The VampIR Book

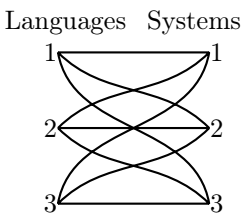
The VampIR Team

Contents

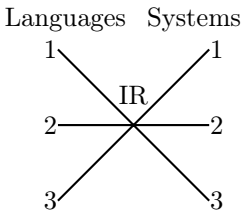
1 What is VampIR and How is it Used?

1.1 Why VampIR?

VampIR is, at its core, a language for defining arithmetic circuits over finite fields. It's intended to be a universal intermediate language supporting many proof systems based on arithmetic circuits. Any higher-level language which intends to compile to arithmetic circuits may target VampIR as an intermediate language.



The need for an intermediate language is obvious. Without an adequate intermediate, systems must support every desired proof system individually. This creates an ecosystem like that depicted in the right figure, where the total work required for interconnectedness is quadratic. This also creates more points of failure and more opportunities for support inconsistencies.



With an adequate intermediate representation, the total amount of work necessary to fill out the ecosystem is only linear. This can be seen in the ecosystem depicted in the left figure. Languages can, instead of targeting a specific proof system, target the intermediate representation. That language would then have support for every proof system targeted by that intermediate representation.

VampIR's goal is to fulfill this need by providing a minimal and expressive interface for describing the core data structure used by most modern zero-knowledge-proof systems. To that end, VampIR aims to be flexible and easy to use but doesn't provide any cryptographic features of its own. It does not presuppose any particular implementation or design for a proof system. VampIR files are sufficiently generic that they may even be used for applications that use arithmetic circuits but are not cryptographic in nature.

1.2 Using VampIR

A very basic example of a VampIR program is the following;

```
// This is a comment!
def x = 10;

x = 10;
```

Here, one can see the two main top-level commands available in VampIR. That first line defines a constant, `x`, to which the value `10` is assigned. Names must start with a letter or an underline, and may contain any number of letters, numerals, or underlines after. The second line is an equation that is expected to be true. Every arithmetic circuit is interpreted as a proposition. Specifically, it is the proposition corresponding to the truth of all the equations appearing in the file which generated it. In this case, the compiled circuit will merely check that `10 = 10`. Notice that every line must end in a semicolon. VampIR does not generally care about white space or newlines (beyond spaces separating individual tokens); this example would be interpreted the same if all newlines were removed and everything was put on a single line.

To compile this circuit, of course, VampIR needs to be up and running. First clone VampIR's directory.

```
$ git clone git@github.com:anoma/vamp-ir
```

VampIR is implemented in Rust and can easily be compiled from source using Rust Cargo.

```
$ cd vamp-ir
$ cargo build
```

This will create the VampIR binary at '/target/debug/vamp-ir' within the main VampIR directory.

VampIR does not possess any cryptographic capabilities on its own. This means that some specific parameters, such as the field size, cannot be determined by VampIR, but are instead decided at compile time. To compile this circuit, a target must first be chosen. For this example, I will choose PLONK [What is the name of the program? How does one access other targets?].

For the sake of this example, I will assume that the lines in our example program are saved into a file called 'ex1.pir', stored within a new folder called 'examples' within the main VampIR directory.

```
$ mkdir examples
$ printf "def x = 10;\n\nx = 10;">examples/ex1.pir
```

Notice the file ends with '.pir', the standard extension for VampIR files. To compile this into a PLONK circuit, one must first set up public parameters.

```
$ target/debug/vamp-ir setup -o examples/params.pp

> * Setting up public parameters...
> * Public parameter setup success!
```

This will create the file 'params.pp' within our 'examples' directory. The `-o` argument indicates an output file and is equivalent to `--output`. One can now create the circuit associated with our file.

```
$ target/debug/vamp-ir compile -u examples/params.pp \
                               -s examples/ex1.pir \
                               -o examples/circuit.plonk

> * Compiling constraints...
> ** Inferring types...
> x[2]: int
> * Reading public parameters...
> * Synthesizing arithmetic circuit...
> * Serializing circuit to storage...
> * Constraint compilation success!
```

This will create our compiled circuit in the file 'circuit.plonk' within the 'examples' directory. The `-u` argument indicates a universal parameter file and is equivalent to `--universal-params`. The `-s` argument indicates a source file and is equivalent to `--source`.

Notice that types for defined expressions are inferred during compilation. VampIR has a simple type system that is mostly implicit. This will be explained in more detail later on. In this simple example, `x` is inferred to be an 'int', that is, an integer that will be interpreted as a field element during compilation.

A zero-knowledge proof of circuit correctness can now be synthesized.

```
$ target/debug/vamp-ir prove -u examples/params.pp \
                             -c examples/circuit.plonk \
                             -o examples/proof.plonk

> * Reading arithmetic circuit...
> * Soliciting circuit witnesses...
> * Reading public parameters...
> * Proving knowledge of witnesses...
> * Serializing proof to storage...
> * Proof generation success!
```

This will create our compiled proof in the file 'proof.plonk' within the 'examples' directory. The `-c` argument indicates a circuit file and is equivalent to `--circuit`.

The last thing one may want to do is verify the circuit.

```
$ target/debug/vamp-ir verify -u examples/params.pp \
                              -c examples/circuit.plonk \
                              -p examples/proof.plonk

> * Reading arithmetic circuit...
> * Reading zero-knowledge proof...
> * Public inputs:
> * Reading public parameters...
> * Verifying proof validity...
> * Zero-knowledge proof is valid
```

This will verify the proof just created. In this case, we've just verified a zero-knowledge proof that $10 = 10$. The `-p` argument indicates a proof file and is equivalent to `--proof`.

Other than `help`, every available VampIR command has now been used; `setup`, `compile`, `prove`, and `verify`. These commands define all current methods for interacting with VampIR. It is a very simple system.

1.3 Proof Validity and Interaction

Values for variables do not need to be given upfront. If our file was instead

```
x = 10;
```

without declaring the value of `x`, VampIR would interpret this unbound variable as an input needing to be specified during proof generation. If this is saved in the file 'ex2.pir' and compiled to a circuit, one can see that VampIR will ask for an input when it's needed.

```
$ printf "x = 10;">examples/ex2.pir
$ target/debug/vamp-ir compile -u examples/params.pp \
                               -s examples/ex2.pir \
                               -o examples/circuit2.plonk

> * Compiling constraints...
> [...]
> * Constraint compilation success!

$ target/debug/vamp-ir prove -u examples/params.pp \
                             -c examples/circuit2.plonk \
                             -o examples/proof2.plonk

> * Reading arithmetic circuit...
> * Soliciting circuit witnesses...
> ** x[2] (private):

$ 9

> * Reading public parameters...
> [...]
> * Proof generation success!
```

It asked for the private value for `x`, to which I input '9'. This should create an invalid proof this time as 9 does not equal 10. If one tries verifying the proof, they will observe that it's invalid.

```
$ target/debug/vamp-ir verify -u examples/params.pp \
                              -c examples/circuit2.plonk \
                              -p examples/proof2.plonk

> * Reading arithmetic circuit...
> [...]
> * Verifying proof validity...
> * Result from verifier: Err(ProofVerificationError)
```

As you can see, a verification error is given.

2 Programming in VampIR

2.1 Basic Arithmetic

VampIR's arithmetic denotations are fairly straightforward, being very similar to what's present in many mainstream languages.

VampIR provides four different methods to denote numbers. One can use

- standard decimal notation
- hexadecimal notation, so long as the number is prefixed by `0x`
- octal notation, so long as the number is prefixed by `0o`
- binary notation, so long as the number is prefixed by `0b`

The following example program demonstrates all the number notations.

```
0xee9a592ba9a9518 = 1074572035719075096;
1074572035719075096 = 0o735151311127246512430;
0o735151311127246512430 =
0b1110111010011010010110010010101110101001101010010101000011000;
```

Notice that VampIR allows for many top-level equations in the same program.

One can add, subtract, and multiply numbers, using, respectively, the infix operators `+`, `-`, and `*`.

```
20 * 3 - 20 = 20 + 20;
```

VampIR uses a standard order of operations where multiplication takes precedence over addition and subtraction, and everything is left-associated otherwise. Further clarification can be made using parentheses.

```
1 + 2 + 3 - 4 * 5 - 6 + 7 =
((((1 + 2) + 3) - (4 * 5)) - 6) + 7;
```

VampIR also allows for division in arithmetic circuits, denoted with the infix operator `/`.

```
15/3 = 5;
```

Division has the same precedence as multiplication, and associates to the left, otherwise.

```
2/3*4/5 = ((2/3)*4)/5;
```

This operation will give an output using the division of the underlying field, meaning constraints may differ in truth value depending on the compilation target. If one uses a prime field of size

52435875175126190479447740508185965837690552500527637822603658699938581184513,

used, for example, in BLS12-381, then the following is valid.

```
2/3 = 0x26a48d1bb889d46d66689d580335f2ac713f36abaaaa1eaa5555555500000001;
```

If one tries dividing by zero, they will get an error at some point in the process depending on where the division occurs. `1/0 = 1;` will produce an error during type inference. `1/x = 1;` will produce an error if 'x' is set to 0 during witness solicitation.

VampIR also possesses a single unary operation in the form of negation, denoted with `-`. Negation must be surrounded by parentheses unless it's iterated with additional negations.

```
(-10) = 0 - 10;
0 - 10 = (---10);
```

Exponentiation is also available via the infix operator `^`.

```
2 ^ 2 = 4;
```

Exponentiation has greater precedence than any other arithmetic operator.

```
5 * 6 ^ 7 = 5 * (6 ^ 7);
```

Only constant exponents are allowed, as they can be translated into iterated multiplication at circuit compilation time. `2 ^ x = 4;` will create an error during type inference.

Exponentiation by 0 will always produce 1, even if the base is 0.

```
2 ^ 0 = 1;
0 ^ 0 = 1;
0 ^ 2 = 0;
```

[Note: this section may describe a bug instead of intended behaviour]

Exponentiation by negatives is a bit weird. They are offset by 1 from what one might expect.

```
2 ^ (-5) = 1 / 2 ^ 4;
6 ^ (-2) = 1 / 6;
5 ^ (-3) = 1 / 5 ^ 2;
```

Exponentiating by -1 will always produce 1, with the exception of 0.

```
2 ^ (-1) = 1;
6 ^ (-1) = 1;
5 ^ (-1) = 1;
0 ^ (-1) = 0;
```

2.2 Functions

The `def` construct allows one to define, not just constants, but functions as well.

```
def square x = x ^ 2;

square 4 = 16;
```

The name of the function is denoted by the first token after `def`. Function application is denoted by juxtaposition, a la some functional languages such as Haskell.

During compilation, the type checker will identify `square`'s type as a function from integers to integers.

```
> square[3]: (int -> int)
```

Functions can have as many arguments as you want.

```
def f x y z = x * z + y;

f 4 5 6 = 29;
```

The type of this function indicates multiple arguments.

```
> f[5]: (int -> (int -> (int -> int)))
```

This tells us that each argument will take in an integer and produce a function, until the third argument, after which it will produce an integer. This indicates that application is left-associative and functions are curried by default (more on this in section 2.4).

```
((f 4) 5) 6 = f 4 5 6;
```

Definitions can contain definitions inside of them.

```
def g x = {
  def k = 20;
  x * k
};

g 3 = 60;
```

Notice the usage of curly braces to define blocks that scope the function. Also, notice that the semicolon for the definition comes after the closing curly brace, and there is no semicolon for the last line before that brace.

Internal `def` statements should be used similarly to `let` statements common in many functional languages. Internal `def` statements can be functions as well.

```
def cube x = {
  def square x = x * x;
  x * square x
};

cube 3 = 27;
```

Types for internal `def` statements are not given during compilation, but they are still checked. There is no limit to the number of lines or nestings within a definition.

```
def power x = {
  def hypercube x = {
    def square x = x * x;
    square (square x)
  };
  def cube x = {
    def square x = x * x;
    x * square x
  };
  cube (hypercube x)
 };

power 2 = 4096;
```

Overlapping names are allowed. References to names will refer to the most recent binding.

```
def x = 4;
def x = 8;
x = 8;
```

During type checking, both variables are listed.

```
> x[2]: int
> x[3]: int
```

The numbers next to the names are, in some sense, the "true" names of the variables. Each variable has a unique number identifying it, allowing the system to avoid confusing variables with the same name.

Equations can also appear in definitions. Definitions may or may not return a value and can act as gates for other values.

```
def g1 x = {x = 4; x};
g1 4;

def g2 x = {x = 10};
g2 10;
```

Looking at the types, one sees something interesting.

```
> g2[5]: (int -> ())
```

`g2` does, in fact, return a value; something of type `()`. This will be explained in section 2.3.

This illustrates that a VampIR file does not need any top-level equations, but will still represent a proposition through the equations enforced by function calls. `g1 5;` will produce an invalid proof, in this case.

Equations aren't enforced if their parent function is never actually called.

```
def j x = {0 = 1; x};

1 = 1;
```

will produce a valid proof because 'j' is never fully instantiated. Equations become part of the circuit when their parent function is fully instantiated. `def k = {0 = 1; 4};` **will** produce an invalid proof since 'k' is already fully instantiated.

Using polynomial constraints, one can create sophisticated, reusable checks. The following, for example, checks that a field element is a boolean; either 0 or 1.

```
def isBool x = {
  (x - 1) * x = 0;
  x
};

isBool 0;
isBool 1;
```

There are a handful of additional arithmetic operators which may be used so long as the circuit can be transformed so they aren't referenced. These operators will be detailed in full in section 3.2. For now, the modulus, `%`, will be used as an example.

```
6 = 15 % 9;
```

will create a valid proof. This is because `15 % 9` can be immediately simplified into `6` during circuit generation. If one attempts to use an uninstantiated variable, that is one without a value defined in the file, they will get an error.

```
6 = x % 9;
```

will generate an error during circuit creation informing us that the modulus is not a supported constraint.

2.3 Tuples

The only first-order data structure provided in VampIR is the tuple. One can use them in place of numbers to check multiple things in parallel.

```
def xs = (1, 2);

xs = (1, 2);
```

During type checking, `xs`'s type is given as

```
> xs[2]: (int, int)
```

This indicates that `xs` is a pair of integers. One can also iterate tuples as much as they want.

```
def xs = (1, 2, 3, 4, 5, 6, 7, 8, 9);

xs = (1, 2, 3, 4, 5, 6, 7, 8, 9);
```

During type checking, one can observe the type

```
> xs[2]: (int, (int, (int, (int, (int, (int, (int, (int, int)))))))))
```

This indicates that tuples with more than two entries are, internally, just nested pairs associated to the right.

```
(1, 2, 3) = (1, (2, 3));
```

`(1, 2, 3) = ((1, 2), 3);` will produce a type error. One cannot compare tuples to numbers, so `1 = (1, 1);` will produce an error. One also cannot treat tuples as numbers, so, for example, `(1, 2) + (3, 4);` will produce an error.

VampIR provides the ability to define functions over tuples using pattern matching.

```
def add (a, b) (x, y) = (a + x, b + y);

add (1, 2) (3, 4) = (4, 6);
```

Many standard tuple construction and manipulation functions can be defined.

```
def fst (x, y) = x;
def snd (x, y) = y;
def third (x, y, z) = z;
def dup x = (x, x);
def swap (x, y) = (y, x);
def assoc (x, (y, z)) = ((x, y), z);
```

As you can see by `assoc`, one can pattern match a tuple within another tuple. Additionally, the fact that n-tuples are nested pairs means that `fst` acts uniformly on tuples; using that definition we'd have `fst (1, 2, 3) = 1`.

The types here tell us something interesting. One sees, for example, that;

```
> fst[4]: (([24], [25]) -> [24])
> dup[13]: ([43] -> ([43], [43]))
> swap[16]: (([50], [51]) -> ([51], [50]))
```

The numbers in square braces denote variable types. These functions do not have a fixed type, but their types can be instantiated to be different types, depending on the context. Different calls within the same program can also have different types. For example, one can have both of these equations in the same file:

```
fst (1, 2) = 1;
fst ((1, 2), 3) = (1, 2);
```

Even though the same function is called on both lines, different types are returned each time. On the first line, it returns an integer while on the second it returns a pair.

VampIR also provides a 0-tuple denoted `()`, analogous to the return value of a unit type in many functional languages or a void type in many imperative languages. It's the closest thing to getting a function to return nothing.

```
def tt = ();
def f x = {
  x = 2;
  ()
};

f 2 = tt;
```

During type inference, the following is shown;

```
 > tt[2]: ()
 > f[4]: (int -> ())
```

indicating that `tt` is an element of the unit type and that `f` will return an element of the unit type on an integer input. This shows that functions that have only equations do the same thing as functions that explicitly return a `()`, as mentioned in section 2.2.

Using units, one can design some of our functions to act like list manipulation programs. That is if one thinks of `(x, y)` as `Cons x y` and `()` as `Nil`, they can encode a list like '[1, 2, 3, 4]' as `(1, 2, 3, 4, ())`. The utility of this lies in the fact that each entry will appear in the head of a pair. This allows us to define more flexible programs. Modifying a few of our previous examples, one can have

```
def swap (x, y, r) = (y, x, r);
swap (1, 2, ()) = (2, 1, ());
swap (1, 2, 3, ()) = (2, 1, 3, ());
swap (1, 2, 3, 4, ()) = (2, 1, 3, 4, ());

def tail (x, y) = y;
tail (1, 2, ()) = (2, ());
tail (1, 2, 3, ()) = (2, 3, ());
tail (1, 2, 3, 4, ()) = (2, 3, 4, ());

def total (x, y, r) = x+y;
total (1, 2, ()) = 3;
total (1, 2, 3, ()) = 3;
total (1, 2, 3, 4, ()) = 3;

def add (a, b, q) (x, y, r) = (a + x, b + y, ());
add (1, 2, ()) (3, 4, ()) = (4, 6, ());
add (1, 2, 3, 4, 5, ()) (6, 7, 8, ()) = (7, 9, ());

def zip (a, b, q) (x, y, r) = ((a, x), (b, y), ());
zip (1, 2, 3, 4, 5, ()) (6, 7, 8, ()) = ((1, 6), (2, 7), ());
```

VampIR ultimately generates a fixed, finite data structure, meaning that there can't be a proper variable-length list type. As such, many list manipulation programs that one might take for granted can't be defined in full generality in VampIR. However, this technique can recover some of that functionality in practice.

Witness solicitation interacts in a somewhat interesting way with pairs. If one compiles the program

```
x = (1, 2);
```

it seems like they will need to input a pair. However, during proof creation, VampIR will ask;

```
> * Soliciting circuit witnesses...
> ** x.0[9] (private):
> ** x.1[10] (private):
```

It splits the variable into two sub-variables, named `x.0` (the first element of `x`) and `x.1` (the second element of `x`). So the compiled circuit does not, in fact, have a hole in the shape of a pair; rather it has two holes corresponding to the elements of the pair.

Without a variable-free first-order datatype for each uninstantiated variable, circuit generation is not possible. If one tries compiling `x = y;` without definitions for x or y, they will get an error indicating such.

2.4 Higher-Order Functions

VampIR has rather robust support for higher-order functions. One can manipulate functions more or less like any other value. One can save a partially instantiated function.

```
def f x y = x * y;
def g = f 2;

g 3 = 6;
```

Notice that `g` is defined in terms of `f` with only one of its two arguments applied. The types are given as

```
> f[4]: (int -> (int -> int))
> g[5]: (int -> int)
```

one can see that `g`'s type is exactly the return type of `f`. This is no different than if `f` returned an integer instead of a function.

One can also write functions that take a function as input.

```
def app2 f x = f (f x);
def times2 x = 2 * x;

app2 times2 3 = 12;
```

Looking at the types;

```
> f[4]: app2[4]: (([10] -> [10]) -> ([10] -> [10]))
```

one can see that `app2` takes a function from the (variable) type `[10]` onto itself, and creates a new function, also from `[10]` onto itself.

As mentioned in section 2.2, the constraints of a function are not called unless the function is fully instantiated. The following program produces a valid proof.

```
def f x y = {0 = 1; x};

f 2;
```

While `f` is called, it's not fully instantiated, and so `0 = 1` does not become part of the generated circuit.

There are many higher-order combinators that may be useful for some applications.

```
def comp f g x = f (g x);
def const x y = x;
def flip f x y = f y x;
def delta f x = f x x;
```

VampIR provides a notation for anonymous functions using the `fun` keyword.

```
(fun x {x}) 15 = 15;
(fun x y {x + y}) 15 20 = 35;
(fun (x, y) {x + y}) (15, 3) = 18;
flip (fun x y {x - y}) 15 3 = (-12);
```

Notice that one can bind multiple arguments and pattern match on tuples using anonymous functions. These can also be used as return values in definitions.

```
def id = fun x {x};

id 2 = 2;
```

one can also place constraints inside anonymous functions.

```
(fun x {x = 2; x}) 2;
(fun x {x = 5}) 5 = ();
```

as you can see, anonymous functions don't need to return data. Like `def`s, `fun`s that don't return a value still return a `()`.

We cannot use functions in comparisons; only first-order data can be compared.

```
(fun x { 2 }) = (fun x { 1 + 1 });
```

will produce an error.

One can also create functions to curry and uncurry our functions

```
def curry f x y = f (x, y);
def uncurry f (x, y) = f x y;

curry (fun (x, y) {x + y}) 3 4 = uncurry (fun x y {x + y}) (3, 4);
```

More generic functions can be defined over tuples;

```
def map f (x, y, z) = (f x, f y, f z);
def zipWith f (x, y, z) (a, b, c) = (f x a, f y b, f z c);
def fold f d (x, y, z) = f x (f y (f z d));

fold (fun x y {x + y}) 0
     (zipWith (fun x y {x - y})
              (map (fun x {3*x}) (1, 2, 3))
              (4, 5, 6)) = 3;
```

VampIR does not give unrestricted access to higher-order functions; any appearing in a program must be simply typable. For example, `fun x { x x };` will produce a typing error. This means that, in particular, one cannot define a fixed-point combinator, which would be necessary to implement general recursion. Still, there are many powerful techniques made available by what is given. In particular, one can define and utilize lambda-encoded data.

If VampIR didn't already have built-in pairs, one could implement them as

```
def pair x y = fun p { p x y };
```

By issuing different arguments to `pair x y`, one can get the underlying components.

```
def fst p = p (fun x y {x});
fst (pair 1 2) = 1;

def snd p = p (fun x y {y});
snd (pair 1 2) = 2;
```

While one cannot compare these pairs using = directly, one can define a bespoke comparison function.

```
def eq p q = {
  fst p = fst q;
  snd p = snd q
};

eq (pair 1 2) (pair 1 2);
```

Further structural manipulation functions can easily be defined.

```
def swap p = p (fun x y {pair y x});
eq (swap (pair 1 2)) (pair 2 1);
```

```
def assoc p =
  fun f {p (fun x yz {yz (fun y z {
      f (pair x y) z
  })})};
eq (fst (assoc (pair 1 (pair 2 3)))) (pair 1 2);
snd (assoc (pair 1 (pair 2 3))) = 3;
```

Functions converting between the two formats can also easily be defined;

```
def lpair (x, y) = pair x y;
def upair p = p (fun x y {(x, y)});
eq (lpair (1, 2)) (pair 1 2);
(1, 2) = upair (pair 1 2);
```

This is all interesting but doesn't give us anything that wasn't already provided. However, a variant of this idea can encode arbitrary length lists.

```
def nil = fun n c {n};
def cons x xs = fun n c {c x (xs n c)};

def ex_list = cons 1 (cons 2 (cons 3 nil));
```

The utility of this construction is that the length of the list is not encoded in the structure of the type, so it can be arbitrarily long. One can also issue arguments into the list, which represents folding a function over the list.

```
ex_list 0 (fun x y {x + y}) = 6;
```

One can define many well-founded recursive functions this way.

```
def append l1 l2 = fun n c { l1 (l2 n c) c };

def map f l n c = l n (fun x xs { c (f x) xs });
```

One cannot define generic translations between lists and built-in tuples, so a bespoke method must be made for each length. But only the translations need to be length-specific. The operations over lists can be generic.

```
def tup2list_2 (x1, x2) = fun n c {c x1 (c x2 n)};
def tup2list_3 (x1, x2, x3) = fun n c {c x1 (c x2 (c x3 n))};
def tup2list_4 (x1, x2, x3, x4) = fun n c {c x1 (c x2 (c x3 (c x4 n)))};

def replaceFst a (x, y) = (a, y);

def rotate_2 (x1, x2) = (x2, x1);
def rotate_3 (x1, x2, x3) = (x3, x1, x2);
def rotate_4 (x1, x2, x3, x4) = (x4, x1, x2, x3);

def list2tup_2 l = l (0, 0)
                     (fun x xs {replaceFst x (rotate_2 xs)});
def list2tup_3 l = l (0, 0, 0)
                     (fun x xs {replaceFst x (rotate_3 xs)});
def list2tup_4 l = l (0, 0, 0, 0)
                     (fun x xs {replaceFst x (rotate_4 xs)});

list2tup_2 (tup2list_2 (1, 2)) = (1, 2);
list2tup_3 (tup2list_3 (1, 2, 3)) = (1, 2, 3);
list2tup_4 (tup2list_4 (1, 2, 3, 4)) = (1, 2, 3, 4);

def sum l = l 0 (fun x y {x + y});
sum (tup2list_2 (1, 2)) = 3;
sum (tup2list_3 (1, 2, 3)) = 6;
sum (tup2list_4 (1, 2, 3, 4)) = 10;
```

Other variants of this construction could be made. Any kind of well-founded tree can be encoded and folded over via similar methods. This opens the door to many powerful functional programming techniques, including free monads, for example.

It should be noted that there are some fundamental limitations to this method. Mainly, it can't interact non-trivially with the underlying field. One cannot, for example, translate a field element into a Church-encoded number or a boolean. While one can certainly implement those in VampIR (in fact, `app2` from earlier is a Church numeral, and `const` is a Church boolean), they cannot be used to actually reason about field elements. The methods described in this section are exclusively useful for data **structure** manipulation, but not data manipulation in general. This is because data structures are fixed at circuit generation time, but field values are not completely fixed until proof generation, which occurs after circuit generation. This implies that the circuit's structure cannot depend on field values.

3 Witness Generation

3.1 The fresh keyword

It's not always practical to calculate values purely using the operations over a finite field. As such, VampIR adds an additional construct for calculating values, called `fresh`.

```
fresh (22) = 22;
```

`fresh` must be followed by a syntactically valid expression surrounded by parentheses. This expression can call all arithmetic operators described in section 2.1, along with additional operators not native to arithmetic circuits which will be detailed in the upcoming sections.

`fresh` can make unrestricted calls to variables within the environment, and it is often used to calculate values involving them.

```
fresh (x / y) = 2;
```

Anything called inside of fresh is essentially a black box to the circuit. The circuit, in this case, does not verify that the result is actually the result of a quotient; rather, it's just a witness value, like any provided by the end user during witness solicitation. `fresh` allows us to calculate such witnesses automatically instead of them being provided by the end user. This means that properties of fresh witnesses need to be enforced by separate constraints in order to prove anything; see section 3.4 for details.

The fact that `fresh` acts as a witness generator can be observed with the following code;

```
6 = fresh (15) % 9;
```

Similar to the situation described in section 2.2, an error declaring that `%` is an unsupported constraint is generated. `fresh (15)` does not get turned into a number; it gets turned into a hole for a witness which is then automatically filled with `15` during proof generation. However,

```
6 = fresh (x % 9);
```

will produce a valid circuit, even if `x` is a free variable which will be filled with a solicited witness.

The syntactic requirements for `fresh` can lead to awkward expressions. Since negation requires its own surrounding parentheses, and fresh requires its own parentheses, `fresh (-1)` will produce a syntax error; one must write `fresh ((-1))` instead.

[Note: It seems to me that the restriction of exponentiation, that it cannot have variables in its exponent, should be lifted inside of fresh. Currently, it is not.]

`fresh` will generally not interact well with higher-order functions or tuples.

```
def id x = x;
def fid = fresh (id);

6 = fid 6;
```

will produce a cryptic error informing us that the application at `fid 6` failed; although it seems to have accepted making a "fresh" version of `id`. We get a similar error from

```
(1, 6) = fresh ((1, 6));
```

Generally, `fresh` should only be used to generate field elements, not data structures. So long as the return type is `int`, `fresh` can use any available functions and capabilities.

```
7 = fresh ((fun (x, y) {x + y}) (1, 6));
```

works just fine, for example.

3.2 Expanded Arithmetic

In addition to the main field operations, there are three more operations available.

\ calculates integer division, rounded toward 0.

```
(3 \ 2) = 1;
(5 \ 2) = 2;
(233 \ 55) = 4;
```

Dividing with negatives can seem to behave strangely, as integers are converted into field elements. Generally, one will observe that `((-k) \ n) = ((-1) \ n) - (k \ n)`.

```
((-3) \ 2) = ((-1) \ 2) - 1;
((-5) \ 2) = ((-1) \ 2) - 2;
((-233) \ 55) = ((-1) \ 55) - 4;
```

and dividing by a negative will usually end up as 0 since it gets interpreted as dividing by a very large number.

```
(2 \ (-1)) = 0;
(2234 \ (-100)) = 0;
(5555555 \ (-2)) = 0;
```

`%` calculates the modulus.

```
(3 % 2) = 1;
(5 % 2) = 2;
(233 % 55) = 4;
```

The modulus has similar strange behaviour when interacting with negatives. Taking the modulus of a negative number can vary based on the underlying field size, as `(-n) % k` will be the same as `p - n % k`, where `p` is the field size. For a prime field of size
52435875175126190479447740508185965837690552500527637822603658699938581184513
we'd have

```
(-6) % 5 = 2;
(-233) % 55 = 10;
```

taking the modulus by a negative will usually end up doing nothing as it's interpreted as taking the modulo by a very large number.

```
(2 % (-1)) = 2;
(2234 % (-100)) = 2234;
(5555555 % (-2)) = 5555555;
```

`|` calculates ordinary division in the underlying field, however, it doesn't generate an error when dividing by 0.

```
15|3 = 15/3;
15|0 = 0;
```

3.3 Conditionals

[TODO]

3.4 Gating Witnesses

By default, any circuit we generate will not know anything about fresh witnesses. They are just numbers, unless we add additional constraints to give them meaning. Consider this function which indicates if a number is not zero.

```
def isntZero x = fresh (1 | x) * x;

isntZero ((-1)) = 1;
isntZero 0 = 0;
isntZero 1 = 1;
isntZero 2 = 1;
```

While this generates a valid proof, what is it proving? Essentially just that we know of inverses for -1, 1, and 2, as well as a number which is 0 when multiplied by 0. Since the correctness of `isntZero` isn't actually checked, we haven't actually proved that any of these numbers are or aren't 0, even though we've calculated the answer correctly. We can fix this by adding additional constraints.

```
def isntZero x = {
  def xi = fresh (1 | x);
  x * (1 - xi * x) = 0;
  xi * x
};
```

That new constraint checks that either `x = 0` or `xi * x = 1`. It's worth thinking carefully about how this constraint is effecting the output of the function. If the input is non-zero, we can calculate its inverse. By definition, this inverse will be 1 when multiplied by the input. The inverse cannot exist if the input is zero, so checking that `xi * x = 1` guarantees that `x` is not zero. If `x` is zero, the constraint becomes trivial. This means the value of `xi` is now unconstrained. However, `xi`, being multiplied by `x` in the output, will not be able to effect the output's value. Ultimately, the constraint forces the output to be unique. If we calculated witnesses incorrectly, then the constraint would fail and the proof would be invalid.

This sort of thinking is at the heart of any meaningful usage of `fresh`. Whenever it's used, we should think of the circuit as defining a relation whose validity ranges over every possible value. By putting constraints on the possible values of a fresh witness, we are narrowing that relation, making the statement of the circuit more specific.

4 Proof Generation and Back-ends

4.1 Choosing a Back-end

A back-end is automatically chosen based on the extensions of the generated files. If we want to generate a Halo2 circuit, we can simply ask it to make a `.halo2` file. Repeating the examples from the introduction, we could have done the following to get a Halo2 circuit.

```
$ target/debug/vamp-ir compile -u examples/params.pp \
                               -s examples/ex1.pir \
                               -o examples/circuit.halo2

> [...]
> * Constraint compilation success!

$ target/debug/vamp-ir prove -u examples/params.pp \
                             -c examples/circuit.halo2 \
                             -o examples/proof.halo2

> [...]
> * Proof generation success!

$ target/debug/vamp-ir verify -u examples/params.pp \
                              -c examples/circuit.halo2 \
                              -p examples/proof.halo2

> [...]
> * Zero-knowledge proof is valid
```

Currently, VampIR supports the following proof systems;

- PLONK via `.plonk`
- Halo2 via `.halo2`
- [TODO]

[TODO: What particularities are there with the different proof systems?]
[TODO: Contoling proofs, e.g. with the -m command.]

5 Appendix A: Cookbook

5.1 Polynomial Logic

The most basic technique for creating propositions in arithmetic circuits is to manipulate polynomials so they have specific roots. The most basic equations are simply checking that a number is a constant, such as;

```
x = 5;
```

However, what if one wanted to say that something is either 5 *or* 6? One can reformulate each as `(x - 5) = 0` and `(x - 6) = 0`. These polynomials can then be multiplied into;

```
(x - 5) * (x - 6) = 0;
```

creating our disjunctive constraint. Similarly, if one have two polynomials, `p` and `q` which must both be equal to `0`, one can form the constraint

```
p^2 + q^2 = 0;
```

The reason one needs to square the values is so that `p`'s (possibly negative) value doesn't cancel out `q`'s (also possibly negative) value. There's also the caveat that squaring and adding very large values could cause those values to exceed the size of the underlying field, so a range check or some other guarantee may need to be made for this to be valid. In most cases, anyone is better off simply stating `p = 0` and `q = 0` as separate constraints.

This basic idea where one treats `0` as denoting truth and use any non-zero value to denote falsity can be extended to a variety of logical operations. Contrast this with the usual presentation of boolean logic, where `0` if false and `1` is true.

5.2 Boolean Logic

One can easily check that a value is a boolean by checking if it's either `0` or `1`.

```
def isBool x = (x * (1 - x) = 0);
```

If one already knows a value is a boolean, they can perform all the standard boolean operations;

```
def negb x = 1 - x;
def andb x y = x * y;
def orb x y = negb (andb (negb x) (negb y));
```

Possibly the most basic predicate over numbers is equality. While this can be checked with `=`, one may need to represent the result with a boolean value instead for later complication. One can test if a number is `0` with the following modification of the program appearing in section 3.4.

```
def isZero x = {
  def xi = fresh (1 | x);
  x * (1 - xi * x) = 0;
  1 - xi * x
};
```

One can then define an equality predicate as

```
def equal x y = isZero (x - y);
```

Through a slight modification, inequality can also be defined.

5.3 Range Checks and Bit Decomposition

One of the most common, and also heaviest, operations necessary for the execution of regular programs is bit decomposition, where a number is decomposed into the binary bits making it up. One of the most common operations necessary for basic arithmetic is checking that a number is within a certain range. Both of these operations are typically done simultaneously. If a number can be decomposed into 8 bits, then it must be in the range $[0, 2^8)$.

To get the nth bit of a number, one merely needs to integer-divide by 2^{n-1}, then mod by 2. Repeating this for every bit one cares about will get us the full decomposition.

```
  def decomp8 x = {
    def x0 = fresh ((x\2^0) % 2); isBool x0;
    def x1 = fresh ((x\2^1) % 2); isBool x1;
    def x2 = fresh ((x\2^2) % 2); isBool x2;
    def x3 = fresh ((x\2^3) % 2); isBool x3;
    def x4 = fresh ((x\2^4) % 2); isBool x4;
    def x5 = fresh ((x\2^5) % 2); isBool x5;
    def x6 = fresh ((x\2^6) % 2); isBool x6;
    def x7 = fresh ((x\2^7) % 2); isBool x7;
    x = x0 + 2*x1 + 2^2*x2 + 2^3*x3 + 2^4*x4 + 2^5*x5 + 2^6*x6 + 2^7*x7;
    (x0, x1, x2, x3, x4, x5, x6, x7)
};
```

Notice that each bit is actually checked to ensure it's a boolean. Additionally, we check that the decomposition is correct prior to returning the tuple. This implementation returns bits in little-endian format, though this can easily be changed to suit the needs of the user.

By modifying the input number prior to checking the range, the range can be modified. For example, the following checks that something is in the range $[-2^7, 2^7)$

```
def intDecomp8 x = {
  decomp8 (x + 2^7)
};
```

This will return what is *almost* the two's complement representation. The main difference is that the last bit, indicating the sign, is the opposite of what it is in usual presentations of two's complement. With it, the sign of a number can be checked.

```
def isNegative8 a = {
  def (a0, a1, a2, a3, a4, a5, a6, a7) = intDecomp8 a;
  1 - a7
};
```

and one can further check if one number is less than another. The following will act as a valid $<$ indicator, so long as both `x` and `y` are in the range $[-2^6, 2^6)$.

```
def less8 a b = isNegative8 (a - b);
```

Using these range checks, an arbitrary range whose width is less than the original can be checked via overlapping checks. The following will check that a number, `x`, is within $[a, b)$, so long as $b > a$ and $b - a < 2^8$.

```
def range8 x a b = {
    decomp8 (x - a); // check x in [a, 2^8 + a)
    decomp8 (x + 2^8 - b); // check x in [b - 2^8, b)
};
```

References