BLAISE COMPILER

by

Austin Laugesen

A project submitted in partial fulfillment of the requirements for graduation with Honors in the Department of Computer Science

Aaron Stump
Honors Project Director

Spring 2011

All requirements for graduation with Honors in the Department of Computer Science have been completed.

Cesare Tinelli
Computer Science Honors Advisor

BLAISE COMPILER

by

Austin Laugesen
Spring 2011

Aaron Stump
Project Director from Computer Science

What if a programming language could have the safety and expressiveness of a functional programming language without the performance hit and complexity of garbage collection? BLAISE, currently in early development, has such properties. Over the last academic year, I helped implement a compiler for a new programming language called BLAISE. Thousands of programming languages exist because each language tries to do something unique or outdo other languages. All languages have a trade-off between machine robustness and programmer convenience because it is hard to provide both. There is hope that BLAISE will provide both robustness, due to the absence of garbage collection, and the convenience of a concise high-level programming language. Furthermore, programmers will find memory errors at compile time, instead of runtime.

The first two steps in creating a programming language are specifying a formal grammar and developing a compiler that generates machine executable code. I implemented three significant parts of the BLAISE compiler: currying support, foreign-function interface (FFI) support, and tail-recursion optimization. Currying is a functional programming staple, therefore it is necessary to implement. Foreign-function interfacing is useful for tying an external code library to an existing language without modifying the compiler. Tail-recursion optimization is a typical component of compilers because it significantly speeds up the execution of tail-recursive code. In addition to compiler development, I evaluated BLAISE's efficiency by comparing it to OCaml in execution time on a well-known algorithm: mergesort. Over the last nine months Dr. Aaron Stump, Dr. Garrin Kimmell, Geoffrey Roughton, Josh Meyer and I successfully created the BLAISE-to-C compiler.

The BLAISE compiler is complete and is ready for a type checking system to be built on top of it. BLAISE code successfully compiles to C code, which can be compiled to produce an executable. On average, BLAISE's execution performance is comparable to or better than OCaml. While implementing BLAISE I learned a lot about programming language design and compiler development. For instance, I learned the difference between under-saturated, saturated and over-saturated function calls because I implemented currying support. Likewise, I implemented and witnessed the exceptional efficiency increase from tail-recursion optimization. Thanks to my individual research project I am a proud contributor to the BLAISE compiler. I helped make it more robust and useful for future research.

Contents

Chapter 1

Introduction

1.1 Motivation

BLAISE is a functional programming language, developed to explore whether or not a programming language can have the safety and expressiveness of functional programming languages without the performance hit and complexity of garbage collection. Many programming languages exist because each language attempts to do something different than other languages. Over the last year I helped implement a new programming language called BLAISE. Anyone who studies programming languages will find that all languages are designed around a trade off between robustness on a machine and convenience for a programmer, because it is hard to provide both. There is hope that BLAISE is both robust, due to explicit memory management, and provides the convenience of a concise functional language. When BLAISE is fully implemented with the proper type-checker and compiler, programmers will find memory errors at compile time, instead of runtime.

1.2 Intro to Functional Programming Languages

Functional programming (FP) languages are designed to reflect the way people think mathematically rather than reflect the structure of the machine on which that code executes. These languages are founded on lambda calculus, a simple model for computation. It is arguably easier to reason formally with programs written in a FP language due to FP's theoretical math foundation [5]. There are a number of reasons to use a FP language and there are a handful of commonly used languages such as Haskell, OCaml, ML, and Scheme.

Functional languages strive to represent everything as pure mathematical functions. For instance, FP languages view the = operator as an expression of an equation. This contrasts with imperative languages such as C, which use = as an assignment operator to store values inside of variables that represent cells in memory. Take the following FP declaration as an example.

```
var x = f(y);
```

This line of code introduces a variable named x and asserts that x is equivalent to f(y) is true. As noted earlier, functional languages abstract code from the machine, so there is no notion of a memory cell in the declaration above. If one were to write the following line

```
var x = x + 1;
```

in a functional language, the program would be nonterminating or the compiler would refuse to compile this code. The previous statement doesn't have a finite solution, whereas in C the value in memory cell x would be incremented [5].

Function definitions can be viewed as assertions much like variables. Consider the following declaration

```
define f = fun(x: int)(y:int).
  x * y
```

which introduces the function f and states that for any x and y f(x,y) is equal to x * y. Notice that the function parameters x and y are not manipulated inside of the function, because FP languages do not allow such operations. In FP, as in mathematics, the value of a function is uniquely determined by its inputs. This is not at all true for imperative languages [5]. Consider invoking a function to get the time of day, generally this sort of function doesn't take any arguments but always returns a different value.

Variables in functional languages cannot be modified after an initial assertion, and therefore repetition cannot make use of loops but uses recursion instead[5]. Let's review the factorial function in both FP and imperative programming. Consider the following functional example:

```
define factorial = fun(x: int).
  case x == 0 of
     1 -> x
  |  0 -> x * factorial(x - 1)
```

Now compare the example above to the imperative example below.

```
int factorial(int x) {
  int prev = -1;
  for(x; x > 0; x--){
    if(prev == -1)
      prev = x;
    else
      prev *= x;
  }
  return prev;
}
```

Clearly, the functional example is much more concise and avoids directly modifying variables with the use of recursion. In contrast, the imperative example takes advantage of the fact variables can be directly manipulated. Every iteration of the loop reassigns the variable prev, thus a loop is used to compute the factorial.

1.3 Memory Management with Garbage Collection

Garbage collection (GC), or automatic memory management, is an algorithm implemented under the hood of many modern programming languages that allows a programmer to easily release memory back to a machine. Every time software runs on a computer it stores information in memory. The tools used to develop the software dictate whether or not memory is handled with GC or handled by the programmer. When software no longer needs to remember information stored in a cell of memory it should give the memory back to the machine. GC makes the process of giving memory back to a machine quite simple for the programmer, improving productivity [7].

Many programming languages use heap storage with GC. Allocated cells of memory that are not reachable in the runtime stack through a chain of pointers are garbage. One commonly used algorithm for GC is mark-and-sweep where garbage is collected for re-use by a traversal algorithm. This algorithm passes through the heap and marks all reachable cells using depth-first search and then collects all of the unmarked cells [2]. Consider the following line of garbage-collected code in C#.

```
Object obj = new Object();
obj = null;
```

Initially, obj is a handle to an object of type 'Object' that consumes memory. Once the variable obj is set to null, C# removes the pointer it had set to the memory containing the 'Object' object. Then sometime later, when C# decides to run its garbage collector it collects the memory that obj had a handle on.

Programmers that use languages with GC don't have to worry about monitoring heap-allocated cells of memory. This means the programmer can write less code that is more straightforward. However, garbage collectors are often slow and expensive, and these straightforward programs are less efficient because they use garbage collection [2].

1.4 The BLAISE Approach

The goal of BLAISE is to create a programming language where GC is unnecessary, since the language provides the programmer with statically enforced abstractions to safely manage memory. At compile time, BLAISE enforces a set of memory management rules that if followed guarantee the absence of memory errors [11]. Other languages have been created with this idea in mind, but typically results have imposed severe restraints on programs, e.g. scoped regions which make code hard to follow, maintain, and reuse [4]. The abstractions enforced by BLAISE are much less restrictive. Dr. Stump's working hypothesis is that "BLAISE's abstractions are flexible enough to accommodate common programming idioms and data structures" [11]. Without GC BLAISE is expected to be highly performant and potentially useful for real-time systems [11].

1.5 Functional Programming with Explicit Memory Management

What happens to memory if a language doesn't have GC? The programmer must explicitly manage memory, which entails writing bookkeeping code to keep track of heap-allocated cells and explicitly freeing them when necessary. This task can be tedious, error prone and often makes software longer and more complex. Explicit memory management can make it hard to modularize code as well as make software more prone to memory leaks [2]. However, BLAISE is different than most languages without GC. First, it is a functional language which is more concise than imperative languages. Second, BLAISE is designed to find memory errors at compile time, opposed to run-time, which will help the programmer debug their code. The following

BLAISE code exemplifies explicit memory management.

```
datatype
  type node
    cons: Cell(number: int)(next: node).node
    nil: Cell().node
  end
end

define mn =
  var x = new Node(5, nil);
  delete x
```

In this example x points to an instantiated 'node' datatype that occupies the newly allocated memory. The 'new' construct allocates memory. 'Delete' demonstrates how the programmer deallocates memory. The memory that x points to is released back to the machine and is no longer accessible from x. Typically, functional languages abstract from the machine as much as possible which means the programmer doesn't have to manage memory. BLAISE is not a typical functional language and requires explicit memory management so it finds memory errors at compile time.

1.6 Static (Compile-Time) Analysis for Safe Memory Usage

The overarching goal of BLAISE is to eliminate the need for GC by giving the programmer statically enforced abstractions to safely manage memory. Static memory management replaces runtime GC [1]. These enforced abstractions allow the compiler to make abstract interpretations on the source code it compiles. Conceptually, source code represents computations in a domain of objects. Abstractly interpreting source code includes using that representation to describe computations in an alternative domain of abstract objects. The results of abstract interpretation gives some information related to the actual computation.

A fairly simple example of abstract interpretation, borrowed from [10], is the rule of signs. The computation -2 * 1 may represent computations in the abstract universe { (+), (-), (+) | (-) } where the semantics of the arithmetic operators is defined by the rule of signs. The abstract execution

$$-2 \ * \ 1 \ => \ -(+) \ * \ (+) \ => \ (-) \ * \ (+) \ => \ (-)$$

proves that -2 * 1 is a negative number. It gives a summary of some portions of the actual execution of a program. Although, the summary is easy to obtain it is not always accurate, consider the following example.

$$-2 \ + \ 1 \ => \ -(+) \ + \ (+) \ => \ (-) \ + \ (+) \ => \ (+) \ | \ (-)$$

The abstract interpretation is unable to determine if the result is positive or negative. Despite not receiving definitive results in certain circumstances, abstract interpretation allows the programmer or the compiler to answer questions that do not need full knowledge of the program executions [3]. With the proper typing system and statically enforced abstractions the BLAISE compiler will be able to reason about memory management.

1.7 Related Work on Cyclone

As mentioned earlier, other programming languages have approached solving the same problem as BLAISE, such as Cyclone, a type-safe dialect of C. Similar to BLAISE, Cyclone provides the performance of a low level language with type safety parallel to languages such as Java. This language uses programmer-supplied annotations, a type system, a flow analysis and run-time checks to verify that programs are type-safe [6]. More recent versions of Cyclone integrate unique pointers into the memory management framework which are utilized by the type system. Unlike BLAISE, Cyclone uses a garbage collector. However, Cyclone's compiler and type system minimize the use of GC and avoids utilizing GC whenever possible [8].

Chapter 2

The BLAISE Language

2.1 Key Constructs

What does BLAISE code look like and what constructs are available for the programmer to use? Also, how does it compare to other conventional functional languages? BLAISE is a functional language that resembles a mix between OCaml and C, which can be observed in the code examples that follow. Typically, every programming language has a way to use variables, and in BLAISE the programmer can assert that a variable is equivalent to an expression, as simply as

```
var x = "Hello!";
```

The statement above asserts that x is equivalent to the string "Hello!". Another common feature in programming is defining and invoking functions. A function definition requires the 'define' keyword followed by a function name, an = sign that is followed by the keyword 'fun' with zero or more parameters, and terminated with a period '.'. Parameters are denoted by parentheses that encase a variable name, colon, and type, e.g. (num : int). The 'define' keyword also denotes bodies of code that need to be executed, these constructs do not have the 'fun' keyword following an = sign. An example is 'mn' below.

```
define f = fun(message: string).
  print_string("I received the following argument: ");
  print_string(message)

define mn =
  f("Here is my message.")
```

As shown, f's definition contains two calls to the print_string function. The only invocation of f appears inside of mn, where it is passed the string "Here is my message.". After execution the strings "I received the following argument: " and "Here is my message" print out to console.

Other constructs worth mentioning include the programmer's freedom to declare custom datatypes, instantiate a datatype, and use 'case <expression> of' constructs for pattern matching datatypes to a specific type and accessing attributes. The example below demonstrates these four constructs.

```
datatype
    type node
      cons: Cell(number: int)(next: node).node
      nil: Cell().node
    end
end

define setNodeNumber = fun(num: int)(n: node).
    case n of
        cons -> n.number = num
    |   nil -> ()

define mn =
    var x = new cons(5, nil);
    setNodeNumber(3, x)
```

The programmer defines a datatype node, where a node is one of two types, cons or nil. Cons nodes have two attributes: *number* of type int and *next* of type node. Nil nodes don't have any attributes, and are akin to enumerated types in C, so the *new* keyword isn't required for instantiation. Further in the program, x becomes equivalent to a node initialized with the values 5 and nil and then passed as an argument to setNodeNumber. Any node passed to setNodeNumber goes into a 'case

<expression> of' statement, which identifies the type of node it is passed, e.g. cons or nil. Those familiar with functional programming may begin to see BLAISE resemblance to OCaml. Other than very specific memory management constraints and mutable datatypes BLAISE is a lot like conventional functional languages.

It is worth noting that the 'case <expression> of' construct is may be used like if-else statements, as shown below.

```
define g = fun(n: int).
  case n == 1 of
       1 -> print_string("true")
  |    0 -> print_string("false")
```

For comparison purposes, here is g's definition written in OCaml:

```
let g (n: int) =
    match n == 1 with
        true -> print_string("true")
    |   false -> print_string("false")
;;
```

Lastly, what would an introduction to a programming language be without a demonstration of "Hello, World!"? One can see that BLAISE is quite succinct and doesn't require a lot of source code to get a program up and running.

```
# Blaise
define mn =
  print_string("Hello World!")
```

Also, for comparison I've provided a "Hello, World!" example in OCaml, which is just one line shorter.

```
(* OCaml *)
print_string("Hello World!")
```

Clearly, BLAISE isn't too complex but provides the programmer with a lot of control.

Chapter 3

The BLAISE Compiler

3.1 Development Timeframe

The implementation of BLAISE started in the summer of 2010 and has included a handful of people: Dr. Aaron Stump, Dr. Garrin Kimmell, Geoffrey Roughton, Josh Meyer and myself. From time to time I may use the term 'we', I am referring to the aforementioned individuals who helped develop this project. Initially, Dr. Stump created an ENBF grammar, Blaise.gra, which can be found in the appendix. Early in the summer Geoffrey started to develop the BLAISE to C compiler. Near the middle of the summer Geoffrey left the BLAISE project for a full-time job. It wasn't until the fall semester when Josh and I picked up BLAISE where Geoffrey left off. During the fall semester Dr. Kimmell was brought on the project to help guide the compiler development.

3.2 Overview - High Level Structure

What is required to execute a new programming language on a computer? First, a formal grammar is needed to specify the syntax of the language. A grammar is used to mold the software necessary for processing source code. Second, a parser is used to make syntactic sense of source code. Parsers verify that source code adheres to the formal grammar, and builds an abstract syntax tree (AST) of the code. Lastly, the AST is handed off to a translator that converts source code into the target language code which a machine can execute. BLAISE source code is translated to C code.

Conveniently, parsers can be automatically generated or manually written. We opted for an auto-generated parser. A lot of auto-generated parsers are created with yacc, 'Yet Another Compiler Compiler', and this is also true for the BLAISE compiler. We use an ocaml variant of yacc, known as ocamlyacc, it turns a BNF grammar into a parser. To add another layer the process, Dr. Stump and Josh Meyer developed gt, Grammar Tool, which turns an EBNF grammar into a BNF grammar that is then given to ocamlyacc, and in return ocamlyacc produces a parser. EBNF grammars are a superset to BNF grammars and are more convenient to write. Blaise.gra, an EBNF grammar, is the only required input for gt to get a parser. Since the BLAISE parser is automatically generated Geoffrey, Josh, and I developed the translator that turns BLAISE source code to C code.

$$\begin{tikzcd}
\text{Blaise.gra} \arrow[r] & \text{Gt} \arrow[r] & \text{BNF grammar} \arrow[r] & \text{OCamlyacc} \arrow[r] & \text{Parser}
\end{tikzcd}$$

From the developer's perspective turning BLAISE source code into an executable file is a matter of passing their code to the BLAISE compiler. Under hood of the BLAISE compiler one will find that there are a number of steps that the machine goes through to create an executable. Initially, source code goes through the parser, which generates an AST. Then the AST undergoes *linearization*, a process which removes certain nested expressions which cannot be directly compiled to C. Following linearization, the AST is given to the code translator that produces C code. This C code is then compiled with gcc which produces an executable.

$$\begin{tikzcd}
\text{Blaise code} \arrow[r] & \text{Parser} \arrow[r] & \text{AST} \arrow[r] & \text{Linearize} \arrow[dlll] \\
\text{Translator} \arrow[r] & \text{C} \arrow[r] & \text{Gcc} \arrow[r] & \text{Executable}
\end{tikzcd}$$

3.3 My Contributions

3.3.1 Currying with Partial Applications

BLAISE is designed to be a functional programming language, and therefore it's necessary to include common features found in conventional functional languages, such as currying. *Currying* allows the programmer to invoke a function with too many, too few, or the expected number for arguments. It also enables the programmer to pass a function as an argument to another function. Hence, the BLAISE compiler supports currying. Admittedly, the examples below are contrived but they serve well for demonstrating how currying works. Consider the following code sample.

```
define g = fun(arg1: 'a)(arg2: 'a)(arg3: 'a) =
  (* function body*)

define mn =
  var x = g(1, 2);
  var y = x(3)
```

Above, g's definition requires three arguments to be passed to g to execute. The first time g is called it is passed two arguments, this is an instance of currying. The function is under-saturated and it is not invoked. Instead, a data-structure, called a *partial application*, is created to remember the function and the arguments passed to it [9]. The resulting partial application is stored in the variable x. Now x can be passed arguments to saturate g so it can be executed. Further in the example one can see that x is passed one argument. Since x is a partial application waiting for one argument it is now able to invoke g with the arguments: 1, 2, and 3. Below is an example of code where the variable y acquires the same value as the variable y above, in the curried example.

```
define mn =
  var y = g(1, 2, 3)
```

As mentioned earlier currying allows the programmer to pass a function as an argument to another function. This means that the compiler must recognize when parameters in a function definition are invoked like a function. Up until now currying has been explained and exemplified where a global scope function is curried. However, there are points where the compiler must deal with unknown functions that are curried. The code below demonstrates how a curried function can be passed as an argument to another function.

```
define f1 = fun(arg1: int)(arg2: int)(arg3: int).
  (arg1 + arg2 + arg3)

define f2 = fun(arg1: function)(arg2: int).
  arg1(arg2)

define mn =
  var x = f1(1, 2);
  f2(x, 3)
```

The variable x takes on the value of a partial application. It is then passed to f2 as an argument. During the invocation of f2 the parameter arg1 acquires the value of x, a partial application. Inside of f2's definition one can see that arg1 is invoked like a function. While compiling a function definition it is important that the compiler notices that a parameter is used like a function. The compiler emits special C code to ensure that the partial application handles itself properly, an example of this can be found in the C code below.

```
void * f1( void * arg1, void * arg2, void * arg3 ) {
f1_goto: {
                void* ___tmp___1;
        ___tmp___1 = (int)arg1 +(int)arg2 +(int)arg3;
        return ___tmp___1;
}
}
aust
partial_app_header * partial_app_header_f2;

void * f2( void * arg1, void * arg2 ) {
f2_goto: {
                void* ___tmp___2;
                void * __otherTMP0;
        void * __arr[] = { arg2 };

        if((((partial_app*)arg1)->numArgsNeeded - 1) == 0){
                int i;
                for(i = 0; i < 1; i++){
                        (((partial_app*)arg1)->args[((partial_app*)arg1)->numArgsNeeded - 1]) = __arr[i];
                        ((partial_app*)arg1)->numArgsNeeded--;
                }
                __otherTMP0 = ((FUN_PTR)((partial_app*)arg1)->header->entry)(arg1);
        }
        else if((((partial_app*)arg1)->numArgsNeeded - 1) < 0){
                __otherTMP0 = arg1;
                 int __count = 0;
                while(__count < 1){
                        (((partial_app*)__otherTMP0)->args[((partial_app*)__otherTMP0)->numArgsNeeded - 1]) = __arr[__count];
                        ((partial_app*)__otherTMP0)->numArgsNeeded--;
                        __count++;
                        if(((partial_app*)__otherTMP0)->numArgsNeeded == 0){
                                __otherTMP0 = ((FUN_PTR)((partial_app*)__otherTMP0)->header->entry)(__otherTMP0);
                        }
                }
        }
        else{
                int __i;
                for(__i = 0; __i < 1; __i++){
                        (((partial_app*)arg1)->args[((partial_app*)arg1)->numArgsNeeded - 1]) = __arr[__i];
                        ((partial_app*)arg1)->numArgsNeeded--;
                }
                __otherTMP0 = arg1;

        }
___tmp___2  =  __otherTMP0;;
        return ___tmp___2  ;
}
}

static void* mn;
void mn0(){
        void* x;
        void* ___tmp___3;
                void * __otherTMP1 = MakePartialApp(3);
((partial_app*)__otherTMP1)->header = partial_app_header_f1;
        ((partial_app*)__otherTMP1)->args[((partial_app*)__otherTMP1)->numArgsNeeded - 1]  = 1;
        ((partial_app*)__otherTMP1)->numArgsNeeded--;
        ((partial_app*)__otherTMP1)->args[((partial_app*)__otherTMP1)->numArgsNeeded - 1]  = 2;
        ((partial_app*)__otherTMP1)->numArgsNeeded--;

        if (((partial_app*)__otherTMP1)->numArgsNeeded == 0){
                __otherTMP1 = ((FUN_PTR)((partial_app*)__otherTMP1)->header->entry)(__otherTMP1);
        }

___tmp___3  = __otherTMP1;;
        x  = ___tmp___3;
        void* y;
        void* ___tmp___4;
                void * __otherTMP2 = f2(x, 3);
___tmp___4  = __otherTMP2;;
        y  = ___tmp___4  ;
        mn  = y ;
}
```

In the emitted C, f2's definition contains quite a bit of code to determine if arg1, a partial application, should be executed when it is passed another argument. The code evaluates whether or not the function has too few, too many, or just enough arguments to execute.

The only time a function may have already been defined but is unknown in certain parts of code is when a function is a parameter for another function. Any function definition that requires a function as an argument does not know which function is passed to it. This results in unknown functions inside the body of function definitions. As displayed above, if a parameter is invoked like a function while compiling a function definition the compiler must produce logic that handles passing arguments to partial applications. The extra logic evaluates whether or not a partial application is ready to execute. In spite of the extra code that is emitted, partial applications are very useful and are a necessity of every functional language.

3.3.2 Tail Recursion Optimization

Anytime the last expression inside of the body of a function definition is a function call to itself, a recursive call, it is *tail recursive*. Execution of this code moves from the bottom of the body to the top of the body of code. This works just like a loop. Normally with recursion the compiled program pushes an environment onto stack memory and moves into the new recursive function call with a free environment to write local variables to. With tail recursion it is not necessary to push onto stack memory; the code can be transformed into a loop. The BLAISE compiler implements this optimzation. Below is a tail recursive function

```
define h = fun(arg1: int).
  # body of function
  h(new_arg1) # tail recursive call
```

When the example function above is compiled to C, a 'goto' statement and target label is used to turn the function into a loop.

```
void * h(void * arg1){
  h_goto:
    /* body of function */
    /* tail recursive call removed */
    arg1 = new_arg1;
    goto h_goto;
}
```

Notice how the recursive call is turned into variable reassignments and a goto statement that moves execution to a target, e.g. 'h_goto:' at top of the function body. Tail recursion optimization saves the machine from consuming unnecessary resources, such as stack memory, and significantly decreases execution time.

3.3.3 Foreign Function Interface

BLAISE has a Foreign Function Interface (FFI), which means functions from an external library can be used in the programmer's source code. This external library is written in C so external functions that are invoked don't need to be translated from BLAISE to C. I originally modified the compiler so arrays could be made in BLAISE. Initially, the compiler to looked for pre-defined function names at every function call. All array function calls were intercepted by logic that utilized specific external functions. I abstracted this idea and re-factored the compiler to have a FFI. Now the compiler looks for external function declarations. These external declarations tell the compiler which functions come from an external library. Every time the programmer invokes a function the compiler checks to see if it is declared external. If it is, the compiler trusts that the programmer is calling a supported external function. Here is a snippet of BLAISE code.

```
external Array_make [ typeInfo ] : [ typeInfo ]
external Array_get [ typeInfo ] : [ typeInfo ]

define mn =
    var arr = Array_make(5,1);
    print_int(Array_get(arr, 0))
```

Foreign Functions are useful because it is easier for program language developers to supplement a programming language without modifying the compiler.

3.3.4 Other Contributions

Debugging

Anyone that has written software knows debugging consumes a lot of time, and given the scale of this project, I spent a fair amount of time debugging code. I learned very quickly that chasing down bugs in a compiler is much more difficult than in the other software projects I've worked on. First, OCaml doesn't have a graphical user interface debugger so I did all of my debugging via injected comments in compiled code. Second, the compiler is built with mutually recursive functions so the same debug comments are printed numerous times. I found tracing through these debug comments can be very disorienting.

More often than not a lot of the bugs introduced to the compiler came from unexpected dependencies between different pieces of code. For example, the function that decides how to print partial applications uses the return value that is passed as an argument to numerous functions. At one point in development the standard return value changed, therefore all of the logic that looked for the standard return value had to change. This experience has helped me reconfirm how important it is to design software to be as modular as possible, especially before implementation, this is discussed in further detail in ’Lessons Learned’.

Develop Test Scripts

Software development has three key aspects - design, develop, and test. Unfortunately, automated-testing for BLAISE was an afterthought that came after quite a bit of manual testing. As discussed in ’Lessons Learned’ automated regression testing is a must-have for long term projects. To my dismay automated regression testing isn’t used much in practice at the University of Iowa. We attempted to set up an automated test infrastructure, but, due to some external complications, BLAISE doesn’t have automated testing. To circumvent our regression testing issue I developed a Bash script, test.sh, that helps developers verify that existing features in the compiler still work. Test.sh executes a series of BLAISE files and prints the output and expected output of each Blaise file. If there is an inconsistency between the true and expected output the developer can easily recognize when a bug needs to be fixed.

3.3.5 Evaluation and Case Studies

BLAISE is meant to provide high performance, therefore it is essential that we evaluate how fast it is compared to an alternative programming language. I ran a series of tests to compare BLAISE to OCaml on a well known algorithm: mergesort. There are two implementations of mergesort. One implementation, found in mergesort.bls and mergesortMutable.ml, which takes advantage of mutable memory. The second implementation, mergesort.ml, follows the conventional functional paradigm and does not mutate memory. Six average execution times were compared: BLAISE with and without gcc optimization, OCaml with mutation with and without GC, and the conventional FP OCaml implementation with and without GC. Figures 3.1 to 3.5 compare average runtimes for all of the implementations.

Comparison to Ocaml

BLAISE executes faster than OCaml on lists larger than 10,000 elements, although, OCaml sorts lists with less than 10,000 elements faster than BLAISE. Each mergesort implementation creates and sorts 100 lists, and the average running time is used to compare different implementations. The following graphs are standardized to the mutable OCaml implementation which is the baseline, '1'. The independent variable is the size of the list to be sorted (number of elements) and time is the dependent variable. The list sizes start at one hundred elements and go up to one million elements. For our purposes we pay most attention to BLAISE's performance with gcc optimization turned on.

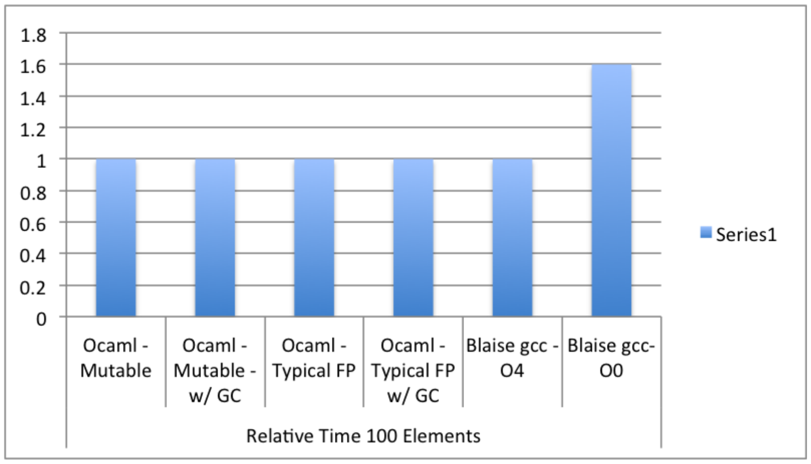

Figure 3.1: *List size = 100 elements.*

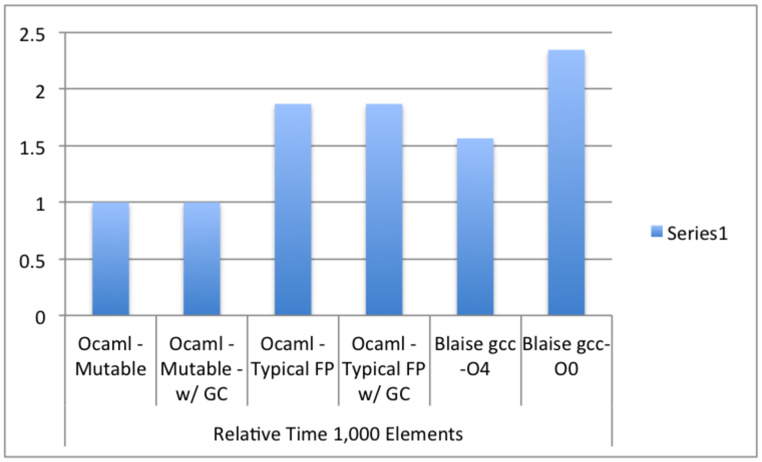

Figure 3.2: *List size = 1,000 elements.*

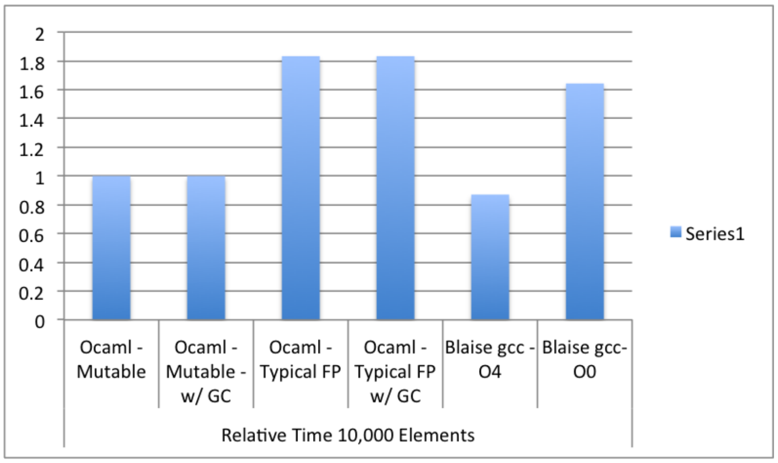

Figure 3.3: *List size = 10,000 elements.*

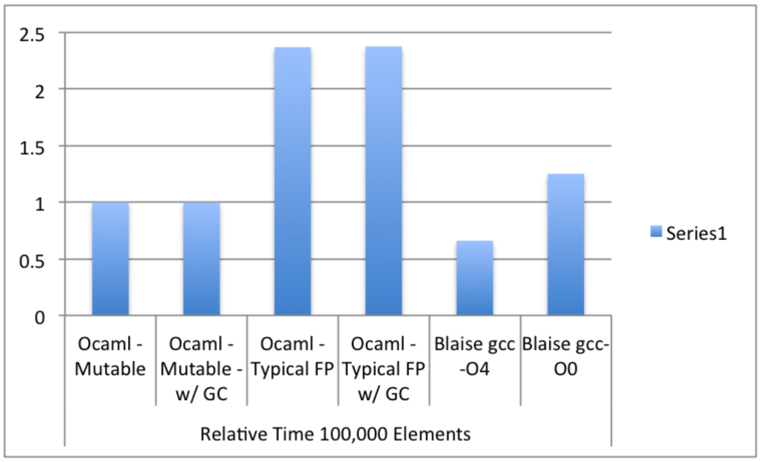

Figure 3.4: *List size = 100,000 elements.*

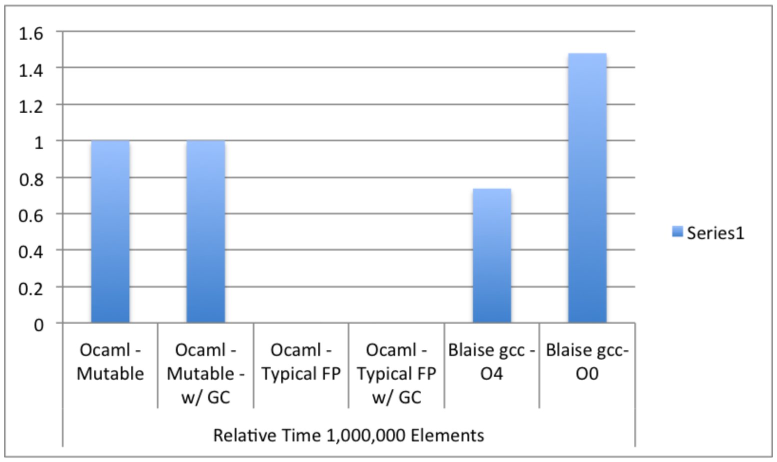

Figure 3.5: *List size = 1,000,000 elements. The conventional OCaml FP implementation is excluded from this graph because the machine experiences a stack overflow, due to the large list size.*

Chapter 4

Conclusion

4.1 Next Steps

The BLAISE Compiler is complete; however, BLAISE still needs a type-checking system before it can be used to find memory errors with static analysis. Also, when BLAISE implementation is complete it needs to go through more benchmark testing on both basic datatype-benchmarks and examples from real-time systems [7]. After this semester Dr. Stump will continue to bring students onto the BLAISE project to finish a type checking system. Then BLAISE will be used to research the practical use of explicit memory management with abstractions and use of static analysis to find memory errors.

This semester marks the end of my undergraduate career. However, I am enrolled in the computer science BS/MCS program, so I will attend the University of Iowa next fall as a graduate student. There is a chance that I will work on BLAISE in the future.

4.2 Discussion

4.2.1 My Experience

The time I spent on BLAISE is invaluable: everyone that attends college should pursue a guided independent study. I believe working on an ongoing project is experience everyone should have before going into industry. Furthermore, I had a great time learning more than I ever imagined about programming language design and implementation. Due to the large amount of time I spent developing BLAISE a lot of the magic behind compilers has been stripped from my mind. Also, I have a new found respect for research, programming languages and compiler developers. Lastly, BLAISE re-inspired my interest in academia and I look forward to graduate school next year.

4.2.2 How does this influence my thinking?

BLAISE development has helped me compare the software development process between two different institutions, academic research and industry. I worked for State Farm Insurance for two and a half years and I spent one academic year working on BLAISE. I've noticed that the academic software development process is much more dynamic, because of the nature of school, students come and go. It is hard to maintain continuity on a project aside from the project advisor. In academia a considerable amount of time can potentially be spent bringing new project members up to speed. Generally, industry employees tend to work and focus solely on a project for a longer period of time: e.g., employees forgo moving to new projects until a milestone is hit. In academia students are around on a semesterly basis with frequent breaks, such as summer and winter break. Furthermore, students are split between coursework and research. There are more obstacles that limit software development productivity in academia compared to industry.

I also observed, based on my experience at State Farm, that it is easier to receive project support by the institution in industry. For example, at State Farm I needed a server to supplement a project. Within a week I was given a server by a support group. At the University of Iowa our research team asked our Computer Support Group (CSG) to configure a server with software useful for automated testing, but we were denied our request. Given CSG's limited resources and ample requests to maintain student technical support our research project wasn't granted support.

This was the first time I've been denied service by a support group. I asked for Hudson 'Continuous Intergration' to be installed on a university server to aid our group's software development process. Hudson is built for automated testing so a developer can configure Hudson to automatically run tests on a project residing in a repository and notify the development team of any problems. To compensate for not acquiring Hudson I created a Bash script, test.sh, which executes various test files and prints out their status. The pitfall of this approach is that it is up to the compiler developers, Josh and I, to manually execute test.sh which tests a slew of BLAISE files. Although this solution isn't perfect it serves as a sufficient workaround for testing. Not receiving support from CSG was a new experience for me. It helped me realize that organizations face hard problems when it comes to managing resources and prioritizing goals. At a high level, The University of Iowa wants to educate students and allow professors to conduct research and publish their results. Although both of these goals are important the university must allocate its finite resources meticulously and as a rule of thumb: top priority goals receive more resources than the rest.

Lastly, the more time I spend on large scale projects I realize that documentation is essential. Regardless of where software development takes place, industry or academia, it is important that a project has specifications that express what the end product will be and is a guide to software developers in terms of product design. In both BLAISE and some larger scale projects at State Farm it would have been nice to have specifications from the beginning of development. Albeit, it can be fun to work on a semi-ad hoc software project. Documentation and high-level designs, e.g. a flow chart, which display how constituents connect to other constituents, is useful. Furthermore, it is important to maintain documentation for an ongoing project which outlines: how a project is assembled, how it is tested, its status, what needs to be done, and who to should be contacted for questions. Documentation preserves valuable knowledge that is useful before, during and after product development.

4.2.3 Lessons Learned

Modularization

Breaking a large project into modular components is extremely helpful especially before project development begins. Software development with a team can be a lot of fun, but it is hard to manage properly. Usually teams are formed to build large software because a team is typically more productive than an individual. A modular design allows members to work on their constituents without having to depend on nuances of their team's code. Without modularization it is hard for teams to continuously communicate important information to all team members. Team members don't want to break one another's code. A strong modular design from the beginning of software development can alleviate un-intended code dependencies later on.

Early in the spring semester I added tail recursion optimization to BLAISE. It worked quite well for a while until a team member changed a piece of code unrelated to tail recursion. A relatively minor code change lead to an unexpected break in the tail recursion optimization. Sadly, this created more work for the team. We had to figure out what caused tail recursion optimization to break and then we had to figure out how to fix the problem. Ideally, with a modular design, a team can anticipate potential side effects from changing code and handle problems as they arise, instead of finding problems weeks after they occur.

Regression testing

Automated testing is a godsend and should be used whenever possible, especially regression testing. Near the end of the fall semester I found that Josh and I were manually compiling scripts from BLAISE to C to verify that the compiler was working properly. This procedure was fine for our initial tests, but as we added more features to the compiler this became a problem. I noticed after a while that our tests were quite narrow in scope. It was common practice to create one BLAISE script that was directly impacted by the most recent feature we worked on, and compile it. We didn't bother to test parts of the compiler that worked in the past. The assumption that 'it worked before, it should work now,' is a bad approach to software development. I learned how this practice isn't too effective while refactoring code that supposedly worked, I found that it had been broken. Automated regression testing brings bugs to the developer's attention much sooner than if the developer stumbles upon it.

References

[1] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: improving region-based analysis of higher-order languages. *SIGPLAN Not.*, 30:174–185, June 1995.

[2] Andrew Appel. Garbage collection can be faster than stack allocation, 1987.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[4] Morgan Deters and Ron K. Cytron. Automated discovery of scoped memory regions for real-time java. In *Proceedings of the 3rd international symposium on Memory management*, ISMM '02, pages 132–142, New York, NY, USA, 2002. ACM.

[5] Benjamin Goldberg. Functional programming languages. *ACM Comput. Surv.*, 28:249–251, March 1996.

[6] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: A type-safe dialect of C. *C/C++ User's Journal*, 23(1), January 2005.

[7] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management, 2005.

[8] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *In Proc. of the 4th international symposium on Memory management (ISMM)*, pages 73–84, 2004.

[9] Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.*, 16:415–449, July 2006.

[10] Michel Sintzoff. Calculating properties of programs by valuations on specific models. In *Proceedings of ACM conference on Proving assertions about programs*, pages 203–207, New York, NY, USA, 1972. ACM.

[11] Aaron Stump. Blaise: Memory-safe programming without garbage collection, 2010. This is an internal document available from the author on request.

Appendices

Appendix A

BLAISE Grammar

A.1 Blaise.gra

```
blaise

(* commands *)
Program : program -> { extern }* { command }*

Extern : extern -> EXTERNAL ID typ COLON tp
Typ : typ -> LSQUARE tp { COMMA tp }* RSQUARE

Datatype : command -> DATATYPE { dt }+ { alias }* { update }* END
Def : command -> DEFINE ID opt_tp EQUALS trm

(* datatype statements *)
Dt : dt -> TYPE ID { TVAR }* { ctor }* END
DtCtor : ctor -> ID COLON cell_tp
DtAlias : alias -> ALIAS ID DOT ID   ID DOT ID
DtUpdate : update -> UPDATE  ID TO ID

(* cell types *)
CellTp : cell_tp -> CELL opt_tparams inputs DOT ID opt_inst

(* terms This has changed*)
Fun : trm -> FUN opt_tparams inputs DOT trm
Rec : trm -> FUN ID opt_tparams inputs COLON tp DOT trm
Case : trm -> CASE atom OF branches
Body : trm -> body
Seq : body -> simple SEMI body
Let : body -> VAR varId EQUALS simple SEMI body
Simple : body -> simple

VarId : varId -> ID opt_tp

(* case branches This has changed*)
BranchesStart : branches -> branch
BranchesCons : branches -> branch BAR branches
BranchesDefault : branches -> UNDERSCORE ARROW body
BranchInt : branch -> INT ARROW body
Branch : branch -> ID ARROW body

(* simple terms.  These behave somewhat like imperative statements. *)
Assign : simple -> derefplus  EQUALS  simple
Update : simple -> UPDATE deref TO ID [ LPAREN comma_atoms_e RPAREN ]
Connect : simple -> CONNECT  derefplus  derefplus
Assert : simple -> ASSERT deref EQUALS deref
Delete : simple -> DELETE deref
Clone : simple -> CLONE deref
Disconnect : simple -> DISCONNECT derefplus
Atom : simple -> atom
Reassign : simple -> ID EQUALS simple

(* deref terms *)
DerefAccess : deref -> ID { DOT ID }*
DerefPlus : derefplus -> ID { DOT ID }+

(* atomic terms This has changed *)
(*Tuple : atom -> LPAREN atom { COMMA atom } + RPAREN*)
New : atom -> NEW ID opt_inst LPAREN comma_atoms_e RPAREN
Arith : atom -> addition
StringLit : atom -> STRINGLIT

(* types This has changed*)
TFun : tp -> TFUN opt_tparams LPAREN comma_atom_tps RPAREN DOT atom_tp
TAtom : tp -> atom_tp
```

```
TVar : atom_tp -> vtype
StringType : vtype -> STRING
TVarType : vtype -> TVAR
IntType : vtype -> INTTYPE
TApp : atom_tp -> ID opt_inst
TTuple : atom_tp -> LSQUARE comma_atom_tps RSQUARE
TTp : atom_tp -> LPAREN tp RPAREN

(* Basic Operations *)
Addition : addition -> factor { ops factor }*

AddOpPlus : ops -> PLUS
AddOpMinus : ops -> MINUS
MulOpTimes : ops -> TIMES
MulOpDivide : ops -> DIVIDE
MulOpMod : ops -> MOD
MulOpGT : ops -> GT
MulOpGTE : ops -> GTE
MulOpLTE : ops -> LTE
MulOpLT : ops -> LT
MulOpEQ : ops -> EQ
MulOpNotEQ : ops -> NEQ
MulOpAND : ops -> AND
MulOpOR : ops -> OR

App : factor -> deref opt_inst LPAREN comma_atoms_e RPAREN
FactorInt : factor -> INT
FactorDeref : factor -> deref
Abort : factor -> ABORT
Trm : factor -> LPAREN trm RPAREN
Unary : factor -> unaryop factor

UnaryNot : unaryop -> NOT
UnaryNegate : unaryop -> MINUS

(* input lists *)
InputsNone : inputs -> LPAREN RPAREN
InputsSome : inputs -> { LPAREN [ consumption_anno ] ID COLON tp RPAREN }+

ConsumeAnno : consumption_anno -> CONSUME
UpdateAnno : consumption_anno -> UPDATE

(* helpers This has *)
OptTParams : opt_tparams -> [ LSQUARE vtype { COMMA vtype }* RSQUARE ]

CommaAtomse : comma_atoms_e -> [ atom { COMMA atom }* ]

OptInst : opt_inst -> [ LSQUARE tp { COMMA tp }* RSQUARE ]

CommaAtomTps : comma_atom_tps -> [ atom_tp { COMMA atom_tp }* ]

(* opt_tp *)
OptTp : opt_tp -> [ COLON tp ]

(* lexical classes *)
DATATYPE="datatype"
EXTERNAL="external"
ABORT="abort"
TYPE="type"
TFUN="Fun"
VAR="var"
COMMA=","
DELETE="delete"
CLONE="clone"
CELL="Cell"
ASSERT="assert"
UPDATE="update"
CONSUME="consume"
CONNECT="connect"
DISCONNECT="disconnect"
TO="to"
END="end"
LT="<"
GT=">"
LTE="<="
GTE=">="
LPAREN="("
RPAREN=")"
LSQUARE="["
RSQUARE="]"
ALIAS="alias"
COLON=":"
EQ="=="
NEQ="!="
AND="&&"
OR="||"
NOT="not"
EQUALS="="
DOT="."
FUN="fun"
CASE="case"
OF="of"
ARROW="->"
```

```
BAR=”|”
NEW=”new”
SEMI=”;”
DEFINE=”define”
STRING=”string”
INTTYPE=”int”
UNDERSCORE=”_”
PLUS=”+”
MINUS=”−”
TIMES=”∗”
MOD=”\%”
DIVIDE=”/”

TVAR = {{ ’\’’ [’a’−’z’ ’A’−’Z’]+ }}
ID = {{ [’a’−’z’ ’A’−’Z’] [’a’−’z’ ’A’−’Z’ ’0’−’9’ ’\’’ ’_’]∗ }}
INT = {{ [’0’−’9’] [’0’−’9’]∗ }}
STRINGLIT = {{ ’”’ ((_#[’\\’ ’”’]) | (’\\’ _ ))∗ ’”’ }}
```

Appendix B

Source Code

B.1 Mergesort.bls - BLAISE

```
# Node that will go inside of the list
datatype
  type node
        cons: Cell(number: int)(next: cons).node
     nil: Cell().node
  end
end

define get_list_length = fun(node: 'a).
    case node of
        nil -> 0
    | cons -> (1 + get_list_length(node.next))

# This function returns a pointer to the second list.
define splitList = fun(node: 'a)(currIndex: int)(startIndex: int).
    (case currIndex < startIndex of
        1 ->
            (case node of
                nil -> nil
            |   cons ->
                        splitList(node.next, (currIndex + 1), startIndex)
            )
        | 0 ->
        (case currIndex == startIndex of
            1 -> (case node of
                        nil -> nil
                    |   cons -> var secondList = node.next;
                                    node.next = nil;
                                   secondList
                    )
            |   0 -> nil
            )
    )

# print every element in a list.
define printList = fun(node: 'a).
    case node of
        nil -> print_string(" ")
    |   cons -> print_int(node.number);
                        print_string(" ");
                        printList(node.next)

define append = fun(l1: 'a)(l2: 'a).
    case l1 of
        nil -> l2
    | cons -> l1.next = l2; l1

# mergeInPlace demonstrates tail recursion.
# and was used to test tail-recursion optimization.

define mergeInPlace = fun(l1: 'a)(l2: 'a)(b: a)(endd: a).
    case l1 of
        nil ->
                (case endd of
                    nil -> l2
                | cons -> endd.next = l2; b
                )
    | cons ->
            (case l2 of
                    nil ->
                        (case endd of
                                nil -> l1
                        | cons -> endd.next = l1; b
                        )
          |    cons ->
                         (case l1.number < l2.number of
                           0 ->
                               (case l2 of
                                   nil -> nil # This should never be reached
                               |       cons ->
                                           var templ2Tail = l2.next;
                                           l2.next = nil;
                                           var tmp = append(endd, l2);
                                           (case b of
                                               nil -> b = tmp
                                           );
                                           mergeInPlace(l1,templ2Tail, b, l2)
                               )
                           | 1 ->
                               (case l1 of
                                       nil -> nil # This should never be reached
                               |       cons ->
                                           var templ1Tail = l1.next;
                                           l1.next = nil;
                                           var tmp = append(endd, l1);
                                           (case b of
                                                   nil -> b = tmp
                                           );
                                           mergeInPlace(templ1Tail,l2, b, l1)
                               )
                        )
          )


define createListHelper = fun(number: int)(l : list).
    case number of
        0 -> (case l of
                        nil -> nil
                |              cons -> l.next = nil
                )
    | _ ->
        (case (number % 2) == 0 of
                1 -> var tmp = new cons((number + 5), nil);
                        (case l of
                            nil -> nil
                        |       cons -> l.next = tmp
                        );
                        createListHelper((number - 1), tmp)
         |      0 -> var tmp = new cons(number, nil);
                            (case l of
                                nil -> nil
                            |   cons -> l.next = tmp
                            );
                            createListHelper((number - 1), tmp)
            )

define createList = fun(number:int).
    var begin = new cons(0,nil);
    createListHelper((number - 1),begin);
    begin

define sort = fun(list: 'a)(listLength:int).
    (case listLength <= 1 of
        1 -> list
    |   0 ->
        var listLengthDivTwo = (listLength/2);
        var list2 = splitList(list, 1, listLengthDivTwo);
        var sub1Sorted = sort(list, listLengthDivTwo);
        var sub2Sorted = sort(list2, (listLengthDivTwo + (listLength \% 2)));

        var mergedList = mergeInPlace(sub1Sorted, sub2Sorted, nil, nil);
        mergedList
    )

define loop = fun(iterationsLeft: int)(sizeOfList: int).
    case iterationsLeft > 0 of
    1 ->
        var lst = createList(sizeOfList);
        var sorted = sort(lst, sizeOfList);
        loop ((iterationsLeft - 1), sizeOfList)

define mn =
    loop(100, 100)
```

B.2 MergesortMutable.ml - Mutated memory - OCaml

```
(*
        Austin Laugesen
        Mergesort
*)

(* Create a list of random numbers up to a specified length *)
let rec create_list (number:int) =
        if(number > 0) then
        (
                if( (number mod 2) = 0) then
                        (number + 5)::(create_list (number -1))
                else
                        number::(create_list (number -1))
        )
        else
        (
                []
        )
;;

let rec splitLists list (currIndex: int) (midPoint: int) =
    match list with
            x::list' -> if(currIndex = midPoint) then (
                                                ([],list)
                                        )
                                        else(
                                                let tmp = (splitLists list' (currIndex + 1) midPoint) in
                                                ( x::fst(tmp) , snd(tmp))
                                        )
        | _ -> (* This point should never be reached. *)
                        ([], [])

;;

(* Merge two lists *)
let rec merge list1 list2 =
        match list1 with
                [] -> list2
        | x::list1' ->
                match list2 with
                        [] -> list1
                | y::list2' ->
                        if(x > y) then
                        (
                                y::(merge list1 list2')
                        )
                        else
                        (
                                x::(merge list1' list2)
                        )
;;


(* Sort a list using Mergesort *)
let rec sort list sizeOfList =
        if((List.length list) <= 1) then
        (
                list
        )
        else
        (
                let twoLists = splitLists list 0 (sizeOfList/2) in
                let sublist1 = fst(twoLists) in
                let sublist2 = snd(twoLists) in

                let sub1Sorted = sort sublist1 (sizeOfList/2) in
                let sub2Sorted = sort sublist2 ((sizeOfList/2) + (sizeOfList mod 2)) in

                (merge sub1Sorted sub2Sorted)

        )
;;

let rec printList list =
        match list with
                [] -> print_string "\n"
        |       x::list' -> print_int x;
                                        print_string " ";
                                        printList list'
;;

let rec loop iterations sizeOfList =
        if(iterations > 0) then
        (
                let list = create_list sizeOfList in
                let sortedlist = (sort list sizeOfList) in
                (*(printList sortedlist); *)
                (loop (iterations - 1) sizeOfList)
        )
;;

(loop 100 1000);;
```

B.3 Mergesort.ml - OCaml

```
(* Create a list of random numbers up to a specified length *)
let rec create_list (number:int) =
    if(number > 0) then
    (
        if( (number mod 2) = 0) then
            (number + 5)::(create_list (number -1))
        else
            number::(create_list (number -1))
    )
    else
    (
        []
    )
;;

(*
    Given a list and two indices create a smaller list consisting of all the elements
    between the specified indices
*)
let rec get_sub_list list curr startIndex endIndex =
    if(curr < startIndex) then
    (
        match list with
            [] -> []
        | x::list' -> (get_sub_list list' (curr + 1) startIndex endIndex)
    )
    else
    (
        if(startIndex = endIndex) then
        (
            []
        )
        else
        (
            match list with
                [] -> []
            | x::list' -> x::(get_sub_list list' (curr + 1) (startIndex + 1) endIndex)
        )
    )
;;

(* Merge two lists *)
let rec merge list1 list2 =
    match list1 with
        [] -> list2
    | x::list1' ->
        match list2 with
            [] -> list1
        | y::list2' ->
                if(x > y) then
                (
                    y::(merge list1 list2')
                )
                else
                (
                    x::(merge list1' list2)
                )
;;

(* Sort a list using Mergesort *)
let rec sort list sizeOfList =
    if((List.length list) <= 1) then
    (
        list
    )
    else
    (
        let sublist1 = (get_sub_list list 0 0 (sizeOfList/2)) in
        let sublist2 = (get_sub_list list 0 (sizeOfList/2) (sizeOfList)) in

        let sub1Sorted = sort sublist1 (sizeOfList/2) in
        let sub2Sorted = sort sublist2 ((sizeOfList/2) + (sizeOfList mod 2)) in

        (merge sub1Sorted sub2Sorted)

    )
;;

let rec printList list =
    match list with
        [] -> print_string "\n"
    |   x::list' -> print_int x;
                        print_string " ";
                        printList list'
;;

let rec loop iterations sizeOfList =
    if(iterations > 0) then
    (
        let list = create_list sizeOfList in
        let sortedlist = (sort list sizeOfList) in
        (* (printList sortedlist); *)
        (loop (iterations - 1) sizeOfList)
    )
;;

(loop 100 100);;
```

B.4 Python Evaluation Script

```
import os
import sys

## required argument for this script: # of iterations of mergesort in respective executed file.
## e.g. python compareTimes.py 100

def formatTimeout(timeOutFile, loops):
        f = open(timeOutFile)

        realSum = 0.0
        userSum = 0.0
        sysSum = 0.0

        count = 0

        for line in f:
                count += 1
                if(count == 1):
                        pass
                elif(count == 2):
                        minsAndSeconds = line.split()[1].split('m')
                        realSum += (int(minsAndSeconds[0]) * 60) + float(minsAndSeconds[1].split("s")[0])

                elif(count == 3):
                        minsAndSeconds = line.split()[1].split('m')
                        userSum += (int(minsAndSeconds[0]) * 60) + float(minsAndSeconds[1].split("s")[0])

                elif(count == 4):
                        count = 0
                        minsAndSeconds = line.split()[1].split('m')
                        sysSum += (int(minsAndSeconds[0]) * 60) + float(minsAndSeconds[1].split("s")[0])

        print("Average time computed from:  " + timeOutFile)

        print("   real: " + str(realSum/ loops))
        print("   user: " + str(userSum/loops))
        print("   sys: " + str(sysSum/loops))

def executeAdotOut(timeOutFile):
        print "call 'time ./a.out' 1 time and store results in " + timeOutFile
        os.system("rm " + timeOutFile)

        os.system("time (./a.out) 2>> " + timeOutFile)
        formatTimeout(timeOutFile, int(sys.argv[1]))

def compileFile(compileCommand, timeOutFile):
        print compileCommand

        # compile Ocaml or Blaise to c
        os.system(compileCommand)

        # compile C code to ./a.out
        if(timeOutFile == "blaise"):
                print "gcc -w -O0 cLibs/* mergesort_CHECK.c "
                os.system("gcc -w -O0 cLibs/* mergesort_CHECK.c ")
                executeAdotOut(timeOutFile + ".timeout")

                print "gcc -w -O4 cLibs/* mergesort_CHECK.c"
                os.system("gcc -w -O4 cLibs/* mergesort_CHECK.c ")
                executeAdotOut(timeOutFile + "Optimized.timeout")
        else:
                print("With GC")
                os.system("export OCAMLRUNPARAM=''")
                executeAdotOut(timeOutFile + ".timeout")
                print("\n Without GC")
                os.system("export OCAMLRUNPARAM='s=1024k,v=0x015'")
                executeAdotOut(timeOutFile + ".timeout")

blaise = "./blaise ../tests/mergesort.bls > mergesort_CHECK.c;"
ocaml = "ocamlopt ../tests/mergesort.ml"
ocamlTRMutable = "ocamlopt ../tests/mergesortMutable.ml;"

compileFile(blaise, "blaise")
compileFile(ocaml, "ocaml")
compileFile(ocamlTRMutable, "ocamlTRMutable")
```