University of Edinburgh
School of Informatics

Random Testing of ML Programs

4th Year Project Report
Software Engineering

Neil Wilkie

March 24, 2005

Abstract: This report describes a project undertaken to devise a testing tool for the Standard ML (SML) language. Reliable bug free software is essential for a number of reasons including public safety, prevention of data loss, economic viability and ease of use. Testing is the most common approach to software quality but it is incredibly labour intensive. Automatic testing solutions are therefore important to maintain high quality within tight time constraints. Several testing tools exist, such as `QuickCheck` for Haskell. A testing tool for SML similar to the `QuickCheck` tool which incorporates automatic generation of test data and systematic checking against specifications would be a useful addition to SML program development. The project was set up to develop such a tool (`QuickECheck`) based on an existing tool (`QCheck`). Development occurred in a number of stages: investigation of program testing techniques and existing tools, investigation of formal testing specifications and first order logic and the creation of an extension to the `QCheck` tool.

Contents

1. Introduction

Software testing is necessary to ensure that programs are bug free before delivery to the user. The testing of purely functional programs is well suited to 'black box' testing on a function by function basis based on the specifications of the program as no program state is carried between functions. Although extremely labour-intensive, testing is the most common approach to software quality [4]. Where time constraints are tight, automatic testing tools are important solutions in maintaining high quality. Automatic testing of functions becomes even more useful when combined with automatic creation of test data; the most simple method of which is random generation. From automatic generation of inputs and automatic batch processing a set of inputs and generated outputs is created which can be checked against test cases generated from the program specification. The interpretation of test results can often be difficult; this is commonly known as the *oracle problem* [6]. The *oracle problem* arises whenever decision procedures for interpreting the test results cannot be easily defined [11]. One way of reducing the problem is by refining specifications towards automated oracles.

At the point in time when the project was first proposed no random testing tool for the Standard ML language existed. The original goal for the project was to create a testing tool similar to the QuickCheck tool for Haskell but for Standard ML. However, between the project selection and the start of the project (2004 summer vacation) the `QCheck/SML` tool was created by Christopher League at Long Island University, Brooklyn, New York. As a tool satisfying the outcomes of the existing project already existed it was deemed unproductive to simply 'reinvent the wheel'. For this reason the aims and goals of the project were adjusted accordingly.

The aims and goals for the revised project were to build upon the work already done on the `QCheck` tool and to improve the way it handles formal specification of tests. The goals for the project were:

- investigate `QCheck`'s inner workings establishing its advantages and limitations
- investigation into the formal specification testing

- create an extension for `QCheck` implementing an increasing number of features:
  - Stage 1 - parse axioms specified in a sub-set of first-order logic, process them and run the test in `QCheck`
  - Stage 2 - extend `QCheck` further to accept axioms at a more complex level and manipulate these axioms to a state where they are able to be processed by `QCheck`
  - Stage 3 - if time allows, extend `QCheck` again to provide a method of quantifying over types for which `QCheck` does not provide a generator.

The main part of the project therefore was the development of the `QCheck` extension, `QuickECheck`. `QuickECheck` development took place in 3 cycles satisfying the first two stages outlined in the project goals. The third stage was not implemented due to time constraints. The development cycles of `QuickECheck` took the form of: the representation of logical axioms within an ML datatype and the parsing of axioms into this datatype, allowing parsed axioms to generate an appropriate matching `QCheck` command to run tests on the specified axiom and finally accepting more complex axioms and manipulating them into a form acceptable in `QCheck` while also providing a basic 'sanity check' for obviously un-useful axioms.

This project report will firstly provide a background to the project by discussing the importance of software verification and formal test specifications. `QCheck` will be explored in depth. Finally the `QuickECheck` extension will be explained in detail along with some examples of its use.

2. Program Testing

2.1 Verification

In any programming development process the issues of validation and verification are important to ensure that a piece of software is correct and useful. Verification is about getting the software right; does the software meet the design? Validation on the other hand is about getting the right software; does this piece of software meet the requirements of the user? Although in a large scale software engineering project both tasks are equally important, for the purposes of this project and the concept of program testing only the problem of verification shall be considered.

Creating correct bug free working software is very important not only for safety critical systems (such as the software controlling an aeroplane or a nuclear power plant) where a bug in a piece of software could lead to injury, loss of life or massive environmental disaster, but also for everyday software where a bug might create a serious problem such as loss of scientific data or prove to be an annoyance for the user (a computer game crashing).In either case unreliable software presents an economic problem for the software developer. For these reasons verification and general testing of software has become an important part of any development methodology.

Verification of a piece of software must begin with a set of requirements stating exactly what the finished product must do. At a general software engineering level these requirements may take the form of loose natural language (eg. 'user stories' within an 'Extreme Programming' [1] development process); graphical representation (eg. 'use case' diagrams from UML [2]); strict formal specification represented in first-order logic (eg. as used in the 'cleanroom software methodology' [3]) or almost any variation in between. For the purposes of this project and of small scale functional programming in general the focus of program specification will be on formal specifications on a function by function basis.

Software testing can be split into several categories of tests which can be com-

[1] "'Extreme Programming' is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation." - xprogramming.com [7]

[2] Unified Modeling Language - An industry standard modelling and specification language created by the Object Management Group [1] used mainly in the design of object orientated software. [12, 3]

[3] The Cleanroom Process Model [10] is based on a formal set of specifications describing the behaviour of the system being developed. [13]

bined to help produce a bug free, correct piece of software. These categories are: unit testing, integration testing, system testing and regression testing. Using functional programming and a cleanroom software methodology, unit testing proves to be the most useful as this directly tests a module or function of the software with respect to the formal specification [6]. Within this scope integration testing is very similar to unit testing as purely functional units carry no state and as such are side effect free. If all units or functions match their formal specifications exactly then they will all integrate perfectly, although this assumes that the formal specifications for each function are correct. System testing concentrates on the system as a whole. Under the limited scope of functional programming and a cleanroom software methodology it is more a test for validating a piece of software. If all units or functions meet their formal specifications then a system test would be testing, not if the software works, but if it meets the user's requirements. Finally regression testing is simply the process of saving all previous tests to run each time a change is made to a program.

2.2 Formal Specification and Testing

Having establishing that requirements for a software system can be written down as a series of first-order logical axioms using the cleanroom methodology, it can be seen that for any more than a simple software solution for the requirements to be written in this way would be a complicated process, especially where in the first instance user requirements are given in another form. However within a functional programming environment where a program exists as a collection of purely functional components specifying requirements in this way can prove to be useful. Specifying functions as mathematical functions based on the input / output behaviour to abstract away from the internal workings of the program can help program correctness throughout development [16].

Using defined formal semantics of the specifications for a program can create the basis for a notion of an exhaustive test set [6]. By creating some hypotheses on the program being tested the success of the test set can be seen as equivalent to the program satisfying the specification. As such an exhaustive test set is generally infinite in order to perform a test a sub domain must be selected through the use of a selection hypothesis. A simple and effective [4] selection hypothesis is the generation of random values. Random testing competes surprisingly well with other systematic methods, especially where the distribution of the data follows that of the test data [4].

[4]This is an especially useful selection hypothesis under the scope of automated testing.

3. QuickCheck and QCheck

3.1 QuickCheck

`QuickCheck` is a lightweight testing tool for the functional programming language Haskell [4]. `QuickCheck` allows the user to provide a specification for their program in a form of properties which their program should satisfy. These properties are described as Haskell functions using functions from the `QuickCheck` library. `QuickCheck` then allows the user to automatically test their program against the specification over a large number of randomly generated values using `QuickCheck`'s built in random data generators. The original paper [4] also provided discussion on a number of associated issues in addition to describing `QuickCheck` and providing some case studies on its use, some issues covered are relevant to this project and others are not.

3.2 QCheck

`QCheck/SML` (`QCheck`) is a lightweight testing tool developed for the SML/NJ (Standard ML of New Jersey) language [19]. It is modelled after the `QuickCheck` tool for Haskell as described in section 2.1 above, with most of the concepts taken directly from the original `QuickCheck` paper [4]. Like `QuickCheck`, `QCheck` allows users to create a specification for a program and set `QCheck` to run a series of random tests automatically.

3.3 QCheck Internals

The source code of `QCheck` can be split easily into 6 distinct categories based on their role within the system. These 6 categories are the main program, the creation of random test data, specifying and evaluating conditional properties, pluggable output styles, user customisable settings and a directory of examples. As each of these categories is important to the functionality, they will be described in detail below.

3.3.1 The Main Program

The main program for QCheck is based around a single main function check' which iterates through a given stream of data, testing the proposition for each item of data in the stream. Additional to this are three functions for each of the ways QCheck can be used, namely check, checkGen and checkOne. Respectively these are the functions which the user calls to use QCheck, check for the case where the user wishes to check the proposition over a user defined set of data, checkGen for when the user wishes to generate test data over which to test and checkOne for the case where the user wishes to check the proposition on a single value.

As an automated testing tool based on QuickCheck the most useful one of these functions is checkGen as we wish to perform an automated test over a random dataset. The checkGen function takes an input of 4 items (or rather 2 tuples), a generator for the dataset, a toString function for the test data, a tag to assign a name to the test and the proposition to test. The generator is used to create the input data as a stream and the check' function is called.

```
val check : ('a, 'b) stream * 'a rep ->
            string * 'a prop -> 'b -> unit
val checkGen : 'a Gen.gen * 'a rep ->
               string * 'a prop -> unit
val checkOne : 'a rep -> string * 'a prop -> 'a -> unit
```

3.3.2 Generating Random Test Data

QCheck provides several generators for creating random data of several different types as well as associated functions for creating random data in the form of various structured types and, at a basic level, functions. The QCheck library provides generators for the types of int, real, Date, text (char and string) and word. Additional to the default generators QCheck also provides a range of special case generators such as Gen.Int.pos which creates only positive integers.

QuickCheck uses Haskell's type classes, allowing arbitrary values for types to be created easily and behind the scenes. As SML does not follow the same type class system, generating random values in SML must be more explicit [3]. Although the supplied generators for basic types are necessary without the combinatory functions, QCheck would not be useful for running any other than the simplest tests. The most useful set of these functions are those for 'zipping' together existing generators to create generators for tuples, the supplied zip functions are:

```
val zip  : ('a gen * 'b gen) -> ('a * 'b) gen
val zip3 : ('a gen * 'b gen * 'c gen) -> ('a * 'b * 'c) gen
val zip4 : ('a gen * 'b gen * 'c gen * 'd gen) ->
           ('a * 'b * 'c * 'd) gen
```

These functions are essential to test a proposition which contains more than 1 free variable. Similar to the 'zip' functions are functions for simulating random generation of functions, functions which 'map' generated input to values to corresponding output values, thus:

```
val map  : ('a -> 'b) -> 'a gen -> 'b gen
val map2 : ('a * 'b -> 'c) -> ('a gen * 'b gen) -> 'c gen
val map3 : ('a * 'b * 'c -> 'd) ->
           ('a gen * 'b gen * 'c gen) -> 'd gen
val map4 : ('a * 'b * 'c * 'd) -> 'e ->
           ('a gen * 'b gen * 'c gen * 'd gen) -> 'e gen
```

Obviously these 4 functions do not cover all possible permutations of inputs and outputs up to the given size. They do not need to, however, as combining the `map` and `zip` functions allows the creation of any 'function generator' with a maximum of 4 inputs to a maximum of 4 outputs. In fact the implementation of `map2`, `map3` and `map4` uses this method; `map` is the only required function, the others are provided purely for ease of use.

3.3.3 Specification and Evaluation of Conditional Properties

`QCheck` acts as a testing tool by allowing users to specify a conditional property using the function (or functions) that they wish to test which should be satisfied for all values within the range they wish to test. The conditional property requires the type `'a prop`, where the polymorphic type `'a` matches that of the input values to be tested, when using a supplied generator this must be of matching type `'a Gen.gen`. The conditional property which the user specifies is of type:

```
type 'a pred = 'a -> bool
```

That is to say that the proposition should be a function testing if the property holds true for an input value of the correct type. To convert this property the required type `'a prop` there is a choice of functions, the two most useful ones being `pred` and `==>`:

```
val pred : 'a pred -> 'a prop
val ==> : 'a pred * 'a pred -> 'a prop
```

The pred function can be used for cases where the conditional property is expressed in a form similar to skolemised predicate calculus. The `==>` function is used in the case of an implication. If the first proposition is satisfied then the second is tested and defines the result of the test. If the first proposition fails then the second proposition will not be tested. In cases where after the maximum number of tests the first proposition has not passed then the outcome of the test is described as dubious, eg.

```
(fn x => false) ==> anotherFunction x
```

3.3.4 Pluggable Output Styles

`QCheck` can provide different ways to show the results from the tests which it runs. There are two provided output styles but it is possible to configure alternate styles by writing functions of the appropriate type. The two styles provided by `QCheck` are the default 'Perl Style' [1] output and an 'CM' style output designed to look like the SML/NJ Compilation Manager output.

The default style provides an easy to read output designed to mimic the Perl unit testing framework. The output provides the name given to the test as an input, the outcome of the test (`OK`, `failed` or `dubious`), the number of passes for the tests and a set number of counter examples. An example of a typical output is:

```
Good Test............OK                                   (100 passed)
```

or

```
BadTest...............Failed                           (80/100 passed)
    counter-examples: 20
                        57
                        45
                        12
                        90
```

3.3.5 User Customisable Settings

To allow a little versatility over the global settings of `QCheck` there are a number of user customisable settings. Most usefully:

[1] Perl - A cross platform programming language used heavily for web programming.

`gen_target` - the number of valid random cases to test (default - 100)
`gen_max` - the maximum number of random cases to consider (default - 400)
`examples` - the number of counter examples to display (default - 5)

3.4 QCheck Examples

3.4.1 Example 1 - Basic Usage (Implication)

This example has been slightly modified from one given in the `QCheck` Paper [8]. Two simple SML functions are to be tested: they check to see if an integer is either even or odd:

```
fun even x = x mod 2 = 0
fun odd x = x mod 2 = 1
```

A mutual test for these two functions is that if an integer x is even, then $x + 1$ will always be odd. Using `QCheck`'s `==>` function the test can be defined as:

```
(fn x => even x) ==> (fn x => odd(x+1))
```

Running the test with an appropriate generator will test each value to see if it is even, if not, discard it or, if it is, test if its successor is odd:

```
checkGen (Gen.Int.int, SOME Int.toString)
         ("even-odd test",
          (fn fn x => even x) ==> (fn x => odd(x+1)))

even-odd test.........ok                              (100 passed)
```

3.4.2 Example 2 - Basic Usage (Predicate)

This example demonstrates the use of running a single skolem style proposal through `QCheck`. It also demonstrates the result when `QCheck` fails a test.

```
checkGen (Gen.Int.int, SOME Int.toString)
             ("even", pred(fn x => even x))
```

This test uses the `even` function defined in Example 1 above to check that all integers are even. This is quite obviously false, and `QCheck` agrees:

```
even..................FAILED                          (48/100 passed)
    counter-examples: ~55
                      ~9869
                      ~51
                      91
                      7433
```

Changing the test slightly however, creates a positive output:

```
checkGen (Gen.Int.int, SOME Int.toString)
         ("even or odd", pred(fn x => even x orelse odd x))

even or odd...........ok                                  (100 passed)
```

3.4.3 Example 3 - Generating Tuples

This test is based upon a test outlined in the `QCheck` paper [8]. This test makes use of the `zip` function outlined above as well as the `Gen.Int.int` generator to generate values of type `(int * int)` for the proposition that the sum of two odd numbers will be even:

```
(fn (x,y) => odd x andalso odd y) ==> (fn (x,y) => even(x+y))
```

As the values for this test are of type `(int * int)` the standard `Int.toString` method is not acceptable, so as a replacement:

```
fn (x,y) => Int.tostring x ^ ", " ^ Int.toString y
```

can be used.

```
checkGen (Gen.zip(Gen.Int.int, Gen.Int.int),
          SOME (fn (x,y) => Int.tostring x ^ ", " ^
                            Int.toString y))
         ("sum of odds is even",
          ((fn (x,y) => odd x andalso odd y) ==>
           (fn (x,y) => even(x+y)))

sum of odds is even...FAILED                          (97/100 passed)
    counter-examples: 897083003, 850430501
                      1073741823, 298807
                      ~845110003, ~715751599
```

Although `QCheck` picks up some results from this test which do not satisfy the proposition, this occurs because the proposition does not account for values which when added create a result above the maximum integer size, or below the minimum. An improvement on the proposition would be to take this into account, as shown below:

```
fn (x,y) => odd x andalso
            odd y andalso
            (x+y) > Int.minInt andalso
            (x+y) < Int.maxInt) ==> (fn (x,y) => even(x+y)
```

3.4.4 Example 4 - Dubious Tests

This is just a small example test to show how `QCheck` responds to dubious tests (Section 3.3.3) where the first proposition in an implication is never satisfied. A very simple, yet contrived example is:

```
(fn x => even x andalso odd x) ==> (fn x => even x)
```

Obviously no value can be both even and odd by definition.

```
checkGen (Gen.Int.int, SOME Int.toString)
         ("even and odd", (fn x => even x andalso odd x) ==>
                          (fn x => even x))

even and odd..........dubious                               (0/0 passed)
```

3.5 QCheck's Limitations

While `QCheck` is a well designed and useful testing tool with a large number of features within the scope of quickly and easily running tests based upon formal specifications it would be useful to be able to input specifications in first-order logic. Additionally the user 'interface' and the program's ease of use could be improved upon, as well as the scope of propositions which can be accepted.

The requirement in `QCheck` to specify propositions to test as functions in SML is counter productive for formal specifications written in first order logic. Being able to enter a formal specification on which to perform a test in first order logic would speed up the process for entering and running tests by cutting out the need to re-specify the propositions into SML functions. For the purpose of running tests quickly and easily on formal specifications the need to specify a generator for the tests is not required as the type information can be inferred from the formal specification. This would also be a benefit for the creation of tuples, as inputs as it can all be 'hidden' from the user; again this information is already conveyed in the formal specification.

In addition to the information required for specifying an appropriate generator `QCheck`'s `checkGen` function also requires information for the 'typesetting' of input values for cases where there are counter values to be displayed. If the generator has been chosen either though the existing manual method or having been inferred from the formal specification of the test, this typesetting information can be easily generated using appropriate `toString` methods for various types.

The final piece of inputted data for `checkGen` is a tag by which to identify a specific test when running large batches. It would be sufficient for this tag to be replaced with formal specification being tested.

4. QuickECheck

After investigating why software testing in general, and formal and random testing for functional programming in particular, is a useful and important task in software development it was necessary to establish how a testing tool for the SML language could be created. As explained in the introduction the original proposal for the project was to create a tool based on the `QuickCheck` tool for Haskell. However, between the project proposal and the start of the project, this task was completed elsewhere in the form of the `QCheck` tool for SML/NJ. This chapter details the design and implementation of an extension for `QCheck` - `QuickECheck`.

After considering the limitations of `QCheck` within the scope of formal testing, it was considered it would be useful if `QCheck` could be adapted to take input in the form of a single logical axiom to test, where this axiom also infers the type information required to produce an acceptable random data generator. `QCheck` can only handle tests which correspond to axioms of the form $\forall y_1, \forall y_2, \ldots, \forall y_3 P$, where P contains no quantifiers; an improvement upon `QCheck` would be to allow the input of complicated axioms containing nested quantifiers and, where appropriate, existential quantifiers. These axioms would be transformed into an acceptable format to test. This design would also implement a 'sanity checker' as not all axioms specify useful tests.

As `QCheck` provides a polymorphic checker and several libraries for generating random data of varying types, it would be useful if `QuickECheck` took an input in the form of an axiom, removed the implied type information to create a matching generator and finally translated the axiom into a form which `QCheck` is able to test, possibly requiring some manipulation of the axiom into a form which `QCheck` can test. `QuickECheck` would essentially be a 'front end' package for `QCheck` simplifying the user's interaction with the program as well as extending the range of tests which can be run.

`QCheck` was written for the SML/NJ dialect of SML, however the implementation of the `QuickECheck` extension is written in the Moscow ML [14] dialect of SML. Moscow ML is a lightweight implementation of SML which implements the entire SML specification as well as most of the SML basis library, although compared with SML/NJ it does not provide as many features additional to the SML specification. It also has the very obvious advantage of being the dialect of SML with which I am most familiar as it is the dialect taught and used at the University of Edinburgh. The difference in dialect between SML/NJ and Moscow ML does not present a problem as `QCheck` provides a make file for the Moscow ML dialect and does not use any special extensions provided by SML/NJ.

On a basic level `QuickECheck` will take an axiom from the user and create an appropriate `QCheck` command as a `string`. This command can then be run within a `mosml` [14] interactive session to run the test This stage should be done completely for the user.

4.1 A Grammar for Logical Axioms, Lexical Analysing and Parsing

In order for `QuickECheck` to take an input in the form of a first-order logical axiom a grammar for this logic must be specified as well as an ML type declaration to represent the axiom. Furthermore a corresponding parser and lexical analyser must be created to make sense of the user's input.

A grammar for specifying a logical axiom should represent all well formed first-order logic expressions. It was investigated during the project if the grammar should contain a basic sub set of ML for specifying basic functions within axioms. It was subsequently decided that this was an unnecessary complication to the idea of simplifying `QCheck`'s user interaction. In addition to the standard first-order form it was necessary to add the type information into the axiom as the information is intended to be removed in order to construct an appropriate data generator. As the input variables for the test are specified by a quantifier along with the variable name, the type of the variable is also specified. The grammar for representing axioms is constructed as follows:

Formula ::= Formula `or` Formula
  | Formula `and` Formula
  | Formula `==>` Formula
  | `~` Formula
  | `forall` s`:`tipe`.`Formula
  | `exists` s`:`tipe`.`Formula
  | Atomic_Formula
  | `(`Formula`)`

Atomic_Formula ::= Expr `=` Expr
  | Expr `>` Expr
  | Expr `<` Expr
  | Expr

```
Tipe ::= int
       | real
       | word
       | char
       | string

Expr ::= AppExpr
       | Expr / Expr
       | Expr * Expr
       | Expr + Expr
       | Expr - Expr
       | Expr mod Expr
       | Expr andalso Expr
       | Expr orelse Expr

AppExpr ::= AExpr
          | AppExpr AExp

AExpr ::= s
        | i
        | #"c"
        | "s"
        | true
        | false
        | (Expr)
        | (VarList)

VarList ::= AppExpr
          | AppExpr, VarList
```

Defining a grammar for the input as logical axioms is important as without a large amount of natural language processing the user is required to enter a recognisable input, but in order to do anything useful to the entered axiom a SML datatype is also required to represent the axiom internally.

```
datatype expr =
    Var of string
  | Vars of expr list
  | Bool of bool
  | Int of int
  | Stringg of string
  | Charr of string
  | Op of string * expr * expr
  | App of expr * expr
  | Andalso of expr * expr
  | Orelse of expr * expr

datatype atomic_formula =
    Equals of expr * expr
  | GreaterThan of expr * expr
  | LessThan of expr * expr
  | Expr of expr

datatype tipe =
    Intt
  | Real
  | Char
  | String
  | Word

datatype formula =
    Disjunction of formula * formula
  | Conjunction of formula * formula
  | Implies of formula * formula
  | Negation of formula
  | Forall of (string * tipe) list * formula
  | Exists of (string * tipe) list * formula
  | Atomic of atomic_formula
```

It is too these datatypes that the parser and lexical analyser will convert the inputted axiom creating an internal representation of type formula. As an example the simple axiom:

```
forall x:int.even x ==> odd(x+1)
```

is parsed as:

```
Forall("x", Intt,
       Implies(Atomic(Expr(App(Var "even", Var "x"))),
               Atomic(Expr(App(Var "odd", Op("+", Var "x",
                                             Int 1))))))
```

Moscow ML provides a lexical generator called `mosmllex` which is closely based on the camllex lexical generator from Caml Light implementation [9]. A lexical generator is a function which creates a lexical analyser. The lexical generator `mosmllex` requires a set of regular expressions and semantic actions to produce a ML code for a lexical analyser. The code used to create the lexical analyser is based on code for a 'Micro-ML' lexical analyser by Don Sannella created for an undergraduate course in Functional Programming and Specification [15], although the tokens and keywords required have been changed to create a lexical analyser for the `QuickECheck` grammar above. The source for creating the lexical analyser as an input to `mosmllex` can be found in the file `Lexer.lex` in Appendix A.

In addition to Moscow ML's `mosmllex` Moscow ML also provides `mosmlyac` which is a corresponding parser generator based upon `camlyacc` from the Caml Light implementation [9] which is based upon the public domain Berkeley `yacc` [5]. When supplied with a context-free grammar and associated semantic actions `mosmlyac` generates a parser for the supplied grammar. This generated parser will define a parsing function for each start symbol in the supplied grammar, taking as arguments a corresponding lexical analyser. In this case the lexical analyser is generated by `mosmllex`. Correspondingly the lexical generator the `QuickECheck` implantation for `mosmlyac` is based upon code for a 'Micro-ML' parser, also from [15], although the context-free grammar is replaced with the grammar defined for acceptable axioms. The source for creating the parser as an input to `mosmlyac` can be found in `Parser.grm` in Appendix A. Additionally the code for creating the lexical analyser and parser in Appendix A contains a commented source defining the internal datatype and additional code required to use the outputted source from `mosmllex` and `mosmlyac`.

4.2 Generating Generators

As outlined in the justification for creating an extension for `QCheck` the required type information for creating a basic set of random generators (that is to allow creation of random generators over the entire range of values, and not using `QCheck`'s special generators such as `Gen.Int.pos`, for creating only positive integers) can be conveyed within the axiom that is being tested. Once the inputted axiom has been parsed into `QuickECheck`'s internal data structure all that is required to do is search the data tree for quantifiers and strip out the type information to create a list of variables and their associated types. This task is completed by the recursive function getVars:

```
val getVars : formula * (string * tipe) list ->
              (string * tipe) list

fun getVars(Disjunction(f0, f1), vars) =
      getVars(f1, getVars(f0, vars))
  | getVars(Conjunction(f0, f1), vars) =
      getVars(f1, getVars(f0, vars))
  | getVars(Implies(f0, f1), vars) =
      getVars(f1, getVars(f0, vars))
  | getVars(Negation(f), vars) =
      getVars(f, vars)
  | getVars(Forall(v, t, f), vars) =
      getVars(f, vars@[(v, t)])
  | getVars(Exists(v, t, f), vars) =
      getVars(f, vars@[(v, t)])
  | getVars(Atomic(a), vars) =
      vars
```

Once a list has been generated containing the variables and associated type information the first part of the QCheck function specifying the generator and toString method for displaying counter examples can be generated:

```
val makeGen : (string * tipe) list -> string
val getGen : tipe -> string
val getToString : tipe -> string

fun makeGen([(v, t)]) =
      getGen(t) ^ ", SOME " ^ getToString(t)
  | makeGen([(v0, t0), (v1, t1)]) =
      "Gen.zip(" ^ getGen(t0) ^ ", " ^ getGen(t1) ^
      "), SOME (fn (t0,t1) => " ^ getToString(t0) ^
      " t0 ^ \", \" ^ " ^ getToString(t1) ^ " t1)"
  | makeGen([(v0, t0), (v1, t1), (v2, t2)]) =
      "Gen.zip3(" ^ getGen(t0) ^ ", " ^ getGen(t1) ^ ", " ^
      getGen(t2) ^ "), SOME (fn (t0,t1,t2) => " ^
      getToString(t0) ^ " t0 ^ \", \" ^ " ^ getToString(t1) ^
      " t1 ^ \", \" ^ " ^ getToString(t2) ^ " t2)"
```

```
  | makeGen([(v0, t0), (v1, t1), (v2, t2), (v3, t3)]) =
      "Gen.zip4(" ^ getGen(t0) ^ ", " ^ getGen(t1) ^ ", " ^
      getGen(t2) ^ ", " ^ getGen(t3) ^
      "), SOME (fn (t0,t1,t2) => " ^ getToString(t0) ^
      " t0 ^ \", \" ^ " ^ getToString(t1) ^ " t1 ^ \", \" ^ " ^
      getToString(t2) ^ " t2 ^ \", \" ^ " ^ getToString(t3) ^
      " t3)"
  | makeGen(vars) = raise Too_Many_Vars

and getGen(Intt) = "Gen.Int.int"
  | getGen(Real) = "Gen.Real.real"
  | getGen(Word) = "Gen.Word.word"
  | getGen(Char) = "Gen.char"
  | getGen(String) =
      "Gen.string (Gen.range(5,20),Gen.charFrom " ^
      "\"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz\")"

and getToString(Intt) = "Int.toString"
  | getToString(Real) = "Real.toString"
  | getToString(Word) = "Word.toString"
  | getToString(Char) = "Char.toString"
  | getToString(String) = "String.toString"
```

The function `makeGen` will use the length of the list of variables and types to choose an appropriate zip function; `zip`, `zip3`, `zip4` or in the case of a single variable no required function. `QCheck` only provides zip functions up to supporting 4 variables. The creation of additional zip functions is an easy although tedious process so a possible improvement upon this design would be to modify the zip function to take a list of generators and create an appropriately sized generator, however in this instance if more than 4 variables are specified then the exception `Too_Many_Vars` is raised. Secondary to creating the appropriate generator the `makeGen` function also creates an appropriate concatenation of `toString` methods separating each variable with a comma. The additional functions `getGen` and `getToString` provide the names of required generators or `toString` methods for the supported types. For the types of `int`, `real`, `word` and `char` the specification of a generator is straightforward; to generate strings requires a little more effort. As a `string` is essentially an array or list of `chars` and can vary in length two generators are required to work to together in order to create the string generator: a generator to specify the length of the string, and another to randomly pick characters to form the string. As a default setting these generators have been set to create a string of length between 5 and 20 characters, and the choice of `chars` has been limited to the upper and lower case Latin alphabet.

As an example of this creation of the code for specifying a generator and `toString` function lets imagine a case where we require a generator for a `real`, a `char` and an `int`. After the `getVars` method has worked its way through the axiom and stripped out the variable names and their types we get the list `[(r, Real), (c, Char), (I, Intt)]` to use as an input to the `makeGen` function. This list will satisfy the 3rd case of the function:

```
"Gen.zip3(" ^ getGen(t0) ^ ", " ^ getGen(t1) ^ ", " ^ getGen(t2) ^
  "), SOME (fn (t0,t1,t2) => " ^ getToString(t0) ^
  " t0 ^ \", \" ^ " ^ getToString(t1) ^ " t1 ^ \", \" ^ " ^
  getToString(t2) ^ " t2)"
```

Which in turn requires the outputs for `getGen` in each of the three cases ("`Gen.Real.real`", "`Gen.char`" and "`Gen.Int.int`") and the outputs for `getToString` in each of the three cases ("`Real.toString`", "Char.toString and "`Int.toString`"). Together this gives the first part of the arguments required to run `QCheck`'s `checkGen` function:

```
Gen.zip3(Gen.string (Gen.Real.real, Gen.char, Gen.Int.int),
SOME (fn (t0,t1,t2) => Real.toString t0 ^ \", \" ^ char.toString t1 ^
                       \", \" ^ Int.toString t2)
```

4.3 Transformation of Axioms

The largest and most complicated element of `QuickECheck` is to transform a possibly complex parsed axiom into a simplified form that `QCheck` can use. As described at the beginning of this chapter `QCheck` accepts functions as inputs relating to the specification that is to be tested. These functions are essentially in Skolem Normal Form (SNF). Any first-order logical formula can be skolemised to a function in SNF equisatisfiable with the original. A simple algorithm for skolemisation can be expressed as [18]:

while F contains an existential quantifier
  begin
    Let F have the form $F = \forall y_1 \forall y_2 ... \forall y_n \exists z G$ for some formula G
    which is rectified and in prenex form and $n \geq 0$;
    Let f be a new function symbol of arity n that does not yet occur in F;
    $F := \forall y_1 \forall y_2 ... \forall y_n G[z/f(y_1, y_2, ..., y_n)]$;
  end;

This algorithm can be implemented in SML as a recursive function which searches through a given formula replacing any existential quantifiers with skolem functions. Source code for such a function can be found in Appendix C. As an example, skolemising the formula

```
forall x:int.exists y:int.y=x+1
```

would create the corresponding non-equivalent but equisatisfiable formula

```
forall x:int.sk0(x)
```

where is `sk0(x)` is the skolem function used to replace `y`.

Although it is possible to implement this algorithm in SML and convert formulae to SNF it is not helpful under the restrictions of creating an automated tool to convert axioms. Although it is possible to insert these functions within the SML program they have no definition, therefore attempting to use them within a test would create errors when executed. Normally skolemisation of formulae requires a calculation of the definition of each skolem function.

Therefore it is not possible to convert formulae fully into Skolem Normal Form completely automatically. However, it is not completely necessary to skolemise the formulae fully. In order for an existential quantifier to prove useful in a specification for random testing it must appear in a negative position within the axiom as it can be substituted for a universal quantifier using the following substitution:

$\exists x P \rightarrow \neg \forall x \neg P$

The definition of positive and negative positions taken as specified in the definition of equality occurrences in [11] is:

$$\begin{array}{rcl} f & := & positive \\ \neg f & := & negative \\ f \wedge g & := & f - positive, g - positive \\ f \vee g & := & f - positive, g - positive \\ f \Rightarrow g & := & f - negative, g - positive \\ \forall x.f & := & f - positive \\ \exists x.f & := & f - positive \end{array}$$

Existential quantifiers in positive positions and likewise universal quantifiers in negative positions do not provide useful tests. As random testing only checks a subset of the input domain, tests which take the form of testing that there exists a value to satisfy a certain case are not useful for random testing. Because the subset of randomly generated values is relatively small compared to the whole input domain, it is extremely unlikely that a value to satisfy the test is generated. An example of a test specification of this sort is:

```
exists x:int.x = 347
```

Even though this test is obviously successful to a human observer when testing over 100 or even 1000 values it is unlikely that this test will succeed. Assuming a truly random generation of 100 values over the domain of all integers specifiable within ML, the chances of generating a single correct value is 1/215474836. It is worth noting however, that under a suitably small domain, tests of this type can prove to be useful; an extreme example would be to test over the domain of boolean values.

Although skolemisation does not provide a solution for converting axioms it does provide a useful indication that a test will not prove to be useful. Where there is a need to skolemise a formula the test is not useful and the user should be alerted. Where there is no need to skolemise the formula, the skolem normal form and prenex normal form are equal. Therefore a suitable solution is to convert formulae to prenex normal form and provide a check that there is no need for skolemisation - that there are no remaining existential quantifiers.

The algorithm for transforming formulae to prenex normal form is straight forward in defining which substitutions can be made and in what order. The problem lies in checking that the manipulated formula cannot undergo any more substitutions at each stage. The algorithm used in QuickECheck is based on that for skolem normal form up until the point of skolemisation from [2] and other sources:

1. 1. Simplify the formula to contain only $\wedge$, $\vee$, $\neg$, $\forall$ and $\exists$ using the substitutions:
$$\begin{array}{rcl} P \Rightarrow Q & \rightarrow & \neg P \vee Q \\ \neg(P \Rightarrow Q) & \rightarrow & (P \wedge \neg Q) \end{array}$$
2. 2. Move negations inwards using the substitutions:
$$\begin{array}{rcl} \neg(\neg P) & \rightarrow & P \\ \neg(P \vee Q) & \rightarrow & P \wedge Q \\ \neg(P \wedge Q) & \rightarrow & P \vee Q \\ \neg\exists x P & \rightarrow & \forall x \neg P \end{array}$$
3. 3. Move quantifiers out:
$$\begin{array}{rcl} (\forall x P \vee Q) & \rightarrow & \forall x(P \vee Q) \\ (\exists x P \vee Q) & \rightarrow & \exists x(P \vee Q) \\ (P \vee \forall x Q) & \rightarrow & \forall x(P \vee Q) \\ (P \vee \exists x Q) & \rightarrow & \exists x(P \vee Q) \\ (\forall x P \wedge Q) & \rightarrow & \forall x(P \wedge Q) \\ (\exists x P \wedge Q) & \rightarrow & \exists x(P \wedge Q) \\ (P \wedge \forall x Q) & \rightarrow & \forall x(P \wedge Q) \\ (P \wedge \exists x Q) & \rightarrow & \exists x(P \wedge Q) \end{array}$$

This algorithm can be implemented by providing for each stage a function to perform the substitution, a function to test if any possible substitutions remain and a recursive function acting like a while loop controlling the number of iterations for which substitutions are executed.

```
val simplify : formula -> formula
val simplified : formula -> formula
val keepSimplifying : formula * bool -> formula

fun simplify(Disjunction(f0, f1)) =
      Disjunction(simplify(f0), simplify(f1))
  | simplify(Conjunction(f0, f1)) =
      Conjunction(simplify(f0), simplify(f1))
  | simplify(Implies(f0, f1)) =
      Disjunction(Negation(simplify(f0)), simplify(f1))
  | simplify(Negation(Implies(f0, f1))) =
      Conjunction(simplify(f0), Negation(simplify(f1)))
  | simplify(Negation(f)) =
      Negation(simplify(f))
  | simplify(Forall(v, t, f)) =
      Forall(v, t, simplify(f))
  | simplify(Exists(v, t, f)) =
      Exists(v, t, simplify(f))
  | simplify(Atomic(a)) =
      Atomic(a)

and isSimplified(Disjunction(f0, f1)) =
      isSimplified(f0) andalso isSimplified(f1)
  | isSimplified(Conjunction(f0, f1)) =
      isSimplified(f0) andalso isSimplified(f1)
  | isSimplified(Implies(f0, f1)) =
      false
  | isSimplified(Negation(f)) =
      isSimplified(f)
  | isSimplified(Forall(v, t, f)) =
      isSimplified(f)
  | isSimplified(Exists(v, t, f)) =
      isSimplified(f)
  | isSimplified(Atomic(a)) =
      true

and keepSimplifying(f, finished) =
      if finished
      then f
      else keepSimplifying(simplify(f), isSimplified(simplify(f)))
```

and similarly for the other stages:

```
val moveNegations : formula -> formula
val negationsMoved : formula -> formula
val keepMovingNegations : formula * bool -> formula

val moveQuantifiers : formula -> formula
val quantifiersMoved : formula -> formula
val keepMovingQuantifiers : formula * bool -> formula
```

Although in terms of strict first-order logic the algorithm and substitutions outlined above are correct, `QCheck` provides special treatment for implications. As detailed in Section 3.3.3 and highlighted in `QCheck` Example 4 (section 3.4.3.1) QCheck provides the `==>` function. Additional to this function `QCheck` also provides the function:

```
val implies : 'a pred * 'a pred -> 'a prop
```

`==>` is really a special case of `implies` with is used in its implementation. The `QCheck implies` (and `==>`) function provides extra functionality by first testing the left hand side, and if successful continuing with the right, if the left had side fails then the test case is discarded and another generated such that the discarded test case does not count towards the total of attempted tests. The difference between this implantation and the formal logical implication is such:

Formal Logic

A	B	Z
0	0	1
0	1	1
1	0	0
1	1	1

`QCheck implies`

A	B	Z
0	0	X
0	1	X
1	0	0
1	1	1

Where A is the left hand side of the implication, B the right hand side and Z the result. X represents the case where `QCheck` would discard a test value.

Although generally this special treatment for implications is useful as it can provide much clearer results than the formal definition does. However, in the case where an existential quantifier (in a negative position as detailed above) has been transformed there is a problem with `QCheck`'s special treatment. If the left hand side of the implication contains a existential quantifier (even in a negative position) then the problems associated with finding a single value within a subset of a large domain return. Another problem with `QCheck`'s `implies` function and its type is that after one implication has been used, there can be no further implications on the left hand side of the implication, further implications are acceptable on the right hand side. Additionally implications must take place at the root of the axiom (ignoring quantifiers); this is also true for nested implications, they must be directly below (and on the right hand side of) another implication.

To make use of `QCheck`'s special treatment of implications it is necessary to manipulate the case where it should be used form when it shouldn't. For the two cases two functions are provided, `simpleTransform` which does not remove implications, and `fullTransform` which does:

```
fun fullTransform(f) =
      keepMovingQuantifiers(keepMovingNegations(
      keepSimplifying(f, false),
      false), false)

and simpleTransform(f) =
      keepMovingQuantifiers(keepMovingNegations(f, false), false);
```

4.4 Giving QCheck the Axiom

At this point a decision needs to be made concerning whether or not the axiom should be transformed fully to prenex normal form, or if the implications should be left in place to be treated specially. As outlined in Section 4.3 above the largest problem when using `QCheck`'s `implies` function is when there exists an existential quantifier within the axiom. After checking for the existence of an existential quantifier the make function decides the appropriate case.

```
fun make(f) = if checkQuantifiers(f)
              then makeImplies(simpleTransform(f), getVars(f, []))
              else makeNoImplies(fullTransform(f), getVars(f, []))

and makeImplies(Forall(v, t, f), vars) = makeImplies(f, vars)
  | makeImplies(Implies(f0, f1), vars) =
      makeImplies2(f0, vars) ^ " implies " ^ makeImplies(f1, vars)
  | makeImplies(f, vars) =
      "pred(fn (" ^ stripVars(vars) ^ ") => " ^ create(f) ^ ")"

and makeImplies2(f, vars) =
      "(fn (" ^ stripVars(vars) ^ ") => " ^ create(f) ^ ")"
```

```
and makeNoImplies(f, vars) =
      "pred(fn (" ^ stripVars(vars) ^ ") => " ^ create(f) ^ ")";
```

As indicated in the previous discussion on the transformation of axioms if an existential quantifier can be substituted appropriately for a universal quantifier then it is possible to create axioms containing no existential quantifiers which would cause problems with `QCheck`'s `implies` function. More discussion on this problem can be found in the Conclusion.

After the axiom has been manipulated into a form which `QCheck` can accept, the corresponding `QCheck` command can be created in a style similar to an un-parser. Up until this point there is still the possibility that the axiom does not define a useful test. An indication of this is that if the axiom contains an existential quantifier then it is located in a positive position, and such the test is not useful. The data tree is searched from the root creating an appropriate string for each node or leaf on the tree.

```
fun create(Disjunction(f0, f1)) =
      create(f0) ^ " orelse " ^ create(f1)
  | create(Conjunction(f0, f1)) =
      create(f0) ^ " andalso " ^ create(f1)
  | create(Implies(f0, f1)) =
      "not (" ^ create(f0) ^ ") orelse " ^ create(f1)
  | create(Negation(f)) =
      "not (" ^ create(f) ^ ")"
  | create(Forall(v, t, f)) =
      "(" ^ create(f) ^ ")"
  | create(Exists(v, t, f)) =
      raise Not_Useful_Test
  | create(Atomic(a)) =
      createA(a)

and createA(Equals(e0, e1)) =
      createE(e0) ^ " = " ^ createE(e1)
  | createA(GreaterThan(e0, e1)) =
      createE(e0) ^ " > " ^ createE(e1)
  | createA(LessThan(e0, e1)) =
      createE(e0) ^ " < " ^ createE(e1)
  | createA(Expr(e)) =
      createE(e)
```

```
and createE(Var(s)) = s
  | createE(Vars(list)) = displayList(list)
  | createE(Bool(b))    = Bool.toString(b)
  | createE(Int(i))     = Int.toString(i)
  | createE(Stringg(s)) = "\"" ^ s ^ "\""
  | createE(Charr(c))   = "#\"" ^ c ^ "\""
  | createE(Op(s, e0, e1)) =
      "(" ^ createE(e0) ^ " " ^ s ^ " " ^ createE(e1) ^ ")"
  | createE(App(e0, e1)) =
      createE(e0) ^ "(" ^ createE(e1) ^ ")"
  | createE(Andalso(e0, e1)) =
      createE(e0) ^ " andalso " ^ createE(e1)
  | createE(Orelse(e0, e1)) =
      createE(e0) ^ " orelse " ^ createE(e1)

and displayList([h])  = createE(h)
  | displayList(h::t) = createE(h) ^ ", " ^ displayList(t)
  | displayList([])   = ""
```

4.5 Providing a Tag Line

As mentioned in section 3.3.1 above the final piece of information `QCheck` requires is a tag by which to identify the test to the user. This needs to be a meaningful unique name which the user can recognise easily when running a large battery of tests. It is therefore sufficient that this tag is simply the axiom in the non-parsed form which it was entered by the user. However there are a couple of problems which need to be monitored. Firstly `QCheck` has the peculiarity that if the tag is longer that 51 characters then the test output appears once for each random value tested, a second problem is that although it is necessary for some of the SML escape sequences to be removed when using the generated `checkGen` command (see section below), escape sequences inserted within the tag must be kept otherwise the Moscow ML interpreter is confused by what is a string and what isn't. These two problems can easily be cleared by creating a pair of functions to search the tag string for possible problems and insert escape sequences where necessary and also truncate the tag to a small enough size.

```
fun trim(s) = implode(crop(explode(s), 0))

and crop(h::t, count) =
      if count < 46
      then if h = #"\""
           then [#"\\", #"\""]@crop(t, count+2)
           else [h]@crop(t, count+1)
      else [#".", #".", #"."]
  | crop([], count) = [];
```

4.6 Running Tests in QCheck

Once the axiom has been parsed, stripped of its type information and the axiom has been manipulated and converted into a form which `QCheck` can use, all that is required is to assemble the `QCheck checkGen` command and run the test. Using the functions for each part of the assembly it is possible to create a string which contains the entire `QCheck` command.

```
fun makeCommand(input) =
      let
        val parsed = parse(input)
  in
        "checkGen (" ^ makeGen(getVars(parsed, [])) ^
        ") (\"" ^ trim(input) ^ "\", " ^ make(parsed) ^ ")"
      end;
```

Once the `QCheck` command has been assembled as a string it can be entered into a Moscow ML Session to be run. This stage could be done by the user; however it would be annoying and defeat the point of an automated testing system if the user was required to sit and copy / paste the command for each test. Another problem with using copy / paste is that as the checkGen command contains quotation marks (") Moscow ML must insert appropriate escape sequences into the string (\"). A simple step around both of these problems is for `QuickECheck` to run the `checkGen` command automatically by first writing the command to a temporary file, then loading the file back into Moscow ML as a program.

```
fun qecheck(input) =
      let
        val outfile = openOut("temp.qec")
      in
        output(outfile, makeCommand(input));
        flushOut(outfile);
        use "temp.qec"
      end;
```

Although this may seem clumsy it does provide the added advantage of fully automated testing, and that the generated `checkGen` command is stored in a temporary file allowing the user to view exactly what is being run in `QCheck` if they so wish. It is unfortunate that SML does not allow values to be read in as sub-routines as found in other languages such as Perl; this is a basic limitation of the SML type system.

4.7 Running QuickECheck

Although `QuickECheck` automatically loads all required libraries when loaded into the Moscow ML interactive session, most importantly `QCheck`, it is required that the Moscow ML interactive session is started using the command line option '`-I`'. This option allows the user to specify additional directories for Moscow ML to search for interface files, bytecode files and source files. This means that any calls to `use`, `load` or `loadOne` will first search the current directory then all directories specified in order of appearance [14]. When running `QuickECheck` the directories for the parser and for `QCheck` must be specified in this way, an example command to run a Moscow ML interactive session using `QuickECheck` would be:

```
mosml -I parser -I QCheck qecheck.sml
```

5. QuickECheck Examples

5.1 Example 1

This first example is a re-run of the simple example taken from the `QCheck` paper as shown in `QCheck` Example 1 above. To recap, the test concerns two simple functions for testing if an integer is even or odd. It is now possible to express the test formally as an axiom:

```
forall x:int.even x ==> odd(x+1)
```

As this axiom contains the type information it is no longer necessary to specify a generator:

```
qecheck "forall x:int.even x ==> odd(x+1)"

[opening file "temp.qec"]
forall x:int.even x ==> odd(x+1).ok                    (100 passed)
> val it = () : unit
[closing file "temp.qec"]
> val it = () : unit
```

As the terminal window output shows the test was successful. Examining the file `temp.qec` shows the `QCheck` function generated by `QuickECheck`:

```
checkGen (Gen.Int.int, SOME Int.toString)
         ("forall x:int.even x ==> odd(x+1)",
          (fn (x) => even(x)) implies pred(fn (x) => odd((x + 1))))
```

Noting that the generator and `toString` function generated are identical to the ones used in `QCheck` Example 1. The function however is slightly different, instead of the `==>` function the more explicit `implies` function is used in conjunction with the `pred` function on its left hand side.

5.2 Example 2

This example shows how `QuickECheck` deals with types other than `ints`, specifically `strings` and `chars`. For this example there are two sets of functions to split strings according to the first instance of a `char`. The first pair of functions returns the substring before the first instance of the `char`, the second pair returns the substring located after the first instance of the `char`.

```
fun getBefore(string, token) =
      implode(splitBefore(explode(string), token, []))

and splitBefore(h::t, token, header) =
      if h = token
      then header
      else splitBefore(t, token, header@[h])
  | splitBefore([], token, header) = header

fun getAfter(string, token) =
  implode(splitAfter(explode(string), token))

and splitAfter(h::t, token) =
      if h = token
      then t
      else splitAfter(t, token)
| splitAfter([], token) = [];
```

A third pair of functions provide a test for whether a `string` contains a given `char`:

```
fun containedIn(string, token) =
      contains(explode(string), token)

and contains(h::t, token) =
      if h = token
      then true
      else contains(t, token)
  | contains([], token) = false;
```

Firstly define a test generating only the `string` portion of the input domain to test that the substring before a given `char` should not contain any instance of that `char`.

```
> qecheck "forall x:string.
             not(containedIn(getBefore(x,#\"a\"), #\"a\"))";

forall x:string.not(containedIn(getBefore(x,#"....ok   (100 passed)
```

Now lets improve upon this test by also randomly generating the char:

```
> qecheck "forall x:string.forall y:char.
             not(containedIn(getBefore(x,y),y))";

forall x:string.forall y:char.not(containedIn(....ok   (100 passed)
```

Inspecting the `temp.qec` file shows the 'zipped' generators as well as the function generated to test the axiom:

```
checkGen (Gen.zip(Gen.string(Gen.range(5,20), Gen.charFrom
          "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"),
          Gen.char),
          SOME (fn (t0,t1) => String.toString t0 ^ ", " ^
                              Char.toString t1))
         ("forall x:string.forall y:char.not(containedIn(...",
          pred(fn (x, y) => ((not(containedIn(getBefore(x,
                                                        y), y))))))
```

Here it can be seen that once the quantifiers were stripped off the axiom required no other transformation. Now that it is established that the function is pulling off the first instance of the character it is worth testing that the length of the two substrings generated is equal to the length of the original string, or one less in the case where a character has been removed:

```
> qecheck "forall x:string.
              forall y:char.(size(getBefore(x,y))+size(
                  getAfter(x,y)))=size(x) or (size(getBefore(x,y))+
                      size(getAfter(x,y))+1)=size(x)";

forall x:string.forall y:char.(size(getBefore....ok  (100 passed)
```

Here the function lives up to expectation and the test is passed successfully.

5.3 Example 3

In the previous two examples none of the tests have specified something which could not be run in `QCheck` in terms of the complexity of the axioms. So far none of the tests have contained existential quantifiers. To demonstrate how `QuickECheck` handles existential quantifiers let's use a simple yet contrived example:

```
> qecheck "forall y:int.(exists x:int.x+1 = y and x>0) ==> y>2";

forall y:int.(exists x:int.x+1 = y and x>0) ==....ok   (100 passed)
```

The test passed, although to understand exactly what was going on let's look at the `temp.qec` file to see the generated `QCheck` command:

```
checkGen (Gen.zip(Gen.Int.int, Gen.Int.int),
          SOME (fn (t0,t1) => Int.toString t0 ^ ", " ^
                              Int.toString t1))
         ("forall y:int.(exists x:int.x+1=y and x>0) ==> ...",
          pred(fn (y, x) => ((not ((x + 1) = y) orelse
                              not (x > 0) orelse y > 2))))
```

Picking apart the command to see the new generated test specification gives:

```
forall y:int.forall x:int.(not ((x + 1) = y) orelse not (x > 0)
                                              orelse y > 2)
```

which is logically equivalent to the initial axiom. Notice that this case has not used `QCheck`'s special treatment of implications as the axiom contains an existential quantifier.

5.4 Example 4

As a fourth and final example to show what `QuickECheck` can do a more complex test involving axioms can be defined as:

```
forall s:string.(exists t:char.appearsOnce(s, t) and even(size(s)))
    ==> not(rev(s) = s)
```

This example makes use of the functions specified in Example 2 as well as two new functions `appearsOnce` and `rev`:

```
fun appearsOnce(string, token) =
      (containedIn(string, token)) andalso
      not (containedIn(getAfter(string, token), token));

fun rev(s) = implode(revList(explode(s)))

and revList(h::t) = revList(t)@[h]
  | revList([])     = [];
```

Running this test in `QuickECheck` yields the expected result that the test passed:

```
> qecheck "forall s:string.
      (exists t:char.appearsOnce(s, t) and even(size(s)))
          ==> not(rev(s) = s)"

forall s:string.(exists t:char.appearsOnce(s,t.....ok  (100 passed)
```

In this case the `QCheck` function generated by `QuickECheck` makes no use of `QChecks` special handling of implications due to the existential quantifier:

```
pred(fn (s, c) => ((not (appearsOnce(s, c)) orelse
                    not (even(size(s))) orelse
                    not(eq(rev(s), s))))))
```

6. Conclusion

Although the initial project was initially thrown astray by the development of `QCheck` between the selection of the project and its beginning, good progress has been made on creating an interesting and constructive project combining background investigation and program development. Although the `QCheck` tool is in itself a useful tool for random testing, `QuickECheck` provides a useful extension allowing a simpler and more explicit interface for `QCheck`. Even though only two out of the three outlined stages for the development of `QuickECheck` were successful, the iterative design process means that there is a working tool available.

Combining the ideas of formal semantics for testing with the existing `QCheck` tool for random testing within the Standard ML language, `QuickECheck` allows users to specify a formal specification which they wish to test in the form of an axiom represented in first-order logic. The grammar for accepted axioms is also designed to convey the type information required to generate a suite of random values to test. In addition to being able to specify simple axioms `QuickECheck` also allows the user to specify axioms containing nested universal quantifiers and existential quantifiers where appropriate, when inappropriate untestable axioms are inputted into the system a warning is conveyed to the user.

If more time had been available there are a number of improvements, large and small, which could be made to `QuickECheck`. The next two sections highlight some of these improvements starting with small improvements mainly focused around usability and moving towards larger improvements focusing specifically on the steps required towards satisfying the third development stage as outlined in the introduction.

6.1 Possible Small Improvements to QuickECheck

As discussed in section 3.3.5 above `QCheck` provides a number of user definable settings for specifying values such as the number of random values to test. In addition to these values `QuickECheck` also has several settings which could theoretically be set by the user, including the input domain for the generation of strings. A possible improvement to `QuickECheck` would be to package all user definable settings together and define functions which allow the user to quickly and easily adjust settings. For instance:

```
val setNumberOfTests : int -> unit
```

Another small improvement to `QuickECheck` would be to extend the range of generators available to include the special generators supplied by `QCheck` such as `Gen.Int.pos` for creating only positive integers. This could be achieved either via a flag in the user settings for different generators or by extending the grammar for inputs and adjusting the type information to allow special cases, ie. `int-pos`.

6.2 Possible Large Improvements to QuickECheck

As highlighted in Section 4.4 there remains an issue associated with how to make the best use of `QCheck`'s `implies` function. Currently `QuickECheck` will check the condition that an axiom does not contain an existential quantifier when distinguishing when to use the `implies` function. Clearly this does not cover all cases where the `implies` function will not work. To improve upon this current 'bug' would require further investigation into the specification of testing as well as a possible re-approach to the transformation of axioms into a form which `QCheck` can accept. The current situation is however more beneficial that the current alternative of substituting all implications for the logical equivalent $P \Rightarrow Q \rightarrow \neg P \vee Q$.

The third development stage intended for `QuickECheck` was to extend it in such a way as to allow quantification over non-standard types including those defined by users, for example, trees. The approach taken in the original `QuickCheck` paper [4] is to leave the process of defining generators for user defined types to the user. It states that although in principle it would be possible to create a pre-processor to create generators, although dismissed this as being too costly on the 'lightweight' tool both in terms of development, ease of use and the requirement to run code through the pre-processor before executing the test.

As noted in both the `QuickCheck` and `QCheck` papers when creating generators for datatypes such as trees non-termination becomes a possible problem and care must be taken. All `QCheck` generators are implemented as functors which would allow a suitably experienced user to create a generator match their own datatype. The problem with adding additional types into the existing `QuickECheck` is that currently the grammar which specifies the parser must be updated and as such the parser regenerated each time a new type is added. A possible solution around this problem would be to define the grammar with a set number of marker types which can be assigned to user defined types, this would not require the grammar or parser to change, but would limit the number of possible user defined types at any one time.

A further extension that was considered as a possible fourth stage near the beginning of the project is to make an attempt at tackling the problems associated with defining equality over non-equality types, the so called oracle problem. Where there is a semantic gap between program specification and program values it is not possible to define a decision procedure of interpreting test results [11]. An example of such a case can be shown by a specification for stacks of integers, in EML:

```
sig
  type stack
  val empty : stack
  val push : int * stack -> stack
  val pop : stack -> stack
  val top : stack -> int
  val isempty : stack -> bool
  axiom forall (a:int,s:stack) => top(push(a,s)) = a
  axiom forall (a:int,s:stack) => pop(push(a,s)) = s
end
```

For this specification the first axiom is fine, integers are an equality type and this can be tested for any given stack implementing the signature. However, where the stack implementation is not available the second axiom cannot be proved. It would be possible to remove and check each element in the stack to confirm that all elements appear in the same order, although the stack could be potentially infinite producing a likewise infinite list of observable consequences (where quantifiers have been omitted for clarity):

$$
\begin{aligned}
isempty(pop(push(x,s))) &= isempty(s) \\
top(pop(push(x,s))) &= top(s) \\
isempty(pop(pop(push(x,s)))) &= isempty(pop(s)) \\
top(pop(pop(push(x,s)))) &= top(pop(s)) \\
isempty(pop(pop(pop(push(x,s))))) &= isempty(pop(pop(s))) \\
top(pop(pop(pop(push(x,s))))) &= top(pop(pop(s))) \\
&\vdots
\end{aligned}
$$

As it is not possible to specify this list of observable in first-order logic as it is infinite [17], it cannot be applied to any formal testing procedure. A possible attempt at a solution to this problem would be to conduct an appropriate approximation to equality for types such as this, serving as guide, but in no way an exact solution.

Bibliography

[1] omg.org, 2005.

[2] M Ben-Ari. *Mathematical Logic for Computer Science.*

[3] S Bennett, J Skelton, and K Lunn. *UML.* Schaum's Outlines. McGraw-Hill, 2001.

[4] K Claessan and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. Technical report, Chalmers University of Technology, 2000.

[5] Bob Corbett. *Berkely Yacc.* http://sourceforge.net/projects/byacc/.

[6] Marie-Claude Gaudel. Testing can be formal, too. *Lecture Notes in Computer Science 915*, 1995.

[7] R.E. Jeffries. xprogramming.com, 2005.

[8] C League. Qcheck/sml. Technical report, Long Island University, 2004.

[9] Xavier Leroy. *Caml Light*, 0.75 edition, 2002.

[10] R. C. Linger. Cleanroom process model. *IEEE Software*, 11(2):50–58, 1994.

[11] P.D.L. Machado. *On Oracles for Interpreting test results against algebraic specifications.* PhD thesis, School of Informatics, 1998.

[12] The Object Management Group, http://uml.org. *UML Resource Page*, 2005.

[13] R Oshana. Quality software via a cleanroom methodology, 1997.

[14] Sergei Romanenko, Claudio Russo, and Peter Sestoft. *Moscow ML Owner's Manual*, version 2.0 edition, 2000.

[15] D. Sannella. Functional program and specificatiom, practical 2. Practical Excerise for undergraduate Computer Science Course.

[16] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. In *Formal Aspects of Computing.* 1997.

[17] D Sannella and A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development.* Cambridge University Press, 2006.

[18] U. Schöning. *Logic for Computer Scientists.* Birkhäuser, 1989.

[19] SML/NJ Fellowship, http://smlnj.org. *SML/NJ.*

Appendix A. Parser Source Code

Contained below is the code required to create the parser for converting axioms into `QuickECheck`'s internal datatype. `Data.sml` provides the defination for the datatype, `Lexer.lex` contains the code required to create a lexer using `mosmllex`, `Main.sml` contains the main code for using the parser and `Parser.grm` contains the code required to create a parser using `mosmlyac`.

A.1 Data.sml

```
(* Data.sml
 *
 * Neil Wilkie 2004-05
 * Edinburgh University Informatics UG4 Project
 *
 * 'Random testing of ML Programs'
 *
 * This is the datatype for representing axioms in ML.
 *)

(* This datatype was added to represent types for varibles within
 * axioms, these are required for QECheck to chose the correct
 * random data generator.
 *)
datatype tipe =
     Intt
   | Real
   | Char
   | String
   | Word


(* This datatype contains the structures required to represent
 * atoms within axioms.
 *)
datatype expr =
    Var of string
```

```
  | Vars of expr list
  | Bool of bool
  | Int of int
  | Stringg of string
  | Charr of string
  | Op of string * expr * expr
  | App of expr * expr
  | Andalso of expr * expr
  | Orelse of expr * expr


(* This datatype was added to represent atomic formulae within
 * axioms.
 *)
datatype atomic_formula =
     Equals of expr * expr
   | GreaterThan of expr * expr
   | LessThan of expr * expr
   | Expr of expr



(* This datatype was added to represent formulae to create
 * axioms.
 *)
datatype formula =
     Disjunction of formula * formula
   | Conjunction of formula * formula
   | Implies of formula * formula
   | Negation of formula
   | Forall of string * tipe * formula
   | Exists of string * tipe * formula
   | Atomic of atomic_formula
```

A.2 Lexer.lex

```
(* Lexer.lex
 *
 * Neil Wilkie 2004-05
 * Edinburgh University Informatics UG4 Project
 *
```

```
 * 'Random Testing of ML Programs'
 *
 * This is the lexer for the lexing and parsing of axioms to a form
 * by which they can be represented in the program. It is based
 * upon a lexer for 'Micro-ML' from a UG3 Functional Programming
 * and Specification practical by Don Sannella.
 *)

{
 open Lexing Parser;

 exception LexicalError of string * int * int

 (* (message, loc1, loc2) *)
 fun lexerError lexbuf s =
     raise LexicalError (s, getLexemeStart lexbuf,
                            getLexemeEnd lexbuf);

 (* Start of outermost comment being scanned *)
 val commentStart = ref 0;

 fun commentNotClosed lexbuf =
     raise LexicalError ("Comment not terminated",
                            !commentStart, getLexemeEnd lexbuf);

 val commentDepth = ref 0;  (* Current comment nesting *)

 (* Scan keywords as identifiers and use this function to
  * distinguish them.
  *)

 fun keyword s =
     case s of
         "true"         => TRUE
       | "false"        => FALSE
       | "andalso"      => ANDALSO
       | "orelse"       => ORELSE
       | "and"          => AND
       | "or"           => OR
       | "forall"       => FORALL
       | "exists"       => EXISTS
       | "int"          => INTT
       | "real"         => REAL
```

```
       | "word"         => WORD
       | "char"         => CHAR
       | "string"       => STRING
       | "mod"          => MOD
       | _              => NAME s;

 }

rule Token = parse
    [` ` `\t` `\n` `\r`]    { Token lexbuf }
  | `~`?[`0`-`9`]+  { case Int.fromString (getLexeme lexbuf) of
                        NONE   => lexerError lexbuf "internal err"
                      | SOME i => INT i
                        }
  | [`a`-`z``A`-`Z`][`a`-`z``A`-`Z``0`-`9`]*
                    { keyword (getLexeme lexbuf) }
  | "(*"            { commentStart := getLexemeStart lexbuf;
                      commentDepth := 1;
                      SkipComment lexbuf; Token lexbuf }
  | "=>"            { DOUBLEARROW }
  | `=`             { EQ }
  | `+`             { PLUS }
  | `-`             { MINUS }
  | `*`             { TIMES }
  | `/`             { DIV }
  | `|`             { BAR }
  | `(`             { LPAR }
  | `)`             { RPAR }
  | `>`             { GRTRTHN }
  | `<`             { LSSTHN }
  | "==>"           { IMPLIES }
  | `~`             { NOT }
  | "."             { DOT }
  | ":"             { COLON }
  | ","             { COMMA }
  | "#"             { HASH }
  | "\""            { QUOTE }
  | eof             { EOF }
  | _               { lexerError lexbuf "Illegal symbol in input" }

and SkipComment = parse
    "*)"                { commentDepth := !commentDepth - 1;
                          if !commentDepth = 0 then ()
```

```
                          else SkipComment lexbuf
                        }
   | "(*"               { commentDepth := !commentDepth + 1;
                          SkipComment lexbuf }
   | (eof | `\^Z`)      { commentNotClosed lexbuf }
   | _                  { SkipComment lexbuf }
;
```

A.3 Main.sml

```
(* Main.sml
 *
 * Neil Wilkie 2004-05
 * Edinburgh University Informatics UG4 Project
 *
 * 'Random testing of ML Programs'
 *
 * Parser for axioms. This code is unchanged from the code written
 * by Don Sannella for a 'Micro-ML' parser.
 *)

(* Parse *)

fun parseExprPlain file stream lexbuf =
    let val expr = Parser.Main Lexer.Token lexbuf
    in
        Parsing.clearParser();
        expr
    end
    handle exn => (Parsing.clearParser(); raise exn);

(* Parse; show offending program piece on error *)

fun parseExprReport file stream lexbuf =
    let val expr =
            Parser.Main Lexer.Token lexbuf
            handle
               Parsing.ParseError f =>
                   let val pos1 = Lexing.getLexemeStart lexbuf
                       val pos2 = Lexing.getLexemeEnd lexbuf
                   in
```

```
                       Location.errMsg (file, stream, lexbuf)
                                       (Location.Loc(pos1, pos2))
                                       "Syntax error."
                   end
             | Lexer.LexicalError(msg, pos1, pos2) =>
                   if pos1 >= 0 andalso pos2 >= 0 then
                     Location.errMsg (file, stream, lexbuf)
                                     (Location.Loc(pos1, pos2))
                                     ("Lexical error: " ^ msg)
                   else
                     (Location.errPrompt ("Lexical error: " ^ msg ^
                                          "\n\n");
                      raise Fail "Lexical error");
    in
        Parsing.clearParser();
        expr
    end
    handle exn => (Parsing.clearParser(); raise exn);

(* Create lexer from instream *)

fun createLexerStream (is : BasicIO.instream) =
  Lexing.createLexer (fn buff => fn n =>
                                 Nonstdio.buff_input is buff 0 n)

(* Parse an expression from a file *)

fun parsefile file =
    let val is     = Nonstdio.open_in_bin file
        val lexbuf = createLexerStream is
        val expr   = parseExprReport file is lexbuf
                     handle exn => (BasicIO.close_in is; raise exn)
    in
        BasicIO.close_in is;
        expr
    end

(* Parse an expression from a string *)

fun parse s =
    let val lexbuf = Lexing.createLexerString s
    in
        parseExprReport "" (BasicIO.std_in) lexbuf
```

```
        (* BasicIO.std_in is not actually used here - the function
           just requires something of that type to be supplied *)
    end
```

A.4 Parser.grm

```
/* Parser.grm
 *
 * Neil Wilkie 2004-05
 * Edinburgh University Informatics UG4 Project
 *
 * 'Random Testing of ML Programs'
 *
 * This is the parser for the parsing of axioms to a form by which
 * they can be represented in the program. It is based apon a
 * parser for 'Micro-ML' from a UG3 Functional Programming and
 * Specification practical by Don Sannella.
 */

%{
open Data;

fun Op2(s,x,y) = App(App(Op s,x),y);
%}

%token <int> INT
%token <bool> BOOL
%token <string> NAME
%token <char> CHARR
%token HASH QUOTE
%token INTT COLON WORD CHAR STRING REAL COMMA
%token PLUS MINUS TIMES DIV MOD
%token ANDALSO ORELSE
%token EQ
%token BAR
%token DOUBLEARROW
%token TRUE FALSE
%token LPAR RPAR
%token NOT
%token OR AND
%token FORALL EXISTS
```

```
%token GRTRTHN LSSTHN
%token DOT
%token IMPLIES
%token EOF
%token JUNK

%left ANDALSO ORELSE    /* lowest precedence */
%left PLUS MINUS
%left TIMES DIV         /* highest precedence  */

%start Main
%type <Data.formula> Main Formula
%type <Data.atomic_formula> Atomic_Formula
%type <Data.tipe> Tipe
%type <Data.expr> Expr AppExpr AExpr
%type <Data.expr list> VarList

%%

Main:
    Formula EOF                        { $1 }
;
Formula:
  | Formula OR Formula                 { Disjunction($1, $3) }
  | Formula AND Formula                { Conjunction($1, $3) }
  | Formula IMPLIES Formula            { Implies($1, $3) }
  | NOT Formula                        { Negation($2) }
  | FORALL NAME COLON Tipe DOT Formula { Forall($2, $4, $6) }
  | EXISTS NAME COLON Tipe DOT Formula { Exists($2, $4, $6) }
  | Atomic_Formula                     { Atomic($1) }
  | LPAR Formula RPAR                  { $2 }
;
Atomic_Formula:
  | Expr EQ Expr                       { Equals($1, $3) }
  | Expr GRTRTHN Expr                  { GreaterThan($1, $3) }
  | Expr LSSTHN Expr                   { LessThan($1, $3) }
  | Expr                               { Expr($1) }
;
Tipe:
  | INTT                               { Intt }
  | REAL                               { Real }
  | WORD                               { Word }
  | CHAR                               { Char }
```

```
  | STRING                          { String }
;
Expr:
    AppExpr                         { $1 }
  | Expr DIV Expr                   { Op("/", $1, $3) }
  | Expr TIMES Expr                 { Op("*", $1, $3) }
  | Expr PLUS Expr                  { Op("+", $1, $3) }
  | Expr MINUS Expr                 { Op("-", $1, $3) }
  | Expr MOD Expr                   { Op("mod", $1, $3) }
  | Expr ANDALSO Expr               { Andalso($1, $3) }
  | Expr ORELSE Expr                { Orelse($1, $3) }
;
AppExpr:
    AExpr                           { $1 }
  | AppExpr AExpr                   { App($1, $2) }
;
AExpr:
    NAME                            { Var $1 }
  | INT                             { Int $1 }
  | HASH QUOTE NAME QUOTE           { Charr $3 }
  | QUOTE NAME QUOTE                { Stringg $2 }
  | TRUE                            { Bool true }
  | FALSE                           { Bool false }
  | LPAR Expr RPAR                  { $2 }
  | LPAR VarList RPAR               { Vars($2) }
;
VarList:
    AppExpr                         { [$1] }
  | AppExpr COMMA VarList           { [$1]@$3 }
;
```

Appendix B. Main QuickECheck Source Code

Source code for the main QuickECheck program contained within the file `qecheck.sml`.

```
(* qecheck.sml
 *
 * Neil Wilkie 2004-05
 * Edinburgh University Informatics UG4 Project
 *
 * 'Random Testing of ML Programs'
 *
 * This is the main code for QuickECheck. All required functions
 * are loaded in here. It is this file which is required to be
 * loaded to use QuickECheck.
 *)

(* Load in some libraries which we are going to need later on. *)

load "Int";
load "Bool";
load "String";
load "Real";
load "Word";
load "Char";
open TextIO;


(* We need to load in the original QCheck stuff. Because we're
 * running mosml with '-I qcheck' we need not worry about
 * directories.
 *)

load "qcheck";
use "qcheck.sml";
open QCheck infix ==>;
open QCheck infix implies;


(* Load the parser and associated things. Because we're running
```

```
 * mosml with '-I parser' we need not worry about directories.
 *)

load "Data";
load "Lexer";
load "Parser";
load "Main";
use "Data.sml";
use "Lexer.sml";
use "Parser.sml";
use "Main.sml";


(* Define some exceptions which might need to be called if there is
 * an error and the user needs to know what's going on.
 *)

exception Not_Useful_Test;
exception Too_Many_Vars;


(* Using the type information from the axiom we can create an
 * appropriate generator and toString function.
 *)

(* Run all the way through the axiom and pull out the varible name
 * and type every time we reach a quantifier.
 *)
fun getVars(Disjunction(f0, f1), vars) =
      getVars(f1, getVars(f0, vars))
  | getVars(Conjunction(f0, f1), vars) =
      getVars(f1, getVars(f0, vars))
  | getVars(Implies(f0, f1), vars) =
      getVars(f1, getVars(f0, vars))
  | getVars(Negation(f), vars) =
      getVars(f, vars)
  | getVars(Forall(v, t, f), vars) =
      getVars(f, vars@[(v, t)])
  | getVars(Exists(v, t, f), vars) =
      getVars(f, vars@[(v, t)])
  | getVars(Atomic(a), vars) =
      vars
```

```
(* Using the list of varible names and types create the generator
 * and toString methods for the QCheck command. Anymore than 4
 * varibles and we're in trouble though. - Possible improvement,
 * create a way to zip together as many generators as it needs.
 *)
and makeGen([(v, t)]) =
      getGen(t) ^ ", SOME " ^ getToString(t)
  | makeGen([(v0, t0), (v1, t1)]) =
      "Gen.zip(" ^ getGen(t0) ^ ", " ^ getGen(t1) ^
      "), SOME (fn (t0,t1) => " ^ getToString(t0) ^
      " t0 ^ \", \" ^ " ^ getToString(t1) ^ " t1)"
  | makeGen([(v0, t0), (v1, t1), (v2, t2)]) =
      "Gen.zip3(" ^ getGen(t0) ^ ", " ^ getGen(t1) ^
      ", " ^ getGen(t2) ^ "), SOME (fn (t0,t1,t2) => " ^
      getToString(t0) ^ " t0 ^ \", \" ^ " ^ getToString(t1) ^
      " t1 ^ \", \" ^ " ^ getToString(t2) ^ " t2)"
  | makeGen([(v0, t0), (v1, t1), (v2, t2), (v3, t3)]) =
      "Gen.zip4(" ^ getGen(t0) ^ ", " ^ getGen(t1) ^ ", " ^
      getGen(t2) ^ ", " ^ getGen(t3) ^
      "), SOME (fn (t0,t1,t2) => " ^ getToString(t0) ^
      " t0 ^ \", \" ^ " ^ getToString(t1) ^ " t1 ^ \", \" ^ " ^
      getToString(t2) ^ " t2 ^ \", \" ^ " ^ getToString(t3) ^
      " t3)"
  | makeGen(vars) = raise Too_Many_Vars

(* Get the generators for each of the supported types. *)
and getGen(Intt)   = "Gen.Int.int"
  | getGen(Real)   = "Gen.Real.real"
  | getGen(Word)   = "Gen.Word.word"
  | getGen(Char)   = "Gen.char"
  | getGen(String) =
      "Gen.string(Gen.range(5,20),Gen.charFrom " ^
      "\"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz\")"

(* Get a toString method for each of the supported types. *)
and getToString(Intt)   = "Int.toString"
  | getToString(Real)   = "Real.toString"
  | getToString(Word)   = "Word.toString"
  | getToString(Char)   = "Char.toString"
  | getToString(String) = "String.toString"


(* Work through the algorithm for manipulating the formula. First
```

```
 * simplify it by removing implications, then push the negations
 * inwards, and finally pull the quantifiers to the outside.
 *)

(* Perform the substitutions to simplify the formula. *)
fun simplify(Disjunction(f0, f1)) =
      Disjunction(simplify(f0), simplify(f1))
  | simplify(Conjunction(f0, f1)) =
      Conjunction(simplify(f0), simplify(f1))
  | simplify(Implies(f0, f1)) =
      Disjunction(Negation(simplify(f0)), simplify(f1))
  | simplify(Negation(Implies(f0, f1))) =
      Conjunction(simplify(f0), Negation(simplify(f1)))
  | simplify(Negation(f)) =
      Negation(simplify(f))
  | simplify(Forall(v, t, f)) =
      Forall(v, t, simplify(f))
  | simplify(Exists(v, t, f)) =
      Exists(v, t, simplify(f))
  | simplify(Atomic(a)) =
      Atomic(a)

(* Have we simplified the formula fully yet? *)
and isSimplified(Disjunction(f0, f1)) =
      isSimplified(f0) andalso isSimplified(f1)
  | isSimplified(Conjunction(f0, f1)) =
      isSimplified(f0) andalso isSimplified(f1)
  | isSimplified(Implies(f0, f1)) =
      false
  | isSimplified(Negation(f)) =
      isSimplified(f)
  | isSimplified(Forall(v, t, f)) =
      isSimplified(f)
  | isSimplified(Exists(v, t, f)) =
      isSimplified(f)
  | isSimplified(Atomic(a)) =
      true

(* Keep preforming substitutions until the formula is
 * simplified.
 *)
and keepSimplifying(f, finished) =
      if finished
```

```
      then f
      else keepSimplifying(simplify(f), isSimplified(simplify(f)))

(* Perform the substitutions to move negations towards the atoms
 * of the formula.
 *)
and moveNegations(Disjunction(f0, f1)) =
      Disjunction(moveNegations(f0), moveNegations(f1))
  | moveNegations(Conjunction(f0, f1)) =
      Conjunction(moveNegations(f0), moveNegations(f1))
  | moveNegations(Implies(f0, f1)) =
      Implies(moveNegations(f0), moveNegations(f1))
  | moveNegations(Negation(Negation(f))) =
      moveNegations(f)
  | moveNegations(Negation(Disjunction(f0, f1))) =
      Conjunction(Negation(moveNegations(f0)),
                  Negation(moveNegations(f1)))
  | moveNegations(Negation(Conjunction(f0, f1))) =
      Disjunction(Negation(moveNegations(f0)),
                  Negation(moveNegations(f1)))
  | moveNegations(Negation(Exists(v, t, f))) =
      Forall(v, t, Negation(moveNegations(f)))
  | moveNegations(Negation(f)) =
      Negation(moveNegations(f))
  | moveNegations(Forall(v, t, f)) =
      Forall(v, t, moveNegations(f))
  | moveNegations(Exists(v, t, f)) =
      Exists(v, t, moveNegations(f))
  | moveNegations(Atomic(a)) =
      Atomic(a)

(* Have the negations been pushed in as far as they can all go? *)
and negationsMoved(Disjunction(f0, f1)) =
      negationsMoved(f0) andalso negationsMoved(f1)
  | negationsMoved(Conjunction(f0, f1)) =
      negationsMoved(f0) andalso negationsMoved(f1)
  | negationsMoved(Implies(f0, f1)) =
      negationsMoved(f0) andalso negationsMoved(f1)
  | negationsMoved(Negation(Negation(f))) =
      false
  | negationsMoved(Negation(Disjunction(f0, f1))) =
      false
  | negationsMoved(Negation(Conjunction(f0, f1))) =
```

```
      false
  | negationsMoved(Negation(Exists(v, t, f))) =
      false
  | negationsMoved(Negation(f)) =
      negationsMoved(f)
  | negationsMoved(Forall(v, t, f)) =
      negationsMoved(f)
  | negationsMoved(Exists(v, t, f)) =
      negationsMoved(f)
  | negationsMoved(Atomic(a)) =
      true

(* Keep pushing the negations inwards until the've all gone as far
 * as they can.
 *)
and keepMovingNegations(f, finished) =
      if finished
      then f
      else keepMovingNegations(moveNegations(f),
       negationsMoved(moveNegations(f)))

(* Perform the substitutions to move all of the axioms as far out
 * of the formula as possible. If we've simpified the formula and
 * we're lucky enough to have no existential quantifiers in
 * positive positions, or universal quantifiers in negative
 * positions then they should all come right out to the front.
 *)
and moveQuantifiers(Disjunction(Forall(v, t, f0), f1)) =
      Forall(v, t, moveQuantifiers(Disjunction(f0, f1)))
  | moveQuantifiers(Disjunction(Exists(v, t, f0), f1)) =
      Exists(v, t, moveQuantifiers(Disjunction(f0, f1)))
  | moveQuantifiers(Disjunction(f0, Forall(v, t, f1))) =
      Forall(v, t, moveQuantifiers(Disjunction(f0, f1)))
  | moveQuantifiers(Disjunction(f0, Exists(v, t, f1))) =
      Exists(v, t, moveQuantifiers(Disjunction(f0, f1)))
  | moveQuantifiers(Disjunction(f0, f1)) =
      Disjunction(moveQuantifiers(f0), moveQuantifiers(f1))
  | moveQuantifiers(Conjunction(Forall(v, t, f0), f1)) =
      Forall(v, t, moveQuantifiers(Conjunction(f0, f1)))
  | moveQuantifiers(Conjunction(Exists(v, t, f0), f1)) =
      Exists(v, t, moveQuantifiers(Conjunction(f0, f1)))
  | moveQuantifiers(Conjunction(f0, Forall(v, t, f1))) =
      Forall(v, t, moveQuantifiers(Conjunction(f0, f1)))
```

```
  | moveQuantifiers(Conjunction(f0, Exists(v, t, f1))) =
      Exists(v, t, moveQuantifiers(Conjunction(f0, f1)))
  | moveQuantifiers(Conjunction(f0, f1)) =
      Conjunction(moveQuantifiers(f0), moveQuantifiers(f1))
  | moveQuantifiers(Implies(f0, f1)) =
      Implies(moveQuantifiers(f0), moveQuantifiers(f1))
  | moveQuantifiers(Negation(f)) =
      Negation(moveQuantifiers(f))
  | moveQuantifiers(Forall(v, t, f)) =
      Forall(v, t, moveQuantifiers(f))
  | moveQuantifiers(Exists(v, t, f)) =
      moveQuantifiers(Negation(Forall(v, t,
                                      Negation(moveQuantifiers(f)))))
  | moveQuantifiers(Atomic(a)) =
      Atomic(a)

(* Have we pulled the quantifiers out as far as we can? *)
and quantifiersMoved(Disjunction(Forall(v, t, f0), f1)) =
      false
  | quantifiersMoved(Disjunction(Exists(v, t, f0), f1)) =
      false
  | quantifiersMoved(Disjunction(f0, Forall(v, t, f1))) =
      false
  | quantifiersMoved(Disjunction(f0, Exists(v, t, f1))) =
      false
  |quantifiersMoved(Disjunction(f0, f1)) =
      quantifiersMoved(f0) andalso quantifiersMoved(f1)
  | quantifiersMoved(Conjunction(Forall(v, t, f0), f1)) =
      false
  | quantifiersMoved(Conjunction(Exists(v, t, f0), f1)) =
      false
  | quantifiersMoved(Conjunction(f0, Forall(v, t, f1))) =
      false
  | quantifiersMoved(Conjunction(f0, Exists(v, t, f1))) =
      false
  | quantifiersMoved(Conjunction(f0, f1)) =
      quantifiersMoved(f0) andalso quantifiersMoved(f1)
  | quantifiersMoved(Implies(f0, f1)) =
      quantifiersMoved(f0) andalso quantifiersMoved(f1)
  | quantifiersMoved(Negation(f)) =
      quantifiersMoved(f)
  | quantifiersMoved(Forall(v, t, f)) =
      quantifiersMoved(f)
```

```
  | quantifiersMoved(Exists(v, t, f)) =
      quantifiersMoved(f)
  | quantifiersMoved(Atomic(a)) =
      true

(* Keep pulling out the quantifiers until they've come as far as
 * they can.
 *)
and keepMovingQuantifiers(f, finished) =
      if finished
      then f
      else keepMovingQuantifiers(moveQuantifiers(f),
 quantifiersMoved(moveQuantifiers(f)))

(* Do a full transformation including removing all the
 * implications.
 *)
and fullTransform(f) =
      keepMovingQuantifiers(keepMovingNegations(
      keepSimplifying(f, false),
      false), false)

(* Do a transformation, but leave the implications in place, we're
 * going to be needing them!
 *)
and simpleTransform(f) =
      keepMovingQuantifiers(keepMovingNegations(f, false), false);


(* Create the function required to preform the test in QCheck by
 * unparsing the formula into the correct form. Use the create
 * functions to unparse each of the different datatypes in the
 * correct way.
 *)

fun create(Disjunction(f0, f1)) =
      create(f0) ^ " orelse " ^ create(f1)
  | create(Conjunction(f0, f1)) =
      create(f0) ^ " andalso " ^ create(f1)
  | create(Implies(f0, f1)) =
      (* If any implications have slipped through this far then
       * we'd better remove them.
       *)
```

```
      "not (" ^ create(f0) ^ ") orelse " ^ create(f1)
  | create(Negation(f)) =
      "not (" ^ create(f) ^ ")"
  | create(Forall(v, t, f)) =
      "(" ^ create(f) ^ ")"
  | create(Exists(v, t, f)) =
      (* If there's any existential quantifiers left then they must
       * be in positive positions and it's not worth going any
       * further. Raise an exception to let the user know.
       *)
      raise Not_Useful_Test
  | create(Atomic(a)) =
      createA(a)

and createA(Equals(e0, e1)) =
      createE(e0) ^ " = " ^ createE(e1)
  | createA(GreaterThan(e0, e1)) =
      createE(e0) ^ " > " ^ createE(e1)
  | createA(LessThan(e0, e1)) =
      createE(e0) ^ " < " ^ createE(e1)
  | createA(Expr(e)) =
      createE(e)

and createE(Var(s)) = s
  | createE(Vars(list)) = displayList(list)
  | createE(Bool(b))    = Bool.toString(b)
  | createE(Int(i))     = Int.toString(i)
    (* Don't forget to put in the escape characters! *)
  | createE(Stringg(s)) = "\"" ^ s ^ "\""
  | createE(Charr(c))   = "#\"" ^ c ^ "\""
  | createE(Op(s, e0, e1)) =
      "(" ^ createE(e0) ^ " " ^ s ^ " " ^ createE(e1) ^ ")"
  | createE(App(e0, e1)) =
      createE(e0) ^ "(" ^ createE(e1) ^ ")"
  | createE(Andalso(e0, e1)) =
      createE(e0) ^ " andalso " ^ createE(e1)
  | createE(Orelse(e0, e1)) =
      createE(e0) ^ " orelse " ^ createE(e1)

and displayList([h])  = createE(h)
  | displayList(h::t) = createE(h) ^ ", " ^ displayList(t)
  | displayList([])   = ""
```

```
(* Get all the variable names and put them in the correct
 * format.
 *)
and stripVars([(v, t)])     = v
  | stripVars((v, t)::vars) = v ^ ", " ^ stripVars(vars)

(* If we don't have any existential quantifiers in the formula then
 * go ahead and use the special case for implications. This isn't
 * perfect, but implies is a very useful function...
 *)
and make(f) = if checkQuantifiers(f)
              then makeImplies(simpleTransform(f), getVars(f, []))
              else makeNoImplies(fullTransform(f), getVars(f, []))

(* Stick the special cases for implications in where we can. *)
and makeImplies(Forall(v, t, f), vars) = makeImplies(f, vars)
  | makeImplies(Implies(f0, f1), vars) =
      makeImplies2(f0, vars) ^ " implies " ^ makeImplies(f1, vars)
  | makeImplies(f, vars) =
      "pred(fn (" ^ stripVars(vars) ^ ") => " ^ create(f) ^ ")"

and makeImplies2(f, vars) =
      "(fn (" ^ stripVars(vars) ^ ") => " ^ create(f) ^ ")"

(* Don't bother with the implies function, just use pred. *)
and makeNoImplies(f, vars) =
      "pred(fn (" ^ stripVars(vars) ^ ") => " ^ create(f) ^ ")"

(* Run through the formula and check for any existential
 * quantifiers.
 *)
and checkQuantifiers(Disjunction(f0, f1)) =
      checkQuantifiers(f0) andalso checkQuantifiers(f1)
  | checkQuantifiers(Conjunction(f0, f1)) =
      checkQuantifiers(f0) andalso checkQuantifiers(f1)
  | checkQuantifiers(Implies(f0, f1)) =
      checkQuantifiers(f0) andalso checkQuantifiers(f1)
  | checkQuantifiers(Negation(f)) =
      checkQuantifiers(f)
  | checkQuantifiers(Forall(v, t, f)) =
      checkQuantifiers(f)
  | checkQuantifiers(Exists(v, t, f)) =
      false
```

```
  | checkQuantifiers(Atomic(a)) =
      true;


(* If the tag is longer than 51 characters then this causes a
 * problem with the output display. We also need to check incase we
 * need to add any escape sequences.
 *)

fun trim(s) =
      let
        val chars = explode(s)
      in
        implode(crop(chars, 0))
      end

and crop(h::t, count) =
      if count < 46
      then if h = #"\""
           then [#"\\", #"\""]@crop(t, count+2)
           else [h]@crop(t, count+1)
      else [#".", #".", #"."]
  | crop([], count) = [];


(* Make up the whole QCheck command by pasring the axiom, creating
 * the generators and toStrings, shove in the possibly trimmed and
 * safe (escape sequences) tag line then finally make the function
 * to be tested.
 *)
fun makeCommand(input) =
      let
        val parsed = parse(input)
  in
"checkGen (" ^ makeGen(getVars(parsed, [])) ^
") (\"" ^ trim(input) ^ "\", " ^ make(parsed) ^ ")"
      end;


(* When we're given an axiom we'd better make the QCheck command
 * then to run it we can write it to a file and bring it back in
 * again.
 *)
```

```
fun qecheck(input) = let
                       val outfile = openOut("temp.qec")
                       val command = makeCommand(input)
                     in
                       output(outfile, makeCommand(input));
                       flushOut(outfile);
                       use "temp.qec"
                     end;
```

Appendix C. Skolemisation Implentation

Assuming that the formula takes the form of prenex normal form the following program will convert the formula to skolem normal form, inserting skolem functions where necessary. Note that the inserted functions however do not have a defination and so the formula cannot be used, in cases where skolem functions are inserted into the formula it would not provide a useful test anyway.

```
(* skolem.sml
 *
 * Neil Wilkie 2004-05
 * Edinburgh University Informatics UG4 Project
 *
 * 'Random Testing of ML Programs'
 *
 * A 'theoretical' implentation of skolemisation. This code
 * requires formula to already be in prenex normal form.
 *)

load "Int";


(* Does the formula contain an existential quantifier? *)
fun hasExists(Disjunction(f0, f1)) =
      hasExists(f0) andalso hasExists(f1)
  | hasExists(Conjunction(f0, f1)) =
      hasExists(f0) andalso hasExists(f1)
  | hasExists(Implies(f0, f1)) =
      hasExists(f0) andalso hasExists(f1)
  | hasExists(Negation(f)) =
      hasExists(f)
  | hasExists(Forall(v, t, f)) =
      hasExists(f)
  | hasExists(Exists(v, t, f)) =
      true
  | hasExists(Atomic(a)) =
      false
```

```
(* Find the first existential quantifier in the formula. *)
and firstExists(Disjunction(f0, f1)) =
      let
        val fEf0 = firstExists(f0)
  in
        if fEf0 = "false"
then firstExists(f1)
else fEf0
      end
  | firstExists(Conjunction(f0, f1)) =
      let
        val fEf0 = firstExists(f0)
  in
        if fEf0 = "false"
    then firstExists(f1)
    else fEf0
      end
  | firstExists(Implies(f0, f1)) =
      let
        val fEf0 = firstExists(f0)
  in
        if fEf0 = "false"
        then firstExists(f1)
else fEf0
      end
  | firstExists(Negation(f))     = firstExists(f)
  | firstExists(Forall(v, t, f)) = firstExists(f)
  | firstExists(Exists(v, t, f)) = v
  | firstExists(Atomic(a))       = "false"


(* Make a list of all universally quantified variables in the
 * formula.
 *)
and getUniversal(Disjunction(f0, f1), list) =
      getUniversal(f1, getUniversal(f0, list))
  | getUniversal(Conjunction(f0, f1), list) =
      getUniversal(f1, getUniversal(f0, list))
  | getUniversal(Implies(f0, f1), list) =
      getUniversal(f1, getUniversal(f0, list))
  | getUniversal(Negation(f), list)     = getUniversal(f, list)
  | getUniversal(Forall(v, t, f), list) = getUniversal(f, list@[(v, t)])
  | getUniversal(Exists(v, t, f), list) = getUniversal(f, list)
```

```
  | getUniversal(Atomic(a), list)       = list


(* Replace the existentially quantified variable with the skolem
 * function.
 *)
and replace(Disjunction(f0, f1), e, app) =
      Disjunction(replace(f0, e, app), replace(f1, e, app))
  | replace(Conjunction(f0, f1), e, app) =
      Conjunction(replace(f0, e, app), replace(f1, e, app))
  | replace(Implies(f0, f1), e, app) =
      Implies(replace(f0, e, app), replace(f1, e, app))
  | replace(Negation(f), e, app) =
      Negation(replace(f, e, app))
  | replace(Forall(v, t, f), e, app) =
      Forall(v, t, replace(f, e, app))
  | replace(Exists(v, t, f), e, app) =
      if v = e
      then replace(f, e, app)
      else Exists(v, t, replace(f, e, app))
  | replace(Atomic(a), e, app) =
      Atomic(replaceA(a, e, app))

and replaceA(Equals(e0, e1), e, app) =
      Equals(replaceE(e0, e, app), replaceE(e1, e, app))
  | replaceA(GreaterThan(e0, e1), e, app) =
      GreaterThan(replaceE(e0, e, app), replaceE(e1, e, app))
  | replaceA(LessThan(e0, e1), e, app) =
      LessThan(replaceE(e0, e, app), replaceE(e1, e, app))
  | replaceA(Expr(e0), e, app) =
      Expr(replaceE(e0, e, app))

and replaceE(Var(v), e, app) = if v = e
       then app
       else Var(v)
  | replaceE(expr, e, app)   = expr


(* Skolemise the formula. First check for existential quantifiers,
 * if there aren't any then the formula is already in skolem normal
 * form, if there are then work through them one by one replacing
 * them with skolem functions.
 *)
```

```
and skolemise(f, i) =
      if hasExists(f)
      then let
     val exists    = firstExists(f)
     val universal = getUniversal(f, [])
           in
             skolemise(replace(f, exists,
           App(Var("sk" ^ Int.toString i),
       Vars(universal))), i+1)
           end
      else f;
```