {
        throw wic;
      }
      thunk = wic.thunk;
    }
  }
};

// thunk -> object
Continuation.InitialContinuationAux = function(thunk) {
  try {
    return thunk();
  } catch(sce) {
    if (sce instanceof SaveContinuationException) {

     var k = sce.toContinuation();
     throw new WithinInitialContinuationException(makeWICThunk(k));

    } else if (sce instanceof ReplaceContinuationException) {

      throw new WithinInitialContinuationException(
                          makeReplaceContinuationThunk(sce));

    } else {
      throw sce;
    }
  }
```

```
};

var makeWICThunk = function(k) {
    return function() {
        var escapingContinuation = k.adjustForEscape();
        return k.reload(escapingContinuation);
    }
};

var makeReplaceContinuationThunk = function(rce) {
    return function() {
        return rce.k.reload(rce.v);
    };
};
```

B.5. CWCC_frame0

```
CWCC_frame0 = function(receiver) {
    ContinuationFrame.call(this);
    this.receiver = receiver;
};
CWCC_frame0.prototype = new ContinuationFrame();
CWCC_frame0.prototype.Invoke = function(return_value) {

     if(this.receiver instanceof Continuation) {
             return Continuation.apply(this.receiver, return_value);
     }

     return this.receiver(return_value);
};
CWCC_frame0.prototype.toString = function() {
     return "[CWCC_frame0]";
};
```

B.6. ContinuationApplication_frame0

```
ContinuationApplication_frame0 = function() {
        ContinuationFrame.call(this);
};

ContinuationApplication_frame0.prototype = new ContinuationFrame();

ContinuationApplication_frame0.prototype.Invoke =
    function(return_value) { return return_value; };

ContinuationApplication_frame0.prototype.toString =
    function() { return "[ContinuationApplication_frame0]"; };
```

B.7. WithinInitialContinuationException

```
WithinInitialContinuationException = function(thunk) {
        this.thunk = thunk;
};



WithinInitialContinuationException.prototype.toString =
    function() { return "[WithinInitialContinuationException]"; };
```

APPENDIX C: COMPLETE TRANSFORMATION CODE

C.1. anormalize.ss

```
#lang s-exp "../lang.ss"

(require "anormal-frag-helpers.ss")
(require "../../collects/moby/runtime/stx.ss")
(require "box-local-defs.ss")
(require "../toplevel.ss")
(require "../env.ss")

;; string with which to name temporary variables
(define temp-begin "temp~a")

;; procedures that we will not consider primitive
;; because they can potentially
;; call arguments that might need a continuation
(define higher-order-prims '
  (andmap argmax argmin build-list build-string compose
          filter foldl foldr map memf ormap quicksort sort))
;; first-order "primitives" not included in toplevel.ss
(define other-prims '(quote set!))

;; stateful hash of primitives and reset procedure
(define prims (make-hash))
(define (reset-prims prim-hash) (set! prims prim-hash))

;; get-struct-defs: (listof s-expr) -> (listof s-expr)
;; takes a list of toplevel statements (a program)
;; returns all struct definitions appearing at toplevel
(define (get-struct-defs program)
  (filter (lambda (statement)
```

```
            (and (cons? statement)
                 (equal? (first statement) 'define-struct)))
          program))

;; generate-prims:(listof s-expr) symbol -> (hash-of symbol boolean)
;; consumes a list of toplevel statements (a program)
;; returns an environment containing all first-order primitives for
;; that program
;;    these are the predefined first-order primitives
;;    and struct primitives
(define (generate-prims program language)
  (let* ([prim-hash (make-hash)]
         [add-key (lambda (key)
                    (if (member key higher-order-prims)
                        (void)
                        (hash-set! prim-hash key #t)))])
    (begin
      (for-each add-key other-prims)
      (for-each add-key (env-keys (get-toplevel-env language)))
      (map (lambda (struct-def)
             (for-each add-key (get-struct-procs struct-def)))
           (get-struct-defs program))
      prim-hash)))

;; primitive-expr?: stx -> boolean
;; consumes a syntax object
;; returns true if the object represents a primitive expression,
;; false otherwise
;;    a primitive expression is either any atomic expression
;;    or a procedure application where the procedure is a
;;    first-order primitive
(define (primitive-expr? expr)
  (or (stx:atom? expr)
      (hash-ref prims (stx-e (first (stx-e expr))) #f)))
```

```
;; gen-temp-symbol: number -> symbol
;; takes a gensym counter and returns a symbol for temporary
;; binding using that gensym
(define (gen-temp-symbol num)
  (string->symbol (format temp-begin num)))

;; fold-anormal-help: (list-of stx) -> linfo
;; consumes a list of syntax objects
;; folds anormal-help across the list, returning linfo where
;; the return
;;    is the list of returns from each call to anormal-help
;;    and the raise is the concatination of the raise lists
;; NOTE: there are some odd-looking calls to reverse
;;       that allow for a foldl so the gensym numbers appear
;;       in a sensable order for testing
(define (fold-anormal-help expr)
  (let ([reversed-result
         (foldl (lambda (an-expr rest-info)
                  (let ([rec-info (anormal-help an-expr)])
                    (make-linfo
                     (cons (linfo-return rec-info)
                           (linfo-return rest-info))
                     (append (reverse (linfo-raise rec-info))
                             (linfo-raise rest-info)))))
                (make-linfo empty empty)
                expr)])
    (make-linfo (reverse (linfo-return reversed-result))
                (reverse (linfo-raise reversed-result)))))

;; anormal-help: stx -> linfo
;; consumes a syntax object representing an expression
;; produces a semantically equivalent expression as linfo
;;    where the return is the final return statement
```

```
;;     and the raise is the other local bindings created
;;     for a-normalization
(define (anormal-help expr)
  ;; if we have an atomic element there's nothing to
  ;; a-normalize
  (if (stx:atom? expr)
      (make-linfo expr empty)
      ;; otherwise we have a list
      (let* ([expr-list (stx-e expr)]
             [first-elt (stx-e (first expr-list))])
        (cond

          ;; if we have a define statement, then
          ;; a-normalize the body and put any raised
          ;; elements in a local inside the define
          [(equal? first-elt 'define)
           (let ([body-info (anormal-help
                             (third expr-list))])
             (make-linfo
              (datum->stx false
                          (list (first expr-list)
                                (second expr-list)
                                (if (empty? (linfo-raise body-info))
                                    (linfo-return body-info)
                                    (list 'local
                                          (linfo-raise body-info)
                                          (linfo-return body-info))))
                           (stx-loc expr))
              empty))]

          ;; if we have a local statement then first
          ;; do a self-contained a-normalize on each definition
          ;; then a-normalize the body and append any new
          ;; definitions to the old list
```

```
[(equal? first-elt 'local)
 (let ([defs (map make-anormal
                  (stx-e (second expr-list)))]
       [body-info (anormal-help (third expr-list))])
   (make-linfo
    (datum->stx false
                (list (first expr-list)
                      (append defs
                              (linfo-raise body-info))
                      (linfo-return body-info))
                (stx-loc expr))
    empty))]

;;for if statements we a-normalize(and pass up the raise)
;;  on the condition and do self-contained a-normalize
;;  on both the then and else clauses to make sure nohing
;;  is evaluated when it shouldn't be
[(equal? first-elt 'if)
 (let ([condition (anormal-help (second expr-list))]
       [then-clause (make-anormal (third expr-list))]
       [else-clause (make-anormal (fourth expr-list))])
   (if (primitive-expr? (linfo-return condition))
       (make-linfo
        (datum->stx false
                    (list 'if
                          (linfo-return condition)
                          then-clause
                          else-clause)
                    (stx-loc expr))
        (linfo-raise condition))
       (let ([temp-symbol (gen-temp-symbol (gensym))])
         (make-linfo
          (datum->stx false
                      (list 'if
```

```
                                    temp-symbol
                                    then-clause
                                    else-clause)
                                (stx-loc expr))
                        (append (linfo-raise condition)
                                (list (datum->stx
                                       false
                                       (list 'define
                                             temp-symbol
                                             (linfo-return condition))
                                       (stx-loc (second expr-list))
))))))))]

;; with and, or, and begin it is easiest to just
;; do a self-contained a-normalize on each sub-expression
[(or (equal? first-elt 'and)
     (equal? first-elt 'or)
     (equal? first-elt 'begin))
 (make-linfo
  (datum->stx false
              (map make-anormal expr-list)
              (stx-loc expr))
  empty)]
;; for quote, define-struct, and require,
;; we don't want to touch anything
[(or (equal? first-elt 'quote)
     (equal? first-elt 'define-struct)
     (equal? first-elt 'require))
 (make-linfo expr empty)]

;; for any other expression, a-normalize
;; each sub-expression
;; then fold across the result,
;; checking if each expression is primitive
```

```
;; if it is, then leave it,
;;     otherwise create a new define binding it
;;     to a temporary variable and add that
;;     to the list of raises
[else
 (let* ([arg-info (fold-anormal-help expr-list)]
        [anormal-expr
         (foldl
          (lambda (an-expr rest-args)
            (if (primitive-expr? an-expr)
                (make-linfo
                  (cons an-expr (linfo-return rest-args))
                        (linfo-raise rest-args))
                (let ([temp-symbol
                       (gen-temp-symbol (gensym))])
                  (make-linfo
                   (cons (datum->stx false
                                     temp-symbol
                                     (stx-loc an-expr))
                         (linfo-return rest-args))
                   (cons (datum->stx false
                                     (list 'define
                                           temp-symbol
                                           an-expr)
                                     (stx-loc an-expr))
                         (linfo-raise rest-args))))))
          (make-linfo empty empty)
          (linfo-return arg-info))])
   (make-linfo
    (datum->stx false
                (reverse (linfo-return anormal-expr))
                (stx-loc expr))
    (append (linfo-raise arg-info)
            (reverse (linfo-raise anormal-expr)))))])))
```

```
;; make-anroaml: stx -> stx
;; consumes a syntax object representing a single
;; stand-alone expression
;; produces a semantically equivalent expression in a-normal form
(define (make-anormal expr)
  (if (stx:atom? expr)
      expr
      (let ([linfo-out (anormal-help expr)])
        (if (empty? (linfo-raise linfo-out))
            (linfo-return linfo-out)
            (datum->stx false
                        (list 'local
                              (linfo-raise linfo-out)
                              (linfo-return linfo-out))
                        (stx-loc expr))))))

;; anormalize: stx:list -> stx:list
;; consumes a syntax object representing a program
;; produces a semantically equivalent program in a-normal form
;; NOTE: this is what is exported, so it resets all stateful fields
;;       and generates the list of primitives to insure it acts
;;       as a pure function
(define (anormalize program)
  (let ([readied (ready-anormalize program)])
    (begin
      (reset-gensym)
      (reset-prims (generate-prims
                    (stx->datum readied) 'language-here))
      (datum->stx false
                  (map make-anormal (stx-e readied))
                  (stx-loc readied)))))
```

```
(provide/contract
 [anormalize (stx:list? . -> . stx:list?)])
```

C.2. fragmenter.ss

```
#lang s-exp "../lang.ss"

(require "anormal-frag-helpers.ss")
(require "anormalize.ss")
(require "../../collects/moby/runtime/stx.ss")

;; strings to prepend onto fragments and name anonymous expressions
(define frag-prepend "f~a_~a")
(define statement-name "statement~a")

;; finfo is used to hold fragmentation information
;;    - return : stx representing the new expression
;;    - fragments : (list-of stx) containing the definitions
;;                  of new fragments
;;    - gensym : a gensym counter counting the number of fragments
(define-struct finfo (return fragments gensym))

;; split is used when splitting a list of definitions inside local
;; fragmentation
;;    - keep : (list-of stx), a list of definitions
;;             to keep (and fragment)
;;    - current : (or stx false), the first value definition,
;;                kept in this fragment
;;    - move : (list-of stx), the reamining value definitions
;;             to fragment out
(define-struct split (keep current move))

;; get-bound-id: stx -> symbol
;; consumes a definition (as a syntax object)
```

```
;; returns the id bound by the define statement
(define (get-bound-id defn)
  (if (stx-begins-with? defn 'define)
      (if (stx:atom? (second (stx-e defn)))
          (stx-e (second (stx-e defn)))
          (stx-e (first (stx-e (second (stx-e defn))))))
      (error 'get-bound-id
             (format "expected definition, found: ~a" defn))))


;; split-def-list: (listof stx) -> split
;; consumes a list of definitions
;; returns a split where the keep is the beginning of the input list
;;     up to (and including) the first value definition
;;     that is not a boxed undefined
;;     the current is that first definition
;;     if it exists (false otherwise)
;;     and the move is everything after that definition
(define (split-def-list def-list)
  (cond
    [(empty? def-list) (make-split empty #f empty)]
    [(and (cons? def-list)
          (stx-begins-with? (first def-list) 'define))
     (let ([components (stx-e (first def-list))])
       (if (or (stx:list? (second components))
               (equal? (stx->datum (third components))
                       '(box 'undefined)))
           (let ([rec-return (split-def-list (rest def-list))])
             (make-split (cons (first def-list)
                               (split-keep rec-return))
                         (split-current rec-return)
                         (split-move rec-return)))
           (make-split (list (first def-list))
                       (first def-list)
```

```
                    (rest def-list))))]
    [else (error 'split-def-list
                 (format "expected list of definitions, found: ~a"
                         def-list))]))


;; fragment-help: stx (list-of symbol) symbol number -> finfo
;; consumes an expression (as a syntax object),
;;   a list of arguments for any new fragments
;;   (generated by finding closures),
;;   the name of the procedure/statement we're fragmenting,
;;   and a counter to tell us how many fragments we've already made
;;   produces finfo where the return is the first fragment
;;   (or the expression itself),
;;   the fragments are all other fragments of the current expression
;;   and the gensym is the index of the next fragment
(define (fragment-help expr args name frag-counter)
  ;; if we have an atomic element there's nothing to fragment
  (if (stx:atom? expr)
      (make-finfo expr empty frag-counter)
      ;; otherwise we have a list
      (let* ([expr-list (stx-e expr)]
             [first-elt (stx-e (first expr-list))])
        (cond

          ;; if we have a LOCAL, then we may have fragments
          ;; unless everything is either a procedure or boxed
          ;; undefine
          [(equal? first-elt 'local)
           (let*
               (;; first split the definition list into
                ;; procedures/boxed undefined
                ;; and temporary statement definitions
                ;; and get the identifiers bound by
```

```
;; the first set of defines
[split-defs (split-def-list
             (stx-e (second expr-list)))]
[new-bound-ids (map get-bound-id
                    (split-keep split-defs))]
;; make a recursive call,
;; that depends on the definitions
[rec-rest
 ;; if there are no temporary value definitions
 ;; at all then just recur on the body of the local
 ;; because we don't need any more fragments
 (if (false? (split-current split-defs))
     (fragment-help (third expr-list)
                    (append new-bound-ids args)
                    name
                    frag-counter)
     ;; otherwise recur on a new procedure fragment
     (fragment-help
      (datum->stx
       false
       (list 'define
             (cons (string->symbol
                    (format frag-prepend
                            frag-counter
                            name))
                   (append new-bound-ids args))
             ;; the contents of the
             ;; new definitions are just
             ;; the local body if we hit the
             ;; last temp definition
             (if (empty? (split-move split-defs))
                 (third expr-list)
                 ;; or a new local with the rest
                 ;; otherwise
```

```
                    (list 'local
                          (split-move split-defs)
                          (third expr-list))))
             (stx-loc expr))
            args
            name
            (add1 frag-counter)))]
    [more-fragments? (stx-begins-with?
                      (finfo-return rec-rest)
                      'define)])
;; now make new linfo with the fragmented
;; local definitions
;;    that we kept, and either a call to
;;      the next fragment or the fragmented
;;      body of the local
;; the fragments are the recursive fragments
;; with the new fragment consed on if it exists
(make-finfo
 (datum->stx
  false
  (list 'local
        (apply append
               (map get-fragments
                    (split-keep split-defs)))
        (if more-fragments?
            (second (stx-e (finfo-return rec-rest)))
            (finfo-return rec-rest)))
  (stx-loc expr))
 (if more-fragments?
     (cons (finfo-return rec-rest)
           (finfo-fragments rec-rest))
     (finfo-fragments rec-rest))
 (finfo-gensym rec-rest)))]
```

```
;; if we have a BEGIN, AND, or OR,
;; then recursively fragment a new
;;    procedure with all but the first expression
;; then return finfo where the return is
;; the same type of statement
;;    that first calls the first instruction
;;    and then the next fragment
;;    and the raise is the rest of the fragments
[(or (equal? first-elt 'begin)
     (equal? first-elt 'and)
     (equal? first-elt 'or))
 (let* ([first-expr
         (fragment-help (second expr-list)
                        args name
                        frag-counter)]
        [rec-rest
         (fragment-help
          (datum->stx
           false
           (list 'define
                 (cons (string->symbol
                        (format frag-prepend
                                (finfo-gensym first-expr)
                                name))
                       args)
                 (if (empty? (rest
                              (rest
                               (rest expr-list))))
                     (third expr-list)
                     (cons (first expr-list)
                           (rest (rest expr-list))))))
          (stx-loc expr))
         args
         name
```

```
            (add1 (finfo-gensym first-expr)))])
      (make-finfo
       (datum->stx false
                   (list (first expr-list)
                         (finfo-return first-expr)
                         (second
                          (stx-e
                           (finfo-return rec-rest))))
                   (stx-loc expr))
       (append (finfo-fragments first-expr)
               (cons (finfo-return rec-rest)
                     (finfo-fragments rec-rest)))
       (finfo-gensym rec-rest)))]

;; if we have a DEFINE, then collect the arguments
;; for the closure, then recursively fragment the body,
;; and return the fragmented body inside the same define
;; (with the fragments in the raise)
[(equal? first-elt 'define)
 (let* ([new-args
         (if (stx:list? (second expr-list))
             (rest (stx->datum (second expr-list)))
             empty)]
        [filtered-args
         (append new-args
                 (filter (lambda (elt)
                           (not (member elt new-args)))
                         args))]
        [rec-rest (fragment-help (third expr-list)
                                 filtered-args
                                 name
                                 frag-counter)])
   (make-finfo
    (datum->stx false
```

```
                    (list 'define
                          (second expr-list)
                          (finfo-return rec-rest))
                    (stx-loc expr))
      (finfo-fragments rec-rest)
      (finfo-gensym rec-rest)))]

;; with an IF statement we need to recursively fragment
;; both the then and else clauses
[(equal? first-elt 'if)
 (let* ([then-info
         (fragment-help (third expr-list)
                        args name
                        frag-counter)]
        [else-info
         (fragment-help (fourth expr-list)
                        args
                        name
                        (finfo-gensym then-info))])
   (make-finfo
    (datum->stx false
                (list 'if
                      (second expr-list)
                      (finfo-return then-info)
                      (finfo-return else-info))
                (stx-loc expr))
    (append (finfo-fragments then-info)
            (finfo-fragments else-info))
    (finfo-gensym else-info)))]

;; if we have any other type of expression
;; then it cannot contain more than one call
;; to anything other than a first-order primitive
;; since everything is already in a-normal form,
```

```
          ;; so simply return it
          [else (make-finfo expr empty frag-counter)])))))


;; get-fragments: stx -> (list-of stx)
;; consumes a syntax object representing a toplevel expression
;; fragments the expression into mini-procedures and returns a list
;;    of those fragments
;; NOTE: assumes the input has already been run through anormalize
(define (get-fragments expr)
  (let* ([name (if (stx-begins-with? expr 'define)
                   (get-bound-id expr)
                   (string->symbol
                    (format statement-name (gensym))))]
         [frag-info (fragment-help expr empty name 0)])
    (reverse (cons (finfo-return frag-info)
                   (finfo-fragments frag-info)))))


;; fragment: stx:list? -> stx:list?
;; consumes a syntax object representing a program
;; returns a semantically equivalent program that has been
;;    completely a-normalized and fragmented
;; NOTE: resets the stateful gensym counter to insure it acts
;;       as a pure function
(define (fragment program)
  (begin
    (reset-gensym)
    (datum->stx false
                (apply append (map get-fragments
                                   (stx-e (anormalize program))))
                (stx-loc program))))
```

```
(provide/contract
 [fragment (stx:list? . -> . stx:list?)])
```

C.3. eliminate-anonymous.ss

```
#lang s-exp "../lang.ss"

(require "anormal-frag-helpers.ss")
(require "munge-ids.ss")
(require "../../collects/moby/runtime/stx.ss")

;; format string to give names to formerly anonymous procedures
(define anon-prepend "anon~a")

;; fold-elim-anon-help: stx:list -> linfo
;; consumes a stx:list
;; returns the result of folding elim-anon-help over
;; the elements of expr
(define (fold-elim-anon-help expr)
  (local [(define reversed-info
            (foldl (lambda (an-expr new-info)
                     (let ([rec-info (elim-anon-help an-expr)])
                       (make-linfo (cons (linfo-return rec-info)
                                         (linfo-return new-info))
                                   (append (linfo-raise new-info)
                                           (linfo-raise rec-info)))))
                   (make-linfo empty empty)
                   (stx-e expr)))]
    (make-linfo (datum->stx false
                            (reverse (linfo-return reversed-info))
                            (stx-loc expr))
                (linfo-raise reversed-info))))

;; elim-anon-help: stx -> linfo
```

```
;; consumes a syntax object with a valid expression inside
;; returns linfo where the return is the same statement
;;     but with all anonymous procedures named, and
;;     the raise is new local definitions to be placed inside
;;     the next binding form
(define (elim-anon-help expr)
  ;; if we have an atomic element, return it
  (if (stx:atom? expr)
      (make-linfo expr empty)
      ;; otherwise we have a list
      (let* ([expr-list (stx-e expr)]
             [first-elt (stx-e (first expr-list))])
        (cond

          ;; lambda expressions get lifted
          [(equal? first-elt 'lambda)
           (let ([new-proc-name
                  (string->symbol
                   (format anon-prepend (gensym)))]
                 [rec-info (elim-anon-help (third expr-list))])
             (make-linfo
              (datum->stx false new-proc-name (stx-loc expr))
              (list
               (datum->stx false
                           (list 'define
                                 (cons new-proc-name
                                       (stx-e (second expr-list)))
                                 (if (empty? (linfo-raise rec-info))
                                     (linfo-return rec-info)
                                     (list 'local
                                           (linfo-raise rec-info)
                                           (linfo-return rec-info))))
                           (stx-loc expr)))))]
```

```
;; define is a binding form, so lift everything inside it
;; then put the new lifts in a local just inside
;; (if they exist)
;; but make sure the definition uses syntactic sugar
;; for procedure definitions so we don't unnecessarily
;; name already-named procedures
[(equal? first-elt 'define)
 (let* ([sugared-def (ensugar expr)]
        [rec-info
         (elim-anon-help
          (third (stx-e sugared-def)))])
   (make-linfo
    (datum->stx false
                (list 'define
                      (second (stx-e sugared-def))
                      (if (empty? (linfo-raise rec-info))
                          (linfo-return rec-info)
                          (list 'local
                                (linfo-raise rec-info)
                                (linfo-return rec-info))))
                (stx-loc expr))
    empty))]

;; local is another binding form
;; so recursively lift both the definitions and the body
;; then add the new bindings to the list of bindings
;; on the local
[(equal? first-elt 'local)
 (let ([new-defs (map elim-anon
                      (stx-e (second expr-list)))]
       [rec-info (elim-anon-help (third expr-list))])
   (make-linfo
    (datum->stx false
                (list 'local
```

```
                                (append new-defs
                                        (linfo-raise rec-info))
                                (linfo-return rec-info))
                            (stx-loc expr))
                    empty))]

          ;; if we have a quote or define-struct, we let it be
          [(or (equal? first-elt 'quote)
               (equal? first-elt 'define-struct)
               (equal? first-elt 'require))
           (make-linfo expr empty)]

          ;; each statement is its own self-contained thing in a
          ;; begin so just map elim-anon across them
          [(equal? first-elt 'begin)
           (make-linfo (datum->stx false
                                   (map elim-anon expr-list)
                                   (stx-loc expr))
                       empty)]

          ;; otherwise fold across each sub-expression
          [else (fold-elim-anon-help expr)])))))

;; elim-anon: stx -> stx
;; consumes an expression as a syntax object and names
;; all anonymous procedures
(define (elim-anon expr)
  (let ([lifted (elim-anon-help expr)])
    (if (empty? (linfo-raise lifted))
        (linfo-return lifted)
        (datum->stx false
                    (list 'local
                          (linfo-raise lifted)
                          (linfo-return lifted))
```

```
                    (stx-loc expr)))))


;; name-anon-procs: stx -> stx
;; consumes a syntax object representing a program
;; produces a syntax object representing a semantically equivalent
;;    program, but with no anonymous procedures
;; NOTE: This procedure only exists to export a self-contained
;;       procedure that resets the state to insure it acts as a
;;       pure function. It is otherwise elim-anon
(define (name-anon-procs expr)
  (begin
    (reset-gensym)
    (elim-anon expr)))



;; lift-struct-defs: stx -> stx
;; consumes a syntax object representing a toplevel program
;; returns a semantically equivalent program with all local
;;    struct definitions raised to top leve
;; NOTE: This procedure munges identifiers to insure no collisions
;;       since it potentially increases the scope of
;;       different definitions
(define (lift-struct-defs expr)
  (let ([lifted (lift-struct-defs-help (munge-identifiers expr))])
    (datum->stx false
                (append (linfo-raise lifted)
                        (stx-e (linfo-return lifted)))
                (stx-loc expr))))



;; fold-lift-struct-defs-help: (list-of stx) -> linfo
;; consumes a list of syntax objects and folds
;; lift-struct-defs-help over the list
;; producing linfo where the return is the lifted return
```

```
;; and the raise is all the raises appended together
(define (fold-lift-struct-defs-help expr-list)
  (foldr (lambda (expr info)
           (let ([output (lift-struct-defs-help expr)])
             (make-linfo (cons (linfo-return output)
                               (linfo-return info))
                         (append (linfo-raise output)
                                 (linfo-raise info)))))
         (make-linfo empty empty)
         expr-list))


;; lift-struct-defs-help: stx -> linfo
;; consumes a syntax object representing an expression
;; produces an linfo where the return is a syntax object
;;     representing the same expression, but missing all
;;     local struct definitions and the raise is
;;     a list of removed struct definitions
(define (lift-struct-defs-help expr)
  ;; we have an atomic element, there cannot be
  ;; any local struct definitions so return the element
  (if (stx:atom? expr)
      (make-linfo expr empty)
      ;; otherwise we have a list
      (let* ([expr-list (stx-e expr)]
             [first-elt (stx-e (first expr-list))])
        (cond
          ;; if we have a local, we need to get
          ;; the local struct definitions
          [(equal? first-elt 'local)
           (let ([struct-defs
                  (filter (lambda (a-def)
                            (equal? (stx-e (first (stx-e a-def)))
                                    'define-struct))
```

```
                    (stx-e (second expr-list)))]
            [other-defs
             (fold-lift-struct-defs-help
              (filter (lambda (a-def)
                        (equal? (stx-e (first (stx-e a-def)))
                                'define))
                      (stx-e (second expr-list))))])
       (make-linfo
        (datum->stx false
                    (if (empty? (linfo-return other-defs))
                        (third expr-list)
                        (list 'local
                              (linfo-return other-defs)
                              (third expr-list)))
                    (stx-loc expr))
        (append struct-defs
                (linfo-raise other-defs))))]
  ;; if we see quote, define-struct (outside a local),
  ;; or require; then don't touch anything
  [(or (equal? first-elt 'quote)
       (equal? first-elt 'define-struct)
       (equal? first-elt 'require))
   (make-linfo expr empty)]
  ;; otherwise recursively fold over each element
  [else (let ([folded-list
               (fold-lift-struct-defs-help expr-list)])
          (make-linfo (datum->stx false
                                  (linfo-return folded-list)
                                  (stx-loc expr))
                      (linfo-raise folded-list)))])))



(provide/contract
```

```
 [name-anon-procs (stx? . -> . stx?)]
 [lift-struct-defs (stx? . -> . stx?)])
```

C.4. box-local-defs.ss

```
#lang s-exp "../lang.ss"

(require "anormal-frag-helpers.ss")
(require "elim-anon.ss")
(require "../../collects/moby/runtime/stx.ss")

;; unbox-ids: stx (list-of symbol) -> stx
;; consumes a symbolic expression and a list of identifiers to
;; unbox returns the symbolic expression with each instance of
;; the identifier wrapped in an unbox
(define (unbox-ids expr ids)
    (let ([contents (stx-e expr)])
      (cond
        [(symbol? contents) (if (member contents ids)
                                (datum->stx false
                                            (list 'unbox expr)
                                            (stx-loc expr))
                                expr)]
        [(cons? contents) (datum->stx false
                                      (map (lambda (an-expr)
                                             (unbox-ids an-expr ids))
                                           contents)
                                      (stx-loc expr))]
        [else expr])))

;; box-locals: stx -> stx
;; consumes a syntax object
;; returns a semantically equivalent expression with
;;   local value definitions created using set-box!
```

```
(define (box-locals expr)
  ;; atomic expressions cannot contain locals, so return expr
  (if (stx:atom? expr)
      expr
      ;; otherwise we have a list
      (let* ([expr-list (stx-e expr)]
             [first-elt (stx-e (first expr-list))])
        (cond
          ;; if we have a local statement
          ;; separate the definitions into explicit
          ;;   lambda expressions and everything else
          ;; change the "everything else" to be defined as
          ;;    (box 'undefined) and set the value using
          ;;    set-box! inside a begin immediately after
          ;;    the definition list
          ;; NOTE: the box of undefined and set-box! works
          ;;       for explicit procedures, but it is not
          ;;       necessary since nothing is evaluated
          ;;       during the definition, so the definitions
          ;;       can all be in the same fragment later
          [(equal? first-elt 'local)
           (let* ([sugared-defs
                   (map ensugar
                        (stx-e (second expr-list)))]
                  [old-val-defs
                   (filter (lambda (a-def)
                             (stx:atom? (second (stx-e a-def))))
                           sugared-defs)]
                  [val-ids
                   (map (lambda (an-expr)
                          (stx-e (second (stx-e an-expr))))
                        old-val-defs)]
                  [boxed-val-defs (map box-locals old-val-defs)]
                  [old-fun-defs
```

```
        (filter (lambda (a-def)
                  (stx:list? (second (stx-e a-def))))
                sugared-defs)]
       [boxed-fun-defs
        (unbox-ids
         (datum->stx false
                     (map box-locals old-fun-defs)
                     (stx-loc (second expr-list)))
         val-ids)])
(datum->stx
 false
 (list 'local
       ;; definitions list is old procedure definitions
       ;; followed by boxes of undefineds for
       ;; the value definitions
       (datum->stx false
                   (append (stx-e boxed-fun-defs)
                           (map (lambda (symb)
                                  `(define ,symb
                                     (box 'undefined)))
                                val-ids))
                   (stx-loc (second expr-list)))
       ;; then use being and set-box! to set
       ;; the values of the undefined definitions
       ;; (not needed if none were there)
       (if (empty? boxed-val-defs)
           (box-locals (third expr-list))
           (cons 'begin
                 (foldr (lambda (a-def rest-expr)
                          (cons
                           (list 'set-box!
                                 (second (stx-e a-def))
                                 (unbox-ids
                                  (third (stx-e a-def))
```

```
                                                val-ids))
                                               rest-expr))
                                        (list (unbox-ids
                                               (box-locals
                                                (third expr-list))
                                               val-ids))
                                        boxed-val-defs))))
                    (stx-loc expr)))]
          ;; if we have a quote, define-struct,
          ;; or require statement, leave it alone
          [(or (equal? first-elt 'quote)
               (equal? first-elt 'define-struct)
               (equal? first-elt 'require))
           expr]
          ;; otherwise map a recursive call across
          ;; each sub-expression
          [else (datum->stx false
                            (map box-locals expr-list)
                            (stx-loc expr))])))

;; ready-anormalize: stx -> stx
;; consumes a syntax object representing a toplevel program
;; produces a semantically equivalent program that, once ANFed
;;     will be ready for fragmentation (identifiers munged, local
;;     struct definitions lifted to top level, no anonymous
;;     procedures, and local value definitions made
;;     using boxes and set-box!)
(define (ready-anormalize expr)
  (box-locals (datum->stx false
                          (map name-anon-procs
                               (stx-e (lift-struct-defs expr)))
                          (stx-loc expr))))
```

```
(provide/contract
 [ready-anormalize (stx:list? . -> . stx:list?)])
```

C.5. munge-identifiers.ss

```
#lang s-exp "../lang.ss"



(require "anormal-frag-helpers.ss")
(require "../rbtree.ss")
(require "../../collects/moby/runtime/stx.ss")

;; format strings to modify symbols
;; to insure unique identifiers later
(define def-prepend "d~a_~a")
(define arg-prepend "a_~a")
(define struct-prepend "s~a_~a")

;; symbol<: symbol symbol -> boolean
(define (symbol< x y)
  (string<? (symbol->string x)
            (symbol->string y)))

;; get-id-tree:
;;         s-expr (tree-of symbol.symbol)->(tree-of symbol.symbol)
;; consumes a symbolic expression and a tree of replacements
;; returns a tree containing all bindings of the old tree
;;  as well as new bindings representing replacements for anything
;;  defined at toplevel relative to the inputted s-expr
(define (get-id-tree expr base-tree)
  (if (not (cons? expr))
      base-tree
      (cond
        [(equal? (first expr) 'define-struct)
```

```
        (let* ([orig-procs (get-struct-procs expr)]
               [new-name (string->symbol (format struct-prepend
                                                 (gensym)
                                                 (second expr)))]
               [new-procs (get-struct-procs (list 'define-struct
                                                  new-name
                                                  (third expr)))])
          (foldl (lambda (old-proc new-proc a-tree)
                   (rbtree-insert symbol< a-tree
                                  old-proc
                                  new-proc))
                 (rbtree-insert symbol< base-tree
                                (second expr)
                                new-name)
                 orig-procs
                 new-procs))]
       [(equal? (first expr) 'define)
        (let ([name (if (cons? (second expr))
                        (first (second expr))
                        (second expr))])
          (rbtree-insert symbol< base-tree name
                         (string->symbol
                          (format def-prepend
                                  (gensym) name))))]
       [else base-tree])))

;; replace-ids: stx (tree-of symbol . symbol) -> stx
;; consumes a syntax object and a tree of replacements
;; returns a new syntax object with all identifiers munged
;;     if they are in the replacement tree or if they are bound
;;     inside the current expression
(define (replace-ids expr replacements)
  (cond
    ;; if we have an identifier we might need to replace it
```

```
[(symbol? (stx-e expr))
 (if (false?
      (rbtree-lookup symbol<
                     replacements
                     (stx-e expr)))
     expr
     (datum->stx false
                 (second
                  (rbtree-lookup symbol<
                                 replacements
                                 (stx-e expr)))
                 (stx-loc expr)))]
;; if we have a list, check the first element of the list
[(stx:list? expr)
 (let* ([expr-list (stx-e expr)]
        [first-elt (stx-e (first expr-list))])
   (cond
     ;; on a define or lambda, get the arguments
     ;; and recursively replace body elements
     [(or (equal? first-elt 'define)
          (equal? first-elt 'lambda))
      (let* ([new-args
              (if (equal? first-elt 'lambda)
                  (stx->datum (second expr-list))
                  (if (stx:list? (second expr-list))
                      (rest (stx->datum (second expr-list)))
                      empty))]
             [new-replacements
              (foldl (lambda (symb a-tree)
                       (rbtree-insert symbol<
                                      a-tree
                                      symb
                                      (string->symbol
                                       (format arg-prepend
```

```
                                                    symb))))
                    replacements
                    new-args)])
    (datum->stx false
                (list (first expr-list)
                      (replace-ids (second expr-list)
                                   new-replacements)
                      (replace-ids (third expr-list)
                                   new-replacements))
                (stx-loc expr)))]
;; with local, get the bindings from the defines
;; recursively replace the defines and the body
;; using those new replacements
[(equal? first-elt 'local)
 (let ([new-replacements
        (foldl get-id-tree
               replacements
               (stx->datum (second expr-list)))])
    (datum->stx false
                (list (first expr-list)
                      (replace-ids (second expr-list)
                                   new-replacements)
                      (replace-ids (third expr-list)
                                   new-replacements))
                (stx-loc expr)))]
;; if define-struct, we only want to munge the
;; struct name
[(equal? first-elt 'define-struct)
 (datum->stx false
             (list (first expr-list)
                   (replace-ids (second expr-list)
                                replacements)
                   (third expr-list))
             (stx-loc expr))]
```

```
        ;; if quote or require, leave it alone
        [(or (equal? first-elt 'quote)
             (equal? first-elt 'require))
         expr]
        ;; otherwise map over the entire expression
        [else (datum->stx false
                          (map (lambda (an-expr)
                                 (replace-ids an-expr
                                              replacements))
                               expr-list)
                          (stx-loc expr))]))]
    ;; if neither an identifier nor a cons, do nothing
    [else expr]))

;; munge-identifiers: stx:list -> stx:list
;; consumes a program as a syntax object
;; returns the same program with all identifiers bound
;;    in that program munged using a gensym counter
;; NOTE: since the gensym counter is stateful, this resets it,
;;       insuring it acts as a pure function
(define (munge-identifiers expr)
  (begin
    (reset-gensym)
    (replace-ids expr (foldl get-id-tree
                             empty-rbtree
                             (stx->datum expr)))))


(provide/contract [munge-identifiers
                   (stx:list? . -> . stx:list?)])
```

C.6. helpers.ss

```
#lang s-exp "../lang.ss"
```

```
(require "../../collects/moby/runtime/stx.ss")

;; lift information (linfo) contains
;;    - return : stx (used as (list-of stx) when folding)
;;               that is the new expression
;;    - raise : (list-of stx)
;;              representing spliced out (or new) expressions
;;              to be raised to a higher level
(define-struct linfo (return raise))

;; stateful gensym operations including a counter,
;; a thunk that adds 1 and returns the old value,
;; and a thunk to reset to 0
;; only the thunks are exported
(define gensym-counter 0)
(define (gensym)
  (begin (set! gensym-counter (add1 gensym-counter))
         (sub1 gensym-counter)))
(define (reset-gensym)
  (set! gensym-counter 0))

;; list-of: (any -> boolean) -> (any -> boolean)
;; consumes a predicate and returns a new predicate that checks
;;    if its input is a list such that each element satisfies
;;    the original predicate
(define (list-of pred)
  (lambda (dat)
    (and (list? dat)
         (andmap pred dat))))

;; s-expr?: datum -> boolean
;; consumes anything and returns true if it is an s-expression,
;; false otherwise
```

```
(define (sexp? expr)
  (or (string? expr)
      (symbol? expr)
      (number? expr)
      (boolean? expr)
      (char? expr)
      ((list-of sexp?) expr)))

#|
;; symb-prepend: string symbol -> symbol
;; consumes a prepend string and an original symbol
;; returns a new symbol with the prepend string prepended
;; to the original
(define (symb-prepend prepend symb)
  (string->symbol (string-append prepend (symbol->string symb))))
|#

;; ensugar: stx -> stx
;; takes a define statement as a syntax object
;; produces a semantically equivalent define statement that is
;; guaranteed to use syntactic sugar if defining a procedure
(define (ensugar a-def)
  (if (stx-begins-with? a-def 'define)
      (let ([stx-list (stx-e a-def)])
        (if (and (stx:atom? (second stx-list))
                 (stx-begins-with? (third stx-list) 'lambda))
            (datum->stx false
                        (list (first stx-list)
                              (cons (second stx-list)
                                    (stx-e
                                     (second
                                      (stx-e (third stx-list)))))
                              (third (stx-e (third stx-list))))
                        (stx-loc a-def))
```

```
            a-def))
      (error 'ensugar
             (format "expected definition as syntax, found: ~a"
                     a-def))))



;; get-struct-procs: s-expr -> (list-of symbol)
;; consumes a struct definition in abstract syntax
;; returns a list of procs generated by defining that struct
(define (get-struct-procs struct-def)
  (list* (string->symbol (format "make-~a" (second struct-def)))
         (string->symbol (format "~a?" (second struct-def)))
         (foldl (lambda (elt rest-procs)
                  (list* (string->symbol (format "~a-~a"
                                                 (second struct-def)
                                                 elt))
                         (string->symbol (format "set-~a-~a!"
                                                 (second struct-def)
                                                 elt))
                         rest-procs))
                empty
                (third struct-def))))



(provide/contract
 [struct linfo ([return (or/c stx? (list-of stx?))]
                [raise list?])]
 [gensym ( -> number?)]
 [reset-gensym ( -> void?)]
 [sexp? (any/c . -> . boolean?)]
 [ensugar (stx? . -> . stx?)]
 [get-struct-procs (sexp? . -> . (list-of symbol?))])
```

Bibliography

Aczel, P. (1977). An introduction to inductive definitions. In J. Barwise (Ed.), *HANDBOOK OF MATHEMATICAL LOGIC*, Volume 90 of *Studies in Logic and the Foundations of Mathematics*, pp. 739–782. Elsevier.

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers: Principles, Techniques, Tools*. Addison-Wesley.

Backus, J. W., F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger (1963). Revised report on the algorithm language algol 60. *Commun. ACM 6*(1), 1–17.

Baker, H. G. (1994). Cons should not cons its arguments, part ii: Cheney on the m.t.a. *Draft Version*.

Bancerek, G. (2003). The fundamental properties of natural numbers. *Journal of Formalized Mathematics 1*.

Barendregt, H. P. (1985). *The Lambda Calculus, Its Syntax and Semantics*. North Holland.

Dijkstra, E. W. (1960). Recursive programming. *Numerische Mathematik 2*, 312–318.

Felleisen, M., R. B. Findler, and M. Flatt (2009). *Semantics Engineering with PLT Redex*. MIT Press.

Felleisen, M., R. B. Findler, M. Flatt, and S. Krishnamurthi (2001). *How To Design Programs*. The MIT Press.

Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen (1993). The essence of compiling with continuations. *Conference on Programming Language Design and Implementation*, 237–247.

Flatt, M. (1996-2005). *PLT MzScheme: Language Manual*.

Friedman, D. P. and M. Wand (1984). Reification: Reflection without metaphysics. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, New York, NY, USA, pp. 348–355. ACM.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1994). Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*.

Hughes, J. (1989). Why functional programming matters. *Comput. J. 32*(2), 98–107.

Knuth, D. E. (1997). *The Art of Computer Programming*. Addison-Wesley.

Krishnamurthi, S. (2007). *Programming Languages Application and Interpretation*. Shriram Krishnamurthi.

Landin, P. J. (1964). The mechanical evaluation of expressions. *The Computer Journal*.

Loitsch, F. and M. Serrano (2006). Compiling scheme to javascript. *ICFP*.

Mano, M. M. and C. R. Kime (2001). *Logic and Computer Design Fundamentals*. Prentice-Hall.

McCarthy, J. A. (2009). Automatically restful web applications: marking modular serializable continuations. *SIGPLAN Not. 44*(9), 299–310.

Moses, J. (1970). The function of function in lisp or why the funarg problem should be called the environment problem. *SIGSAM Bull.* (15), 13–27.

Naur, P. (1963). The design of the gier algol compiler part i. *BIT Numerical Mathematics 4*, 124–140.

Pettyjohn, G., J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen (2005). Continuations from generalized stack inspection. *International Conference on Functional Programming*.

Reynolds, J. C. (1993). The discoveries of continuations. *LISP AND SYMBOLIC COMPUTATION: An International Journal*.

Sandewall, E. (1971). A proposed solution to the funarg problem. *SIGSAM Bull.* (17), 29–42.

Schanzer, E. and PLT-RacketTeam. The bootstrap program.

Schinz, M. and M. Odersky (2001). Tail call elimination on the java virtual machine. *Proceedings of Babel&aps01*.

Sipser, M. (2006). *Introduction To The Theory Of Computation*. Course Technology, Cengage Learning.

Sperber, M., R. K. Dybvig, M. Flatt, A. van Straaten, R. B. Findler, and J. Matthews. Revised6 report on the algorithmic language scheme.

Sussman, G. J. and G. L. Steele (1975). Scheme: An interpreter for extended lambda calculus. *MIT AI Lab*.

Tarditi, D., P. Lee, and A. Acharya (1990). No assembly required: Compiling stan-

dard ml to c. Technical report, School of Computer Science, Carnegie Mellon University.

TheApacheTeam. Apache http server project.

Turbak, F. and D. Gifford (2008). *Design Concepts in Programming Languages*. MIT Press.

Yoo, D., Z. Zhang, E. Cecchetti, B. Hickey, S. Krishnamurthi, C. Derici, and N. Zimmt. Wescheme environment.