Rethinking VampIR

Anthony Hart[a]

[a]Heliax AG

* *E-Mail: anthony@heliax.dev*

Abstract

This paper provides an overview of VampIR v0.1.3 and outlines potential modifications for future versions. Specifically, it contains proposals for streamlined versions of the fresh function and constraint generation, a proposal for removing the type system, and a short proposal for dealing with imports. It ends with an overview of implementation improvements made in light of the Haskell port.

Keywords: VampIR; Arithmetic Circuits; Compilers; Proposals;

(Received July 31, 2023; Published: August 21, 2023)

1. Current VampIR

If we were to retroactively turn VampIR into a calculus, we could describe its syntax as follows.

$$
\begin{aligned}
\text{VampIR} ::=\ & \text{Arith} \\
\mid\ & \text{VampIR} \ = \ \text{VampIR} \\
\mid\ & \text{VampIR};\ \text{VampIR} \\
\mid\ & () \\
\mid\ & \text{VampIR},\ \text{VampIR} \\
\mid\ & \text{fst} \mid \text{snd} \\
\mid\ & \text{VampIR} :: \text{VampIR} \mid [] \\
\mid\ & \text{hd} \mid \text{tl} \\
\mid\ & \text{fresh VampIR}^{+} \\
\mid\ & \text{iter } n \text{ VampIR} \\
\mid\ & \text{fold VampIR VampIR VampIR} \\
\mid\ & \lambda v.\text{VampIR} \\
\mid\ & \text{VampIR VampIR} \\
\mid\ & \text{VampIR op VampIR} \mid \ -\text{VampIR} \mid f \mid v \\
\text{op} ::=\ & + \mid \times \mid - \mid / \\
f \in\ & \mathbb{F} \\
\text{v} \in\ & \text{string}
\end{aligned}
$$

Here $\mathbb{F}$ is the field associated with the circuit. VampIR^{+} is the same as VampIR, except op can also be $\%$ or $\div$, which represent the non-field operations of modulus and integer division. Also note that the real VampIR lacks built-in fst, snd, hd, and tl operations. Instead, such operations are implemented via pattern matching, however, since pattern matching can only be done on tuples and lists, this ends up being equivalent to what I present here.

VampIR has a simple type system. The syntax of types can be defined as;

$$
\begin{aligned}
\text{Type} ::=\ & v \\
\mid\ & \text{Type} \ \to \ \text{Type} \\
\mid\ & \text{Type} \ \times \ \text{Type} \\
\mid\ & [\text{Type}] \\
\mid\ & () \\
\mid\ & \mathbb{F}
\end{aligned}
$$

VampIR relies on a notion of first-order type for some determinations. The set of first-order types can be defined inductively as follows;

$$1st := () + \mathbb{F} + [1st] + 1st \times 1st$$

The type-checking rules of VampIR are;

$$\frac{x:T \quad y:U}{(x;\ y):U} \qquad \frac{f\in\mathbb{F}}{f:\mathbb{F}} \qquad \frac{op\in\{+,*,-,/,\div,\%\} \quad x:\mathbb{F} \quad y:\mathbb{F}}{(x\ op\ y):\mathbb{F}} \qquad \frac{T\in 1st \quad x:T \quad y:T}{(x=y):()}$$

$$\frac{}{():()} \qquad \frac{x:T \quad y:U}{(x,y):T\times U} \qquad \frac{}{\mathsf{fst}: T\times U\to T} \qquad \frac{}{\mathsf{snd}: T\times U\to U} \qquad \frac{x:T\vdash y:U}{\lambda x.y:T\to U} \qquad \frac{x:T\to U \quad y:T}{(x\ y):U}$$

$$\frac{}{[]:[T]} \qquad \frac{x:T \quad l:[T]}{(x::l):[T]} \qquad \frac{}{\mathsf{hd}:[T]\to T} \qquad \frac{}{\mathsf{tl}:[T]\to[T]} \qquad \frac{}{\mathsf{fold}: U\to(T\to U\to U)\to[T]\to U}$$

$$\frac{n\in\mathbb{N}}{\mathsf{iter}\ n: T\to(T\to T)\to T}$$

The type-checking rules prevent many ill-formed terms, but they do not prevent something like hd[], which will give an error during evaluation.

The types of free variables can vary, but each must have a unique type. VampIR requires that the type of a variable be inferable from context. For example, $x = 1$ will allow one to infer that $x : \mathbb{F}$. Such variables will always represent (tuples or lists of) public and/or private variables. Variables representing tuples will be split. For example, x, if it were known to be a tuple, will be split into $(x.1, x.2)$, where $x.1$ and $x.2$ are new variable names. Splitting will continue until all variables are field elements. A free variable cannot be a function, it must be in a first-order type.

Evaluation is fairly simple, but it's designed to have side effects making it look less intuitive than it might otherwise. While evaluating, there's a globally available stack of constraints, which I'll denote χ. The correctness of evaluation requires the term to be well-typed. Missing patterns from evaluation should, mostly, be eliminated by type checking.

A top-level VampIR program should be of type (). Given that restriction, the following defines the set of constraints represented by the program;

$$\begin{aligned}
[[()]] &= () \\
[[\mathsf{iter}\ (n+1)\ f\ x]] &= [[f\ (\mathsf{iter}\ n\ f\ x)]] \\
[[(x;y)]] &= [[x]];[[y]] \\
[[(x,y)]] &= ([[x]],[[y]]) \\
[[\mathsf{fst}\ (x,\ y)]] &= [[x]] \\
[[\mathsf{snd}\ (x,\ y)]] &= [[y]] \\
[[\mathsf{iter}\ 0\ f\ x]] &= [[x]] \\
[[(x::y)]] &= ([[x]]::[[y]]) \\
[[\mathsf{hd}\ (x::l)]] &= [[x]] \\
[[\mathsf{tl}\ (x::l)]] &= [[l]] \\
[[\ \mathsf{fold}\ n\ f\ []\ ]] &= [[n]] \\
[[\mathsf{fold}\ n\ f\ (x::l)]] &= [[n\ x\ (\mathsf{fold}\ n\ f\ l)]] \\
[[(x1,\ x2)=(y1,\ y2)]] &= [[x1=y1]];[[x2=y2]] \\
[[(x1::\ x2)=(y1::\ y2)]] &= [[x1=y1]];[[x2=y2]] \\
[[x=y]] &= [[\ [[x]]=[[y]]\ ]], \text{where } x,y:1st, \text{but not } x,y:\mathbb{F} \\
[[x=y]] &= (), [[x]]=[[y]] \text{ gets pushed onto } \chi,\ \text{where } x,y:\mathbb{F} \\
[[(\lambda x.f)y]] &= [[\ f[y/x]\ ]] \\
[[x\ op\ y]] &= r, \text{where } x,y\in\mathbb{F}, \text{and } r \text{ is the obvious result} \\
[[\mathsf{fresh}\ x]] &= v, \text{where } v \text{ is a fresh variable}
\end{aligned}$$

Some things never need to be evaluated; for example, an unapplied lambda expression on its own should never occur during the evaluation of a well-formed program, so it does not need to be considered. Any unconsidered pattern should produce an error, though most such forms are already eliminated by type checking.

The nature of evaluation makes semantics somewhat nonintuitive. As a program is evaluated, it will output more and more constraints. There isn't any control over the constraints themselves as they are not manipulable objects of the language. Through the ability to ignore function calls, it is possible to prevent constraints from propagating (e.g. any constraint produced by f in iter $0\ f\ x$), but it is not always obvious when this will happen. The overall meaning of a program will be that, for a fixed set of public variables, there exists an assignment to the private variables such that the conjunction of all the constraints in χ is true. Since χ is produced procedurally by side effects, the semantics is not immediately compositional. Since the order of evaluation does not matter (a consequence of the monoidal nature of conjunction), one may be able to recover a compositional semantics, but such semantics is not immediately apparent in the evaluation model.

2. Proposal: Remove the type system

The current type system in VampIR has led to limitations and the need for workarounds. While it is not inherently flawed, there is potential for a more versatile design. A key constraint is the inability to represent certain recursive programs due to the type check process. While solutions have been made to address specific instances, like lists and numbers, these are band-aid solutions that don't address the core issue: the limitations inherent in the type system itself.

If lists were removed from the language, they could still be implemented using lambda expressions. However, destructors such as the head and tail of the list cannot be iterated since such do not type check. Built-in lists and the accompanying hd, tl, and fold functions were added to compensate for this. A similar issue comes with numbers. Natural numbers could also be lambda encoded, but iterated predecessors don't type check, limiting their usage. iter was added to compensate; it just interprets its natural number argument as a church numeral; something that can already easily be implemented. This same issue would come up with any recursive type. Binary trees can be lambda encoded, and, in fact, one can fold over them as well already in VampIR. You can even take the left and right branches, but you can't iterate taking left and right branches because such wouldn't type check. Some version of this issue emerges with all recursive types. VampIR has no solution, currently. The way things have been done in the past suggests extending the language, perhaps with a full-fledged recursive type system. However, I would prefer a different solution: get rid of the type system.

The type system does not prevent exponentially sized circuits from occurring; just infinitely sized ones. We can easily deal with exponentially sized circuits by stopping evaluation early; I see no reason why this isn't a fine way to deal with infinite circuits on the occasion they occur. VampIR is, in particular, an intermediate language. This makes facilities to prevent the developer from shooting themselves in the foot less important than it might be. It's already the user's responsibility to make sure VampIR code is sensible; giving them the additional responsibility to make sure the code doesn't loop infinitely is trivial in comparison. The logic errors that the simple type system prevents are, perhaps, not all trivial, but they aren't substantial either. By the nature of a simple type system, the guarantees it gives aren't hard to keep track of by hand.

Using lambda encodings for all data structures would allow essentially all the faculties of functional programming to be made available; from data structures to complex mutually recursive algorithms. As it stands, the type system filters this into a less useful subset than end users might otherwise want to use.

3. Proposal: Remove DSL and make fresh simpler

The design of VampIR has an understated issue related to 'fresh'. During evaluation, the body of 'fresh' is discarded, but VampIR saves it as a method to determine the value of the newly generated variable. This design doesn't impact circuit semantics but serves as a convenience tool. Instead of users manually inputting values for private variables, 'fresh' provides automatic calculation instructions. To aid this, 'fresh' utilizes a DSL, essentially an extended version of VampIR with additional operations, like modulus, which aren't easily represented in arithmetic circuits. This enables 'fresh' to compute values, with subsequent constraints verifying the accuracy of these values.

So, what's the problem? The fact that fresh uses a manually created DSL means that users cannot use arbitrary programs to calculate values that need to be filled with fresh. This has led to much discussion for how or whether this language should be extended. It's not hard to imagine this discussion going on forever; adding new things whenever they come up. Alternatively, there's a path where we make the DSL computationally universal. But both of these wouldn't even necessarily work, as the correct value for a private variable may need to be supplied by an oracle that cannot be implemented in a VampIR program for whatever reason. This can be dealt with by declaring a private variable manually and submitting it at proof generation time. But this begs the question of why fresh needs to exist at all. If it's a reasonable feature of convenience, can it be made better?

My proposal is this: remove the DSL entirely and have fresh just store a name and signature. Take this example of a typical usage of fresh;

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

def range5 a = {
 def a0:a1:a2:a3:a4:[] =
   fresh (((a\1) % 2):((a\2) % 2):((a\4) % 2):((a\8) % 2) :((a\16) % 2):[]);
    def a0 = bool a0;
    def a1 = bool a1;
    def a2 = bool a2;
    def a3 = bool a3;
    def a4 = bool a4;
    a = a0 + 2*a1 + 4*a2 + 8*a3 + 16*a4;
    (a0, a1, a2, a3, a4, ())
};
```

Here, fresh is calculating a list of values, and those values are calculated using integer division and the modulus operation. My suggestion is that this be changed into something like the following;

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

def range5 a = {
    def a0:a1:a2:a3:a4:[] = fresh (range5_fr a);
    def a0 = bool a0;
    def a1 = bool a1;
    def a2 = bool a2;
    def a3 = bool a3;
    def a4 = bool a4;
    a = a0 + 2*a1 + 4*a2 + 8*a3 + 16*a4;
    (a0, a1, a2, a3, a4, ())
};
```

When this program is compiled, it will also come with a list of named signatures representing functions that need implementation. This program will contain a signature named range5_fr that takes one argument and produces a list of five elements. The length may or may not be a part of the signature, but the basic input and output types are. When proving a circuit, implementations for every function in the signature must be provided as well. Exactly what this will look like will depend on the language, but one can easily imagine implementing a function that takes a name and a list of arguments and calculates the output as needed for the circuit in just about any language. The prove function would essentially have an additional argument which is a partial function of type string * [int] -> [int]. Some languages with no support for higher-order functions will need something more complicated, but this shouldn't be a show-stopper in any case. This system would be less convenient for some examples than the current system, but it would allow full flexibility and allow some simplifications in the language. It would require having VampIR lean more towards being a library than it currently does, but that's not necessarily a bad thing.

4. Proposal: Explicit Constraints

The difficulty in VampIR's semantics stems from its non-compositional nature. To fix this, I propose adding a type of constraint to the language. Lambda expressions shouldn't generate constraints; rather, they should return sets of constraints. I propose the following modified language (note: this is leaving out things we may want to keep, but I'm ignoring them for simplicity).

$$
\begin{aligned}
\text{VampIR} ::= {} & \text{Arith} \\
\mid {} & \text{VampIR} \ = \ \text{VampIR} \\
\mid {} & \text{VampIR} \wedge \text{VampIR} \\
\mid {} & \text{Type} \ \times \ \text{Type} \\
\mid {} & \lambda v.\text{VampIR} \\
\mid {} & \text{VampIR VampIR} \\
\mid {} & \text{VampIR op VampIR} \mid \ - \text{VampIR} \mid f \mid v \\
\text{op} = {} & + \mid \times \mid - \mid / \\
f \in {} & \mathbb{F} \\
\text{v} \in {} & \text{string}
\end{aligned}
$$

with the type system

$$
\begin{aligned}
\text{Type} ::= {} & v \\
\mid {} & \text{Type} \ \rightarrow \ \text{Type} \\
\mid {} & \text{Type} \ \times \ \text{Type} \\
\mid {} & \mathbb{B} \\
\mid {} & \mathbb{F}
\end{aligned}
$$

and type-checking rules

$$
\frac{f \in \mathbb{F}}{f : \mathbb{F}} \quad
\frac{op \in \{+, *, -, /, \div, \%\} \quad x : \mathbb{F} \quad y : \mathbb{F}}{(x \ op \ y) : \mathbb{F}} \quad
\frac{T \in 1st \quad x : T \quad y : T}{(x = y) : \mathbb{B}} \quad
\frac{x : \mathbb{B} \quad y : \mathbb{B}}{x \wedge y : \mathbb{B}}
$$

$$
\frac{x : T \quad y : U}{(x, y) : T \times U} \quad
\frac{}{\mathsf{fst} : T \times U \rightarrow T} \quad
\frac{}{\mathsf{snd} : T \times U \rightarrow U} \quad
\frac{x : T \vdash y : U}{\lambda x.y : T \rightarrow U} \quad
\frac{x : T \rightarrow U \quad y : T}{(x \ y) : U}
$$

With these in place, we can give a sensible interpretation to any program of type $\mathbb{B}$; it will be, essentially, a set of constraints.

$$\begin{aligned}[[x = y]] &= \{x = y\} \\ [[x \wedge y]] &= [[x]] \cup [[y]] \\ [[(x1, x2) = (y1, y2)]] &= [[x1 = y1]] \cup [[x2 = y2]] \\ &\ldots\end{aligned}$$

By having functions return an explicit set of constraints, we don't need to worry about procedural order, etc. This will make function types more complicated, but it will also make them more intuitive. For example

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

In current VampIR this function has type $\mathbb{F} \to \mathbb{F}$, despite the fact that it has the side effect of generating a constraint. In my proposed change, this would become;

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

and have type $\mathbb{F} \to \mathbb{B} \times \mathbb{F}$. This changes later usage; an example of a range check would become;

```
def range5 a = {
    def a0:a1:a2:a3:a4:[] = [...];
    def (c0, a0) = bool a0;
    def (c1, a1) = bool a1;
    def (c2, a2) = bool a2;
    def (c3, a3) = bool a3;
    def (c4, a4) = bool a4;
    def c5 = (a = a0 + 2*a1 + 4*a2 + 8*a3 + 16*a4);
    (c0 & c1 & c2 & c3 & c4 & c5
    , (a0, a1, a2, a3, a4, ()) )
};
```

There are alternatives to this. This typical pattern is an instance of a state monad; it may be better to implement state monad combinators and assume all functions are under a (possibly trivial) constraint set. Function composition would union the produced states together, etc. This would be just as compositional, but it would be, perhaps, less tedious than what I presented here. We can, however, implement such combinators in VampIR already, so that may be unnecessary.

There are additional potential benefits to what I presented here. For many functions, it's not always obvious when one should do prefix constraint checks. In the case of bit-vector computations, if one doesn't know if the inputs are actually constrained to 0 and 1 then one needs to check this. If we already know this (maybe we already checked this with an earlier operation, and the output of this op is guaranteed to still be a bitvector by construction) then we don't need to add prefix checks. In the current version of VampIR this is handled by having multiple versions of the same function, one with checks and the other without. This modification allows one to ignore generated constraints, reducing the need for redundancy.

5. Proposal VampIR Imports

As it stands, VampIR does not currently support any kind of import system. This stems, largely, from the fact that simply combining two files will force the top-level constraints of both files to be combined. Despite this, most VampIR files consist mainly of reusable functions. VampIR should be modified to allow for imports. This would allow much better code reuse, allowing several files to, for example, import range checks rather than forcing each to re-implement them

There are several ways this could be done. The simplest would be to simply ignore all top-level constraints for imports. We could also make a second file format that specifically only has definitions and no top-level constraints.

My preferred approach is this: There should be two kinds of VampIR files; one that has top-level constraints and another that contains only definitions. These should not be allowed to mix and match. This would prevent spurious constraints and would encourage programmers to abstract functionality into reusable definitions.

6. VampIR Haskell

Recently, a port of VampIR from Rust into Haskell was started. As part of this, many important implementation differences have been made. This section will briefly summarise them.

In the original implementation of VampIR, parsing was done through a PEG grammar. There was a .pest file which was then used to define a datatype in which parsed modules were stored. The Haskell implementation uses parser combinators as provided by the megaparsec library. This significantly reduces the parsing code from over a thousand lines in Rust to less than 200 in Haskell. This is far more maintainable. In the original implementation of VampIR, there were periodic stack overflows when certain files were parsed; it's not clear at this point if the parser combinators will be more efficient in this respect.

The original version of VampIR parsed everything into the same format used for computation. The Haskell implementation has a separation between a simpler "core" language and a more complicated type representing VampIR ASTs. This allows things like pattern matching to be removed prior to evaluation or type checking. This also allows the core expressions to use higher-order abstract syntax, allowing us to reuse Haskell's binding features instead of implementing them anew.

In the original version of VampIR, all expression constructors were wrapped in a type annotation. These annotations were blank by default, and only filled during type checking. After type checking, these annotations were removed, but the wrappers themselves were always present. This made the language data types more complicated than necessary, and type annotations themselves are simply not necessary since VampIR's type system is simple enough that a most general type can always be efficiently inferred. The type checker in the Haskell implementation infers the most general type, and attempts unification to check for valid types.

In the original, compiled VampIR circuits came equipped with a large dictionary of definitions that told the system how to calculate intermediate circuit values. As an example, an expression like;

```
x + (y * z) = 2
```

would be turned into something like

```
x + v = 2
v = y * z
```

Additionally, separate from the actual circuit sent to Halo2 or Plonk, there would be a definition telling the system that v's value can be calculated by multiplying y and z. This is both obviously redundant and significantly complicates things like optimization as one needs to keep track of, not just the circuit, but also ancillary definitions. The Haskell implementation uses constraint propagation to infer what values unassigned variables must take based on the 3ac constraints they are a part of. Any values which could have been inferred by simply evaluating the circuits can also be inferred like this, with no change in asymptotic efficiency. This would also significantly simplify any future optimization work, although the Haskell implementation does not do any optimization, currently.

References

AG, H. "Geb Lisp Implementation." 2023a
AG, H. "VampIR Rust Implementation." 2023b
AG, H. "VampIR Haskell Implementation." 2023c
AG, H. "Juvix Haskell Compiler." 2023d