Anoma Research Topics | Technical Report

Juvix to VampIR Pipeline

Lukasz Czajka[a]

[a]Heliax AG

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

Abstract

This report explores two alternatives to Geb for Juvix-to-VampIR compilation. The first alternative is a straightforward approach based on full normalisation, which may be implemented relatively quickly and used as a comparison baseline for all other approaches. The second alternative is based on a pipeline of several compiler transformations that together convert Juvix programs into a form that can be directly translated to VampIR input.

Keywords: Juvix; Vamp-IR; Geb; compilation; normalisation; arithmetic circuits;

(Received April 25, 2023; Published: August 16, 2023; Version: August 21, 2023)

Contents

1. Introduction

VampIR* is a language for writing arithmetic circuits with support for Halo2† and Plonk-ish proving systems. Juvix targets VampIR as one of its backends, currently through an external Geb-to-VampIR compiler‡. Juvix programs

*https://github.com/anoma/vamp-ir
†https://github.com/zcash/halo2.git
‡https://github.com/anoma/geb

are first translated into the JuvixCore[§] language and then transformed in a series of steps into a form that can be translated directly into Geb.

This report describes two possible alternative Juvix-to-VampIR compilation approaches which do not depend on Geb.

1. Normalisation approach (Section 5).

   Relatively easy to implement. Perhaps the most straightforward approach possible. Could serve as a baseline comparison for other approaches and the quickest way to get a working Juvix-to-VampIR compiler, but suffers from the (super-)exponential blow-up problem.

2. Transformation pipeline (Section 6).

   Based on a pipeline of traditional compiler transformations which together convert JuvixCore programs into a form that can be directly translated to VampIR. A crucial transformation is that of defunctionalisation which removes higher-order functions (see Appendix B.3). The pipeline is more robust than the normalisation approach, but more time-consuming to implement. It probably would be more efficient than unoptimised Geb, but it is less general and harder to extend.

The conclusion of this report is the recommendation to implement the normalisation approach in order to have a first working version of Juvix-to-VampIR compilation and a simple baseline compilation strategy against which other approaches can be compared. The transformation pipeline from Section 6 can be considered as an "emergency" alternative approach in case of significant unforeseen issues with Geb in the future.

In the rest of this report, we first present a brief overview of the features supported by Juvix, JuvixCore, Geb 0.4.1 and VampIR (Section 2). Then, we discuss the fundamental (super-) exponential blow-up problem for highly-branching programs and its implications for any compilation strategy (Section 4). In Section 3, we discuss the issues with and limitations of the current implementation of Geb. Sections 5 and 6 describe the normalisation approach and the proposed transformation pipeline. Section 7 presents a comparison of the two approaches with Geb and each other. Section 8 contains some recommendations and their justifications. The appendices contain technical details relevant to the normalisation approach (Appendix A) and the transformation pipeline (Appendix B), and discussions of a few more tangential technical issues (Appendix C).

Remark 1 (Update) As of Juvix version 0.3.5, the normalisation approach has been implemented. It exhibits the limitations predicted in this report with regard to compiling highly-branching programs. The report has been slightly updated to reflect the current state of the implementation of the normalisation approach and of Geb.

2. Language features overview

The following table provides an overview of the language features supported by Juvix, JuvixCore, Geb version 0.4.1, and VampIR.

Feature	Juvix	JuvixCore	Geb	VampIR
General recursion (functions)	Yes	Yes	No	No
First-class functions	Yes	Yes	Yes	No[a]
Inductive data types	Yes	Yes	No	No
Finite data structures	Yes	Yes	Yes[b]	No[c]
Prenex polymorphism	Yes	Yes	No	No[d]
Higher-rank polymorphism	Some	Yes	No	No
Primitive integer type	Yes	Yes	No	Yes[e]
Errors	Yes	Yes	Yes	No

[a] VampIR higher-order and anonymous functions are not fully "first-class" because they cannot nontrivially interact with the field elements - they are essentially compilation-time-only.
[b] Encodable via products and coproducts.
[c] Pairs and lists in VampIR cannot nontrivially interact with the field elements. In general, finite JuvixCore data structures cannot be translated into them.
[d] Polymorphic functions are supported, but not polymorphic data types which can interact non-trivially with field elements. Hence, monomorphisation is still necessary to translate from JuvixCore.
[e] VampIR "integers" are the field elements, not integers strictly speaking.

3. Current state of the Geb implementation (v0.4.1)

Currently, Geb lacks the following features necessary for Juvix integration.

- Primitive integers and arithmetic operations.
- Support for bounded algebraic data types (needed for recursive data structures).
- Polymorphism (Juvix could alternatively implement monomorphisation).

These features are currently being worked on by the Geb team.

[§] http://github.com/anoma/juvix

4. The branching problem

Compiling highly-branching functional programs to circuits may result in (super-)exponential circuit size blow-up. This branching problem is a fundamental limitation on any direct non-interpretive compilation strategy. The difficulty lies in the fact that circuits encode all potential computation routes.

The proposed pipeline would translate Juvix programs into VampIR programs, with the size of the program increasing only by a factor polynomial in the unrolling depth limit. However, when compiling the generated programs into circuits, VampIR needs to insert the bodies of functions at all call sites, which can cause the size of the circuit to become (super-)exponential in comparison to the size of the VampIR input program. The recursive inlining done by VampIR is essentially a full normalisation process, which involves the sharing of common normal subexpressions of type integer.

The branching problem occurs, in particular, with recursive functions that have more than one recursive function call occurrence in the body. For example, consider the `filter` function defined as follows.

```
filter : (a -> Bool) -> List a -> List a
filter p Nil = Nil
filter p (Cons x xs) = if p x then Cons x (filter p xs) else filter p xs
```

The `filter` function would be completely expanded up to the given recursion depth limit, with each recursive call being expanded independently, leading to a circuit size that increases exponentially with the recursion depth limit. In this example, the blow-up could be avoided by hoisting the common subexpression `filter p xs`, as follows.

```
filter p (Cons x xs) =
  let xs' = filter p xs
  in if p x then Cons x xs' else xs'
```

However, this is not always possible if there are two or more recursive calls with essentially different arguments.

Fortunately, there is a large class of functional programs that can be guaranteed to not cause an exponential blow-up (assuming the circuits are DAGs not trees): those which use recursion linearly, i.e., for which there is only one recursive call occurrence in each function body. Note that folds over non-branching data structures (e.g. lists) fall into this category. In fact, the class of functions definable with linear recursion is quite large (though it might be difficult or inconvenient to reformulate programs into this format). All primitive recursive functions on natural numbers are definable with first-order linear recursion.

For first-order linearly recursive programs, the size of the final circuit DAG should be at most proportional to the program size multiplied by a factor polynomial in the recursion unrolling depth limit, with the degree of the polynomial depending on the number of functions in the program source.

For higher-order linearly recursive programs, to guarantee the lack of blow-up, one also needs to require that functions provided as arguments to higher-order recursive functions are themselves not recursive.

With the normalisation approach discussed in the next section, to obtain an absolute guarantee for the lack of blow-up, one also needs to restrict to programs which use only numbers and booleans.

5. The normalisation approach

One can prove that, for appropriate reduction rules, any closed JuvixCore normal-form lambda-term t of type $\mathrm{Int} \to \ldots \to \mathrm{Int} \to \mathrm{Int}$ must have the form $t = \lambda x_1 \ldots x_n.b$ where b is an expression built up from:

- the variables $x_1, \ldots, x_n$,
- number literals,
- arithmetic operations and comparisons,
- if-then-else,
- runtime error nodes.

See Appendix A for a more detailed analysis. After removing the error nodes, any lambda-term in such a form can be directly translated to VampIR.

This suggests a straightforward compilation strategy with two main steps:

1. Unrolling of recursive functions (already implemented).
2. Normalisation (partly implemented).

Once recursion is unrolled, all typed terms are strongly normalising, and normalisation can be performed without looping. After normalisation, the runtime error nodes need to be removed, but the effort required is low (see Appendix B.7).

For normalisation to work on anything but the tiniest toy programs one needs to use the Normalisation by Evaluation (NbE) technique.[¶] One also needs to take care to propagate "stuck" runtime error nodes[‖], share common normal subexpressions of type Int using lets and perform appropriate permutative conversions (explained below). Implementing NbE with these modifications is more involved than a naive repetition of β-reductions. The implementation effort is still much smaller than for the remaining parts of the pipeline from Section 6.

The problem with the normalisation approach is that normalising a (simply-typed) lambda-term may increase its size super-exponentially. In fact, the blow-up may be non-elementary, that is, the size of the output may be greater than

$$2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}$$

for any tower of 2s, where n is the size of the original term. See (Sørensen and Urzyczyn, 2006, Section 3.7).

Naive normalisation without sharing of subexpressions of type Int may cause an exponential blow-up of the program size even for numeric first-order linearly recursive programs. For example, normalising the n-times composition of $\lambda x.fxx$ results in a normal form of size exponential in n. A remedy is to reduce $(\lambda x.t)s$ to $\mathrm{let}\, x := s \,\mathrm{in}\, t$ when s is a non-constant normal form of type Int. After normalisation, the lets need to be hoisted (see Appendix B.5).

If normalisation is implemented to share subexpressions of type Int, it can be guaranteed that numeric first-order linearly recursive programs do not cause an exponential blow-up. Nonlinearly recursive programs may still increase in size super-exponentially.

Unfortunately, it is not possible to use the trick with lets for non-numeric data types. First-order linearly recursive programs that use data structures other than numbers or booleans may still cause a super-exponential blow-up.

Because JuvixCore has case-expressions on booleans (i.e., "if-then-else"), in order to obtain the desired kind of normal forms the reduction rules need to be extended with appropriate *permutative conversions* which move around case and let-expressions to expose "stuck" redexes (see (Troelstra and Schwichtenberg, 1996, Chapter 6)). Permutative conversions may cause an exponential blow-up for linearly recursive programs even if common subexpressions of type Int are shared. This happens because for non-boolean `A1`, `A2`, a case-expression

```
case (case M of {C1 -> A1; C2 -> A2}) of
  D1 -> B1
  D2 -> B2
```

needs to be converted into

```
case M of
  C1 -> case A1 of
          D1 -> B1
          D2 -> B2
  C2 -> case A2 of
          D1 -> B1
          D2 -> B2
```

to expose potential redexes in the case analysis on `A1`, `A2`. But this duplicates `B1` and `B2`.

However, if we ultimately target circuits and the blow-up does not happen during Juvix-to-VampIR compilation, it may happen when VampIR tries to generate the final circuit. VampIR needs to essentially fully inline all functions, which amounts to full normalisation (with sharing of common normal subexpressions of numeric type). With the pipeline from Section 6 the blow-up is delayed until VampIR generates the circuit. With the normalisation approach, it already happens when compiling Juvix to VampIR.

Nonetheless, it is a major disadvantage that the (super-)exponential blow-up happens already when compiling to VampIR, which could make it impossible in practise to generate any VampIR input at all. This would prevent VampIR from optimising the programs in the particular special cases when the blow-up could be avoided. Also, if parts of the circuit programs are to ultimately be interpreted in a VM, then the right VM/interpreter code format is probably closer to Geb/VampIR than to the initial JuvixCore without further transformations.

6. The transformation pipeline

The proposed pipeline for compiling Juvix to VampIR directly, without going through Geb, consists of several JuvixCore transformations which together convert a JuvixCore program into a form that can be straightforwardly translated into the input format of VampIR. The transformations remove step-by-step the features of JuvixCore not supported by VampIR (see Section 2).

1. Lambda-lifting (already implemented).
   - Removes anonymous lambda-abstractions by converting them into top-level named function definitions.

[¶] The difference between evaluation and normalisation is that evaluation does not reduce under binders, e.g., $\lambda x.(\lambda y.y)x$ is a value but not a normal form. Evaluation is what mainstream functional programming languages do (e.g., Haskell, OCaml, Lisp). Normalisation is what dependently-typed proof assistants do (e.g., Coq, Agda, Idris).

[‖] E.g. reducing $(\mathrm{if}\,\mathrm{error}\,M\,N)$ to error.

2. Monomorphisation (necessary for polymorphism).
    - Instantiates all type parameters in functions and datatypes with concrete types.
3. Defunctionalisation.
    - Removes higher-order functions.
4. Unrolling of recursive functions (already implemented).
    - Unrolls recursion up to some specified depth.
5. Hoisting of lets.
    - Moves lets upwards, out of applications.
6. Encoding of datatypes.
    - Encodes datatypes into tuples of numbers.
7. Encoding of runtime errors.
    - Encodes runtime errors with pairs.

See Appendix B for more detailed descriptions of the transformations.

After performing the above transformations, we are left with a collection of non-recursive first-order function definitions of the form

```
fun f x1 ... xn =
  let y1 = v1
      ...
      ym = vm
  in
  v
```

where $x_1, \ldots, x_n, y_1, \ldots, y_m$ have the types of numbers or tuples of numbers, and $v_1, \ldots, v_m, v$ are applicative expressions built from:

- variables,
- number literals,
- fully applied functions,
- arithmetic operations and comparisons,
- tuple operations (tuple creation and destruction).

Any definition of this form can be directly translated to a VampIR function definition. VampIR supports local definitions, arithmetic and tuples (at circuit generation time).

7. Comparison

7.1. Efficiency. The proposed pipeline relies on traditional compiler transformations to convert Juvix programs into a form that can be straightforwardly translated to VampIR. The transformations are "syntactic" and relatively "low-level". Such an approach might give better control over the details of the generated VampIR code and make it easier to generate good VampIR code (which can be compiled to smaller circuits that need less proving and verification time). The code generated with the proposed transformation pipeline should be reasonably efficient without further optimisations.

The normalisation approach essentially "fully inlines" the program by normalizing it. This may result in super-exponential blow-up at compilation stage and thus failure to compile. But the code that does compile without blow-up should be reasonably efficient.

Efficiency of Geb-generated code improved since the first version of this report, but Geb is still missing some features to allow for compilation from Juvix and a thorough comparison.

7.2. Generality. Geb is more abstract and general, which in the long term might make it more amenable to modular, structured extensions, and to formal verification. For example, according to Terence Rokop supporting higher-rank polymorphism is not problematic. With the proposed pipeline, it is not clear how to handle higher-rank polymorphism. Also, adding support for dependent types may be more challenging with the proposed pipeline than with Geb.

7.3. The branching problem. The Geb approach, in the long run, might be better suited to address the branching problem. Interpreting (parts of) the programs might avoid circuit size blow-up at the cost of significant interpretation overhead. We are not sure to what extent the current version of Geb avoids the branching problem.

8. Conclusion

We recommend the following.

1. Regardless of any other choices, implement the normalisation approach from Section 5 as a reference baseline.

   Justification:

   - The normalisation approach is perhaps the most straightforward compilation method possible for Juvix-to-VampIR. It can serve as a reference point for more sophisticated methods.
   - The normalisation approach can be implemented relatively quickly.
   - Implementing the normalisation approach will give the Geb team more time to design the incorporation of the features needed by Juvix in a structured and modular manner.

2. Continue the work on Geb. In case of significant unexpected difficulties with the Geb approach, implementing the pipeline from Section 6 could be considered.

   Justification:

   - Geb is more general than the proposed pipeline, which in the long term may make it easier to extend it with higher-rank polymorphism, dependent types or other features. Potential formalisation might also be easier.
   - Geb might make it easier to solve the branching problem.

Acknowledgements

The author expresses his gratitude to the organisers of the Valencia Heliax retreat 2023, where the possibility of direct compilation from Juvix to VampIR was discussed with Jonathan Prieto-Cubides and Murisi Turesenga. Jan Mas Rovira implemented let-hoisting used in the normalisation pipeline. Also, thanks to the entire VampIR team for their ongoing support and updates during the implementation process. The Geb team, in particular Terence Rokop, clarified many Geb-related issues. Last but not least, special thanks to Jonathan Prieto-Cubides for reviewing this report and suggesting improvements.

References

Sørensen, M.H. and Urzyczyn, P. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006
Troelstra, A. and Schwichtenberg, H. *Basic Proof Theory*. Cambridge University Press, 1996
Heliax AG. "Geb Lisp Implementation." 2023a
Heliax AG. "VampIR Rust Implementation." 2023b
Heliax AG. "Juvix Haskell Compiler." 2023c

A. Analysis of normal forms

We consider the problem of compiling a Juvix program with the main function of type $\mathrm{Int} \to \mathrm{Int}$ into a one-argument VampIR function f. One would then use this function e.g. in a VampIR equation $fx = y$ with the input variables x, y.

A generalisation to multiple arguments of type Int is straightforward. To handle first-order algeraic data types as the arguments/result of the main function, one needs to specify how to encode them into (tuples of) numbers.

Note that we restrict only the type of the main function. Other parts of the program can use arbitrary types, be polymorphic, higher-order, etc.

We consider an extended polymorphic lambda calculus without recursion which essentially corresponds to JuvixCore after compilation of pattern matching and the unrolling of recursion. For the sake of brevity, we omit some features of JuvixCore (integer comparisons, if-then-else, errors) but the analysis extends to them with small modifications.

In the presence of case-expressions on booleans and integer comparisons (or other functions which produce booleans from integers), the reduction rules need to be extended with appropriate permutative conversions (see (Troelstra and Schwichtenberg, 1996, Chapter 6)).

The types τ, σ are:

- primitive integer type Int,
- type variables α, β,
- parameterised inductive types $I\tau_1 \dots \tau_n$,
- function types $\tau \to \sigma$,
- type quantification $\forall \alpha.\tau$.

The terms t, s, r are:

- integer literals n, m,
- arithmetic operations $t \odot s$ where $\odot \in \{+, -, *, /\}$,
- variables x, y,
- application ts,
- lambda-abstraction $\lambda x : \tau.t$,
- type application $t\tau$,
- type abstraction $\Lambda\alpha.t$,
- inductive type constructors c,
- case-expressions: $\mathrm{case}\, t \,\mathrm{of}\, \{c_i x_1 \dots x_{k_i} \Rightarrow t_i \mid i = 1, \dots, n\}$.

The reductions are:

$$(\lambda x : \tau.t)s \to_\beta t[x := s]$$
$$(\Lambda\alpha.t)\tau \to_\beta t[\alpha := \tau]$$
$$\mathrm{case}\, c_m s_1 \dots s_{k_m} \,\mathrm{of}\, \{c_i x_1 \dots x_{k_i} \Rightarrow t_i \mid i = 1, \dots, n\} \to_\iota t_m[x_1 := s_1, \dots, x_{k_m} := s_{k_m}]$$

We omit the standard typing rules. The arithmetic operations have type $\odot : \mathrm{Int} \to \mathrm{Int} \to \mathrm{Int}$.

Lemma 2 *Any closed (i.e. without free variables) $\beta\iota$-normal term t of type $\mathrm{Int} \to \mathrm{Int}$ must have the form $t = \lambda x : \mathrm{Int}.s$ where s is a $\beta\iota$-normal term of type Int built up from:*

- *the variable x,*
- *integer literals n, m,*
- *arithmetic operations $\odot$.*

Proof. This is an easy lambda-calculus exercise in normal form analysis. Because $t : \mathrm{Int} \to \mathrm{Int}$ is $\beta\iota$-normal and closed, it must be a lambda-abstraction $\lambda x : \mathrm{Int}.s$ with $s : \mathrm{Int}$ also $\beta\iota$-normal and $\mathrm{FV}(s) = \{x\}$. Then one proves by induction on the size of $\beta\iota$-normal $s : \mathrm{Int}$ with $\mathrm{FV}(s) = \{x\}$ that s has the required shape. The crucial observation is that if s is an application $s = hs_1 \dots s_m$ then h cannot be a lambda-abstraction (because s is β-normal), so it must be an arithmetic operation (because there are no other terms available having a function type with target Int). To exclude the possibility that s (or h or t above) is a case-expression, one also needs to show by induction on $\beta\iota$-normal $r : I\tau_1 \dots \tau_m$ with $\mathrm{FV}(r) = \{x\}$ that $r = cr_1 \dots r_n$ for some constructor c of I. □

Any term in the form specified by the above lemma can be directly translated into a VampIR function definition.

B. Pipeline transformations

In this appendix, we describe in more detail the transformations for the pipeline proposed in Section 6.

B.1. Lambda-lifting.

- **Necessity:** required
- **Workload:** already implemented

Lambda-lifting removes anonymous lambda-abstractions by converting them into top-level named function definitions.

For example,

```
f y z = map (\x -> x + y) z
```

is transformed into

```
lam y x = x + y
f y z = map (lam y) z
```

This transformation has already been implemented for the WebAssembly pipeline.

B.2. Monomorphisation.

- **Necessity:** required for polymorphism
- **Workload:** medium

Monomorphisation instantiates all type parameters in functions and datatypes with concrete types.

For example,

```
data List a = Nil | Cons a (List a)

map : (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

is transformed into

```
data List_Nat = Nil_Nat | Cons_Nat Nat List_Nat
data List_String = Nil_String | Cons_String String List_String

map_Nat_Nat : (Nat -> Nat) -> List_Nat -> List_Nat
map_Nat_Nat f Nil_Nat = Nil_Nat
map_Nat_Nat f (Cons_Nat x xs) = Cons_Nat (f x) (map_Nat_Nat f xs)

map_String_Nat : (String -> Nat) -> List_String -> List_Nat
map_String_Nat f Nil_String = Nil_Nat
map_String_Nat f (Cons_String x xs) = Cons_Nat (f x) (map_String_Nat f xs)
```

provided that `map` occurs in the program with $a = b = \mathrm{Nat}$ and with $a = \mathrm{String}, b = \mathrm{Nat}$.

Note that not all Juvix programs can be monomorphised, because Juvix supports some higher-rank polymorphism. For example, the following is a valid Juvix definition

```
f : {A : Type} -> ({B : Type} -> B -> B) -> A -> A;
f x := x x;
```

It is not possible to (directly) monomorphise `f` because $\lambda x.xx$ is not simply-typable.

With this approach, any program with non-monomorphisable higher-rank polymorphism would cause a compilation error. One could also consider a more general approach to monomorphisation, which would duplicate the higher-rank polymorphic function arguments and move the type quantifiers to the top, e.g., first changing the above `f` to

```
f : {A B C : Type} -> (B -> B) -> (C -> C) -> A -> A;
f x1 x2 := x1 x2;
```

and adjusting all call sites appropriately. At this point, it is not clear how to formulate this transformation in full generality.

According to Terence Rokop, higher-rank polymorphism is not a problem for Geb, in principle. Geb is expected to support higher-rank polymorphism in the future.

B.3. Defunctionalisation.

- **Necessity:** required
- **Workload:** medium

Defunctionalisation removes higher-order functions by representing them as data structures and encoding unknown function applications as case distinctions on this data structure.

For example,

```
map : (Nat -> Nat) -> List Nat -> List Nat
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)

z = map (g 0) (map f xs)
```

is transformed into

```
data Fun_Nat_Nat = F | G Nat

app_Nat_Nat : Fun_Nat_Nat -> Nat -> Nat
app_Nat_Nat F x = f x
app_Nat_Nat (G x) y = g x y

map : Fun_Nat_Nat -> List Nat -> List Nat
map f Nil = Nil
map f (Cons x xs) = Cons (app_Nat_Nat f x) (map f xs)

z = map (G 0) (map F xs)
```

assuming that `f`, `g` are the only functions in the entire program which occur as the head of a partial application of type `Nat -> Nat`.

Note that, in general, one cannot use VampIR higher-order functions to encode Juvix higher-order functions, because VampIR higher-order functions are compile-time only and cannot interact nontrivially with the field elements. Thus, they cannot be, e.g., directly stored in a data structure encoded as a tuple of field elements.

In many cases, one could optimise by using VampIR higher-order functions instead of defunctionalising, but not in general. In terms of the effect on the final circuit, however, this seems equivalent to doing the specialisation optimisation on the JuvixCore level (i.e. duplicating the higher-order functions with each known function argument "pasted in"). The downside of this optimisation is that it might result in program/circuit size blow-up (see also Sections 4,5). It should probably be performed only selectively.

B.4. Unrolling of recursive functions.

- **Necessity:** required for recursion
- **Workload:** already implemented

Unrolls recursion up to some specified depth.

For example,

```
fact x = if x == 0 then 1 else x * fact (x - 1)
```

is transformed into

```
fact0 x = error "recursion too deep"
fact1 x = if x == 0 then 1 else x * fact0 (x - 1)
fact2 x = if x == 0 then 1 else x * fact1 (x - 1)
fact3 x = if x == 0 then 1 else x * fact2 (x - 1)
...
fact{D} x = if x == 0 then 1 else x * fact{D-1} (x - 1)
fact x = fact{D} x
```

This transformation has already been implemented for the Geb pipeline.

B.5. Hoisting of lets.

- **Necessity:** required
- **Workload:** low

Hoisting of lets moves lets upwards, out of applications and other subexpressions.

For example,

```
f (let x = let z = g a in z + 2 in x * b)
```

is transformed into

```
let z = g a
in
let x = z + 2
in
f (x * b)
```

B.6. Encoding of datatypes.

- **Necessity:** required
- **Workload:** medium

Elements of each datatype (i.e. the constructors) need to be encoded into (tuples of) numbers (field elements). Any unambiguous bit encoding of constructors can be used. For example,

- the first 8 bits for the constructor tag (indicating which constructor),
- the remaining bits for the encodings of constructor arguments in the order from left to right.

Note that after defunctionalisation all data types are first-order, so every constructor argument is itself a constructor or an integer. For finite bounded datatypes, the maximum number of bits needed for the encoding can be calculated at compile time. If the number of bits needed exceeds the number of bits available in a field element, then we use an appropriate tuple of field elements. For unbounded datatypes, one needs to set some bound on the depth of the data structure and calculate the number of bits based on that. Pattern matching on data types is then encoded by appropriate arithmetic and tuple operations.

For finite datatypes which are not too deep, a simpler and more readable encoding by nested tuples can be used. Let n be the maximum number of constructor arguments in any inductive type in the entire program. Then a constructor with k arguments is encoded by an $n+1$-tuple where:

- the first element is the tag,
- the next k elements are the encodings of the constructor arguments,
- the remaining $n-k$ elements are arbitrary elements of appropriate tuples (they're irrelevant since they'll never be accessed).

For the types to match, any integer that is a constructor argument must also be encodable as an $n+1$-tuple, e.g., by encoding it in the tag field.

The nested-tuple encoding is a bit simpler but not very efficient with datatypes of larger depth.

B.7. Encoding of runtime errors.

- **Necessity:** required
- **Workload:** low

Runtime errors need to be encoded, for example, using pairs.

For instance,

```
f : Int -> Int
f x = if x == 0 then error "Runtime error" else x - 1
```

is transformed into

```
f : (Int, Int) -> (Int, Int)
f (1, x) = (1, x)
f (0, x) = if x == 0 then (1, 0) else (0, x - 1)
```

where the first component of the pair indicates whether a runtime error occurred.

C. Other technical issues

C.1. Recursion unrolling before defunctionalisation. Unrolling recursion without defunctionalising first may result in counterintuitive effective recursion depth limit. This is because with higher-order functions one can "cheat" the recursion depth limit. For instance, consider the following Juvix program:

```
iterate : {A : Type} → (A → A) → Nat → A → A;
iterate f zero x := x;
iterate f (suc n) x := f (iterate f n x);

plus : Nat → Nat → Nat;
plus := iterate suc;

mult : Nat → Nat → Nat;
mult m n := iterate (plus n) m 0;

exp : Nat → Nat → Nat;
exp m n := iterate (mult m) n 1;
```

If one unrolls `iterate` up to depth k without defunctionalising (i.e. independently of its function argument), then the effective recursion depth limit for the entire program (and the size of the normal form) can be proportional to k^k, because the `exp` function iterates an iteration of iterations, each with idependent depth limit k. Note that the normal-form size blow-up may happen even if `exp` is never actually expected to be called with arguments that would result in exponential recursion depth.

After defunctionalisation, the depth limits will depend on each other – in `iterate` the calls via `app` to `plus` and `mult` will ensure a decreasing depth limit for them.

C.2. Geb encoding of higher-order functions. As far as we understand, Geb essentially encodes higher-order functions as field elements in a generic way (i.e., not depending on the particular program but encoding the entire lambda-term generically) and later essentially "interprets" them upon application (via the `eval` morphism associated with the exponential object). The difference with defunctionalisation seems to be that in defunctionalisation we just do a simple switch to choose between a finite number of functions that were present in the original program and compiled "normally", while when the entire lambda-term is encoded as a data structure all of it needs to be "interpreted". This might be much less efficient than defunctionalisation. At this point, we are unable to investigate this more thoroughly, because the Geb implementation of higher-order functions is not yet finished.

However, this issue will mostly disappear after Geb implements the planned optimisation of using higher-order VampIR functions for Geb higher-order functions. Since VampIR higher-order functions are compile-time only, there are some situations where they cannot be used to implement Geb higher-order functions. In such rare situations, the issue may persist.