University of Edinburgh
School of Informatics

Compiling Links Server-Side Code

4th Year Project Report
Computer Science

Steven Holmes

April 1, 2009

Abstract: Links is a functional programming language for creating web applications. From a single source program Links produces Javascript for the client, SQL for database access, and an interpreted intermediate language for the server. This report describes the implementation of a compiler for the Links intermediate language that targets OCaml. The goal is to improve the performance of server-side code. The compiler is evaluated and found to improve performance significantly over the interpreter. In addition, the report describes a new algorithm for efficiently boxing values that was developed as part of the compiler.

Acknowledgements

I would like to thank my supervisors Sam Lindley and Phil Wadler for their extensive help and guidance.

Contents

1. Introduction

Links[1] is a language that is designed to make web programming easier. Web programs usually consist of three tiers: the client front-end, the server in the middle, and a database back-end. This causes problems, as all three tiers are programmed using different languages with different styles. Links allows programmers to write code in a single functional language for all three tiers. From a single source program Links produces Javascript for the client, SQL for database access, and an interpreted intermediate language for the server.

This report describes the development of a compiler for the Links intermediate language that targets OCaml. Currently Links is interpreted on the server, and this is slow, particularly for CPU-intensive algorithmic code. The primary purpose of the compiler is to improve the speed at which Links can run on the server, making it more practical to write computationally expensive server-side code.

1.1 What Has Been Done

A compiler from the Links intermediate language (*IR*) to OCaml[2] has been implemented, along with a small library of primitive functions and a run-time system.

The compiler currently lacks web 2.0-style Javascript to server function calls, SQL query generation and concurrency. However, it can successfully compile programs that do not use these features. Compiled programs can be run either interactively or as CGI programs under a webserver.

Additionally, a new boxing optimisation was designed during development of the compiler. This is intended to efficiently add boxing to compiled code.

1.2 Report Structure

The report is split into several chapters. The first gives some background about the project in order to place the undertaken work in context. This includes describing the Links language, its current implementation, and a brief introduction to various concepts and ideas upon which the project relies.

The second chapter describes a simple pretty-printer for the intermediate representation used by the Links, along with an example of pretty-printed output.

The third chapter introduces the compiler, first giving a high-level description followed by more in-depth details. Details of the translation are given, followed by a descriptions of how values are boxed and represented. The primitives library and run-time system are then presented.

The fourth chapter introduces a boxing optimisation. It describes the algorithm at a high level.

The fifth chapter evaluates the compiler by benchmarking it against the interpreter. It compares the execution speed of compiled and interpreted code and shows that the project goal of improving performance has been achieved.

2. Background

2.1 Notation

To illustrate concepts, code samples are dotted throughout the report. Links and OCaml code are typeset using a monospace font.

```
let square x = x * x
```

Examples involving the IR and target languages defined in Figures 4.1 and 4.2 are typeset like so

$$\textbf{letrec}\ square = x * x$$

and algorithms operating on IR and target terms are typeset using pattern-matching. Vector notation, $\vec{x}$ is used to denote lists. The *cons* operator for lists is written as $x_{new} :: \vec{x}$, and this same notation is used for pattern matching against the head and tail of a list. A list of elements is written as $[x_1, x_2, x_3]$. The empty list is written []. This notation for lists is occasionally abused by leaving out the square brackets when it is obvious what is going on.

2.2 Links the Language

2.2.1 Web Programming

Web programming is traditionally split into three tiers. These are the front-end, usually written as a mixture of HTML and Javascript, the middle tier, written in a server-side language such as PHP, Java, Python or Ruby, and the back-end, which is normally a database queried with SQL.

This proliferation of languages and mindsets can cause problems. Effective web programmers must be proficient in several paradigms and must continually switch between them. It is also difficult to share code between tiers, which can lead to code duplication and an unnatural program structure.

Furthermore, the three tiers must communicate. This often involves the programmer juggling a myriad of different types between languages, usually writing the tedious interfacing and conversion code by hand. It can also involve attempting to reconcile fundamentally different views of the world—for instance, cramming SQL's relational view of data into the object-oriented view of Java or Python.

Links is a language designed to overcome these problems. It allows programmers to write code for all three tiers in a single typed functional language. It compiles to Javascript and HTML for the client, and to an interpreted intermediate language with automatically generated SQL on the server. Communication between the tiers is handled transparently and in a type-safe way.

2.2.2 CGI

CGI[3] is a standard interface that allows web servers to execute external programs in order to service a request from a client. The server communicates with the external program via a set of *environment variables* that give details of the request. *CGI parameters* can be passed as part of the environment. These are a set of name-value pairs provided by the client request, usually representing a form post. A CGI program may inspect these parameters along with the rest of the environment, and may consult databases, read files etc. before deciding how to respond. The response is written to standard output and is then delivered to the client.

CGI is the most widely supported method for providing dynamic web pages and web applications. Both compiled and interpreted Links programs run with CGI.

2.2.3 Database Queries

Links allows the programmer to write database queries in a functional style, using effect annotations in the type-system to guarantee statically that code intended for the database compiles into efficient SQL.

Database querying has recently undergone an overhaul. A new system of typing and rewrite rules is now used that allows Links to produce efficient SQL queries while at the same time supporting the high-level abstractions of functional programming.

This system requires significant run-time support. It it is not possible to statically compile all queries into SQL, so a query compiler must be embedded into the Links run-time system. This is not unlike the Microsoft .NET LINQ[4] system.

Due to this requirement for run-time support, implementing querying in the compiler would be a substantial piece of work. Therefore, no attempt was made to include SQL generation.

2.2.4 Concurrency

Links has an implementation of concurrency. The programmer is able to spawn new processes and send messages between them using a mailbox system.

Links implements concurrency in a serialised way: Occasionally the running process is suspended and a new process takes its place.

The compiler supports the mechanisms required for concurrency, but the run-time system currently lacks support. This would be fairly straightforward to add.

2.2.5 AJAX calls

In Links, functions are annotated to say whether they should be run on the client or on the server (or both). Functions which should be run on the client are compiled to javascript. Links allows client functions to seamlessly call functions on the server and vice-versa. This functionality takes advantage of modern browsers' *XMLHTTPRequest* ability, which allows Javascript code to make asynchronous HTTP requests to fetch data from a server. This data can then be used to update pages in-place rather than requiring a browser refresh.

These asynchronous requests are known as *AJAX* requests, and are a common component of many *web 2.0*-style web applications. In traditional sites, the browser makes a single request per page. Web 2.0 sites generally act in a more dynamic, event-driven way, and the ability to update content dynamically allows web 2.0 sites to more closely emulate desktop applications.

To use AJAX, normally an explicit HTTP request is made with Javascript to a resource on the server. This can involve a lot of interface and type-conversion code. The fact that Links can eliminate this tedium with its ability to seamlessly call functions on the server from the client and vice-versa means it is particularly well-suited for programming web 2.0 applications.

The compiler currently lacks support for AJAX calls. The mechanics are implemented, but the necessary interfacing code between the Javascript client code and the compiled server code does not yet exist.

2.3 The Current Links Implementation

To serve web pages, Links runs as a CGI program. Upon invocation, a Links source program is read and compiled into an intermediate representation (IR) tree. The client portions of the code are then compiled to Javascript by a

Javascript compiler, and the server-side portion is run by directly interpreting the IR tree. Any output is then passed to the browser.

Links also supports being run from the command line. Given an input file, it is treated as a program, and the server-side component is executed immediately. The value this returns is then printed.

Links also supports running in a read-eval-print loop that allows expressions to be entered and evaluated immediately.

2.4 Links and Continuations

2.4.1 Continuations

Conceptually, a continuation represents "the rest of a computation"—that part of a computation that control should pass to next. In this way, continuations are a representation of a program's current state—they capture what has been done so far, and what still needs to be done.

In some programming languages, Links included, continuations are first-class. The current continuation at any point can be *captured* and bound to a variable. It can then be passed around like any other variable.

Captured continuations can be used as a powerful form of control flow. When a continuation is *invoked*, execution jumps to the point and state at which the continuation was captured. In Links, continuations are captured using the *escape* construct, which binds the current continuation to a variable. Continuations are represented as single-argument functions, and applying them invokes the continuation. The argument becomes the value returned by the escape expression.

The function in Figure 2.1 shows how to perform non-local escape using an explicit continuation. It returns **true** if the passed-in list contains all true values, otherwise it returns **false**.

The continuation is bound to the variable *esc* inside the escape construct. The function that is mapped over the list will invoke the continuation with argument **false** if it finds that the current item is **false**. This causes execution to immediately continue just after the escape expression, with *ret* bound to **false**. If the *map* call completes, the continuation was not invoked, and so there are no **false** values in the list and **true** is returned.

```
fun all_true(l) {
    var ret = escape esc in {
        var l = map(
            fun (x) {
                if (not(x)) esc(false) else true
            },
        l);
        true
    };
    ret
}
```

Figure 2.1: A function demonstrating the use of continuations for non-local escape.

2.4.2 How Continuations Are Used in Links

Concurrency in Links is implemented with continuations. Each process can be represented by its continuation, and every so often the current process is "swapped out", its continuation being swapped with that of a waiting process.

Continuations are also used for client-server communication. When a client function is called from the server (or vice-versa), the current continuation is saved behind the scenes and the call is made on the next client request. When (and if) a result is obtained, the stored continuation is invoked, allowing the caller to continue with the result of the function call. This hides the stateless nature of HTTP and makes client-server or server-client calls appear to the programmer like normal, stateful function application.

A similar idea is used for form processing. *Formlets*[5], the standard means for abstracting web forms in Links, operate by saving the current continuation when rendering a form and then invoking that continuation when the form is posted in order to process the data.

2.4.3 Serialized Continuations

The idea of using continuations to elegantly handle the stateless nature of HTTP is not new[6][7][8]. However, most implementations store the server-side continuations in memory between client requests. Continuations are quite heavy-weight, and many may need to be stored, which can place strain on server resources. It can also affect user experience—continuations will be discarded from memory at some point, often after a certain time of client inactivity. This can cause pages to become invalid. Storing continuations in this way also impedes scalability. In

order to scale an application beyond a single server, techniques for sharing continuations amongst servers must be implemented. This can become a bottleneck.

Links solves these problems by *serialising* continuations into a string and then storing them on the client. With each request, the client sends a serialised continuation, which is then deserialised and applied. This means the server does not need to store any state at all, which allows trivial scaling to any number of web servers.

2.5 ANF and CPS

Continuation-passing style[9], or CPS, is a style of programming in which continuations are passed around explicitly. Continuations are represented as functions, and instead of an expression returning a value, as in direct-style, it calls its continuation function with the result, which then continues with the computation.

CPS is sometimes used as the intermediate representation in compilers. One of the reasons is that it guarantees that all arguments to functions are trivial—either variables or constants, never more complicated expressions. This can simplify program transformations.

The compiler's target language is in CPS form because it gives explicit access to the current continuation. It is therefore easy to make the continuation available to programs as a first-class value, as required by Links. The current continuation is made available to the program via a construct **call/cc** (call with current continuation) which, given a function, calls it passing in the current continuation, thus making it available to the programmer.

In CPS, all function calls are tail-calls. This is because "what to do next" is explicitly passed to the function as its continuation.

A Normal Form (ANF) is another restricted form for programs, and is direct-style. ANF[10] was created to try and keep only those aspects of CPS that are helpful for compilation. For instance, in ANF, like in CPS, arguments to a function must be simple, and as a result intermediate values used during computation must be named.

The ANF restrictions can often make optimisations such as data-flow analysis easier, and ANF is in fact closely related to an intermediate language used for optimisation in imperative-language compilers called SSA[11].

2.5.1 The Links Prelude

In addition to primitives, a set of functions written in Links is made available to every program. This serves the purpose of a standard library, and is called the Prelude.

2.6 The Compiler and Where It Sits

2.6.1 Choice of Target

The choice of target for a compiler is important. It determines how, where and how efficiently generated code can run, what tools are available, and how easy it is to gain access to foreign libraries and interfaces.

Several possibilities for the target language were considered.

2.6.1.1 C--

C--[12] is a C-like language designed to be targeted by compilers. It aims to be a portable assembly language that

> should serve as the interface between high-level compilers and retargetable, optimizing code generators.[1]

It is slated to become the default target for the GHC Haskell compiler at some point. However, at present it is rather bare-bones: it does not do much optimisation, and supports generating machine-code for only one architecture.

2.6.1.2 LLVM

LLVM[13] is a a compiler infrastructure that is designed around optimisation of code. It provides a language and machine independent typed instruction set for compilers to target. The target language is low-level, and while compiling with LLVM would be interesting, the difficulties inherent in compiling to a low-level instruction set would significantly complicate the compiler.

[1] Taken from the C-- home page at http://www.cminusminus.org/

2.6.1.3 The JVM

The Java Virtual Machine (JVM) provides a stack-based object-oriented virtual machine with garbage-collection.

However, the JVM lacks tail-call elimination, which is essential for a functional language like Links. It is possible to emulate tail-call elimination, but this can be inefficient and awkward[14].

Parametric polymorphism is also not directly supported, and shoe-horning a functional language with features such as polymorphic variant types into an object-oriented target can be clumsy[15]. These problems can (and have) been solved, and there exist a number of functional languages targeting the JVM[16].

2.6.1.4 .NET

Like the JVM, Microsoft's .NET's virtual machine (known as the CLR) is stack-based. It also is the target of several functional programming languages[17]. It is less explicitly object-oriented than the JVM and supports tail-call optimisation[18].

There exists an extended assembly language for the CLR called ILX[19]. This aims to make it easier to target functional programming languages at the CLR, and provides features like first-class functions, parametric polymorphism and variants (although not polymorphic variants).

2.6.1.5 OCaml

OCaml is a functional language featuring a high-performance optimising native compiler that can target many architectures.

OCaml is also the basis for F#, a functional language for Microsoft's .NET platform. Both languages share a common core, which makes it easy to produce code that compiles with either (for example, the F# compiler itself is written so as to compile as either OCaml or F#). Targetting both F# and OCaml gives flexibility and portability, allowing Links code to run natively on many architectures (via OCaml), or to run on a well-established "enterprise" system already in use by many organisations (via F# and .NET).

Compiling a functional language to another functional language is also significantly simpler than targetting an imperative language. For these reasons, it was decided the compiler would target OCaml.

The compiler produces code that will compile as F#. However, the run-time system in a couple of places (unavoidably) uses features incompatible with F#

(specifically functors). This should be straightforward to fix as F# has conditional compilation that can compile different blocks of code depending on whether OCaml or F# is being used.

The only esoteric feature that compilation relies on is the ability to serialise closures (this is needed to serialise continuations). It has been checked that F# supports this.

2.6.2 How The Compiler Fits In

The Links compiler uses the same front-end as the interpreter. The front-end does parsing, type-checking and desugaring of Links code into an IR tree. This IR tree is then fed into the compiler, which compiles it to a simple untyped language which is pretty-printed into OCaml.

The compiler code is essentially independent of most of the rest of the Links implementation, relying only on the IR format and various simple utilities.

It may have been possible to re-use other parts of the code, particularly the Javascript compiler, but a decision was taken early on that the compiler code should be as independent as practicable. Links was (and still is) evolving rapidly, and making the compiler code independent insulated its development from most of the rest of the ongoing development. This meant less time and frustration was spent integrating changes required by the compiler with the rest of the code and vice-versa.

3. The Pretty Printer

To aid understanding of the intermediate representation (IR), as well as to facilitate debugging, a pretty-printer was developed for IR trees.

Developing the pretty-printer required a good understanding of the IR and its scoping rules, so it was well worthwhile as a preparation for writing the compiler.

3.0.3 Implementation

The pretty-printer simply traverses the IR and uses a set of pretty-printing combinators[20] to build a textual representation of the tree.

At present it does not print typing information, as there was no pressing need for it to do so. There is already code for pretty-printing types inside Links, and this could be re-used or adapted for the pretty-printer if it was required in the future.

3.0.4 An Example

A simple Links function is given in Figure 3.1. The output the pretty-printer produces on its IR tree is shown in Figure 3.2

Variables are printed as "name/ID" pairs. The name is the human-readable variable name (if one exists—if not one is generated from the ID by prepending an underscore) paired with the internal integer representation of the variable.

Infix operators are printed in prefix style. This is to emphasise their syntactic structure within the IR and because in the IR function application and operator application are treated the same way.

Bindings are grouped before the computation they are used in, as they are in the IR.

```
fun factorial(n) {
   if (n <= 1)
      1
   else
      n * factorial(n - 1)
}
factorial(10)
```

Figure 3.1: A simple Links function to calculate the factorial of a number n

```
let
  rec
    fun factorial/1467 __1466/1466 =
      let
        val n/1465 = __1466/1466
      in
        if
          (<=/16 n/1465 1)
        then
          1
        else
          let
            val __1468/1468 = factorial/1467 (-/2 n/1465 1)
          in
            (*/3 n/1465 __1468/1468)
  end
in
  factorial/1467 10
```

Figure 3.2: The result of pretty-printing the IR representation of the code in 3.1

4. The Compiler

4.1 Overview

The compiler is invoked via a function that is called from main Links file. This takes an IR tree and returns a formatted string containing the compiled OCaml code.

The compiler walks the IR and translates it into a much simpler untyped target language tree. It can produce code in either CPS or direct-style. Boxing of values is performed during translation.

OCaml code is then easily generated by walking and pretty-printing the target language tree. This code is returned as a string.

4.2 The IR

The Links IR is in ANF. To help illustrate ideas, a simplified version, detailed in Figure 4.1, will be used in this report:

The important constructs are *computations* M, *tail computations* P, *bindings* B and *values* V. Branches are not tail-computations, but are labeled as such in the compiler IR. This is for convenience as it simplifies the front-end's transformation of Links code into ANF.

A *computation* consists of a list of bindings along with a tail computation P. Each binding is in scope for the bindings that follow it, and also for P. A *value bind* binds a variable x to a tail computation. A *function bind* binds a list of mutually recursive functions. Each function in the list is in scope for every other. A *return* simply returns a value. A *branch* first tests its condition value V. If this is **true**, it executes the true branch M_1, otherwise it executes the false branch, M_2. A *call/cc* calls its value V, which should be a function, passing in the current continuation. A *program* is a closed computation (one that does not have any free variables).

Variable names in the IR are unique—that is, any variable name is bound in at most a single place.

The structure of this simple language follows closely the structure of the actual IR used in the compiler, but for ease of explanation omits many constructs.

$$
\begin{array}{lcll}
M & ::= & \textbf{let } \vec{B} \textbf{ in } P & \textit{(computation)} \\
\\
B & ::= & \textbf{val } x \ = \ P & \textit{(value bind)} \\
 & | & \vec{F} & \textit{(function bind)} \\
\\
P & ::= & \textbf{return } V & \textit{(return)} \\
 & | & V(\vec{V}) & \textit{(call)} \\
 & | & \textbf{if } V \textbf{ then } M_1 \textbf{ else } M_2 & \textit{(branch)} \\
 & | & \textbf{call/cc } V & \textit{(call/cc)} \\
\\
F & ::= & \textbf{fun } x(\vec{x}) \ = \ M & \\
\\
V & ::= & x \mid c & \\
\\
x & ::= & \text{variable} & \\
c & ::= & \textbf{true} \mid \textbf{false} \mid \text{integer} &
\end{array}
$$

Figure 4.1: The simple IR used throughout the report to illustrate concepts.

For instance, the actual IR supports several more value types, like polymorphic variants, lists, records and XML nodes, and allows control structures like case.

In Links, the IR is typed, but the compiler ignores types as they are not required during translation.

4.3 Pure Application

A feature of the compiler IR is "pure application". This is an optimisation that allows applications of certain pure functions to act like ANF values. For instance, if *foo* and *bar* are marked as pure, the following, which is normally illegal in ANF, is allowed

$$foo(bar(a))$$

The proper ANF version of this is

$$\textbf{let } tmp = bar(a) \textbf{ in}$$
$$foo(tmp)$$

Pure application, while useful for compiling to Javascript, makes life more awkward because the assumption that all function arguments are simple disappears. Therefore a transformation was written on the compiler IR to remove pure applications and turn them into proper ANF applications. Links has a generic IR

$$
\begin{array}{rcll}
M & ::= & \textbf{let } x \ = \ M_1 \textbf{ in } M_2 & \textit{(value bind)} \\
 & | & \textbf{letrec } \vec{F} \textbf{ in } M & \textit{(function bind)} \\
 & | & \textbf{if } M_1 \textbf{ then } M_2 \textbf{ else } M_3 & \textit{(branch)} \\
 & | & M(\vec{M}) & (\textit{call}) \\
 & | & \lambda \vec{x}.M & \\
 & | & x \mid c & \\
 & & & \\
F & ::= & x(\vec{x}) \ = \ M & \\
 & & & \\
x & ::= & \text{variable} & \\
c & ::= & \textbf{true} \mid \textbf{false} \mid \text{integer} &
\end{array}
$$

Figure 4.2: The target language that will be used throughout the report.

transformation framework that makes walking the IR tree while keeping track of types easy. This was used to write the transformation.

4.4 The Target Language

The target language is essentially a simple, untyped syntax tree for a small subset of OCaml. A simplified version is represented below. The actual target language is not significantly more complicated. In addition it has constructs for case expressions, lists, pairs, triples, and for raising exceptions.

A *value bind* binds a variable x to a computation M. x is then in scope for M_2. A *function bind* binds a mutually recursive list of functions, all in each others' scope, and in the scope of M. A *branch* tests its condition M_1, and if it is **true** executes M_2, and otherwise executes M_3.

This language is not syntactically restricted to ANF. This is convenient, as it allows both CPS and ANF to be easily represented, and also allows us to "cheat" a little when compiling... for instance, it is often useful to treat applications of boxing and unboxing functions as values, which is not normally allowed in ANF.

The compiler's target language does not directly have constructs for all Links values (for instance, records, XML nodes and variant types). These values must be simplified and encoded during translation.

```
type value =
  | Bool of bool
  | Int of int
  | Function of (value -> value)
```

Figure 4.3: A *value* type suitable for boxing our simple target language.

4.5 The Value Datatype

The Links and OCaml type-systems are incompatible, so in order to avoid difficulties, all values are boxed into a *value* datatype. A minimal version suitable for the target language given above is shown in Figure 4.3. In the compiler, this *value* type contains an entry for every possible Links value. Boxing is described below. Boxing essentially allows the compiler to pretend that OCaml is untyped.

4.6 Transformations

The compiler supports compiling both to direct-style and CPS. This is achieved by two separate classes that derive from a base class containing common code. These classes, called respectively *direct* and *cps* are encapsulated along with their base class in a module called *Translator*.

The **call/cc** operator is only supported when compiling to CPS. This means that features that rely on having access to the current continuation, like Formlets, will not be available when compiling to direct-style.

Direct-style compilation is still useful, however, as it is often easier to follow direct-style code, which can aid debugging. Also, as discussed in Further Work, it would be possible to add **call/cc** to direct-style compilation some time in the future.

4.6.1 The Direct-style transformation

Figure 4.4 outlines the direct-style transformation of the IR to the target language. It consists of 3 functions. $[\![\cdot]\!]_M$ takes an IR computation or tail computation and produces a target term. $[\![\cdot]\!]_B$ takes an IR binding term and a target term and produces a target term. $[\![\cdot]\!]_V$ takes an IR value and returns a target term.

The translation is performed by applying $[\![\cdot]\!]_M$ to an IR program, and yields an equivalent program in the target language.

$$[\![\textbf{val } x = P]\!]_B\ R = \textbf{let } x = [\![P]\!]_M \textbf{ in } R$$

$$[\![[\textbf{fun } x_1(\vec{x_1}) = M_1, \ldots, \textbf{fun } x_n(\vec{x_n}) = M_n]]\!]_B\ R = \\ \quad \textbf{letrec } [x_1(\vec{x_1}) = [\![M_1]\!]_M, \ldots, x_n(\vec{x_n'}) = [\![M_n]\!]_M] \textbf{ in } R$$

$$[\![B_1 :: \vec{B}]\!]_B\ R = [\![B_1]\!]_B\ ([\![\vec{B}]\!]_B\ R)$$
$$[\![[]]\!]_B\ R = R$$

$$[\![\textbf{let } \vec{B} \textbf{ in } P]\!]_M = [\![\vec{B}]\!]_B\ [\![P]\!]_M$$

$$[\![\textbf{if } V \textbf{ then } M_1 \textbf{ else } M_2]\!]_M = \textbf{if } [\![V]\!]_V \textbf{ then } [\![M_1]\!]_M \textbf{ else } [\![M_2]\!]_M$$

$$[\![\textbf{return } V]\!]_M = [\![V]\!]_M$$

$$[\![V([V_1, \ldots, V_n])]\!]_M = [\![V]\!]_V([[\![V_1]\!]_V, \ldots, [\![V_n]\!]_V])$$

$$[\![\textbf{call/cc } V]\!]_M = \textbf{error}$$

$$[\![V]\!]_V = V$$

Figure 4.4: The direct-style transformation consisting of functions that take IR terms and produce direct-style target code.

The list of bindings that accompanies a computation in the IR is split up, with each binding being "pushed down" to be a continuation of the previous one, with the tail-computation being pushed down furthest:

```
let
  val x = 1
  val y = 2
  val z = 3
in
  P
```

becomes

```
let x = 1 in
  let y = 2 in
    let z = 3 in
      P
```

This pushing-down is achieved via the argument R (standing for *rest*) to the bindings transformation.

The rest of the translation is straightforward. In the actual compiler, the translation of values is more complicated because some Links primitive types do not have direct representations in the target language and must be transformed into their target language representation. This is described in 4.9.

call/cc is not supported, and if it is encountered by the compiler then code to raise an OCaml exception is emitted. This means that compiled code raises a run-time rather than a compile-time error. As parts of the Links prelude rely on **call/cc** , emitting a compile-time error would necessitate two versions of the prelude, one for direct-style and one for CPS.

The target language retains important properties of the source language: Variables are still unique and single-assignment, and the terms are still in ANF, albeit this is now by convention rather than enforced by syntactic restriction.

4.6.2 The CPS-transformation

The CPS transformation is given in 4.5. It is not much more complicated than the direct-style translation as transforming ANF into CPS is fairly simple.

There are three functions, one for computations, one for bindings and one for values. Transformation of values is exactly the same as with direct-style, so in the compiler the value transformation code is shared via subclassing between the direct-style and CPS transformers.

$$[\![\textbf{val}\ x = P]\!]_B\ R = [\![P]\!]_M\ \lambda x.R$$

$$[\![[\textbf{fun}\ x_1(\vec{x_1}) = M_1, \ldots, \textbf{fun}\ x_n(\vec{x_n}) = M_n]]\!]_B\ R = \\ \quad \textbf{letrec}\ [x_1(\kappa :: \vec{x_1}) = [\![M_1]\!]_M\ \kappa, \ldots, x_n(\kappa :: \vec{x_n}) = [\![M_n]\!]_M\ \kappa]\ \textbf{in}\ R$$

$$[\![B_1 :: \vec{B}]\!]_B\ R = [\![B_1]\!]_B\ ([\![\vec{B}]\!]_B\ R)$$
$$[\![[]]\!]_B\ R = R$$

$$[\![\textbf{let}\ \vec{B}\ \textbf{in}\ P]\!]_M\ K = [\![\vec{B}]\!]_B\ ([\![P]\!]_M\ K)$$

$$[\![\textbf{if}\ V\ \textbf{then}\ M_1\ \textbf{else}\ M_2]\!]_M\ K = \textbf{if}\ [\![V]\!]_V\ \textbf{then}\ [\![M_1]\!]_M\ K\ \textbf{else}\ [\![M_2]\!]_M\ K$$

$$[\![\textbf{return}\ V]\!]_M\ K = K([\![V]\!]_V)$$

$$[\![V([V_1, \ldots, V_n])]\!]_M\ K = [\![V]\!]_V([K, [\![V_1]\!]_V, \ldots, [\![V_n]\!]_V])$$

$$[\![\textbf{call/cc}\ V]\!]_M\ K = [\![V]\!]_V((\lambda\kappa, x.K(x)), K)$$

$$[\![V]\!]_V = V$$

Figure 4.5: The CPS transformation, consisting of three functions that take IR terms and transform them into target language terms in CPS form.

The transformation on computations takes a continuation argument which is a function of one argument, K. Applications are translated to pass the current continuation as an extra argument. Return invokes the current continuation with its value. Conditionals simply pass on the continuation to their two computation branches. Let-bindings are transformed into a continuation which is invoked with the value of the let's tail-computation. Functions are transformed to take an extra continuation argument, κ, and κ then becomes the continuation of the function body. Call/cc becomes an application of its value to the current continuation (wrapped in a λ that accepts and discards its continuation argument).

4.7 Avoiding Name Clashes

In the Links IR, each variable is represented as an integer. These integers are associated with their Links names at binding time. As it walks through the IR, the compiler keeps an environment mapping variable integers to Links names. This allows the Links names to be looked up and included as part of the variable name in the target language, which improves the readability and debugability of the resultant code.

However, it is important that the generated variable names do not clash with OCaml built-ins and that they contain only characters allowed as part of OCaml identifiers. Names are therefore generated as follows:

- An underscore is prefixed to the Links name. As no OCaml built-ins start with an underscore, this ensures that identifiers will not clash with OCaml names.
- The variable's integer representation is post-fixed to the variable name. This ensures the variable name remains unique.
- The result is then passed through a *wordify* function that replaces symbols invalid in OCaml identifiers with words.

For example, the Links variable $*$ with integer representation 3 would be transformed into *_star_3*.

4.8 Boxing

The type-systems for Links and OCaml are incompatible in various ways. In order to escape from OCaml's type-system, all Links values are boxed into the OCaml *value* datatype.

Boxing is implemented by various boxing and unboxing functions that are part of the run-time system. Boxing functions take the OCaml representation of a Links type and return a boxed value of type *value*. Contrariwise, unboxing functions take a boxed value of type *value* and return the original unboxed value.

There are boxing and corresponding unboxing functions for every primitive Links type. Calls to these boxing and unboxing functions have to be inserted during translation. To achieve this, the *Translator* module containing the cps and direct classes is parametrised with a module of type *Boxer*. During translation, calls are made to functions in this *Boxer* module.

Each construct in the IR that may require boxing or unboxing has a corresponding function in *Boxer* to perform the boxing transformation. These functions are of type $code \rightarrow code$, taking the fragment of the target language representing the construct and modifying it to be correctly boxed.

There are two modules of type *Boxer*. One, *FakeBoxer*, adds no boxing or unboxing calls at all. This is handy for debugging, as when trying to track down a problem it is often clearer to look at the code without boxing. The other module, *CamlBoxer*, does the necessary boxing for OCaml.

Values are always kept boxed and only unboxed when required. This is naive, as no analysis is done to determine whether boxing of values is actually necessary. Chapter 5 presents an algorithm that performs more optimised boxing.

Boxing for OCaml is mostly straight-forward. For instance, literals are boxed immediately

$$\mathbf{true}$$

becoming

$$box\_bool(\mathbf{true})$$

and conditionals require their condition to be unboxed before testing

$$\mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3$$

becomes

$$\mathbf{if}\ unbox\_bool(M_1)\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3$$

4.8.1 Boxing Functions

Boxing functions and function application is more complicated. Functions take boxed arguments and return a boxed result. This gives them types of the form $(value * \ldots * value) \rightarrow value$

To support higher-order functions, it must be possible to box functions with an arbitrary number of arguments. This is achieved by transforming functions so they have the type *value* → *value*. Functions of this type can then be boxed into the *value* datatype.

Single-argument functions already have the type *value* → *value*. Multiple-argument functions are transformed to this type by currying, so that applying a function to multiple arguments becomes a series of single-argument applications, each resulting in a *value* type that is either a boxed function accepting the next argument or a boxed result if the function has been fully applied.

For example, the function:

$$f(x_1, x_2, x_3) = M$$

becomes

$$f(x_1) = box\_func\ (\lambda x_2.box\_func\ (\lambda x_3.M))$$

where *box_func* boxes a function of type *value* → *value*. This results in f having type *value* → *value*.

When calling functions, the opposite process occurs: The curried functions are unboxed and applied to the arguments:

$$(unbox\_func\ ((unbox\_func\ (f\ x))\ y))\ z$$

When boxing functions in a **letrec** , it is important to ensure that it is the boxed version of functions that is always in scope. To achieve this, each function in a **letrec** is boxed inside all mutually recursive function bodies, as well as in the code after. An example is in Figure 4.6.

4.8.2 Alternative Ways to Box Functions

There are other possible ways to box functions. One alternative to is to represent arguments as OCaml arrays. This would give all functions the type *value array* → *value*, which could be easily boxed. OCaml supports pattern-matching on arrays, which would make the syntax for arguments much clearer than above:

```
let x a b = a + b
```

would become

```
let x [|a; b|] = a + b
```

and calling it would be just as easy:

$$\textbf{letrec}\ [x(a,b) = M_1, y(c,d) = M_2]\ \textbf{in}\ M_3$$

$$\begin{array}{l}
\textbf{letrec}\ [\\
\quad x(a) = \\
\qquad \textbf{let}\ x = box_func(x)\ \textbf{in} \\
\qquad \textbf{let}\ y = box_func(y)\ \textbf{in} \\
\qquad\quad M_1, \\
\quad y(b) = \\
\qquad \textbf{let}\ x = box_func(x)\ \textbf{in} \\
\qquad \textbf{let}\ y = box_func(y)\ \textbf{in} \\
\qquad\quad M_2 \\
]\ \textbf{in} \\
\quad \textbf{let}\ x = box_func(x)\ \textbf{in} \\
\quad \textbf{let}\ y = box_func(y)\ \textbf{in} \\
\qquad M_3
\end{array}$$

Figure 4.6: An example of boxing mutually recursive functions: both functions are boxed inside each others' bodies, and also in the code after.

```
x [|4; 2|]
```

This could also potentially be more efficient, as there is less (explicit) unboxing during function calls. However it was decided that the currying approach of 4.8.1 was more natural.

4.9 Value Representations

Links includes several primitive value types that must be represented in a workable way in OCaml. This section gives details of these representations and why they were chosen.

4.9.1 Variants

Links supports polymorphic variants, which means that variant tags do not belong to a specific type. Instead, a type is inferred for each tag usage and the Links type-system ensures that they are used in a consistent way. Variants therefore should be represented in a way that doesn't associate them with a particular type. In the compiler, variants are represented by pairs of strings and values. For instance, the Links variant

```
Pounds(42)
```

would be represented as

```
("Pounds", box_int(42))
```

in OCaml. This allows for easy translation of switch pattern matching:

```
switch (money) {
    case Pounds(i) -> i
    case Dollars(i) -> i
}
```

becomes

```
match money with
  | ("Pounds", i) -> i
  | ("Dollars", i) -> i
```

OCaml does support polymorphic variants, but they are strictly less expressive than those in Links, making them an unsuitable representation. Furthermore, it must be possible to easily box variant values, meaning they should have a consistent representation of a single type.

4.9.2 Records and Tuples

Links supports record types, which are a collection of fields indexed by name. For example

```
(country="Scotland", capital="Edinburgh")
```

Records and tuples are the same thing in Links. Tuples are just records indexed by an integer sequence beginning at 0.

Records are represented in the compiler using *StringMap*, an OCaml data structure that maps strings to *value* types. Records are constructed as a series of nested StringMap.add operations

```
(StringMap.add
   "country" (box_string "Scotland")
   (StringMap.add "capital" (box_string "Edinburgh")
      StringMap.empty))
```

they are indexed by looking up the a field name in the StringMap.

4.9.3 XML

Links supports XML nodes as a native data type. They are represented as a list of type *xmlitem*. This list is built at run-time by calling a *build_xmlitem* function. An xmlitem value is either a string, which represents a text node, or a tuple consisting of a node name, a list of attributes and a list of xmlitem children:

```
type xml = value list
and xmlitem =
  | Text of string
  | Node of string * (string * string) list * xml
```

4.10 From The Target Language To OCaml

As the compiler's target language essentially represents the syntax tree for a subset of OCaml, converting it to OCaml is easy. The tree is walked, and OCaml is produced using the same pretty-printing combinators that were used for the pretty-printer.

The resulting string is a closed OCaml expression. This is wrapped with a function called *entry*, and an OCaml main function is added that will pass this entry function to the *run* method of the run-time system when the code is executed.

The resulting code is then printed. It can be compiled with OCaml by linking against a small run-time system and library of primitives.

4.11 The Primitives Library

Compiled executables are linked with a small library of primitive functions written in OCaml.

This does not include all the functionality of the interpreter primitive library. Notably missing are support for regular expressions and for concurrency.

4.11.1 Generating Wrappers

Translation of function application makes no distinction between calling primitive functions and non-primitive functions. This means that primitives must behave exactly like non-primitives. They must therefore be boxed themselves, take boxed arguments and return boxed results. For instance, the primitive _int_add

```
let _int_add x y = x + y
```

would become

```
let _int_add = (box_func (
  fun x -> (box_func (
    fun y -> box_int ((unbox_int x) + (unbox_int y))))))
```

This is tedious for a human to write. Therefore, library functions are written in a natural style and boxed stubs are generated for them at compile-time. These stubs are created using a table giving details of each library function. An example table entry is

```
"+", "_int_add", (["unbox_int"; "unbox_int"], "box_int"), false
```

The fields are, respectively, the Links name for the primitive, the OCaml function name, the boxing information (which consists of a list of unboxing functions, one for each argument, and a single boxing function for the return value), and a boolean indicating whether the function needs to be passed the current continuation.

Generating function stubs for the library at compile-time allows the same OCaml code to be used for direct-style and CPS. Library functions are always written in direct-style, and when a stub is generated during CPS compilation, it accepts an extra continuation and applies it to the return value of the library function.

```
let _int_add x y = x + y

let _plus_1 = (box_func
  (fun k -> (box_func
    (fun arg_0 -> (box_func
      (fun arg_1 ->
        ((unbox_func k)
          (box_int (u_add (unbox_int arg_0) (unbox_int arg_1)))))))
```

Figure 4.7: The natural, direct-style definition of a primitive function followed by the boxed CPS stub generated from its primitive table entry.

Some functions (for instance, *exit*) require the current continuation to be passed in. This is indicated in the table entry for the primitive, and in this case the CPS stub generated passes the continuation as an extra argument rather than applying it. In direct-style compilation, the stub will raise an exception if called.

It is important that the generated stubs receive the name that they will be referred to by later. The stub name is therefore generated in the same way as other variables, as described in 4.7. For instance, the stub for the Links + primitive would be given the name *_plus_1*. A complete CPS stub for + is shown in Figure 4.7.

4.12 The Run-time System

The compiler's run-time system contains the functions for boxing and unboxing values, some utility functions used by the compiler, the *value* datatype and the code required to run the executable.

4.12.1 CGI and Command-line interface

When a compiled executable is run, its main function is passed to the run-time's *run* function. This checks whether the program is being run using CGI or on the command-line. It does this by checking for the presence of CGI environment variables.

If the program is being run on the command line, its main function is immediately executed, and the result it returns is printed to standard output.

If it is being run as a CGI program, first the CGI parameters are parsed and stored into a mutable variable so that primitives can access them (some primitive

functions related to formlets require access to the CGI parameters). If the special continuation parameter, $\_k$ is present, it is assumed to be a serialised continuation, and is deserialised and applied. Otherwise, the program's *run* function is called.

In either case, a value of type *Page* is returned. This is turned into XML by passing it to a function *renderPage* and then printed to standard output, which goes to the client.

Serialisation and deserialisation of continuations is achieved in CPS using OCaml's Marshal module, which is capable of serialising closures.

4.12.2 Printing

The run-time system takes advantage of the tags boxing provides to implement polymorphic printing of values.

5. Optimised Boxing

Boxing can sometimes be expensive, and also leads to ugly code. It is therefore worthwhile to try and eliminate as much boxing and unboxing as possible. This chapter attempts to give an intuitive overview of a new boxing algorithm that boxes correctly, but tries to do so with as few explicit boxings/unboxings as possible.

The algorithm operates on the target language, as this is a little simpler than operating on the IR. In the actual compiler, where the range of built-in types is larger, the analysis part of the optimisation would need to take place on the IR. This is because some value types do not have direct target language analogues.

The input is assumed to be in ANF with unique variable names. This is the case for terms generated using the direct-style transformation given in Figure 4.4. Terms also should not contain λ functions.

The algorithm consists of a boxing transformation that adds boxing to terms. To help it do this, it takes advantage of knowledge gained by two analysis phases.

The transformation treats boxed and unboxed versions of a variable separately rather than as two states of a single variable. This has the effect of splitting a variable x into two: a boxed version, x_b and an unboxed version, x_u.

When a boxed value is required, x is replaced by x_b and when an unboxed version is required, it is replaced by x_u. This has the effect of hoisting boxing and unboxing up to where x is bound. Whether x_u, x_b, or both are made available is determined by an environment Γ that, for each variable says whether it is used boxed, unboxed, both, or whether it doesn't matter.

Γ maps each variable to a member of a lattice $\mathsf{L} = \{\top, \bot, \mathcal{B}, \mathcal{U}\}$ with partial order $\bot \sqsubset \mathcal{U} \sqsubset \top$ and $\bot \sqsubset \mathcal{B} \sqsubset \top$. $\Gamma\ x$ is $\mathcal{U}$ if x is used only unboxed, which means only x_u need be available. Similarly, it is $\mathcal{B}$ if it is used only boxed, which means only x_b need be available. $\top$ means that it is used both boxed and unboxed, and therefore both x_u and x_b must be available. $\bot$ means it is used neither boxed or unboxed, in which case it doesn't matter of which of x_u and x_b is available and the most convenient can be picked.

At a binding **let** $x = M_1$ **in** M_2 it is useful to know whether the body M_1 can readily provide boxed or unboxed values. By *readily* we mean that it can provide the value without having to explicitly box or unbox.

If x is required only boxed and M_1 can readily provide a boxed value then x_u can simply be assigned straight from M_1. This works similarly for boxed values. The **let** binding tells M_1 whether it requires a boxed or an unboxed value, and

M_1 is obliged to provide what it is asked for; but by knowing what M_1 can *easily* provide, the **let** can ask for things that are already available and so prevent unnecessary boxing/unboxing.

For each variable x bound with **let** $x = M_1$ **in** M_2, the environment Π maps x to a member of L. If $\Pi\ x = \mathcal{U}$ then only unboxed values are readily available from M_2. If it equals $\mathcal{B}$ then only boxed values are available, and if it equals $\top$ then both boxed and unboxed values are available. $\bot$ does not occur.

To calculate Γ, we need to know what it means to say a variable is used boxed or unboxed.

A variable is used *boxed* if it

1. Is passed to a function that requires a boxed value as its argument.
2. May be copied to a variable that is used only boxed.

A variable is used *unboxed* if it

1. Is passed to a function that requires an unboxed value as its argument.
2. May be copied to a variable that is used only unboxed.
3. Appears in the conditional of an **if** term.

The effect of rule (2) is to propagate usages backwards across copies. For example

$$\textbf{let } x = y \textbf{ in } M$$

if x is used only unboxed in M then y is marked as being used unboxed too. That means that y will be able to readily supply an unboxed value to x. In effect, unboxing is being hoisted so it occurs earlier.

Rule (1) necessitates that we know whether a function takes boxed or unboxed values as arguments. The environment Ω maps every function to a tuple $\langle \vec{\mathsf{L}}, \mathsf{L} \rangle$ consisting of a list of L values, one per argument, and a single L value for the return value. This tuple is known as a *schema*. $\mathcal{B}$ means boxed and $\mathcal{U}$ means unboxed. No other values are allowed. By inspecting Ω, it is possible to tell whether a variable passed as an argument to a known function is used boxed or unboxed. Similarly, it is possible to tell what value a function can readily provide by inspecting whether its return value is $\mathcal{B}$ or $\mathcal{U}$. Unknown functions always return boxed values and take boxed arguments.

Each function f is split into two versions. A *worker* function f_u that may take and return boxed and unboxed values and a safe *wrapper* function f_b that takes and returns boxed values. The wrapper function can be easily generated using the

information in Ω. Whenever a known function is called, f_u is used, and whenever the function escapes (through a copy, for instance) f_b is used.

Γ, Π and Ω are calculated by abstract interpretation in two analysis phases. The first pass computes Γ and the arguments part of Ω. The second pass computes Π and the return part of Ω.

6. Evaluation

6.1 Execution Speed

The main aim of the project was to improve the execution speed of server-side Links code. To determine whether this had been achieved, execution times were measured on a variety of benchmarks for both interpreted and compiled Links code.

6.1.1 Benchmark Setup

The benchmarks were conducted on a minimally loaded 2.16 GHz Intel Core 2 Duo Macbook Pro with 2 gigabytes of RAM. Each result is the average of 10 runs with the highest and lowest times thrown away.

The Links interpreter has the ability to record time statistics for the run of a program. This facility was used to ensure that only the time taken to interpret the Links IR was measured, without including the time taken to read, parse, type-check and compile the source code. These disregarded steps can take a significant amount of time and so could adversely affect the perceived interpretation speed. Ignoring them is also fair, as in real-world operation Links caches the generated IR tree, so the overheads of parsing, type-checking and compilation only apply to the first execution of a program.

Code was added to the compiler's run-time system so that when a particular environment variable was set for program execution (*MEASURE_PERFORMANCE*), the execution time was recorded. Execution time was measured in the same way as for interpreted code.

The performance statistics were written to standard error. A python script was created that automatically ran the benchmarks, parsed the run-time for each run and then calculated the average and the standard deviation. For programs that required CGI, this was emulated by setting appropriate environment variables.

6.1.2 Benchmark Programs

The tests were conducted using several fairly standard micro-benchmarks, along with some Links-specific benchmarks that measure rendering and form-processing speed.

sumlist	12.677	0.005	0.067	0.000
tak	6.578	0.004	0.084	0.000
quicksort	15.619	0.006	0.048	0.000
fib	7.450	0.008	0.201	0.000
table render	13.150	0.010	1.700	0.001
formlet render	6.721	0.005	0.353	0.001

Table 6.1: Benchmark results for execution time of interpreted and compiled code. Values are times in seconds.

sumlist generates a list of consecutive numbers, then sums the list. It tests integer arithmetic and list processing.

tak calculates the takeushi function. This tests recursion and function calls.

quicksort generates a list of consecutive numbers then sorts them using the quicksort function. The version of quicksort used is the one in the Links prelude, which is implemented using higher-order functions for comparisons. This tests higher-order functions, list deconstruction/construction and recursion. The pivotal element is always chosen to be the left-most. Thus, the benchmark always runs with quicksort's worst-case performance.

fib is the usual naive recursive implementation of the fibonacci function. It tests recursion and function calls.

table render generates a large HTML table of consecutive integers. It is supposed to measure the raw speed of building up and rendering XML.

formlet render renders several formlet forms on a page. This tests formlet rendering speed.

Parameters were chosen for the size of the benchmark problems to ensure that the interpreter benchmarks would complete in a reasonable amount of time.

6.1.3 Results

As can be seen, compiled code significantly outperforms interpreted code, but much less so on the rendering and form-processing benchmarks than on the micro-benchmarks. This suggests that compiling makes a massive difference to algorithmic code and gives a much smaller benefit when building and rendering pages.

7. Conclusion

The goal of the project was to increase the performance of server-side Links code by implementing a compiler. A compiler has successfully been created, and benchmarks show that compiling gives a clear and substantial performance gain.

A new boxing optimisation has also been developed, but unfortunately due to lack of time has neither been implemented in the compiler nor described in great detail in this report.

7.1 Further Work

There is lots of scope for further work. Support for Javascript, concurrency and database querying would go a long way to making the compiler usable in the real world.

It would be interesting to add continuations to the direct-style compiler and compare performance against CPS. A method for doing this is described in[8] and its implementation for Javascript described in[21]

F# support would give Links the ability to run on .NET. As the compiler already produces F#-compatible code, this would just require some conditional compilation in parts of the run-time system, which would not be difficult.

The boxing optimisation presented in the report should be implemented and benchmarked, and the compiler should be benchmarked against the interpreter on some more substantial programs. Currently this is hampered by the lack of SQL and Javascript support in the compiler.

Bibliography

[1] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, "Links: Web Programming Without Tiers," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4709, p. 266, 2008.

[2] [Online]. Available: http://caml.inria.fr/

[3] D. Robinson and K. Coar, "Rfc 3875, the common gateway interface (cgi) version 1.1," October 2004.

[4] E. Meijer, B. Beckman, and G. Bierman, "LINQ: reconciling object, relations and XML in the .NET framework," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM New York, NY, USA, 2006, pp. 706–706.

[5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, "An idiom's guide to formlets," Technical Report EDI-INF-RR-1263, School of Informatics, University of Edinburgh, 2008, Tech. Rep.

[6] S. Krishnamurthi, P. Hopkins, J. McCarthy, P. Graunke, G. Pettyjohn, and M. Felleisen, "Implementation and use of the PLT Scheme web server," *Higher-Order and Symbolic Computation*, vol. 20, no. 4, pp. 431–460, 2007.

[7] C. Queinnec, "Inverting back the inversion of control or, continuations versus page-centric programming," *ACM SIGPLAN Notices*, vol. 38, no. 2, pp. 57–64, 2003.

[8] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen, "Continuations from generalized stack inspection," in *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, vol. 40, no. 9. ACM New York, NY, USA, 2005, pp. 216–227.

[9] A. Appel, *Compiling with Continuations*. Cambridge: Cambridge University Press, 1992.

[10] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen, "The essence of compiling with continuations," in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. ACM Press New York, NY, USA, 1993, pp. 237–247.

[11] M. Chakravarty, G. Keller, and P. Zadarnowski, "A Functional Perspective on SSA Optimisation Algorithms," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 2, pp. 347–361, 2004.

[12] S. JONES, N. RAMSEY, and F. REIG, "C-: A portable assembly language that supports garbage collection," *Lecture notes in computer science*, pp. 1–28.

[13] [Online]. Available: http://llvm.org/

[14] M. Schinz and M. Odersky, "Tail call elimination on the Java Virtual Machine," *Electronic Notes in Theoretical Computer Science*, vol. 59, no. 1, pp. 158–171, 2001.

[15] N. Perry and E. Meijer, "Implementing functional languages on object-oriented virtual machines," in *Software, IEE Proceedings-[see also Software Engineering, IEE Proceedings]*, vol. 151, no. 1, 2004, pp. 1–9.

[16] N. Benton, A. Kennedy, and G. Russell, "Compiling standard ML to Java bytecodes," *ACM SIGPLAN Notices*, vol. 34, no. 1, pp. 129–140, 1999.

[17] N. Benton, A. Kennedy, and C. Russo, "Adventures in interoperability: the SML .NET experience," in *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM New York, NY, USA, 2004, pp. 215–226.

[18] E. Meijer and J. Gough, "Technical Overview of the Common Language Runtime," *language*, vol. 29, p. 7.

[19] D. Syme, "ILX: Extending the. NET Common IL for Functional Language Interoperability," *Electronic Notes in Theoretical Computer Science*, vol. 59, no. 1, pp. 53–72, 2001.

[20] C. Lindig and G. D. Gbr, "Strictly pretty," 2000.

[21] F. Loitsch, "Exceptional Continuations in JavaScript."