| galois |

Introduction to the Formal Verification Course

Given a valid function specification S_f and an implementation I_f of that function *f*, a crucial question is whether $S_f = I_f$ on all inputs. Up to now industry has relied on verification testing to answer this question. This approach is quite difficult to manage for the following reasons: 1) a complete set of input data must be devised to test all execution paths of *f*; 2) it is unlikely that a complete set of input data is known; 3) even if a complete set of input data is known, execution of all paths in the testing process may cost too much time; 4) much human interaction is required in the testing process and humans make mistakes of many kinds (for example consider the lapses that resulted in the failure of the first Arianne 5 launch – see, e.g. https://www.esa.int/Newsroom/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report).

While testing can result in a high level of confidence that $S_f = I_f$, formal verification can raise confidence to a higher level, in some cases to certainty. Quite a few students, seeing formal verification for the first time, have commented that it runs on black magic and there is no entry point. There is certainly no entry point but it hardly runs on black magic. The following tiny example, from algebra, illustrates what formal verification is doing. Consider the equation

$2x = 10$

what is the value of *x*? Using an approach similar to testing one might try various values of *x* and test whether the equation is correct for those values until, in this case, 5 is found. But, there are rules of algebra – in this case dividing both sides of = by the same number to get an equivalent equation. Formal verification uses rules of logic to the same effect. Formal verification does not validate the specification but, if the specification is valid, formal verification can verify (or 'prove') correctness of function implementations, protocols, various properties that the function has, and it can find counterexamples that explain why an implementation or even a specification does not meet its intended functionality.

The purpose of this sequence of lessons is to illustrate what formal verification is capable of. Learning rules of logic is beyond the scope of this sequence. Several tools are available for formal verification. They fall into the classes Interactive Theorem Prover (ITP) and Automatic Theorem Prover (ATP). The benefit of ITP is powerful proof procedures, such as induction, but proofs typically require some intervention, and therefore a level of expertise, by the user. The benefit of ATP is typically user intervention is not needed or some minor amount of intervention is needed to speed things up but some powerful proof procedures are missing. In this sequence only ATP is considered and, in particular, the tools Cryptol and Software Analysis Workbench, by Galois, will be used. These tools are under development but are robust enough for the intended purpose of this sequence.

Install the Package and Environment

This version of the package is set up for Linux and Linux commands are used to install the package as well as run normal utilities such as text editors. The package consists of files and

directories that are bundled in one `7zip` file named `cryptol-course.7z`. That file is available for download. Open firefox (similar instructions apply for other browsers), write the following into the address bar: `http://gauss.ececs.uc.edu/cryptol-course.7z` and hit Enter. There should appear a dialog box with two choices: 1) Open with Archive Manager 2) Save File. If unsure of what to do choose 1) and wait for a dialog box showing the word 'Extract' at the top and '`ElemCourse`' under 'Name'. Click 'Extract'. A file chooser opens to allow putting the package anywhere in your directory structure but, for now, just click 'Extract'. The result is folder `ElemCourse` and file `cryptol-course.7z` in directory `Downloads` shown graphically as in Figure 1. Click or double click (depending on how your Linux is configured) on `ElemCourse` to enter the package directory as shown graphically in Figure 2.

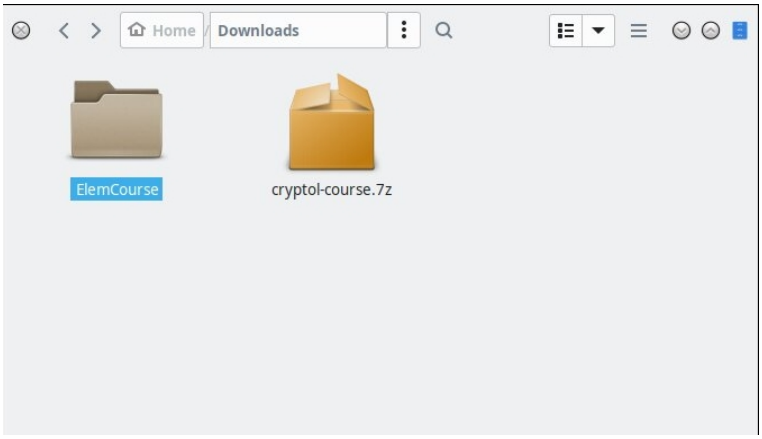

Figure 1*: bundled package and package directory* `ElemCourse`

Figure 2*: Folder* `ElemCourse` *contains package files*

Explore the subdirectories of `ElemCourse`. Directory `bin` contains a number of tools to support formal verification including Cryptol (`cryptol` and `cryptol-bin`) and the Software Analysis Workbench (`saw`). File `runterminal` is a script for running a `gnome-terminal` – something a user will not use directly. All other files are SMT solvers that are used by `cryptol` and `saw`. Directory `jdk-lin` contains `Openjdk` version 14 for running the java code that supports a point-and-click interface for accessing lesson content. Directory `src` contains subdirectories, each of which contains files with lesson descriptions, labs, solutions to labs,

manuals and other content, one subdirectory for each lesson plus subdirectories for images, sounds, and pdf files common to all lessons. File run starts the lesson sequence. Right click on run to bring up a menu. Choose ‘Run as a Program’. The index to the lessons shows up as in Figure 3.

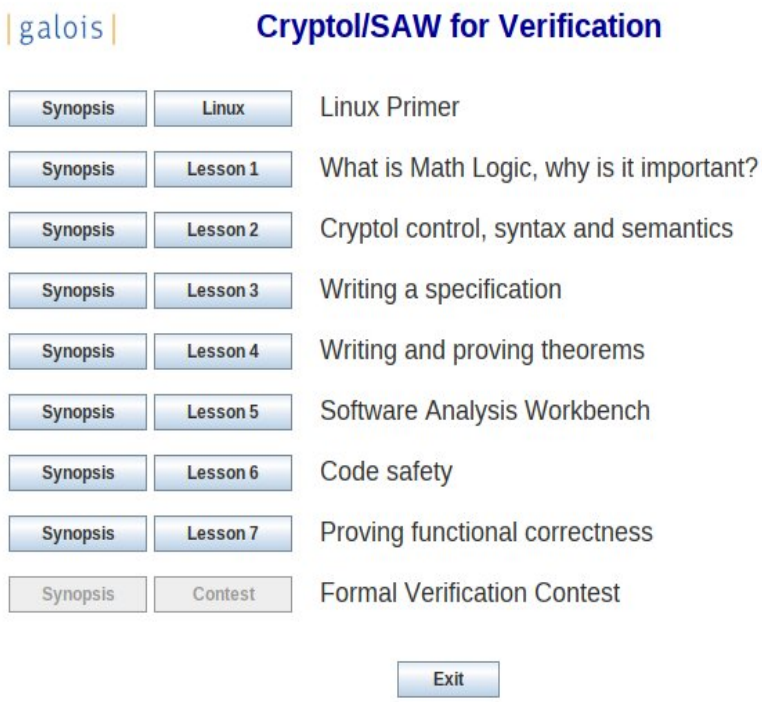

Figure 3: *Index to the lessons*

Example: Completing a Lesson

Each lesson has a synopsis which says a bit about what the lesson is intended to accomplish. Click a synopsis button to open a synopsis. Click a lesson button to access a lesson. A lesson may have several parts in which case it shows up as an index as well with synopses to explain intent. At some point, clicking lesson buttons will bring up content for a lesson. An example is shown in Figure 4 for a sub-lesson of ‘Writing a Specification’ then sub-sub-lesson 3.3 which shows how to write a specification for an sn74181 Arithmetic Logic Unit.

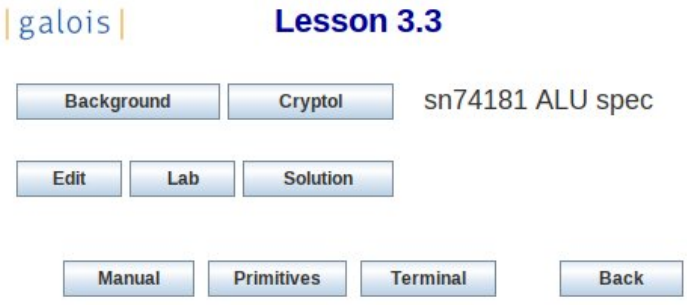

Figure 4: Content access pane for an ALU specification in Cryptol

Try it: click the ‘lesson 3’ button from the main index dialog, then the ‘lesson 3.3’ button from the ‘Writing a Specification’ dialog to arrive at the dialog of Figure 4. You will approach this lesson by first clicking the `background` button and looking at the background material which introduces you to information that will be needed to solve the problem at hand. In this case the background material is the Texas Instruments official spec sheet for a SN74AS181A Arithmetic Logic Unit. The spec sheet includes a wiring diagram as well as outputs on given inputs. Your task is to write an equivalent specification in Cryptol and prove that the wiring of the circuit produces outputs that are identical to those given in the TI spec sheet.

Tasks for doing this are found in the `lab` document (click the ‘Lab’ button to get it). In this case the lab document, Exercise 1, shows how to build a Cryptol specification from a wiring diagram. Exercise 2 illustrates how to build Cryptol functions that correspond to functions stated in the Texas Instruments spec sheet. Exercise 3 asks to write a function showing equivalence of the result of Exercise 1 and Exercise 2. You will be able to build on top of previous lessons to achieve this. If you need some help, code for this is presented in the lab directory. To see what code is there click the ‘Edit’ button to bring up a file chooser as shown in Figure 5.

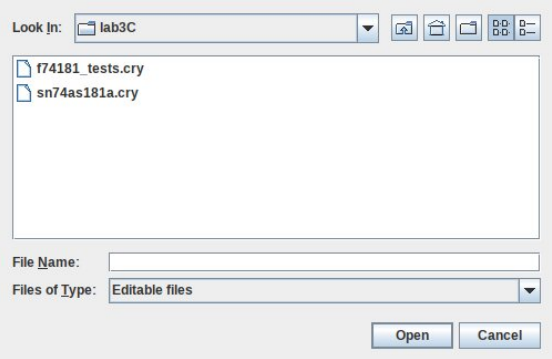

Figure 5: *File chooser - pick a file to edit*

You may select a file from the chooser – the file opens in a text editor. You may examine the contents of the file in the text editor to see where your help needs are met. You can run the code. In this case, the extension cry means both files are cryptol files. To run the code start Cryptol by clicking the ‘Cryptol’ button to bring up the Cryptol tool as shown in Figure 6.

Figure 6: *Cryptol has been started*

File `sn74as181a.cry`, for example, may be loaded into Cryptol with `:l sn74as181a.cry`:

```
Cryptol> :l sn74as181a.cry
Loading module Cryptol
Loading module sn74as181a
sn74as181a>
```

Inspection of the loaded file using the text editor will reveal a number of things you might try. You may run `:t f74181_netlist` to find the type of that function. The result is:

```
sn74as181a> :t f74181_netlist
f74181_netlist :
   Bit -> Bit -> Bit -> Bit -> Bit -> Bit -> Bit -> Bit -> Bit -> Bit -> Bit -> Bit -> Bit -> [8]
```

Knowing the type, you may run that function with crafted inputs according to the required input types and check outputs (in this case the output is an 8 bit number corresponding to output bits f_0,f_1,f_2,f_3,cout,p,g,a,b, in that order – `1` and `0` represent bit values in the following).

```
sn74as181a> f74181_netlist 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0x58
sn74as181a> :s base=2
sn74as181a> f74181_netlist 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0b01011000
```

The default number format is hexadecimal, hence `0x58`, but changing the format to binary with `:s base=2` results in `0b01011000` so, for example, `cout` is 1.

There are three lines in file `sn74as181a.cry` that begin with the word `property`. These are propositions that may be proved. You may prove that `f74181_fout` is correct for selectors s_0==0, s_1==1, s_2==1, s_3==0, m==1 for example, with `:prove xorWorks`, or that `f74181_spec` is correct for selectors s_0==1, s_1==0, s_2==0, s_3==1, m==0 with :prove addingWorks or, most importantly, you may prove equivalence as desired with `:prove itWorks` (see Figure 7 for these examples).

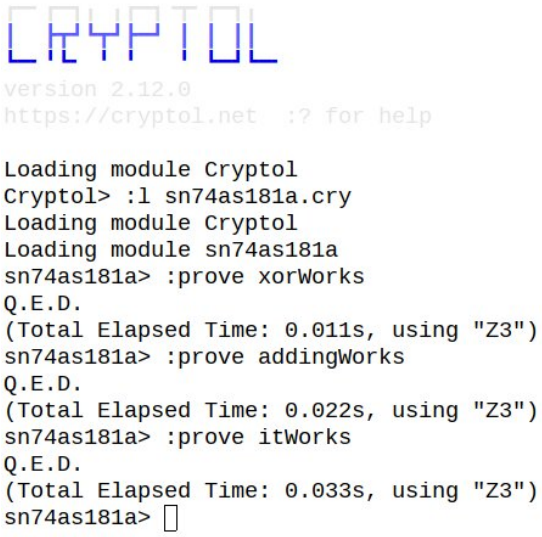

Figure 7: *Cryptol checks equivalence of ALU functions with wiring diagram*

Additional help may be found by clicking the 'Solution' button. In that case a working solution is usually presented and sometimes that solution is in the lab directory. Clicking the 'Terminal' button brings up a gnome-terminal that allows the use of Linux commands to operate on the contents of the lab directory. For example, you may wish to edit your own Cryptol, Java, C, or C++ code and compile and run in some cases. For Lesson 3.3 bring up the terminal and run the command `ls` to see what files are in the lab directory (see Figure 8).

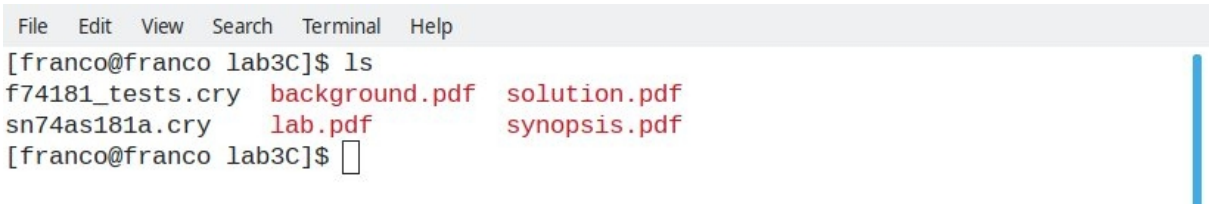

Figure 8: *A terminal invoked on the lab directory with contents shown using command* `ls`

You can use `evince` to view the pdf files plus do a lot of other things in the terminal. The Linux tutorial reveals many of those things.

Additional help is accessible from the two buttons on the left in the bottom row – in this case they are labeled 'Manual' and 'Primitives'. For this example clicking 'Manual' brings up a Cryptol manual. Later, when using the Software Analysis Workbench, 'Manual' brings up the SAW manual. Clicking 'Primitives' in this case brings up a tutorial on Cryptol primitive types. Later, 'Primitives' becomes 'Tutorial' which brings up the SAWscript tutorial when clicked.

The 'Back' button returns to the previous index dialog box.

What the Lessons are intended to do

Linux Primer

Linux offers numerous utilities that can create, manipulate, and delete directories and files. These can be used most easily in a terminal. With these utilities you will be able to create and explore numerous options that will extend your understanding of material beyond what exists in this course.

Lesson 1: What is Math Logic, why is it important

Logic manipulations can be reduced to algebra which makes it possible to build mechanical systems to prove theorems symbolically. Math Logic provides the rules and methods for proving properties of software, hardware and systems. Some of the basic concepts of logic are here such as soundness and completeness, axioms, rules of inference, and mechanical proof systems. Automation improves confidence since many logic mistakes that are made by humans are taken out of the equation. Examples of mechanized reasoning are given, such as Syllogistic logic and Propositional logic. Conjunctive Normal Form logic expressions are shown to have fundamental importance and inference rules such as resolution are presented to show how problems can be expressed in logic and solved efficiently in many cases. Some puzzles are solved in logic to illustrate problem formulation and analysis of output. The Lab shows why binary search is error prone if midpoints are found with something resembling `(low+high)/2`, a computation that has been used by educators for decades. With Math Logic a counterexample is found without the user even asking whether something is wrong – observe that test cases are built to catch possible bad states which have to be known so Math Logic already is an improvement over testing as it avoids lapses in human thinking. The Solution presented proves binary search correct with a midpoint calculation that is something like `low + (high-low)/2`.

Lesson 2: Cryptol control, syntax, and semantics

Cryptol is a relatively safe and functional specification language. It may be used to solve complex problems as well as C or Java with much less apparent control code. The aim of this sequence of Lessons is to learn Cryptol well enough to write specifications and to solve some problems on the side. There are five Lessons in this sequence to help with this.

Lesson 2.1, 'Commands and built-in functions'

The background material documents Cryptol commands and built-in functions and type operations. In the Lab directory is file `examples.cry` which contains numerous examples using said functions, commands, and operations. The Lab contains 10 exercises asking to use Cryptol to solve some elementary problems. The Solution page shows solutions to those exercises.

Lesson 2.2, 'Types'

The background material discusses type classes that allow describing behaviors that are shared by more than one type. Examples of type classes are `Eq` (equal) and `Cmp` (compare) but there are others. Cryptol introduces type variables that allow expressing interesting (arithmetic) constraints on types such as lengths of sequences or relationships between lengths of sequences. These will be very important in handling some solutions which otherwise may not be possible due to the very strict typing rules of Cryptol. Simpler examples are presented here. Basic data types are presented including sequences and tuples, with examples. The Lab consists of eight simple exercises and the Solution page shows solutions to those.

Lesson 2.3, 'Data Structures'

Shows how to build data structures in Cryptol, analogous to structs, arrays, etc. in C. A simple puzzle illustrates how data structures can be used to advantage: see, on Page 4, `property solution states` which solves the puzzle on logic considerations alone (from the perspective of the user) in the absence of control code. The Lab directory contains `warmup.c` and `warmup.cry` to compare structures in both languages, `foxChickenCorn.cry` which is a complete solution to the claasic puzzle, and `solutions.cry` which are solutions to the Lab exercises. Lab exercises ask to build solutions to some problems using user-defined data structures.

Lesson 2.4, 'Cryptol comprehensions'

Explains the most important structure in Cryptol, and how to use it. The background material contains numerous examples, ending with a rather complex and interesting one. The Lab consists of some challenging exercises where you must find workarounds for the strong typing rules of Cryptol (hints are given).

Lesson 2.5, 'Cryptol functions'

Shows how to create Cryptol functions and several examples, some infinite, are given. There are numerous Cryptol functions in the Lab directory, based on the background material. The Lab asks to solve a difficult puzzle. The Solution page shows how to do it.

This sequence of lessons should be sufficient preparation for the next sequence which illustrates how to write a specification in Cryptol.

Lesson 3: Writing a specification in Cryptol

This sequence of five lessons shows how to write specifications for five examples. In two cases, namely the ZUC mobile phone pseudo-random sequence generator and the Texas Instruments sn74181 Arithmetic Logic Unit, the Cryptol specification is developed from a published, human readable specification. Alongside the specifications are required properties and Cryptol is used to prove those properties to hold or not.

Lesson 3.1, 'Specification from Human Readable to Cryptol: ZUC'

Two Cryptol specifications of ZUC version 1.4 and version 1.5 are presented in the Lab directory as `ZUC_v1.4.cry` and `ZUC_v1.5.cry`. The human readable, ETSI/SAGE specification of version 1.5 is the background material. One of two remarkable features of this lesson is to see how closely the Cryptol code matches the functions stated in the ETSI/SAGE specification. In fact, the Cryptol code is structured to match: for example, function `LFSRWithInitialisationMode(u)` defined in ETSI/SAGE is implemented as `LFSRWithInitializationMode (u, ss)` in Cryptol and so on. As example, the line in ETSI/SAGE

$v=2^{15}s_{15}+2^{17}s_{13}+2^{21}s_{10}+2^{20}s_{4}+(1+2^{8})s_{0}$ mod $(2^{31}-1)$;

is implemented in `ZUC_v1.5.cry` as

```
v = add [s <<< c | s <- ss @@ [15,13,10,4,0,0] | c <- [15,17,21,20,8,0]]
```

where the mod $(2^{31}-1)$ addition function '`add`' is not formally defined in ETSI/SAGE and must be implemented in Cryptol, appearing at line 13 of `ZUC_v1.5.cry`. You can study ETSI/SAGE and the code and see that the code naturally follows from the ETSI/SAGE spec. The second remarkable feature of this lesson is that a bug is found in ETSI/SAGE version 1.4, implemented as `ZUC_v1.4.cry`, and it is a bug that Cryptol finds immediately whereas testing would be highly challenged to find it. A safety requirement of any protocol like ZUC is

that two different initialization vectors, with the same key, should produce two different initialization sequences. Cryptol code for expressing this is (actually, the code is only looking at the first output of `InitializeZUC`, which is sufficient, hence the '`@ 1`'):

```
property ZUC_isResistantToCollisionAttack k iv1 iv2 =
  if iv1 != iv2
  then InitializeZUC (k, iv1) @ 1 != InitializeZUC (k, iv2) @ 1
  else True
```

Open Cryptol, load `ZUC_v1.4.cry`, then run `:prove ZUC_isResistantToCollisionAttack`:

```
Loading module Cryptol
Cryptol> :l ZUC_v1.4.cry
Loading module Cryptol
Loading module Main
Main> :prove ZUC_isResistantToCollisionAttack
Counterexample
ZUC_isResistantToCollisionAttack
  0x49000000b774000000003f0000a60094
  0x850070004b00000000007c0e00df5ac1
  0xb10070004b00000000007c0e00df5ac1
  = False
(Total Elapsed Time: 0.431s, using "Z3")
```

Incredibly, a counterexample is found in less than ½ second, showing that v1.4 is defective. The size of the key and initialization vectors is 128 bits. Imagine how long it would likely take to get this result with testing! Load `ZUC_v1.5.cry` which has a fix for this bug and get this:

```
Main> :l ZUC_v1.5.cry
Loading module Cryptol
Loading module Main
Main> :prove ZUC_isResistantToCollisionAttack
Q.E.D.
(Total Elapsed Time: 1.334s, using "Z3")
```

The property is shown to hold in just 1.334 seconds. Of course, this is not the whole story as this property must be shown to hold for all outputs of `InitializeZUC`. We don't go any further with this here because the objective is to demonstrate two remarkable properties of the Cryptol specification language and that objective has been met.

Lesson 3.2, 'Specification of DES'

DES is an old symmetric key cryptographic scheme that is no longer used but TripleDES, derived from DES is. Background materials explains both. The code `DES.cry` and `Cipher.cry` in the Lab directory implement the specification according to the background material. As in Lesson 3.1 there is a straightforward translation from the human generated description of DES and TripleDES to Cryptol code. An obvious required property is that for any key, decrypting an encrypted message produces the message. This is expressed in Cryptol for DES:

```
DESworks : [64] -> [64] -> Bit
property DESworks key pt = (DES.decrypt key (DES.encrypt key pt)) == pt
```

and for TripleDES:

```
TripleDESworks : [64] -> [64] -> [64] -> Bit
property TripleDESworks k1 k2 pt =
  (DES.decrypt k1 (DES.encrypt k2 (DES.decrypt k1 (DES.encrypt k1 (DES.decrypt \
  k2 (DES.encrypt k1 pt)))))) == pt
```

Using prover `abc` `DESworks` proves in less than a second.

```
DES> :s prover=abc
DES> :prove DESworks
Q.E.D.
(Total Elapsed Time: 0.758s, using "ABC")
```

TripleDESworks proves in 3.4 seconds. The default prover, Z3, does not do well on these properties. Thus, this lesson reveals something about choosing the right provers for the problem at hand: if one prover is not working, try another. To find out the available provers do this:

```
DES> :s prover=?
Prover must be cvc4, yices, z3, boolector, mathsat, abc, offline, any, sbv-cvc4,
sbv-yices, sbv-z3, sbv-boolector, sbv-mathsat, sbv-abc, sbv-offline, sbv-any,
w4-cvc4, w4-yices, w4-z3, w4-boolector, w4-abc, w4-offline, or w4-any
```

The interesting feature of this lesson is in the Lab exercises. Some cryptographic schemes are vulnerable to weak keys. A weak key, in the case of the 16 round DES algorithm, can be such that all per round keys, which are computed from the weak key, are the same. Another can be such that encrypting a message then encrypting the encrypted message produces the message. Cryptol is able to find such weak keys quickly. The Lab exercises establish the problems and the Solution page shows solutions. For example, to find weak keys of the first kind use this code:

```
// Returns a sequence of 1s iff all numbers in (expandKey key) are the same
func key = z
  where
    z = [ x == y | x <- (expandKey key) | y <- (drop `{1} (expandKey key)) ]

// Exercise 1: weak keys
property WeakKeys key = (func key) == ~zero
```

Then, in Cryptol, do this:

```
solution> :s prover=z3
solution> :s satNum=all
solution> :sat WeakKeys
Satisfiable
WeakKeys 0x1e1e1e1e0e0e0e0e = True
WeakKeys 0x0000000000000000 = True
WeakKeys 0x0000000000000100 = True
WeakKeys 0x0000000100000100 = True
WeakKeys 0x0100000100000100 = True
...
Models found: 1024
(Total Elapsed Time: 7.044s, using "Z3")
```

So, Cryptol does more than just check that properties hold.

Lesson 3.3, ‘Specification of sn74181 ALU’

This ALU is hardware with two 4 bit inputs and 5 selector bits that are set to choose the operation that is applied to the two inputs. The goal is to show that the circuit wiring produces output that matches the functionality stated in the Texas Instruments specification. The Lab document shows how to translate a wiring diagram to Cryptol. Functions stated in the TI spec are easy to implement in Cryptol. You put the two together to show that they are equivalent – that is, for all inputs, the outputs of the circuit match what the TI spec says the output should be.

Lesson 3.4, 'Multiplier in assembly'
The background material presents a piece of code in assembly for a Mostek processor plus legal operations on registers. The code intends to multiply two 8 bit numbers placing the output into two 8 bit registers. The goal is to prove that the multiplier multiplies correctly. The Lab walks you through the steps needed to solve the problem. The Cryptol code expresses state and then steps from one state to another depending on the output of the previous state. This is a different construction that what has been seen up to this point.

Lesson 3.5, 'Bounded Model Checking'
Speaking of state, many problems require infinite state changes. An example would be an automotive ECU. Such problems might be handled by some form of temporal logic. Cryptol can handle bounded versions of such problems. The background information shows how this is done. The Lab exercise asks to model the operation of an elevator in Cryptol so that some specified properties hold up to a certain point.

Lesson 4, 'Writing and proving theorems'
Some of the principles of writing and proving theorems are presented. Due to its strong typing, Cryptol can sometimes catch subtle errors that might not be noticed otherwise. In the case of implementing a well-known function to check whether two sequences are permutations of each other, one might replace 'removed' locations with a -1 to satisfy the strong typing. But that raises the problem of whether -1 must now be forbidden from input sequences. Switching to a well-known sorting algorithm, sequence sizes change during execution and the strong typing does not tolerate that: a workaround is to apply infinite padding to the input. A sorting example that does this is shown in this lesson but the result is an infinite sequence. This is where type variables come in:

```
bsort : {n, a} (Cmp a, Integral a, Literal 1 a, fin n) => [n]a -> [n]a
bsort lst = take `{n} (bb (lst#[-1,-1…]))
```

In the above, the given list to sort, `lst`, is padded with an infinite sequence of `-1` (observe `lst#[-1,-1…]`) and the padded list is input to (an omitted function) `bb` which returns the sorted list followed by infinite padding with `-1`. The padding is removed by the `take `{n}` – look at the type specification for `bsort` and observe the `n` shows up as the input and output sequence length in `[n]a`. The `take `{n}` just grabs the first `n` elements of the `bb` output which happens to be the sorted sequence, exactly. The only problem is the type specification needs to be crafted in advance. That is not too hard, though. Use something in place of `n` in the `take `{n}` then load the function then perform `:t` to see how Cryptol defines the type specification and add the `n` to the type specification as above and put the `n` in `take `{n}`. Lab exercises provided are intermediate level and should help familiarize you with the meaning of verification in Cryptol. In particular, one exercise has Cryptol proving the equivalence of two functions although they are not equivalent – but this is perfectly OK in that particular case

Lesson 5, 'Software Analysis Workbench'
This is an introduction to SAW. This sequence of lessons shows that SAW can prove equivalence between functions implemented in different languages or between a given function and a Cryptol specification of that function, regardless of how the function is implemented.

Lesson 5.1, 'Introduction to SAW'

A simple C function is given and Cryptol specifications of add, with and without pointers, are created. The C code is compiled using clang to llvm, an intermediate representation. Ways to deal with C pointers are shown.

Lesson 5.2, 'Equivalence checking'
Defines the problem of finding the first 1 in a given 32 bit sequence. The Lab exercises ask to implement several different algorithms to solve this problem then show that all are functionally equivalent with SAW. You are tasked to create and run the SAW file that solves this problem.

Lesson 5.3, 'Equivalence across languages'
Again considers the problem of finding the first 1 in a given 32 bit sequence but provides solutions in C, Java, and Cryptol. The Lab exercises introduce and-inverter-graphs and ask to show equivalence of all the functions.

Language 5.4, 'Salsa20'
Salsa20 is an encryption/hash function. Provided is a C implementation and specification. You are tasked to create a Cryptol specification from the C specification and to prove that your specification matches. This is beginning to get quite advanced.

Lesson 6, 'Code safety'
Code should be written to solve a problem and nothing more. The 'nothing more' part is safety. Kinds of safety are: thread safety, type safety, and memory safety. Thread safety does not apply yet to Cryptol so this lesson treats type safety and memory safety. Of course, the best examples of these come from the C language.

Lesson 6.1, 'Memory safety'
The Lab directory is filled with C and Cryptol code and SAWscript. The background has numerous examples of memory unsafe C code. You are tasked to determine whether given C code is memory safe or not by writing Cryptol specifications and proving or disproving equivalence.

Lesson 6.2, 'Type safety'
The background has several examples of type unsafe C code. You are tasked to use Cryptol to find a counterexample that shows that a particular C application is not type safe. You are to create Cryptol code and SAWscript that uses that code to prove a modified application is type safe.

Lesson 7, 'Proving functional correctness'
By this time you have already seen something about proving functional correctness. This sequence of two lessons is an advanced version.

Lesson 7.1, 'Proving functional correctness'
The Lab directory is filled with C and Cryptol code and SAWscript. The background material shows how SAW can handle things like global variables, C structs, and C pointers. Quite a lot of SAWscript snippets are presented to show how to handle these advanced cases. At the end of the background material several helpful SAWscript snippets are presented. Lab exercises help in practicing the use of these snippets.

Lesson 7.2, 'SHA512'
Very ambitious lab to prove correctness of an implementation of the SHA512 hash algorithm.