SH Project - A Scheme Compiler

Asad Zaidi

University of St Andrews

14/01/2018

1 Abstract

The aim of this SH Project was to implement a compiler for a subset of Scheme detailed in the paper "An incremental approach to Compiler Construction", in which the author briefly describes its components across 24 sections. The compiler is written in Scheme coupled with a runtime in C. Originally, the paper targets x86, but my project targets LLVM IR instead. This allows me take a more modern approach as the paper is dated from 2006, to take advantage of many of the features that are bundled with LLVM including its optimiser and it's ability to compile into many other architectures.

2 Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 7,371 words long, including project specification and plan.

3 Introduction

Approaching compiler construction can be a daunting task. Ghuloum argues that too much emphasis is placed on "individual passes of the compiler", often losing focus on the bigger picture. A humorous example is supplied by Ghuloum in which Appel's book on compiler construction presents 11 chapters on the different stages, but only 6 pages on "Putting it all together".

I selected this paper as Ghuloum aimed to provide an alternative approach to compiler construction, an incremental one. Furthermore, Ghuloum aims to challenge students by engaging them in compiler elements which are often omitted such as lexical scoping, heap allocated objects and garbage collection by compiling Scheme.

However, an issue with this paper is that is dated from 2006 and many students have attempted this already. The paper targets x86 assembly. My goal was to adopt a more modern approach towards compiler construction by targeting LLVM instead while also documenting the process to highlight the differences between using x86 and LLVM. Furthermore, my goal was to gain a deeper insight into compiler construction, investigate advanced memory management and optimisation techniques. Finally, my aim was to learn Scheme, a programming language I was unfamiliar with at the start.

4 Objectives

I implemented a subset of Scheme's core forms, extended forms and a large collection of primitives in addition to the variety of elements required to construct them. In addition, I developed a sophisticated C runtime and a testing framework to compliment them. Furthermore, I extended the project by applying advanced memory management techniques including garbage collection and various optimisation techniques.

However, I was unable to meet my final primary object of a fully self-hosting compiler as this goal was far too ambitious. The task required an infeasible quantity of time and effort. However, I found that by pursuing this goal at the beginning, I researched and accomplished tasks which I found to be impossible initially like the code transformations for lambdas and assignable variables.

5 Project Layout

The project is organised across many files.

```
 - scheme.scm       - The Scheme compiler.
 - runtime.c        - The complimentary runtime.
 - run.scm          - Reading file contents and running compiler.
 - test_driver.scm  - The testing framework.
 - Makefile         - A Makefile to drive compilation and tests.
 - tests            - A required directory used by the Makefile.
```

5.1 Usage

The implementation of Scheme used is “guile” which is by default installed on the lab machines.

To run the tests:

```
make test
```

To compile a new program by typing into the shell:

```
make clean
make
```

To compile and run a specific file:

```
make clean
make file=filename
```

The previous compilation commands generates a binary named “program”, to run it execute the following command:

```
./program
```

To cleanup:

```
make clean
```

6 Runtime

The compiler generates a file named “program.ll” which contains LLVM code and contains an entry function named “scheme_entry”. The LLVM static compiler is used to compile the LLVM code into assembly for the current architecture and passed through a native assembler to produce the object file “program.o”.

The runtime is situated in “runtime.c” and is also compiled into object code to produce “runtime.o”. The runtime calls the scheme_entry function and unpacks the packed value that is returned by the entry function in order to display it.

These object files are linked together to produce the binary named “program” which can then be executed to run the compiled Scheme program.

The runtime features a garbage collector which is discussed in further detail later featuring a simple incremental memory allocator, some helper functions which are called by the compiled binary and code to display immediate values such as pairs, vectors and numbers.

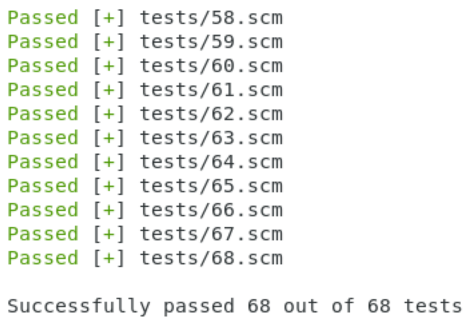
7 Test Framework

The first element in Ghuloum’s strategy for building an incremental Scheme compiler from scratch was to implement a testing framework. By implementing a testing framework, a feature can be developed and immediately tested to verify its operations.

This task involved implementing a procedure called “test” which accepts a symbolic expression and compares it to an expected value and returns success or failure. The test framework was extended in order to deal with the code transformations used in code optimisations and in order to test for output of different types.

The tests can be found in “test_driver.scm” and can be run using “make test”.

Figure 1: The testing framework



```
Passed [+] tests/58.scm
Passed [+] tests/59.scm
Passed [+] tests/60.scm
Passed [+] tests/61.scm
Passed [+] tests/62.scm
Passed [+] tests/63.scm
Passed [+] tests/64.scm
Passed [+] tests/65.scm
Passed [+] tests/66.scm
Passed [+] tests/67.scm
Passed [+] tests/68.scm

Successfully passed 68 out of 68 tests
```

The tests function by entering a new directory, copying the current makefile into it and writing the Scheme code to a file. The compiler is ran and the input is passed to the “compile_program” procedure. The LLVM generated by the compiler is written to a file which is passed to the LLVM static compiler to produce assembly for the given architecture. This is assembled and linked to generate a binary. The output generated by the binary is then compared with the expected output value printing a pass or fail message.

Ghuloum’s approach to starting the Scheme compiler by writing a testing framework was very insightful. Firstly, the testing framework allowed me to have confidence in the operations of my Scheme compiler. I was able to ensure that adding a new change to the compiler did not impact the operations of any of the previous features. Secondly, this allowed me to take the incremental approach of writing a feature and testing it. Finally, the incremental nature of adding a features and tests allowed me remain organised and to be able to accurately evaluate which phase of the compiler construction I was in.

The testing framework also allows me to evaluate my project. Currently, the testing framework features 68 tests with a 100% pass rate . Tests are included for every major feature of the compiler with the exception of garbage collection, which is demonstrated in the report thus allowing me to have confidence in my Compiler.

8 Immediate Values

This section discusses the task of implementing immediate values which constitute of integers, characters, booleans and the empty list. It also gives an insight at the code emitted by the compiler. During the compilation process, immediates are packed into a "value and tag form" and returned.

Table 1 shows the masks, tags and sizes of the packed immediates. For example, the value 42 is shifted left twice to produce the packed value 168.

Type	Mask	Tag	Size
Integer	0x03	0x00	30 bits
Character	0xFF	0x0F	24 bits
Boolean	0x7F	0x1F	1 bit
Empty List	0xFF	0x2F	0 bits

Table 1: Masks, tags and sizes of immediate values.

The following example demonstrates how the immediate value "42" is compiled.

```
>42

 define i64 @scheme_entry()   /* Header
 {
   entry:
   %tmp = alloca i64              ..  */

   store i64 168, i64* %tmp   /* Body */

   %ret = load i64, i64* %tmp /* Footer
   ret i64 %ret
 }                                ..  */
```

As there are no registers or an operand stack in LLVM, a variable representing a memory location named "tmp" is designated as the return value holder similar to how the "eax" register holds the return address in the cdecl calling convention. The variable is allocated in the header. At the end of the program, the value stored in tmp is returned. In the runtime, the “scheme_entry” function is called and displays the value returned. An advantage of using LLVM is not having to use a specific calling conventions, allocating registers and managing register cleanup in the context of caller save and callee save registers.

To supplement the immediate values, the runtime was extended to display the immediate value returned by the program. In order to display the immediate value, the type of the value is discerned by applying a mask and comparing the tag. The original value can be obtained once the type is known by bit shifting.

Originally Ghuloum’s approach to this assumes 32 bit sized registers and integers allowing for up to 30 bits of fixnums as his paper targets x86. In other implementations like MIT Scheme, fixnums are at least 24 bits in size. However, this size was insufficient for me and it was necessary for me to increase the word size in my compiler to 64 bits in order to store memory addresses of 64 bits in size which were not present during Ghuloum’s time. Although integers are usually 32 bits in size in many implementations, memory addresses can be 32 or 64 bits depending on the architecture and storing memory addresses is necessary to return pointers to objects on the heap.

This was trivially adjusted in LLVM by switching the types from i32 to i64 in LLVM.

9 Primitives

The compiler was then extended by adding unary primitives. Unary primitives are procedures which are natively defined in the environment and accept a single argument. Arguments are emitted first and the result stored in temporary variables. The result is computed and the result is returned in tmp. The process of loading operands into temporary variables is commonly shared between every primitive. However, the final computation varies between primitives. For example, in the unary primitive add1, the computation involves incrementing by one.

```
>(add1 42)

 define i64 @scheme_entry()      /* Header
 {
   entry:
   %tmp = alloca i64                     .. */
   store i64 168, i64* %tmp      /* Emit 42 */
   %var10 = load i64, i64* %tmp
   %var11 = add i64 %var10, 4    /* Add   1 */
   store i64 %var11, i64* %tmp
   %ret = load i64, i64* %tmp    /* Footer
   ret i64 %ret                          .. */
 }
```

These unary primitives can be combined to form some semblance of a program. In this example, two unary primitives are combined to increment and decrement the number 42 by 1. The result is 42 as expected. In x86 assembly, the code emitted resembles a stack machine. However in LLVM, a new label must be generated for each temporary variable. This is trivially accomplished by incrementing a global variable every time a new label is required.

```
>(add1 (sub1 42))

 define i64 @scheme_entry()
 {
   entry:
   %tmp = alloca i64
   store i64 168, i64* %tmp
   %var12 = load i64, i64* %tmp
   %var13 = sub i64 %var12, 4
   store i64 %var13, i64* %tmp
   %var10 = load i64, i64* %tmp
   %var11 = add i64 %var10, 4
   store i64 %var11, i64* %tmp
   %ret = load i64, i64* %tmp
   ret i64 %ret
 }
```

A comparison can be made between x86 and LLVM. To emit the equivalent code in x86 would involve involve simply adjusting the value stored in the eax register by adding or subtracting it. Hence, implementing unary primitives in x86 would be much simpler in LLVM. However, the advantage of using LLVM is revealed when emitting code for a binary primitive. Emitting a binary primitive in x86 involves emitting the code for an operand. This is followed by copying the value from the eax register to the stack. The stack pointer must then be subtracted. The code for the second operand is emitted and finally the result of the primitive can be computed. On the other hand, in LLVM the results emitted by the operands are simply copied into temporary variables.

10 Let and the Local Variables

The let primitive allows us to bind an expression to a symbol in a body.

```
(let ((var1 exp1)      /* Bindings
      (var2 exp2) ..)     ..       */
         body ..)      /* Body     */
```

A simple example shows the internal workings of the let primitive.

```
>(let ((x 1) (y 2)) (add x y))

define i64 @scheme_entry()       /* Header
{
entry:
%tmp = alloca i64                                  .. */

// Bindings

store i64 4, i64* %tmp           /* Emit 1 into
%var10 = load i64, i64* %tmp     symbol x (var10) .. */

store i64 8, i64* %tmp           /* Emit 2 into
%var11 = load i64, i64* %tmp     symbol y (var11) .. */

// Body

store i64 %var11, i64* %tmp      /* Load x
%var12 = load i64, i64* %tmp                       .. */

store i64 %var10, i64* %tmp      /* Load y
%var13 = load i64, i64* %tmp                       .. */

%var14 = add i64 %var13, %var12  /* Addition
store i64 %var14, i64* %tmp                        .. */

%ret = load i64, i64* %tmp       /* Footer
ret i64 %ret                                   .. */
}
```

The compiler was extended by an environment, a dictionary which maps symbols to LLVM variable names ([x var10], [y, var11], ..). Binding a variable in a let expression emits the right hand side expression and stores its value into a variable. This process is repeated for each variable and the dictionary is populated.

Ordinarily, if we had used x86, we would have had to construct stack frames, manage the stack pointer and map symbols to offsets from current frame pointer. This process is more tedious from a compiler writer's perspective. However, LLVM easily supports local variables as they can be trivially represented using LLVM variables.

```
 store i64 4, i64* %tmp        /* Emit 1 */
 %var10 = load i64, i64* %tmp  /* Load into symbol x (var10) */
```

After the binding's have been used to populate the environment, the environment is available in the body of the expression. When a symbol is referenced, a load referencing the symbol's LLVM variable is emitted. In x86, referencing a symbol would involve loading from an offset in memory so the procedure is trivial in both LLVM and x86. However, in x86 a clean up is required at the end of the let expression. In x86, the let expression needs to construct a stack frame and allocates some memory on the stack to hold the values of the variables and this process must be reversed.

```
store i64 %var11, i64* %tmp     /* Emit x (var11)
%var12 = load i64, i64* %tmp                  .. */
```

Slightly more complex programs can be constructed in which we can store the result of an expression into symbols. Ghuloum's technique for constructing an environment for variable bindings and emitting references as loads is a fairly common technique that is flexible in nature. The environment is populated recursively in nested let expressions and the technique works well from a compiler writer's perspective.

11 Let*

The let* extended form was not mentioned in Ghuloum's paper. However, the let* extended form is a very useful extension to deal with a small obstacle that is present in ordinary let bindings. In an ordinary let binding, it is not possible to reference another variable present in the same set of variable bindings. To accomplish this with a let binding involves nesting multiple let expressions and this not particularly elegant.

The syntax of the let* expression is identical to the let primitive, except the variables can be referenced in consecutive bindings without the usage of nesting. The following example illustrates the syntax of the let* expression.

```
(let* ((var1 exp1)      /* Bindings
       (var2 var1)
       (var3 var2))      ..         */
          body ..)      /* Body     */
```

The let* expression is implemented as a code transformation. The implementation involves rewriting the let* expression as a nested let expression. The code transformed version looks like the following example.

```
(let ((var1 exp1))
(let ((var2 var1))
 (let ((var3 var2))
  body ..)))
```

Although this was not mentioned in Ghuloum's original paper, this experiment gives an insight into code transformations which will be more heavily involved later when implementing more complex elements of the compiler like lambdas and variable assignment.

12 Begin

Originally, in Ghuloum's paper the begin expression is omitted. However, I believed that sequencing was a vital primitive to include. I could not imagine an implementation of Scheme in which you could not sequence instructions. Implementation wise, the begin expression emits the code for each expression sequentially. Finally, the return value of the final expression is returned. The following example demonstrates the syntax of the begin expression.

```
(begin exp1 exp2 ...)
```

13 Conditionals

The conditional primitives include if and cond.

```
(if condition consequent alternative)
```

A short example shows the implementation.

```
// A program to return the lower of x and y.
(let ([x 3]
      [y 5])
      (if (< x y) x y))
define i64 @scheme_entry()
{
 entry:
 %tmp = alloca i64
 store i64 12, i64* %tmp
 %var10 = load i64, i64* %tmp              // Let bindings
 store i64 20, i64* %tmp
 %var11 = load i64, i64* %tmp
 store i64 %var11, i64* %tmp
 %var17 = load i64, i64* %tmp
 store i64 %var10, i64* %tmp
 %var18 = load i64, i64* %tmp
 %var19 = icmp slt i64 %var18, %var17
 %var20 = zext i1 %var19 to i64
 %var21 = shl i64 %var20, 7
 %var22 = or i64 %var21, 31
 store i64 %var22, i64* %tmp
 %var15 = load i64, i64* %tmp
 %var16 = icmp eq i64 %var15, 159          // Compare Conditional
 br i1 %var16, label %lab12, label %lab13  // Branch
 lab12:                                    // Conseq
 store i64 %var10, i64* %tmp
 br label %lab14
 lab13:                                    // Altern
 store i64 %var11, i64* %tmp
 br label %lab14
 lab14:
 %ret = load i64, i64* %tmp
 ret i64 %ret
}
```

Unique labels are generated for the Consequent and Alternative branches. The conditional is compared with the packed value of true (159). This is followed by a branching instruction which jumps to the Consequent on equality or to the Alternative otherwise. Implementing conditionals increases the variety of programs that can be written. Furthermore, they can be complimented with conditional operators and many of the other extended forms and primitives to write sophisticated programs hence making conditionals a suitable choice to implement by Ghuloum.

Branching in LLVM is more robust and configurable. In x86, a compare instruction is called which performs a subtraction and this is followed by a variety of branching instructions. In LLVM the sequence of a compare followed by a branch is identical. However, the condition for branching, “eq”, is present in the compare instruction. The branching instruction also requires a consequent and alternative branch. In x86, if the condition in the branching instruction is not met, the program counter simply continues to the next instruction. However, in LLVM an alternative branch is required. From a compiler writer’s perspective, this allows leaving the task of organising and arranging branches to the LLVM static compiler.

14 Lambdas

Now that our program contains many useful primitives, the next task is to implement lambdas. Compiling lambdas is sophisticated task and involves some complex elements. In this section I discuss the challenges involved in compiling lambdas and how they were solved. In order to implement lambdas, I visited several external resources. One, which was a blog on the topic of Closure Conversion [4].

The first challenging aspect is defined by what lambdas are. The syntax of a lambda is expressed in the following example, a lambda describes an anonymous functions, a function that does not have a name.

```
(lambda (formal args) body)
```

Primcall	Description
(lambda? (formal args) body)	Constructs a lambda

Table 2: Primitives related to lambdas.

However, a lambda does not simply compile into a function pointer. This is because Scheme has lexical scoping. It is possible for an expression in the body of a lambda to contain a free variable. A free variable is one which is not defined in the formal arguments of the lambda expression. A free variable can outlive the scope that it was defined in. Currently we have been using a model where the lifespan of variables exist solely within the scope of where it was defined, this particular model suits the stack machine like structure we are exploiting to implement local variables. However, this model is not suitable for the life span of free variables.

```
The variable x is free in the body of the second lambda expression.

(lambda (x)
 (lambda (y)
  (add x y)))
```

Instead, a lambda compiles into a "closure" which is a data structure that contains a pointer to a function and the values of its free variables. During the construction of the closure, the variables that become free in the body of the lambda are present in the scope. The values of these variables is captured in the closure. A pointer to this closure is emitted. In order to call the closure, the function is extracted and a pointer to the free variables is passed to the function. The free variables are immutable at this point, but later become mutable when assignment is implemented.

14.1 Code Transformation

In order to accomplish this task, the lambda expression is transformed into an intermediate form in order to identify the elements of the lambda expression. This is necessary as we need to extract the function component of the lambda which is lifted up and emitted as a function in LLVM. In addition, the lambda is replaced by a closure which will contain a pointer to the function that was lifted up. Furthermore, the local variables and the free variables need to be identified as they are accessed in different manners. Table 3 shows the intermediate forms.

Primcall	Description
(closure lvar fvar ...)	Emits a pointer to a Closure Object on the heap.
(funcall expr args)	Calls the expr with the given args.
(labels bindings expr)	Maps function names to bindings.
(code lvar fvar expr ...)	A lambda that has been lifted.

Table 3: Primitives related to lambda intermediate forms.

14.2 Example

To demonstrate the code transformation, we shall compile a small example and explain the separate elements. This following example features a lambda expression which accepts a single argument and returns a lambda which is capable of summing the two free variables x and y which are present in it's body. In order to keep this section as simple as possible, the act of calling the function is left until the next section.

```
(let ((x 5))
  (lambda (y) (lambda () (add x y))))
```

14.2.1 Transformation into Free Variables Analysed Form

After the process of free variable analysis (currently skipped), the free variables are lifted to the top. My compiler expects code in the free variable analysed form. In this form, an extra parameter is added to store free variables. In this example, the outer lambda's parameter is populated by x and the inner lambda's parameter is populated by x and y.

```
After Free Variable Analysis

(let ((x 5))
  (lambda (y) (x) (lambda () (x y) (add x y))))
```

14.2.2 Closure Conversion Transformation

```
(labels
   ([f0 (code () (x y) (add x y)])         /* Lvar to
    [f1 (code (y) (x) (closure f0 x y))])       Code Bindings */
(let ((x 5)) (closure f1 x)))
```

In this final transformation, a top level Labels form is constructed. The lambdas are transformed to closure forms, assigned a name e.g. “f0” referred to as an lvar, assigned a LLVM function name "fun10" and its code collected in the top level Labels construct recursively. The term lvar is used by Ghuloum similar to way in which the left hand identifier refers to the right hand expression in a variable binding.

14.2.3 Code form

```
(code (formal vars) (free vars) body)
```

The code for the labels form is then emitted, each Code construct is emitted as a separate LLVM function. Several differences between LLVM and x86 can be highlighted here. In x86, a label must be emitted followed by the code of the function. The function must also be present in the text section of the binary. There is no notion of function arguments built into the language and the return type of the function is not captured as the return value is simply returned in the eax register. However, LLVM function are similar to C functions. Each function declaration involves a return type, a list of typed function arguments and a function name. This is far more elegant and simplistic than its x86 counter part. In x86, extra care must be taken to use a particular calling convention, to be aware of caller save and callee save registers and several other elements including constructing stack frames and saving and restoring frame pointers which can be tedious from a compiler writer's perspective.

The next challenge involves emitting local and free variables. In order to distinguish between local variables and free variables, the environment was updated. In the code expression's body, formal function arguments are emitted in the same manner as local variables. That is by emitting a load from a local LLVM variable which stores the value of the function argument. However, free variables are stored as an offsets from the closure pointer in the environment. In order to access these, the runtime was updated with a function to retrieve an entry from the global closure pointer given an offset.

The following example is the code emitted for the function labelled "f1".

```
// f0
define i64 @fun10()
{
 . // Omitted
}
// f1 -> (code (y) (x) (closure f0 x y))
define i64 @fun11(i64 %arg0) // Accepts 1 Argument
{
entry:
%tmp = alloca i64
%var15 = call i64 @hptr_ptr(i64 2)
call void @hptr_inc(i64 2)
// Store Function Pointer on Heap
call void @hptr_inc(i64 ptrtoint (i64 ()* @fun10 to i64))
// Access Variables through Closure Ptr
call i64 @hptr_get_freevar(i64 0, i64* %tmp)
%var19 = load i64, i64* %tmp
call void @hptr_inc(i64 %var19)
store i64 %arg0, i64* %tmp
%var20 = load i64, i64* %tmp
call void @hptr_inc(i64 %var20)
store i64 %var15, i64* %tmp
%ret = load i64, i64* %tmp
ret i64 %ret
}
define i64 @scheme_entry()
{
. // Omitted
}
```

15 Function Calls

To call a closure the funcall primitive is used. The closure being called has a function pointer element and holds the values of the free variables as well. In order to call a function, the previous value of the global closure pointer is saved and the global closure pointer is set to the function being called. The function pointer is extracted. It is stored as a number and first requires a cast to a function pointer. A call to the function pointer is issued. After the function call, the saved closure pointer is restored. The function can access free variables as offsets from the closure pointer and local variables as references to LLVM variables.

In the cdecl calling convention used by Ghuloum, arguments need to be pushed onto the stack and then a call to the function is issued. This means that issuing function calls in x86 is much simpler. As the function pointer needs to be extracted from the closure, calling the function in LLVM is a littler trickier. First the function pointer which is stored as a long needs to be casted back into a function pointer. In order to call the function, it's return type and the types of its arguments need to be specified both which are i64. On the contrary, the stronger type system alleviates the risk of many of the mistakes that can be made during the code emitting stage.

```
Input:

(let ([f (lambda (x y) () (mul x y) )])
     (funcall f 3 4))

Closure Converted:

(labels ((f10 (code (x y) () (mul x y)))) (let ((f (closure f10))) (funcall f 3 4)))


Code:

define i64 @fun10(i64 %arg0, i64 %arg1)
{
entry:
%tmp = alloca i64
... // Omitted
ret i64 %ret
}
define i64 @scheme_entry()
{
entry:
%tmp = alloca i64
... // Omitted
store i64 %var15, i64* %tmp                      // Put Clsr Ptr in Tmp
%var22 = load i64, i64* %tmp                     // Load New Clsr Ptr
%var25 = call i64 @hptr_get_clsptr()             // Save Prev Clsr Ptr
call void @hptr_set_clsptr(i64 %var22)           // Set New Clsr Ptr
%var23 = call i64 @hptr_closure_lab()            // Load Fcn Ptr from Clsr Pointer
%var24 = inttoptr i64 %var23 to i64 (i64, i64)*  // Cast Fcn from Long to Fcn Ptr
%var26 = call i64 %var24(i64 %var20, i64 %var21) // Call Fcn Ptr
call void @hptr_set_clsptr(i64 %var25)           // Restore Prev Clsr Ptr
store i64 %var26, i64* %tmp
%ret = load i64, i64* %tmp
ret i64 %ret
}
```

16 Datastructures and the Heap

Pairs, Vectors, Closures and Strings are data structures that cannot be represented by a single fixed width number. Hence, the runtime was extended by adding a heap. A contiguous region of memory is allocated to store runtime objects that cannot be represented by an immediate. By implementing these data structures, the programs that can be expressed in our implementation of Scheme increases and their quality can be more sophisticated. For example, combining pairs or vectors and closures allows many sorting algorithms can be implemented. Furthermore, the implementation of vectors allows us to implement assignable variables in the next section.

In order to compile data structures, the objects are constructed on the heap and a pointer to them is emitted. By aligning the allocated objects to a multiple of 8, the last 3 bits of a pointer can be used to store a tag, these can be found in Table 4. To minimise the effort required to implement these, I extended the runtime with functions to construct these objects give sufficient arguments. The process in x86 and LLVM would be similar as the code emitted by the compiler will simply call the helper functions in the runtime.

Type	Mask	Tag
Pairs	0x07	0x01
Closures	0x07	0x02
Vectors	0x07	0x05
Strings	0x07	0x06

Table 4: Masks and Tags of data structures.

16.1 Pairs

The primitives required for the construction of pairs include "pair?", "cons", "car" and "cdr". A pair is a collection of two objects. If the 2nd item in a pair is a list, then the pair is referred to as a list. Pairs are constructed using the "cons" primitive which increases the heap pointer twice and copies the elements of the pair into the new cells which have been allocated. A list constructed using pairs is equivalent to a linked list. The "car" and "cdr" primitives return the first and second elements.

Primcall	Description
(pair? p)	Returns true if p is a Pair.
(cons a b)	Constructs (a . b)
(car '(a . b))	Returns a
(cdr '(a . b))	Returns b

Table 5: Primitives related to Pairs.

```
;; Construct an improper list

>  (cons 3 (cons 4 5))

>> (3 4 . 5)
```

In memory, pairs are organised as the following.

```
0 [val 1]
1 [val 2]
```

The following example shows the code emitted. The construction of pairs and accessing elements on the heap are performed through the use of functions in the C Runtime.

```
>(let* ([p (cons 3 4)]
        [x (car p)]
        [y (cdr p)])
        (mul x y))
define i64 @scheme_entry()
{
entry:
%tmp = alloca i64
// var11 = 3
store i64 12, i64* %tmp
%var11 = load i64, i64* %tmp
// var12 = 4
store i64 16, i64* %tmp
%var12 = load i64, i64* %tmp
// Call C Function hptr_con(var11, var12) to construct Pair
%var13 = call i64 @hptr_con(i64 %var11, i64 %var12)
// Store pair in var10
store i64 %var13, i64* %tmp
%var10 = load i64, i64* %tmp
// Load pair into var16
store i64 %var10, i64* %tmp
%var16 = load i64, i64* %tmp
// Call C function hptr_car(var16) to retrieve Car of Pair
%var17 = call i64 @hptr_car(i64 %var16)
store i64 %var17, i64* %tmp
// Store Car of Pair in var15
%var15 = load i64, i64* %tmp
// Load pair into var19
store i64 %var10, i64* %tmp
%var19 = load i64, i64* %tmp
// Call C function hptr_cdr(var16) to retrieve Cdr of Pair
%var20 = call i64 @hptr_cdr(i64 %var19)
store i64 %var20, i64* %tmp
// Store Cdr of Pair in var18
%var18 = load i64, i64* %tmp
// Load Car
store i64 %var18, i64* %tmp
%var21 = load i64, i64* %tmp
// Load Cdr
store i64 %var15, i64* %tmp
%var22 = load i64, i64* %tmp
// Multiply Car * Cdr
%var23 = mul  i64 %var22, %var21
%var24 = sdiv i64 %var23,   4
store i64 %var24, i64* %tmp
// Return result
%ret = load i64, i64* %tmp
ret i64 %ret
}
```

16.2 Vectors

A vector is analogous to an array in C. It represents a fixed size array of memory which can be indexed and its elements can be mutated. Vectors can be used to represent assignable variables, this is because a vector represents an indirection of a single level. We can use this trick to use a vector of size 1 to represent an assignable variable that can outlive its scope as it will be stored on the heap. The primitives for the construction of pairs include "vector?", "make-vector", "vector-ref" and "vector-set!".

```
;; Produce a Vector of size 3 populated with character A.

>  (make-vector 3 #\A)

>> #(#\A #\A #\A)
```

Primcall	Description
(vector? vec)	Returns true if vec is a Vector..
(make-vector len val)	Constructs #(val, val, ..) of size len.
(vector-ref vec ind)	Accesses vec[ind]
(vector-set! vec ind val)	Sets vec[ind] = val.

Table 6: Primitives related to Pairs.

```
0 [vec len]
1 [val 1]
2 [val 2]
  .
  .
```

16.3 Closures

A closure represents a closed lambda expression and is constructed during the closure conversion code transformation step. It is an Object that is also constructed on the heap. A closure stores the function pointer followed by the values of the free variables allowing the free variables to outlive the scope they were defined in.

```
0 [fcn ptr]
1 [free v1]
2 [free v2]
  .
  .
```

16.4 Strings

Strings are stored in the same manner as vectors, the string's length is stored in the first slot. This is followed by the characters which are stored sequentially after. Strings in my implementation are limited as they are not editable. However, this functionality is similar to mutating elements in a vector and hence the implementation required to edit them is simple.

```
>  (make-string 5 #\A)
>> "AAAAA"
```

```
0 [str len]
1 [val 1]
2 [val 2]
  .
  .
```

17 Assignment

Variables in scheme can be mutated by the use of the “set!” procedure.

Primcall	Description
(set! var val)	Assigns var val.

Table 7: Primitives related to Assignment.

17.1 Code Transformation

Our implementation contains local variables, however they are immutable. In this section, we adjust our implementation by applying a code transformation and using a trick involving using vectors of size 1 to implement variable assignment.

The following example illustrates the challenges of assignment.

```
(let ((f (lambda (c) ()
       (cons (lambda (v) (c) (set! c v))
               (lambda () (c) c)))))
(let ((p (funcall f 0)))  // Call f (0) to retrieve Pair and set C to 0
     (funcall (car p) 12) // Call ((Car p) 12) to set C to 12
     (funcall (cdr p))))  // Call ((Cdr p)) to retrieve C
```

The example consists of an outer lambda “f(c)” which returns a Pair of lambdas. The former mutates c and the latter which returns c. The lambdas are executed in the following order. First, c is set to 0. It is then mutated to 12 and finally the value is returned. Comments in the example highlight the different stages.

The first challenge is that the variable C is a local variable, so it is “stack allocated”. The scope of the variable is local to it’s body and the variable will no longer exist outside of its scope. However, the variable can outlive the lambda it was declared in. Secondly, the value of c must be mutable such that its mutated value is visible to those who hold a reference to it.

These two challenges can be tackled through the use of a code transformation. It is important to note that variables are defined in a let or a lambda. The code transformation works firstly by performing a depth first search to find the assignable variables. Assignable variables can be identified as they are enclosed by the "set!" procedure. A list of assignable variables is populated and this list is used in the next step.

Once assignable variables are found, code transformation can be performed. The general idea by Ghuloum is to replace the body of a lambda or let where the assignable variable is defined by a let expression which copies the value of the variable into a vector of size 1. An alternate name for this code transformation is "Boxing". The code transformation involved was easier to implement having experienced it before while implementing lambdas.

Any references made to that variable are done through the vector version of the variable. Assignment via **set!** is replaced by **vector-set!**. Variable access is replaced by the usage of **vector-ref**. The usage of the vector accomplishes our task of letting the variable's life span exist outside of its scope. The indirection allows each closure to observe the same version of the variable, changes made to the variable are shared to its observers.

```
After code transformation.

(let ((f (lambda (c) ()
     (let ((c (make-vector 1 c)))
        (cons (lambda (v) (c) (vector-set! c 0 v))
     (lambda () (c) (vector-ref c 0)))))))
(let ((p (f 0)))
     ((car p) 12)
     ((cdr p))))
```

To evaluate, Ghuloum's strategy for implementing variable assignment and the closure conversion compliment each other very well. A flaw of the closure conversion technique is that it copies the values of variables, this means the value of the variable is not shared between copies of the closure. However, this flaw is corrected by using the assignment technique in which a vector of size 1 is used to add one layer of indirection, the vector representing the variable points to the same location in memory thus allowing the variable to be shared. Hence, by using a suitable complimentary technique, this flaw is amended.

18 Optimiser

The compiler features two optimisers, a constant propagator. The propagator is coupled with an interpreter which evaluates mathematical expressions at compile time. The aim of the optimizer is to minimise computation performed at runtime in order to increase performance and perhaps reduce the size of the emitted code to optimise for storage. The optimisations used are trivial, but impactful.

18.1 Constant Propagation

The first optimisation is constant propagation. The aim of this optimisation is to eliminate many loads and stores during run time. Without this optimisation, a known value will be repeatedly fetched from memory incurring a small time penalty from loading from cache or memory repeatedly. Time is saved by using the value directly in the computation. This optimisation is implemented as a code transformation and is illustrated in the following example.

In this example, the computation produced by the addition is a constant and it stored in the variable x. Every time the variable x is referenced, a load from memory is issued during runtime, this is not efficient.

```
(let ([x (add 1 2)] [y 3]) (add x y))
```

The first step involves applying the interpretive evaluator to compute the value of the addition in the let bindings. This allows constants to be located even if they are not explicit and need to be computed. Otherwise, the compiler would be limited to explicit constants limiting the optimisations that could be performed.

```
(let ([x 3] [y 3]) (add x y))
```

Once the constants have been identified, they need to be propagated into the body of the let expression. This is implemented as a search which populates a dictionary mapping variables to constants. Once the dictionary is populated, the variables in the dictionary are omitted from the variable bindings as the store into an LLVM variable is no longer needed. Finally, symbols in the body are replaced by their values in the dictionary demonstrated in the next example. This allows the compiler to simply emit its value during compile time, for example the value 6 is emitted for our example.

```
(let () 6)
```

This results in optimising away several loads and stores as the compiler simply emits the code for the immediate 6 as opposed to fetching a symbol's value from memory.

18.2 Compile time evaluation

Common mathematical expressions with constant values can be evaluated at compile time and the value can be emitted as opposed to code required to compute the value. The code transformation is recursive. If a mathematical expression is encountered and its arguments are constants, it is replaced by its value.

```
(let () (add 4 3))
(let () (mul 4 3))
(let () (div 4 3))
(let () (sub 4 3))
```

The compiler simply interprets these computations during compile time. The interpreter is currently able to perform this for multiplication, addition, division and subtraction. Perhaps the scope of this can improved by allowing a similar transformation for conditional expressions which can also be computed at compile time. The following example demonstrates the transformation in which the mathematical expression is replaced by its value.

```
(let () 7)
(let () 12)
(let () 1)
(let () 1)
```

19 Garbage Collection

The collector uses the mark and sweep algorithm.

A shadow stack is created on which local variables in the let bindings are pushed onto. At the end of the scope, the variables are popped from the stack. Pointers on this stack represent the roots of the run time objects. In the following example, code is emitted to push var1 and var2 onto the shadow stack, followed by executing the body and finally popping var1 and var2 from the shadow stack. The term shadow stack is used because it's a technique made available by LLVM for garbage collection. However, it is not possible for me to use this technique unless I develop my compiler using LLVM's tooling and libraries as opposed to emitting LLVM IR manually like I have. So, I have manually implemented a similar technique in my C runtime and named it after LLVM's equivalent.

```
(let ([var1 expr1] [var2 expr]) body)
```

A depth first search is initiated starting from the roots following children nodes. Nodes are marked as they are visited by marking them with a bit pattern. I exploited the fact that pointers are 48 bits in size allowing the remaining 16 bits to be used for tagging. During the sweeping phase, the heap is examined and all non-marked objects are "collected" as they are no longer reachable.

As a sophisticated memory allocator is not present, it is not possible to reclaim memory addresses that have been collected. A further extension to the project would be to write a sophisticated memory allocator which is capable of reclaiming memory addresses. As of now, the collector simply lists the memory addresses that need to be reclaimed for testing purposes.

19.1 Testing Garbage Collection

In order to test garbage collection, a new primitive was added and the runtime was updated to collect gc metrics. The metrics collected highlight the count of marked objects in memory and collected memory locations giving an insight to what the collector accomplished during a run and how it performed. This is coupled with the fact that we can trigger garbage collection manually thus allowing us to verify the functionality of our collector.

Primcall	Description
(gc)	Executes the garbage collector manually.

Table 8: Primitives related to gc.

In this first example gc is executed before the first let expression completes, the marking portion of the collector marks 1 vector and 28 pairs which are still reachable. This is because the vector “a” has been pushed onto the shadow stack and the pairs present inside of it are reachable. As these objects are marked, they are not collected. This means our collector does not collect objects which are currently live and in scope and currently on shadow stack which is the expected behavior.

```
(begin
 (let ([a (make-vector 10 (cons 1 2))]) a (gc))
)

Marked Vect 1
Marked Clsr 0
Marked Pair 10
Collected 3 memory locations
```

In this final example, gc is executed after the first let expression completes and collects all of the garbage memory locations. After the let expression is executed all of its local variables go out of scope, the vector is popped from the shadow stack. During the marking phase of the compiler, as the vector is no longer on the shadow stack, the vector and the pairs are not marked. During the sweeping phase, all unmarked memory addresses are “collected”. The metrics show that all of the allocated memory was “collected” which is the expected behaviour.

```
(begin
 (let ([a (make-vector 10 (cons 1 2))]) a)
 (gc)
)
```

```
Marked Vect 0
Marked Clsr 0
Marked Pair 0
Collected 13 memory locations
```

20 Similar Projects

As this paper is dated from 2006, similar projects already exist. The projects in general follow the paper as given and target the x86 architecture. An example of a given implementation is supplied in the bibliography [2]. This particular example is much more advanced than my project and offers many more complex code transformations including continuation passing style conversion and alpha conversion (a variable renaming conversion).

However, a large difference exists between my project and projects implementing the compiler described in the paper. This is because my compiler targets LLVM IR as opposed to x86 which means my compiler takes a more modern approach. A flaw involved with targetting x86 is that the existing implementations are locked to this architecture. However, my compiler can easily be extended to compile to a variety of architectures offered by the many backends for LLVM or to take advantage of the LLVM optimiser to produce highly optimised code for a particular architecture.

Furthermore, my compiler approaches the compilation process differently. Using x86 involves lots of book keeping, managing stack frames and keeping track of the various registers involved, for example the stack pointer and the base pointer. However, my implementation avoids dealing with these issues and takes advantages of many of the modern features available in LLVM.

21 Conclusion

To conclude, I incrementally implemented and thoroughly tested a compiler for a subset of Scheme guided by Ghuloum's paper on compiler construction. It consists of several interesting elements, featuring a subset of Scheme's core forms, extended forms and many primitives.

In doing so I furthered my insight into compiler construction, learnt Scheme and managed to tackle a lot of challenges which I felt were impossible initially. In particular, translating a lexically scoped language with closures into a lower level language like LLVM, dealing with heap allocated objects and the code transformations involved in variable assignment.

Furthermore, I documented the process of using LLVM. By adopting a more modern approach on a paper from 2006, I discovered and utilised alternative techniques that do not require book keeping memory locations and registers, bizarre calling conventions and allows my compiler to emit code that can be run on various architectures.

Finally, I extended my project by investigating advanced memory management techniques and optimisation techniques, this led to the addition of mark and sweep garbage collection and optimisation techniques such as constant propagation and compile time evaluation.

The extensions to a compiler until it's completion are numerous and allow for a huge scope. Suggested extensions in the original paper include a full numeric tower, user-defined macros, a module system, transforming into continuation passing style and tail call optimisation.

References

[1] An Incremental Approach to Compiler Construction.
`http://scheme2006.cs.uchicago.edu/11-ghuloum.pdf`

[2] A similar project based on the paper, but targets x86.
`https://github.com/namin/inc`

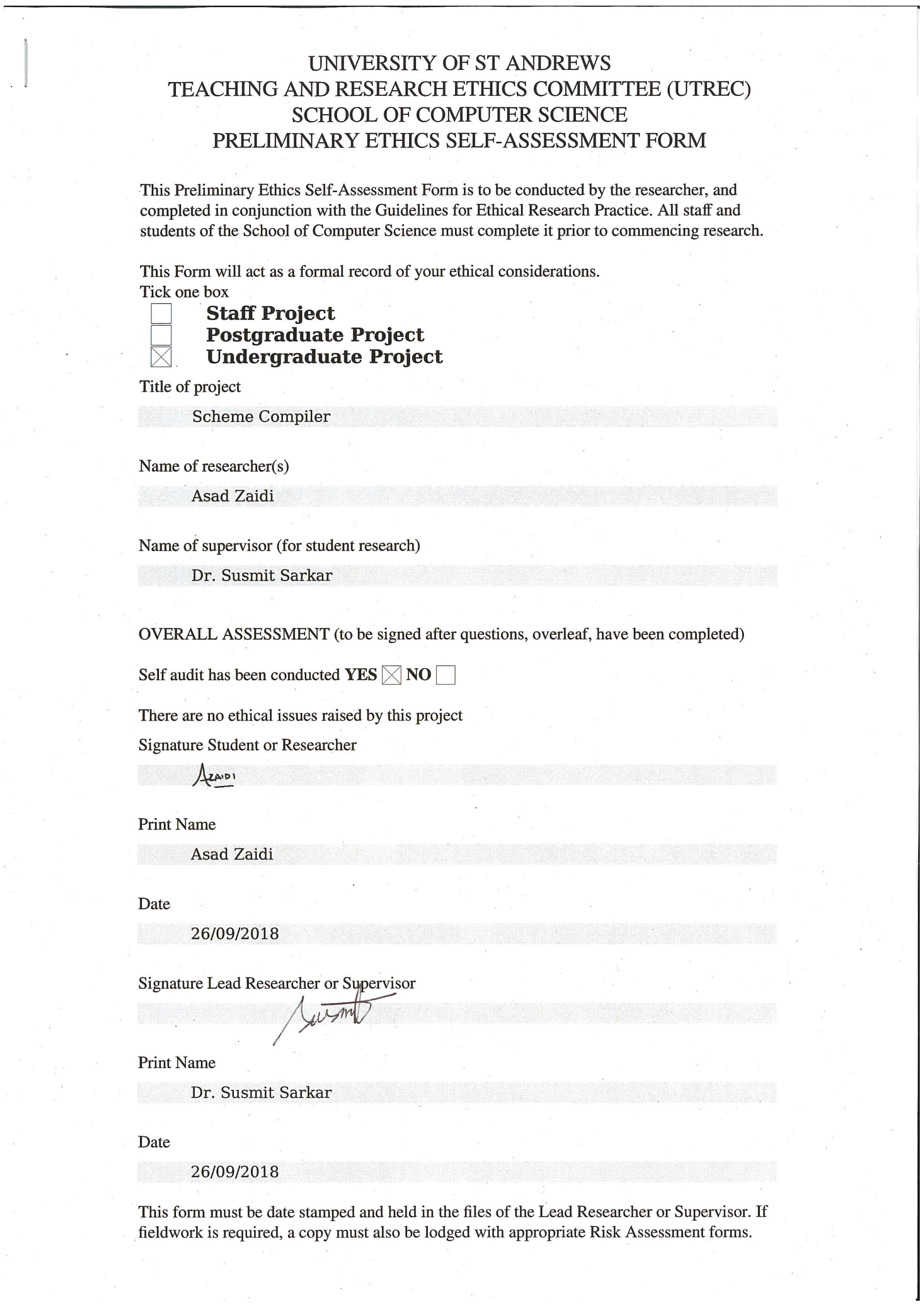
[3] An extended tutorial.
`http://www.cs.indiana.edu/∼aghuloum/compilers-tutorial-2006-09-16.pdf`

[4] Closure Conversion.
`http://matt.might.net/articles/closure-conversion/`

[5] Closure Conversion in C.
`http://matt.might.net/articles/compiling-scheme-to-c/`

Appendix

Preliminary Ethics



UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

☐ **Staff Project**
☐ **Postgraduate Project**
[✓ **Undergraduate Project**]

Title of project

Scheme Compiler

Name of researcher(s)

Asad Zaidi

Name of supervisor (for student research)

Dr. Susmit Sarkar

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted [✓ **YES**] **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher

Azaidi

Print Name

Asad Zaidi

Date

26/09/2018

Signature Lead Researcher or Supervisor

[illegible]

Print Name

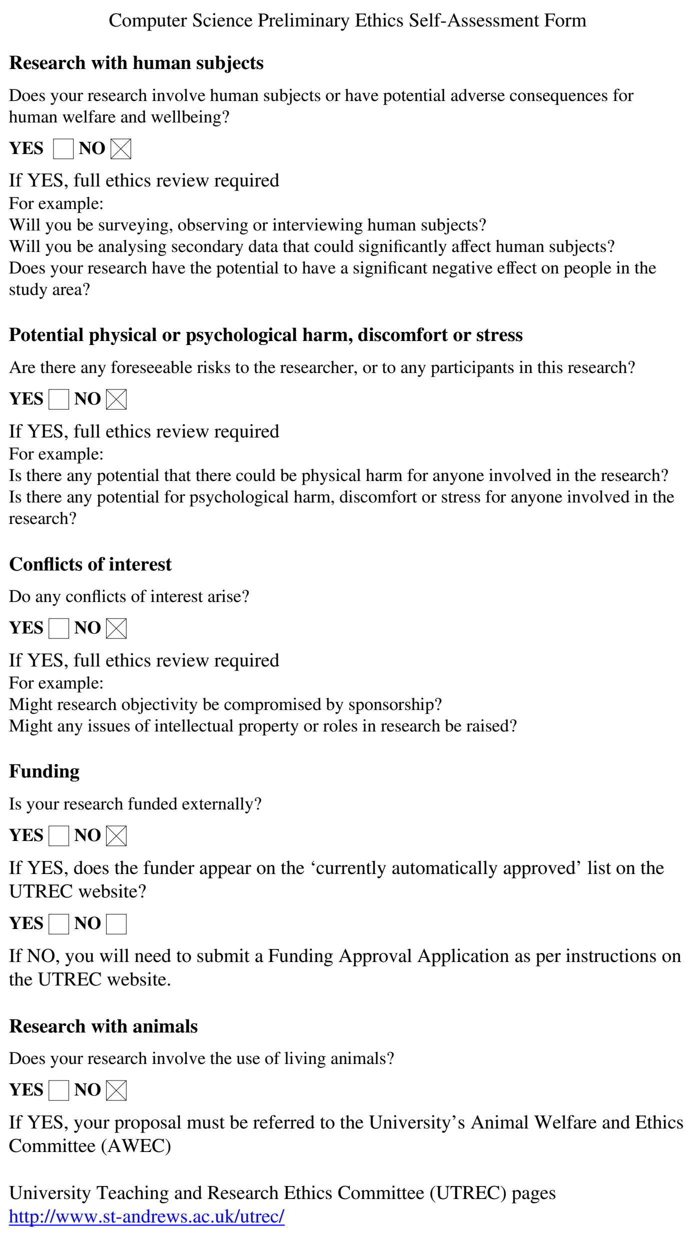
Dr. Susmit Sarkar

Date

26/09/2018

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms.

Preliminary Ethics



Computer Science Preliminary Ethics Self-Assessment Form

Research with human subjects

Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing?

YES **[✓ NO]**

If YES, full ethics review required
For example:
Will you be surveying, observing or interviewing human subjects?
Will you be analysing secondary data that could significantly affect human subjects?
Does your research have the potential to have a significant negative effect on people in the study area?

Potential physical or psychological harm, discomfort or stress

Are there any foreseeable risks to the researcher, or to any participants in this research?

YES **[✓ NO]**

If YES, full ethics review required
For example:
Is there any potential that there could be physical harm for anyone involved in the research?
Is there any potential for psychological harm, discomfort or stress for anyone involved in the research?

Conflicts of interest

Do any conflicts of interest arise?

YES **[✓ NO]**

If YES, full ethics review required
For example:
Might research objectivity be compromised by sponsorship?
Might any issues of intellectual property or roles in research be raised?

Funding

Is your research funded externally?

YES **[✓ NO]**

If YES, does the funder appear on the 'currently automatically approved' list on the UTREC website?

YES **NO**

If NO, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

Research with animals

Does your research involve the use of living animals?

YES **[✓ NO]**

If YES, your proposal must be referred to the University's Animal Welfare and Ethics Committee (AWEC)

University Teaching and Research Ethics Committee (UTREC) pages
http://www.st-andrews.ac.uk/utrec/