Anoma Research Topics | Technical Report

Geb Pipeline v5.0

Your Namehere[a,b], Second Equal[b], and Last Author [a]

[a]Institution A, with some address.
[b]Institution B, with its other address.

* *E-Mail: youremail@someinstitute.com*

Abstract

At Heliax, we are developing a compiler stack to facilitate the creation of decentralized applications using high-level functional programming languages. This stack comprises a series of compilers that begin with Juvix and culminate in various targets. One such target is arithmetic circuits, represented via VampIR, an intermediate language for such circuits. This report highlights the Geb project, a component of this pipeline, and details the process of compiling JuvixCore into VampIR through the Geb compiler. To aid its adoption and implementation, we provide a categorical overview of the mathematical foundations of the Geb project and insights into its current Lisp-based implementation. The objective of this report is to guide future implementations and improvements of the Geb project.

Keywords: Geb ; Juvix ; VampIR ; compilers ; category theory ; semantics of PL ; Lambda calculus ; Arithmetic circuits;

(Received August 8, 2023; Published: August 21, 2023)

Contents

1. Introduction

The Geb project was initiated to compile functional programs into high-level representations of arithmetic circuits using categorical methods. This categorical approach specifies compilation procedures via internal language theorems, thus shifting focus from syntax to suitable categorical structures.

The core of the approach taken here rests on the observation that compilation procedures, which preserve type structure, can be linked to functors that maintain corresponding structures between categories representing those programming languages. One canonical example is specified in Lambek-Scott [1], dealing with the fact that the languages possessing simply-typed Lambda calculus (STLC) structure form a category which is equivalent to the category of Cartesian Closed Categories (CCC).

Therefore, instead of operating with syntax, we can represent our languages as categories and construct functors between them as also described in Elliot.[2] This approach simplifies compiler formalization due to the direct mathematical meaning of the constructions. Furthermore, functor properties inherently provide several beneficial features, including automatic type-checking for morphism composition.

The approach outlined in this document is supported by a reference implementation in Lisp. This Lisp-based Geb implementation signifies the project's initial completed stage: the compilation of functional programs written

DOI: 10.5281/zenodo.8262747

in the JuvixCore language (regarded as an appropriate untyped Lambda calculus with optional typing) to VampIR, an in-house language for arithmetic circuits. As part of the implementation, we also provide an interactive user environment equipped with extra features such as interpreters, program type-checkers, and a Geb visualizer available at the following address, release v0.4.

https://github.com/anoma/geb.

The general aim of the paper is to provide an overview of pipeline design that is accessible to people without delving very deep in the Lisp implementation. Primarily, we aim to:

1. Make all components of the pipeline precise by describing their intended semantics and paraphrasing the constructions in ML-like code with explicit typing.

2. Describe all the compilation steps using diagrams for readability and intelligibility.

3. Highlight important interactive features of the current implementation (see Appendix A).

2. Language/Category Specs

The central challenge is to effectively specify the compilation of programs, written in a high-level functional programming language, into an intermediate arithmetic circuit language. Here, this corresponds to the translation from Juvix to the VampIR language, with a specific focus on the compilation of JuvixCore to VampIR.

$$\begin{tikzcd}
\text{Juvix} \arrow[r] \arrow[dr, dashed] & \text{JuvixCore} \arrow[d] \\
 & \text{VampIR}
\end{tikzcd}$$

- JuvixCore: A Lambda calculus (with possible typing) incorporating inductive types and primitive integers, which defines the strict execution semantics of Juvix.

- VampIR: An intermediate language for arithmetic circuit specification, akin to LLVM IR. It features useful constructors such as tuples, arrays, mapping and folds, currying, and circuit synthesis to Plonkup and Halo2.

The first step in this is to provide a compilation from a sub language of Juvix, namely JuvixCore. The way we do this is by breaking up the procedure into compilations through languages we name Lambda, Geb, and Seq$\mathbb{N}$.

$$\begin{tikzcd}
\text{JuvixCore} \arrow[r] & \text{Lambda} \arrow[r] & \text{Geb} \arrow[r] & \text{Seq}\mathbb{N} \arrow[r] & \text{VampIR}
\end{tikzcd}$$

Here we are concerned only with the steps starting from Lambda. The Juvix compiler already supports compiling JuvixCore to a suitable representation in Lambda. Hence, the pipeline we will be interested in and the one written in Lisp is presented in the bottom diagram.

$$\begin{tikzcd}
\text{Lambda} \arrow[r] & \text{Geb} \arrow[r] & \text{Seq}\mathbb{N} \arrow[r] & \text{VampIR}
\end{tikzcd}$$

The current section is devoted to the specifications of the languages used in the compilation with the VampIR section devoted not to the description of VampIR itself, but of the library of functions needed for any compiled code to compile on valid inputs.

Let us provide a summary of the language features of each language in the compiler stack, as originally presented in.[3]

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	Yes[e]	Yes[f]
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] Geb has a separate type for n-bit strings for any $n \in \mathbb{N}_{>0}$

[f] VampIR "integers" are the field elements, not integers strictly speaking.

2.1. Geb . The finite version of Geb implemented here answers two questions:

- What are the minimal/optimal operations needed to present polynomials which VampIR operates with?
- What are the minimal/optimal bounding operations between functional programming languages?

The answer seems to be that we need some sort of addition, multiplication, 0, and 1 as structures to be used. For the usual arithmetic to hold, we also need some sort of distributivity law. Exponentiation should follow in a similar fashion to how we define arithmetic exponentiation from multiplication in $\mathbb{N}$. [*] Finally, we need some notion of a "context" or a "domain" - as is usual in the syntax/semantics duality - in order to be able to produce variables.

Hence, this brings us to the first definition of Geb .

Definition 1 Geb *is the language spanned by the empty type* `so0` *, unit type* `so1` *and sum/product types appropriately named* `coprod` *and* `prod` .

Here is a definition as one would write using Agda-like syntax:

```
data substobj : Set where
  so1 : substobj
  so0 : substobj
  coprod : substobj → substobj → substobj
  prod : substobj → substobj → substobj

variable
    x y z : substobj

data _G->_ : substobj → substobj → Set where
   comp : (y G-> z) → (x G-> y) → (x G-> z)
   -- composition
   id : (x : substobj) → (x G-> x)
   -- identity function
   initial : (x : substobj) → (so0 G-> x)
   -- absurd
   terminal : (x : substobj) → (x G-> so1)
   -- constant function
   mcase : (x G-> z) → (y G-> z) → ((coprod x y) G-> z)
   -- casing
   prod  : (z G-> x) → (z G-> y) → (z G-> (prod x y))
    -- product analysis
   distribute : (prod x (coprod y z)) G-> (coprod (prod x y) (prod x z))
    -- distributivity
   ->left : x G-> (coprod x y)
    -- left injection
   ->right : y G-> (coprod x y)
    -- right injection
   <-left : (prod x y) G-> x
    -- left projection
   <-right : (prod x y) G-> y
    -- right projection

subtmorph : substobj → substobj → Set
substmorph a b = a G-> b
```

Remark 2 *In the implementation, we use the name* `substmorph` *for the type above but for readability we use* `G->` *here.*

Here `substobj` is the collection of the types of our language and `f : substmorph a b` stands for a term of `b` in the context of `a` [†].

Another way of thinking about Geb is to specify the categorical structure instead of the syntactical one.

Definition 3 Geb *is the category freely spanned by an initial object, terminal object, and finite (co)products alongside the distributivity law.*

[*] Of course multiplication can be defined similarly by addition, but it is not optimal to do so due to exponentiation of terms. Moreover, it seems that having explicit exponentiation in a language through which we compile is inefficient, as it would need to compile to some separate VampIR function.

[†] There are of course extra equalities to subject the terms to, e.g. making terminal terms unique or composition associative on the nose, yet for the sake of the current project, we do not implement these, and if needed, reductions are done by hand. There is also an explicit reducer in development.

Remark 4 *The discussion below uses a more advanced categorical vocabulary. Anyone satisfied by the previous two definitions can instead go directly to 7.*

Recall that a category of models of a language is a category in which objects are models of said language and morphisms are mappings that maintain the language structure[‡]. This definition is equivalent to considering the term model of a language, which is the initial category of its corresponding category of models[§].

However, the semantics of Definition 3 are unclear. Note that taking coproducts with `so0` is equivalent to the identity functor[¶] while taking products with `so0` to a constant functor at `so0`[‖]. Similarly, taking products with `so1` is equivalent to the identity functor[**]. Therefore, intuitively, the only way to make "new" types in Geb is by taking coproducts of `so1` with itself over and over. There is actually a category which is specified by being generated by filtered colimits over finite inclusions between such coproducts of the terminal object: FinSet. It is indeed generated by simply adding extra 1-element sets to the empty set. So, there is the third definition of Geb.

Definition 5 Geb *is* FinSet.

Let us make this correspondence exact:

Theorem 6 FinSet *is an initial weak model of the* Geb *language. That is, it is equivalent to the initial model of the essentially algebraic theory of category theory with initial, terminal objects, finite (co)products structure, and distributivity. Moreover, the equivalence preserves all relevant categorical structure.*

Proof. We can instead deal with the skeleton $Sk(\text{FinSet}) = \text{FinOrd}$, the category of finite ordinals, since clearly the equivalence preserves all relevant structure. Now suppose that $(C, 1_C, 0_C, +_C, \times_C)$ is a distributive category with initial/terminal object.

Note that $(\bigoplus_n 1_C) \times_C (\bigoplus_m 1_C) = \bigoplus_{n\times m} 1_C$ by induction on n.

- When $n = 0$ since the initial object is strict, then the product is just 0_C which is also the null coproduct.
- For the inductive step, consider the chain of isomorphisms below.

$$\begin{aligned}
\bigoplus_{1+n} 1_C \times_C \bigoplus_m 1_C &\cong \left(\bigoplus_n 1_C +_C 1_C\right) \times_C \bigoplus_m 1_C \\
&\cong \left(\bigoplus_n 1_C \times_C \bigoplus_m 1_C\right) +_C \bigoplus_m 1_C && \text{(By distributivity prop.)} \\
&\cong \bigoplus_{n\times m} 1_C + \bigoplus_m 1_C && \text{(By induction hypothesis on } n\text{)} \\
&\cong \bigoplus_{(1+n)\times m} 1_C.
\end{aligned}$$

Now we prove that FinOrd is our weakly initial model. To do so, we need to construct a functor of type FinOrd → C that preserves our structure, and we also need to prove that it is unique up to a natural isomorphism.

First, we construct the functor.

Recall that both FinSet and FinOrd constitute the weakly initial category of the theory of categories with finite coproduct on one object[††]. Given that C has coproducts and 1_C, we can hence construct the functor $F\colon \text{FinOrd} \to C$, which preserves this structure along with the terminal object. We claim that F is our needed functor. We claim that the product structure is also preserved. Given $n \times m$ proceed by induction on n:

- If $n = 0$, then we have,

$$F(0 \times m) = F(0) \cong 0_C \cong 0_C \times_C F(m) \cong F(0) \times_C F(m).$$

- For the inductive step, we use the previous observation:

[‡] For example, the category of models of groups is the category of groups and group morphisms.

[§] For more abstract nonsense on the topic one can consult [4] and. [5]

[¶] Similar to how adding 0 is the same as the identity function.

[‖] Distributive categories have strict initial objects, similar to how multiplying by 0 gives 0.

[**] Similar to how multiplying by 1 gives back the original term.

[††] https://ncatlab.org/nlab/show/FinSet#universal_properties

$$
\begin{aligned}
F((1+n)\times m) &= F((n\times m)+m) \\
&\cong (F(n)\times_C F(m)) +_C F(m) && \text{(By induction and } F \text{ preserves coproducts)} \\
&\cong \left(\bigoplus_n 1_C \times_C \bigoplus_m 1_C\right) +_C \bigoplus_m 1_C && \text{(By definition of } F\text{)} \\
&\cong \bigoplus_{n\times m} 1_C + \bigoplus_m 1_C \\
&\cong \bigoplus_{(1+n)\times m} 1_C \\
&\cong \bigoplus_{1+n} 1_C \times_C \bigoplus_m 1_C \\
&\cong F(1+n)\times F(m).
\end{aligned}
$$

Since everything the functor hits is a coproduct of the terminal object up to an isomorphism and F preserves all coproduct structure, it preserves all other morphism structure, including product universal morphisms and distributivity, i.e. it is an actual functor of models.

The only thing left to prove is the uniqueness property of F.

However, by the same argument, since everything the functor hits is a coproduct of 1_C and F preserves coproduct structure and is a unique functor doing so up to natural iso, this also proves uniqueness up to iso. □

Hence, Definition 5 is fully valid and when one talks of Geb , one can instead think about a nice presentation of FinSet. Hence a term of `substobj` is just a finite set and a term of `a G-> b` is just a function between corresponding finite sets. As expected, this equivalence actually provides us with the existence of a lot of extra structure.

Corollary 7 *Geb is cartesian closed.*

That is, for any two objects `a, b : substobj` we get an object `(sohomobj a b) : substobj` which is an object of Geb representing all terms of `b` in context `a` , or, in other words, it is the function type from type `a` to type `b` . It also comes equipped with the usual curry isomorphism and evaluation maps:

```
  curry : ((prod x y) G-> z) → (x G-> (sohomobj y z))
  eval : ((sohomobj x y) G-> x) → y
  uncurry : (x G-> (sohomobj y z)) → ((prod x y) G-> z)
```

Remark 8 *The actual implementation of said types and terms is not of importance for the report but we need these functions in order to compile* JuvixCore *code to* Geb *, which is why these are mentioned. For explicit constructions, one can consult the Agda folder*[‡‡].

Now we "extend" Geb by adding natural numbers with the following trick: consider all n-bit wide numbers. Evidently there are 2^n of these, which means that their set is in bijection with the finite ordinal of cardinality 2^n. In other words, we introduce objects $nat - widthn$ for every $n > 0$ which are really just isomorphic copies of 2^n coproducts of `so1` . So Geb already had the natural number functionality, it was just highly inefficient before introducing explicit constructors.

```
data substobj : Set where
...
  nat-width : Nat → substobj
  -- nat-width n is the set of n-bit natural numbers

data substmorph : substobj → substobj → Set where
  ...
  nat-add : (n : Nat) → ((nat-width n) G-> (nat-width n))
  --addition
  nat-mult : (n : Nat) → ((nat-width n) G-> (nat-width n))
  --multiplication
  nat-sub : (n : Nat) → (nat-width n)  G->  (nat-width n)
  --subtraction
  nat-div : (n : Nat) → (nat-width n)  G-> (nat-width n)
  --division
  nat-concat : (n m : Nat) → ((prod (nat-width n) (nat-width m))
                               G-> (nat-width (n+m)))
  -- concatenation of bits
```

[‡‡] https://github.com/anoma/geb/blob/main/geb-agda/Geb_Hom.pdf

```
nat-inj : (n : Nat) → ((nat-width n) G-> (nat-width (n + 1)))
-- injects an $n$-bit numbers as an $n+1$ bit number
nat-decompose : (n : Nat) → ((nat-width (n + 1))
                             G-> (prod (nat-width 1) (nat-width n)))
-- decomposes an n+1 bit number into the top bit and the rest of the bits
one-bit-to-bool : (nat-width 1) G-> (coprod so1 so1)
-- isomorphism sending bit 0 to the left copy of so1, and
-- 1 to the right copy.
nat-eq : (n : Nat) → ((prod (nat-width n) (nat-width n))
                      G-> (coprod so1 so1))
--equality predicate
nat-lt : (n : Nat) → ((prod (nat-width n) (nat-width n))
                      G-> (coprod so1 so1))
--"less-than" predicate
```

With this, we have described all the specification of Geb relevant for the current report.

Remark 9 *Be careful when using the term "Geb" as it can point to many things. The* Geb *language as it is currently implemented in the pipeline is just a subcomponent of what the final version of* Geb *will look like. The intention is that* Geb *will become a language that will be able to articulate the entirety of the pipeline currently presented in CL. The reason we call this category* Geb *here is that this category will be a core component specifying the higher-order* Geb *category, which itself will correspond to a larger programming language. However, in this report we always use "Geb" to describe* Geb *as given in Definitions 1-3.*

2.2. Lambda. The Lambda language is an extended version of STLC with types taken from Geb [§§]. However, the terms are introduced untyped first and become typed once the context is introduced via specific functions such as `ann-term` .

Definition 10 Lambda *is STLC with types from* `substobj` *extended by error terms* `err` *of arbitrary type, as well as natural number arithmetic with fixed bit width.*

Here is the definition as one would type it in Agda:

```
data lambda : Set where
    absurd : substobj → lambda
    -- beta reduction for empty type
    unit : lambda
    -- unique element of unit type so1
    left : substobj → lambda → lambda
    -- left inclusion into coproduct
    right : substobj → lambda → lambda
    -- right inclusion into comporoduct
    case-on : lambda → lambda → lambda → lambda
    -- beta elimination for coproducts
    pair : lambda → lambda → lambda
    -- term introduction for products
    fst : lambda → lambda
    -- left projection
    snd : lambda → lambda
    -- right projection
    lamb : List substobj → lambda → lambda
    -- function type term introduction
    -- takes in a list of types for iterated function type use
    app : lambda → List lambda → lambda
    -- function type elimination
    --takes in a list of terms for iterated elimination
    index : Nat → lambda
    -- variable via de Bruijn indices
    err : substobj → lambda
    -- error term of arbitrary type
    plus : lambda → lambda → lambda
    -- arithmetic operations
    times : lambda → lambda → lambda
    minus : lambda → lambda → lambda
    divide : lambda → lambda → lambda
```

[§§] The latter part is the reason why we described Geb before talking about Lambda, even if the compilation order is different.

```
    bit-choice : List bool → lambda
    -- natural number choice
    lamb-eq : lambda → lambda → lambda
    -- equality of natural numbers
    lamb-lt : lambda → lambda → lambda
    -- "less-than" predicate on natural numbers
```

Of course, a lot of the terms produced in this way will be ill-typed. For example, indices in an empty context are ill-defined. So, as mentioned before, we have the functionality of typing the terms of Lambda, namely the function `ann-term` as in the Lisp implementation each term has a slot for types which we can fill in recursively. We also have a predicate `well-defp` that checks whether a given Lambda term is well defined according to the standard typing rules examples of which are provided below.

- `unit` is of type `so1`.
- All the arithmetic operation terms are deemed well-typed iff the provided terms are of same bit-width.
- The natural number predicates have return type `coprod so1 so1`

2.3. Seq$\mathbb{N}$. At this stage, we need a way to talk about abstract VampIR functions in terms of inputs and outputs. Seq$\mathbb{N}$ is supposed to act as a canonical model for that.

Definition 11 Seq$\mathbb{N}$ *is the category whose objects are finite sequences of natural numbers* $(x_1, \cdots, x_n)$ *and* $f : (x_1, \cdots, x_n) \to (y_1, \cdots, y_m)$ *is a* VampIR *function that takes in* n *inputs with the* i*-th entry being of bit-size* x_i *and outputs an* m*-tuple with* i*-th entry being of bit-size* y_i*. Hence, it makes sense to also call* Seq$\mathbb{N}$ *a category of Vamp-IR programs.*

However, we do not yet have a precise specification of VampIR as it is still in active development, what we are aiming for is simply to specify some subcategory of the one described above which will be sufficient for interpreting faithfully all the Geb programs we have.

We now present an explicit definition of Seq$\mathbb{N}$ in terms of a free category spanned by the following objects/morphisms[¶¶].

```
seqnobj = List Nat

variable
  x y z w : seqnobj

data _S->_ : seqnobj → seqnobj → Set where
    composition : (y S-> z) →  (x S-> y) → (x S-> z)
    --composition
    id : (x : seqnobj) → (x S-> x)
    --identity
    fork-seq : (x : seqnobj)  → (x S-> (append x x))
    --morphism copying the entry twice
    parallel-seq : (x S-> y) → (z S-> w) →
                   ((append x z) S-> (append y w))
    -- given two functions, parallizes them
    drop-nil : (x : seqnobj) → (x S-> (0 :: []))
    -- given an object, just drops everything to a nil entry
    remove-right : (x : seqnobj) → ((append x (0 :: [])) S-> x)
    -- removes a right-positioned zero entry
    remove-left : (x : seqnobj) → ((append (0 :: []) x) S-> x)
    -- removes a left-positioned zero entry
    drop-width : (x y : seqnobj) → (x S-> y)
    -- on entry, removes extra bits assuming x > y pointwise
    inj-length-left : (x y : seqnobj) → (x S-> (append x y))
    -- injects an entry producing nothing on the left side
    inj-length-right : (x y : seqnobj) → (y S-> (append x y))
    -- injects an entry producing nothing on the right side
    inj-size : (x y : Nat) → ((x :: []) S-> (y :: []))
    -- if x < y injects an x-bit number as y-bit
    branch-seq : (x S-> y) →  (x S-> y) → ((1 :: x) S-> y)
    zero-bit : (0 :: []) S-> (1 :: [])
    -- produces 0
```

[¶¶] Operations such as `::` and `append` are usual operations on lists in the definition.

```
one-bit : (0 :: []) S-> (1 :: [])
-- produces 1
shift-front : ((x_1 :: ... :: x_n) : seqnobj) → (k : Nat) →
              ((x_1 :: ... :: x_n) S->
               (x_k :: x_1 ... x_k-1 :: x_k+1 :: ...))
-- shifts the k-th entry in front
seqn-add : (n : Nat) → ((n :: n :: 0) S-> (n :: 0))
-- basic arithmetic operations
seqn-subtract : (n : Nat) → ((n :: n :: 0) S-> (n :: 0))
seqn-multiply : (n : Nat) → ((n :: n :: 0) S-> (n :: 0))
seqn-divide : (n : Nat) → ((n :: n :: 0) S-> (n :: 0))
seqn-nat :: (n m : Nat) → ((0 :: []) S-> (n :: []))
-- produces an n-wide natural number m
seqn-concat : (n m : Nat) → ((n :: m :: []) S-> ((n + m) :: []))
-- concatenates two n and m wide bits
seqn-decompose : (n : Nat) → ((n :: []) S-> (1 :: (n - 1) :: []))
-- takes the top bit off of the input and places it in separate slot
seqn-eq : (n : Nat) → ((n :: n :: []) S-> (1 :: 0 :: []))
-- equality predicate
seqn-lt : (n : Nat) → ((n :: n :: []) S-> (1 :: 0 :: []))
-- less-than predicate
```

Remark 12 *Note the pseudocode used in* `seqn-shift` *definition. Also, note the unusual codomain of* `seqn-eq` *and* `seqn-lt`*. In this category* $(1, 0) \cong (1)$*. For a more detailed discussion, see Section 3.2.*

The `parallel-seq` function is actually the product of the corresponding morphisms, and therefore we use the notation `parallel-seq f g` $= f \times g$.

2.4. VampIR.

VampIR is an intermediary language for arithmetic circuits. As the final point of the compilation, it does not need to be fully specified by the pipeline [***]. The only thing we practically need is a collection of functions we will be using in the compilation procedure which we provide[†††].

- Check that x is a boolean, then print it

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

-
  ```
  def base_range 0 = { []};
  ```

-
  ```
  def next_range range a = {
    def a0 = bool (fresh (a % 2));
    def a1 = fresh (a / 2);
    a0 : (range a1)
  };
  ```

- Take natural number n and integer m, check that m is n-bit then print n-bit representation of m.

  ```
  def range_n n = { iter n next_range base_range };
  ```

- Head of the list

  ```
  def hd (h:t) = { h };
  ```

- Tail of the list

  ```
  def tl (h:t) = { t };
  ```

- n-th element of the list

  ```
  def n_th lst n = { hd (iter n tl lst) };
  ```

- Give 0 if a is negative and 1 otherwise given an expected bit-size

[***] Even if we do have the specification in the code covering some principal aspects https://github.com/anoma/geb/blob/main/src/vampir/spec.lisp.

[†††] The functions are written in VampIR syntax v0.1.3.

DOI: 10.5281/zenodo.8262747

```
def negative n a = {
    nth (range_n (n + 1) (a + (2 ^ n))) n
};
```

- Check that the sum of two n-bit numbers is an n-bit number then prints the sum

  ```
  def plus_range n x1 x2 = {
    range_n n (x1 + x2);
    x1 + x2
  };
  ```

- Check that the multiplication of two n-bit numbers is n-bit then prints the sum.

  ```
  def mult_range n x1 x2 = {
    mult_range n (x1 * x2);
    x1 * x2
  };
  ```

- Check that the subtraction of two n-bit natural numbers is non-negative then prints the subtraction

  ```
  def minus_range n x1 x2 = {
    negative n (x1 - x2) = 1;
    x1 - x2
  };
  ```

- Check that "a" is nonzero

  ```
  def iszero a = {
    def ai = fresh (1 | a);
    def b = 1 - (ai * a);
    a * b = 0;
    1 - b
  };
  ```

- `def combine-aux x y = { x + (2 * y) };`

- Combines a bit-representation of a number into decimal form

  ```
  def combine xs = { fold xs combine_aux 0 };
  ```

- `def take_base lst = { [] };`

- `def take_ind take (h:t) = { h:(take t) };`

- `def take n = { iter n take_ind take_base };`

- `def drop_ith_rec take (h:t) = { h:(take t) };`

- Drop n-th element of a list.

  ```
  def drop_ith n = { iter n drop_ith_rec (fun (h:1) {1}) };
  ```

3. Compilation Steps

We have outlined the specifications of the intermediary languages that we will be using to compile through, and we are now ready to present our main pipeline. The compilation diagram is presented below.

$$\begin{tikzcd} \text{Lambda} \arrow[r, "\text{to-cat}"] & \text{Geb} \arrow[r, "\text{to-seqn}"] & \text{Seq}\mathbb{N} \arrow[r, "\text{to-circuit}"] & \text{VampIR} \end{tikzcd}$$

The compilations are appropriately named to-cat, to-seqn, and to-circuit after their codomains. Actually, to-seqn and to-circuit are generics, in the sense that their names stand for various compilations. For example, to-seqn is originally a compilation from Geb to Seq$\mathbb{N}$, yet if we feed this function with Lambda code, it actually compiles [‡‡‡] that code to Seq$\mathbb{N}$: first, applying to-cat and then the original to-seqn. Similarly for to-circuit.

[‡‡‡] With certain settings

DOI: 10.5281/zenodo.8262747

$$\begin{tikzcd}
\text{Lambda} \arrow[r, "\text{to-cat}"] \arrow[rr, bend right=15, "\text{to-seqn}"'] \arrow[rrr, bend right=25, "\text{to-circuit}"'] & \text{Geb} \arrow[r, "\text{to-seqn}"] \arrow[rr, bend left=30, "\text{to-cicruit}"] & \text{Seq}\mathbb{N} \arrow[r, "\text{to-circuit}"] & \text{VampIR}
\end{tikzcd}$$

Remark 13 *The interactive use of the pipeline is best done through the lambda package.*

```
CL-USER> (in-package geb.lambda.trans)

TRANS> (to-cat nil (lamb (list (coprod so1 so1)) (index 0)))
(<-left (+ s-1 s-1) s-1)

TRANS> (to-seqn (lamb (list (coprod so1 so1)) (index 0)))
(COMPOSITION (REMOVE-RIGHT (1 0)) (PARALLEL-SEQ (ID (1 0)) (DROP-NIL (0))))

TRANS> (to-circuit (lamb (list (coprod so1 so1)) (index 0)) :identity_fun)
(def identity_fun x1 x2 x3 = {
   range_n 0 x3;
   range_n 0 x2;
   range_n 1 x1;
   (x1) };
)
```

Here, nil stands for the empty context we feed the compiler when we move from Lambda *to* Geb *. All other compilations are assumed to occur in the empty context. To compile to* VampIR *we also need to specify the name of our function as above.*

Remark 14 *In the proceeding discussion, all possible info is replaced by categorical counterparts for readability.* prod *will be denoted by* $\times$*,* coprod *by* $+$*, elements of* substmorph *by actual arrows etc. wherever possible without losing information. Similarly, sohomobj will now be denoted by explicit arrows.*

3.1. Lambda $\longrightarrow$ Geb . The core part of the compilation from Lambda to Geb is a canonical one: the initial morphism from the term model of the Lambda language to Geb in the category of STLC models.

The description of the compilation will be made less formal to facilitate understanding, lambda-abstraction, and function application terms will be assumed to be one argument only.

We define the compilation to-cat : Lambda$\rightarrow$ Geb with precise typing being

```
to-cat : List substobj → lambda  →  substmorph
```

as a function taking in a lambda term and a list of Geb objects serving as a context and spitting out a Geb morphism with typing

$$\text{to-cat}(A_1, ..., A_n)(\text{t} : B) : A_1 \times ... \times A_n \times 1 \to B.$$

We proceed by induction on t , assuming that we are always given context $A_1, ..., A_n := \gamma$ and defining $A_1 \times ... \times A_n \times 1 := \Gamma$[§§§].

Table 1: Compilation from Lambda to Geb .

Lambda	Geb
(absurd B (term : so0))	$$\begin{tikzcd} & 0 \arrow[dr, "!"] & \\ \Gamma \arrow[ur, "\text{to-cat}(\gamma)(\text{term})"] \arrow[rr, "\text{to-cat}(\gamma)(\text{t})"'] & & B \end{tikzcd}$$
(unit : so1)	$$\begin{tikzcd} \Gamma \arrow[r, "!"] \arrow[r, bend right, "\text{to-cat}(\gamma)(\text{t})"'] & 1 \end{tikzcd}$$
(left A (term : B) : B + A)	$$\begin{tikzcd} & B \arrow[dr, "\iota_1"] & \\ \Gamma \arrow[ur, "\text{to-cat}(\gamma)(\text{term})"] \arrow[rr, "\text{to-cat}(\gamma)(\text{t})"'] & & B + A \end{tikzcd}$$

[§§§] That is, any context is interpreted in Geb as an iterated product with so1 on the right.

DOI: 10.5281/zenodo.8262747

(right A (term : B)) : A + B

$$\begin{tikzcd}
 & B \arrow[dr, hook, "\iota_2"] & \\
\Gamma \arrow[ur, "\text{to-cat}(\gamma)(\text{term})"] \arrow[rr, "\text{to-cat}(\gamma)(\text{t})"'] & & A+B
\end{tikzcd}$$

(case (on : A + B)
(ltm : C)
(rtm : C)) : C

$$\begin{tikzcd}
A\times\Gamma \arrow[dr, "\text{to-cat}(\gamma)(\text{ltm})"] & & B\times\Gamma \arrow[dl, "\text{to-cat}(\gamma)(\text{rtm})"'] \\
\Gamma\times A \arrow[u, "\cong"] \arrow[r, "F"'] & C & \Gamma\times B \arrow[l, "G"] \arrow[u, "\cong"']
\end{tikzcd}$$

$$\begin{tikzcd}
\Gamma\times(A+B) \arrow[r, "\text{distribute}"] & (\Gamma\times A)+(\Gamma\times B) \arrow[d, "{[F,G]}"] \\
\Gamma \arrow[u, "\text{to-cat}(\gamma)(\text{on})"] \arrow[r, "\text{to-cat}(\gamma)(\text{t})"'] & C
\end{tikzcd}$$

(pair (ltm : A) (rtm : B)) : A × B

$$\begin{tikzcd}
 & A\times B \arrow[dl, bend right, "\pi_1"'] \arrow[dr, bend left, "\pi_2"] & \\
A & \Gamma \arrow[u, bend left, "\langle F,G\rangle"] \arrow[u, bend right, "\text{to-cat}(\gamma)(\text{t})"'] \arrow[u, phantom, "="] \arrow[l, "F"'] \arrow[l, bend left, "\text{to-cat}(\gamma)(\text{ltm})"] \arrow[l, phantom, "\|"] \arrow[r, "G"] \arrow[r, bend right, "\text{to-cat}(\gamma)(\text{rtm})"'] \arrow[r, phantom, "\|"] & B
\end{tikzcd}$$

(fst (term : A × B)) : A

$$\begin{tikzcd}
 & A\times B \arrow[dr, "\pi_1"] & \\
\Gamma \arrow[ur, "\text{to-cat}(\gamma)(\text{term})"] \arrow[rr, "\text{to-cat}(\gamma)(\text{t})"'] & & A
\end{tikzcd}$$

(snd (term : A × B)) : B

$$\begin{tikzcd}
 & A\times B \arrow[dr, "\pi_2"] & \\
\Gamma \arrow[ur, "\text{to-cat}(\gamma)(\text{term})"] \arrow[rr, "\text{to-cat}(\gamma)(t)"'] & & B
\end{tikzcd}$$

(lamb (tdom : A) (term : B)) : A → B

$$\begin{tikzcd}
 & A\times\Gamma \arrow[dr, "\text{to-cat}(A,\Gamma)(term)"] & \\
\Gamma\times A \arrow[ur, "\cong"] \arrow[rr, "F"'] & & B
\end{tikzcd}$$

$$\begin{tikzcd}
\Gamma \arrow[r, "\text{curry}(F)"] \arrow[r, bend right, "\text{to-cat}(\gamma)(\text{t})"'] & (A\to B)
\end{tikzcd}$$

(app (fun : A → B) (term : A)) : B

$$\begin{tikzcd}
 & \Gamma \arrow[dl, bend right, "\text{to-cat}(\gamma)(\text{fun})"'] \arrow[dl, bend left, "F"] \arrow[dl, phantom, "//"] \arrow[dr, bend left, "\text{to-cat}(\gamma)(\text{term})"] \arrow[dr, bend right, "G"'] \arrow[dr, phantom, "//"] & \\
(A\to B) & & A
\end{tikzcd}$$

$$\begin{tikzcd}
 & (A\to B)\times A \arrow[dr, "\text{eval}"] & \\
\Gamma \arrow[ur, "\langle F,G\rangle"] \arrow[rr, "\text{to-cat}(\gamma)(\text{t})"'] & & B
\end{tikzcd}$$

(index i) : A_i

$$\begin{tikzcd}
A_1\times\ldots\times A_n\times\mathsf{so1} \arrow[r, "\pi_n"] \arrow[r, bend right, "\text{to-cat}(\gamma)(t)"'] & A_i
\end{tikzcd}$$

DOI: 10.5281/zenodo.8262747

m = bitv in decimal notation

`(bit-choice bitv) : nat-width n`

$$\begin{tikzcd}
 & 1 \arrow[dr, "\text{nat-const n m}"] & \\
\Gamma \arrow[ur, "!"] \arrow[rr, "\text{to-cat}(\gamma)(\text{t})"'] & & \text{(nat-width n)}
\end{tikzcd}$$

```
(plus (ltm : natwidth-n)
      (rtm : nat-width n))
      : nat-width n
```

$$\begin{tikzcd}
 & \Gamma \arrow[dl, bend right, "\text{to-cat}(\gamma)(\text{ltm})"'] \arrow[dl, bend left, "F"] \arrow[dl, phantom, "\parallel"] \arrow[dr, bend left, "\text{to-cat}(\gamma)(\text{rtm})"] \arrow[dr, bend right, "G"'] \arrow[dr, phantom, "\parallel"] & \\
\text{(nat-width n)} & & \text{(nat-width n)}
\end{tikzcd}$$

$$\begin{tikzcd}
 & \text{(nat-width n)} \times \text{(nat-width n)} \arrow[dr, "\text{nat-add}"] & \\
\Gamma \arrow[ur, "\langle F,G \rangle"] \arrow[rr, "\text{to-cat}(\gamma)(\text{t})"'] & & \text{(net-width n)}
\end{tikzcd}$$

```
(lamb-eq    (ltm : nat-width n)
       (rtm : nat-width n))
        : (coprod so1 so1)
```

$$\begin{tikzcd}
 & \Gamma \arrow[dl, bend right, "\text{to-cat}(\gamma)(\text{ltm})"'] \arrow[dl, bend left, "F"] \arrow[dl, phantom, "\parallel"] \arrow[dr, bend left, "\text{to-cat}(\gamma)(\text{rtm})"] \arrow[dr, bend right, "G"'] \arrow[dr, phantom, "\parallel"] & \\
\text{(nat-width n)} & & \text{(nat-width n)}
\end{tikzcd}$$

$$\begin{tikzcd}
 & \text{(nat-width n)} \times \text{(nat-width n)} \arrow[dr, "\text{nat-eq}"] & \\
\Gamma \arrow[ur, "\langle F,G \rangle"] \arrow[rr, "\text{to-cat}(\gamma)(t)"'] & & 1+1
\end{tikzcd}$$

Remark 15 *Other arithmetic operations and `lamb-lt` that get compiled accordingly to the last two steps. There is also an `err` term in the language that we have skipped for simplicity of presentation.* Lambda *term with `err` subterm is interpreted through a different pass. The process of compiling terms with `err` is analogous to the one previously described, but with the addition of wrapping all elements recursively in a Maybe monad.*

3.2. Geb $\longrightarrow$ Seq$\mathbb{N}$. The compilation of Geb to Seq$\mathbb{N}$ is more complex as we now explicitly deal with establishing a functor. We need two functions: one presenting Geb types as sequences and another transforming Geb morphisms into Seq$\mathbb{N}$ morphisms.
The encoding of types is tricky. We want `coprod so1 so1` to stand for (1) since it should represent one bit. The most evident way of taking coproducts is to index them with a bit, so that on 0 the element is of the left type and the right one on 1. That is, taking the coproduct means adding an extra bit of information. Since `coprod so1 so1` stands for (1), the original types should be of 0-bit width. That is, `so1` stands for (0). Similarly, `so0` stands for (0). Clearly, `nat-width n` should correspond to (n). We also have arithmetic functions of form

$$\texttt{nat-add n : (nat-width n)} \times \texttt{(nat-width n)} \rightarrow \texttt{(nat-width n)}$$

which we want to be a VampIR function taking in two n-bit inputs. So we want to interpret products `prod a b` as a tuple where we append the list of interpreted `a` to the list of interpreted `b`. Hence, for the `nat-add n` domain, the result will be (n, n).
The hardest part is interpreting the coproducts. Given a morphism with domain `coprod a b` we want something that takes either input from `a` or an input from `b` and uses branching the way it is usually done in VampIR. As mentioned before, to use branching, we need the top of the sequence to have a bit on top, with 0 indicating an input from `a` and 1 indicating an input from `b`. So, for example, we might have `coprod (nat-width n) (nat-width n)` interpreted as $(1, n)$ so that at 0 we consider the second input to be from the left copy of numbers, and on 1 from the right.
Yet what if the interpretations of a and b differ in bit-sizes? For example, when we consider `coprod (nat-width 2) (nat-width 3)`? Then we probably should just take the largest bit-width and interpret it as $(1, 3)$. What if they differ in lengths as `coprod (nat-width 2) (prod (nat-width 2) (nat-width 2))`? Take the longest sequence: $(1, 2, 2)$. What if both occur? Then take the longest sequence, take their product point-

wise and get the maximal bit-width of each entry. So coprod (nat-width 2) (prod so1 (nat-width 3)) becomes $(1, (\max(2, 0)), (\max(0, 3)) = (1, 2, 3)$
So we define a function width: substobj $\to$ seqnobj by induction using Agda-like notation:

```
width : substobj  → seqnobj
width so1 = (0 :: [])
width so0 = (0 :: [])
width (nat-width n) = (n :: [])
width (prod x y) = (append (width x) (width y))
width (coprod x y) = (1 :: (zipWith max (width x) (width y)))
```

Now we are ready to define the compilation of morphisms via diagrams. Recall that seqn-parallel is the product of morphisms in our category. Hence we write $f \times g$ instead of parallel f g for appropriate morphisms.
We define to-seqn : Geb $\to$ SeqN with formal typing

```
to-seqn : {x y : substmorph} → (x G-> y) → ((width x) S-> (width y))
```

We denote the interpretation of the variables a, b, c in Seq$\mathbb{N}$ by $(a_1, \ldots, a_n), (b_1, \ldots, b_m), (c_1, \ldots, c_k)$ accordingly.
Proceed by induction on t : substmorph .

Geb	Seq$\mathbb{N}$
id a	$$\begin{tikzcd} (a_1, \ldots, a_n) \arrow[r, "\text{id}"] \arrow[r, bend right, "\text{to-seqn(f)}"'] & (a_1, \ldots, a_n) \end{tikzcd}$$
comp (f : b G-> c) (g : a G-> b)	$$\begin{tikzcd} (a_1, \ldots, a_n) \arrow[r, "\text{to-seqn g}"] \arrow[rr, bend right, "\text{to-seqn(f)}"'] & (b_1, \ldots, b_m) \arrow[r, "\text{to-seqn f}"] & (c_1, \cdots, c_k) \end{tikzcd}$$
init a	$$\begin{tikzcd} (0) \arrow[r, "\text{drop-width}"] \arrow[r, bend right, "\text{to-seq (f)}"'] & (a_1, \ldots, a_n) \end{tikzcd}$$
terminal a	$$\begin{tikzcd} (a_1, \ldots, a_n) \arrow[r, "\text{drop-nil}"] \arrow[r, bend right, "\text{to-seqn}(f)"'] & (0) \end{tikzcd}$$
->left a b	• $n = m$ $$\begin{tikzcd} & (0, a_1, \ldots, a_n) \arrow[d, "\text{zero-bit} \times (\bigotimes_n \text{inj-size})"] \\ (a_1, \ldots, a_n) \arrow[ur, "\text{inj-length-right}"] \arrow[r, "\text{to-seqn(f)}"'] & (0, \max(a_1, b_1), \ldots, \max(a_n, b_m)) \end{tikzcd}$$ • $n < m$ $$\begin{tikzcd} (0, a_1, \ldots, a_n) \arrow[r, "\text{inj-length-left}"] & (0, a_1, \ldots, a_n, 0, \ldots, 0) \arrow[d, "\text{zero-bit} \times (\bigotimes_m \text{inj-size})"] \\ (a_1, \ldots, a_n) \arrow[u, "\text{inj-length-right}"] \arrow[r, "\text{to-seqn(f)}"'] & (0, \max(a_1, b_1), \ldots, \max(a_n, b_n), \ldots, b_m) \end{tikzcd}$$

->right a b	• $n = m$ $$\begin{tikzcd} & (0, a_1, \dots, a_n) \arrow[d, "\text{one-bit} \times (\bigotimes_n \text{inj-size})"] \\ (a_1, \dots, a_n) \arrow[ur, "\text{inj-length-right}"] \arrow[r, "\text{to-seqn(f)}"'] & (0, \max(a_1, b_1), \dots, \max(a_n, b_m)) \end{tikzcd}$$ • $n < m$ $$\begin{tikzcd} (0, a_1, \dots, a_n) \arrow[r, "\text{inj-length-left}"] & (0, a_1, \dots, a_n, 0, \dots, 0) \arrow[d, "\text{one-bit} \times (\bigotimes_m \text{inj-size})"] \\ (a_1, \dots, a_n) \arrow[u, "\text{inj-length-right}"] \arrow[r, "\text{to-seqn(f)}"'] & (0, \max(a_1, b_1), \dots, \max(a_n, b_n), \dots, b_m) \end{tikzcd}$$
mcase (f : a G-> c) (g : b G-> c)	• $n = m$ $$\begin{tikzcd} & (\max(a_1, b_1), \dots, \max(a_n, b_n), ..., \arrow[dl, "\text{drop- nil}"'] \arrow[ddl, "F"] \arrow[dr, "G"] \arrow[r] & {} \\ (a_1, \dots, a_n) \arrow[d, "\text{to-seqn(f)}"'] & & {} \\ (c_1, \dots, c_k) & & \end{tikzcd}$$ $$\begin{tikzcd} (1, \max(a_1, b_1), \dots, \max(a_n, b_m)) \arrow[r, "\text{branch-seq F G}"] \arrow[r, bend right, "\text{to-seqn(t)}"] & {} \end{tikzcd}$$ • $n < m$ $$\begin{tikzcd} & (\max(a_1, b_1), \dots, \max(a_n, b_n), ..., \arrow[dl, "\text{drop-width} \times \text{drop-nil}"'] \arrow[dddl, bend left, "F"] \arrow[dr, bend right, "G"] \arrow[r, "\text{drop}"] & {} \\ (a_1, \dots, a_n, 0) \arrow[d, "\text{remove-right}"'] & & {} \\ (a_1, \dots, a_n) \arrow[d, "\text{to-seqn f}"'] & & \\ (c_1, \dots, c_k) & & \end{tikzcd}$$ $$\begin{tikzcd} (1, \max(a_1, b_1), \dots, \max(a_n, b_n), ..., b_m)) \arrow[r, "\text{branch-seq F G}"] \arrow[r, bend right, "\text{to-seqn(t)}"] & {} \end{tikzcd}$$
<-left a b	$$\begin{tikzcd} & (a_1, \dots, a_n, 0) \arrow[dr, "\text{remove-right}"] & \\ (a_1, \dots, a_n, b_1, \dots, b_m) \arrow[ur, "\text{id} \times \text{drop-nil}"] \arrow[rr, "\text{to-seqn(t)}"'] & & (a_1, ..., a_n) \end{tikzcd}$$
<-right a b	$$\begin{tikzcd} & (0, b_1, \dots, b_m) \arrow[dr, "\text{remove-left}"] & \\ (a_1, \dots, a_n, b_1, \dots, b_m) \arrow[ur, "\text{drop-nil} \times \text{id}"] \arrow[rr, "\text{to-seqn(t)}"'] & & (b_1, ..., b_m) \end{tikzcd}$$
pair (f : a G-> b) (g : a G-> C)	$$\begin{tikzcd} & (a_1, \dots, a_n, a_1, \dots, a_n) \arrow[dr, "\text{(to-seq f)} \times \text{(to-seq g)}"] & \\ (a_1, \dots, a_n) \arrow[ur, "\text{fork-seq}"] \arrow[rr, "\text{to-seqn (f)}"'] & & (b_1, \dots, b_m, c_1, \dots, \end{tikzcd}$$
distribute : (prod (a (coprod b c))) G-> (coprod (prod a b) (prod a c))	$$\begin{tikzcd} (a_1, \dots, a_n, 1, \max(b_1, c_1), \dots) \arrow[r, "\text{shift-front}"] \arrow[r, bend right, "\text{to-seqn(t)}"'] & (1, a_1, \dots, a_n, \max(b_1, c_1), \dots \end{tikzcd}$$
nat-add	$$\begin{tikzcd} (n, n) \arrow[r, "\text{seqn-add}"] \arrow[r, bend right, "\text{to-seqn(t)}"'] & (n) \end{tikzcd}$$
nat-const n m	$$\begin{tikzcd} (0) \arrow[r, "\text{seqn-nat n m}"] \arrow[r, bend right, "\text{to-seqn(t)}"'] & (n) \end{tikzcd}$$

nat-inj n	$\begin{tikzcd} (n) \arrow[r, "\text{inj-size}"] \arrow[r, bend right, "\text{to-seqn(t)}"'] & (n+1) \end{tikzcd}$
nat-concat n m	$\begin{tikzcd} (n,m) \arrow[r, "\text{seqn-concat}"] \arrow[r, bend right, "\text{to-seqn(t)}"'] & (n+m) \end{tikzcd}$
one-bit-to-bool	$\begin{tikzcd} (1) \arrow[r, "\text{inj-length-left}"] \arrow[r, bend right, "\text{to-seqn(t)}"'] & (1,0) \end{tikzcd}$
nat-decompose	$\begin{tikzcd} (1+n) \arrow[r, "\text{seqn-decompose}"] \arrow[r, bend right, "\text{to-seqn(t)}"'] & (1,n) \end{tikzcd}$
nat-eq n	$\begin{tikzcd} (n,n) \arrow[r, "\text{seqn-eq}"] \arrow[r, bend right, "\text{to-seq(t)}"'] & (1,0) \end{tikzcd}$

3.3. Seqℕ → VampIR.

Finally, we need to showcase how a Seqℕ morphism is interpreted in VampIR by producing actual code. We break this up into several parts: first, we showcase what sort of tuples get computed in VampIR backend on valid inputs using the to-vampir function and then use those tuples to formulate an actual VampIR function with constraints whose inputs will be variables we feed into our function.

Remark 16 *We do not remove the inputs which are* 0*-bit long and instead produce* 0*-es whenever we land there. These entries will be later removed.*

We define to-vampir taking in a Seqℕ morphism $t : (a_1, \ldots, a_n) \to (b_1, \ldots, b_m)$ and n-long list of inputs where the i-th input is of bit-range a_i producing an m-tuple in VampIR where the i-th entry is of bit-size b_i.
We proceed by induction on a Seqℕ morphism t.

Seqℕ	Inputs	Inductive Hypotheses	Output
id (a1,...,an)	(x1,...,xn)	None	(x1,...,xn)
composition (f : (b1,...,bm) S-> (c1,...,ck)) (g : (a1,...,an) S-> (b1,...,bm))	(x1,...,xn)	to-vampir (g) (x1,...,xn) = (g1,..,gm)	to-vampir (f) (g1,...,gm)
parallel-seq (f : (a1,...,an) S-> (b1,...,bm)) (g : (c1,...,ck) S-> (d1,...,di))	x1,...,x(n+m)	to-vampir (f) (x1,...,xn) = (f1,....,fn) to-vampir (g) (x(n+1),...,x(n+m)) = (g1,....,gm)	(f1,...,fn,g1,...,gm)
branch-seq (f : (a1,...,an) S-> (b1,...,bm)) (g : (a1,...,an) S-> (b1,...,bm))	(x0,...,xn)	to-vampir (f) (x1,...,xn) = (f1,...,fm) to-vampir(g)(x1,...,xn) = (g1,...,gm)	(((1-x0) * f1) + (x0 * g1), ...,((1-x0) * fm) + (x0 * gm)
fork-seq (a1,...,an)	(x1,...,xn)	None	(x1,...,xn,x1,...,xn)
drop-nil (a1,...,an)	(x1,...,xn)	None	(0)
remove-right (a1,...,an)	(x1,...,xn,x(n+1))	None	(x1,...,xn)
remove-left (a1,...,an)	(x1,...,xn,x(n+1))	None	(x2,...,x(n+1))
inj-length-left (a1,...,an) (b1,...,bm)	(x1,...,xn)	None	(x1,...,xn,0,...,0)
inj-length-right (a1,...,an) (b1,...,bm)	(x1,...,xm)	None	(0,...,0,x1,...,xm)
inj-size n m	(x1)	None	(x1)
zero-bit	Any	None	(0)
one-bit	Any	None	(1)
shift-front (a1,...,an) k	(x1,...,xn)	None	(xk,x1,...,x(k-1),x(k+1),...,x
seqn-add n	(x1,x2)	None	(plus_range n x1 x2)
seqn-substract n	(x1,x2)	None	(minus-range n x1 x2)
seqn-multiply n	(x1,x2)	None	(mult_range n x1 x2)
seqn-divide n	(x1,x2)	None	(x1 / x2)

DOI: 10.5281/zenodo.8262747

```
def name x1 ... xn = {
    range a1 x1;
    ...
    range an xn;
    X }
```

seqn-nat n m	Any	None	(m)
seqn-concat n m	(x1,x2)	None	((x1 * (2 m̂)) + x2)
seqn-decompose n	(x1)	None	(n_th (n - 1) (range_n n x1), combine (drop_ith (range n n x
seqn-eq n	(x1,x2)	None	(iszero (x1 - x2), 0)
seqn-lt n	(x1,x2)	None	(negative n (x1 - x2), 0)

We define to-circuit as a function that takes a term t : (a1,...,an) S-> (b1,...,bm) , along with a name name , and produces a VampIR circuit as follows.
The expression X above is an abbreviation for the function to-vampir f (x1,...,xn) with any output of 0-bit length removed.

Remark 17 *Some functions, such as drop-width do not have interpretations that match their intended semantics. However, due to the nature of the compilation, whenever we compile starting with* Lambda*, we never, e.g., drop integers into a smaller bit-width, hence making any operations on the outputs superfluous. The report also skips implemented optimization steps.*

4. Future work

The current presentation of Geb merely signifies the project's initial stage. A principal objective is to enhance Geb to support not just Lambda, but richer type systems such as dependently typed languages. Moreover, the final version ought to be able to represent the current pipeline internally. We plan to run benchmarks against the Juvix-VampIR pipeline, as detailed in[3], using this existing implementation.

Acknowledgements

The fundamental theories and conjectures related to Geb are attributed to Terence Rokop who is also the author of the Idris implementation of Geb. The Lisp implementation reference was initially introduced by Jeremy Ornelas, the current repository maintainer, and later expanded upon by the first author. Anthony Hart contributed to the implementation by correcting and adding an alternative pipeline to the original version of Geb without explicit natural numbers. Finally, the Lambda language showcased herein, along with enhancements to typechecking, resulted from a collaboration with the Juvix team.

References

1. Lambek, J. and Scott, P.J. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, USA, 1986. ISBN 0521246652
2. Elliott, C. "Compiling to Categories." *Proc ACM Program Lang*, 2017. **1**(ICFP). doi:10.1145/3110271
3. Czajka, L. "Juvix to VampIR Pipeline." 2023. doi:10.5281/zenodo.8247642
4. Kapulkin, K. and Lumsdaine, P.L. "The homotopy theory of type theories." *Advances in Mathematics*, 2018. **337**:1–38. doi:https://doi.org/10.1016/j.aim.2018.08.003
5. Adamek, J. and Rosicky, J. *Locally Presentable and Accessible Categories*. London Mathematical Society Lecture Note Series. Cambridge University Press, 1994. doi:10.1017/CBO9780511600579
6. Team, T.A.D. "Agda 2.6.3 documentation." 2023
7. AG, H. "Geb Lisp Implementation." 2023
8. AG, H. "VampIR Rust Implementation." 2023
9. AG, H. "Juvix Haskell Compiler." 2023

A. Interactive Lisp API

There are several resources that are not utilized directly in the pipeline, yet are heavily used in testing, debugging, and the interactive environment of the Lisp implementation. This section is devoted to three main features that are important for anyone interested in interacting with the code and further development.

A.1. Interpreters. Geb and Seqℕ are defined as categories and as such what they possess is a specification of morphisms but not their evaluations. For example, Geb can specify a morphism `nat-add 2` yet does not have a straightforward way of being evaluated on inputs.
The implementation introduces an interpreter, namely a generic function `gapply`. It takes in a morphism from Geb or Seqℕ alongside an appropriate input and spits out the result of applying the morphism to it as if evaluating a function. `gapply` on Geb takes in a list of "terms", `so1` as a term of the terminal object, `left t` for a left inclusion of some term into a coproduct, `right t` for a similar right inclusion, and natural numbers as terms of natural number types. Product terms are generated via a fed-in list.
For example, we can evaluate the compiled Lambda code, recalling that there is an extra `so1` appearing in the domain:

- Evaluating the not function in the left part of bool gives the right part.

  ```
  CL-USER> (in-package geb.lambda.trans)
  TRANS> (gapply (to-cat nil
                  (lamb (list (coprod so1 so1))
                        (case-on (index 0)
                                 (right so1 (index 0))
                                 (left so1 (index 0)))))
          (list (geb:left so1) so1))
  (right s-1)
  ```

-
  ```
  TRANS> (gapply (to-cat nil
                  (lamb (list (coprod so1 so1))
                        (case-on (index 0)
                                 (right so1 (index 0))
                                 (left so1 (index 0)))))
          (list (geb:left so1) so1))
  (left s-1)
  ```

- Evaluating the +1 function at 1 gives 2

  ```
  TRANS> (gapply
           (to-cat nil
                   (lamb
                      (list
                          (nat-width 16))
                      (plus (index 0)
                            (bit-choice #*0000000000000001))))
                 (list 1 so1))
  2
  ```

- Evaluating the +1 function at 1 gives 2

  ```
  TRANS> (gapply (to-cat
                    nil
                    (lamb (list (nat-width 16))
                          (plus (index 0)
                               (bit-choice #*0000000000000001))))
                  (list 2 so1))
  3
  TRANS> (gapply (to-cat
                    nil
                    (lamb (list (nat-width 16))
                          (plus (index 0)
                                (bit-choice #*0000000000000001))))
                 (list 3 so1))
  4
  ```

```
CL-USER> (in-package geb)
GEB> (well-defp-cat (comp so0 so0))
T
GEB> (well-defp-cat (comp so0 so1))
; Evaluation aborted on #<SIMPLE-ERROR "(Co)Domains
; do not match for ~A" {1007FE8183}>.

CL-USER> (in-package geb.lambda.trans)
TRANS> (to-circuit (app (unit) (list (unit))) :name)
; Evaluation aborted on #<SIMPLE-ERROR "not a
; well-defined ~A in said ~A" {1009E8D9B3}>.
```

- As mentioned, gapply also works on Seqℕ terms. Here, the inputs should always be lists of natural numbers.

```
TRANS> (gapply
          (to-seqn
            (lamb
              (list (nat-width 16))
              (plus (index 0)
                    (bit-choice #*0000000000000001))))
          (list 2 0))
(3)
```

A.2. Type-checking. The way in which Geb and Seqℕ are defined in Lisp allows the formation of many ill-typed terms. For example, comp f g can be computed in the implementation even if f and g have mismatching (co)domains.
Although implementing the pipeline in a language such as Haskell will solve this problem, for debugging purposes, we still have a generic well-defp-cat, that can be applied to both Geb and Seqℕ, morphisms checking whether they are well defined and telling the user the first mismatch it notices.
Similarly, we have the usual type-checker for Lambda named well-defp, working as just a predicate in the usual manner of STLC type-checkers. It is actually used in the pipeline during compiling from Lambda producing a compilation error if the term being fed-in is mistyped.

A.3. Visualizer. One of the advantages that categorical semantics provides to programmers is the ability to canonically reason using diagrams rather than pure syntax. Jeremy Ornelas has introduced a visualizer for Geb, which uses our semantic structure to user's advantage.
The visualizer can take in any Geb term and produce a representation of it in diagram form, allowing for readability and providing great debugging features which were actively used in the early stages of the project.
The visualization uses usual notation for category-theoretic constructions other than for universal morphism from/to (co)products, which are presented as separate arrows showcasing the casing/pairing involved.
After getting a term, the visualizer then produces a separate window with an inspectable diagram.

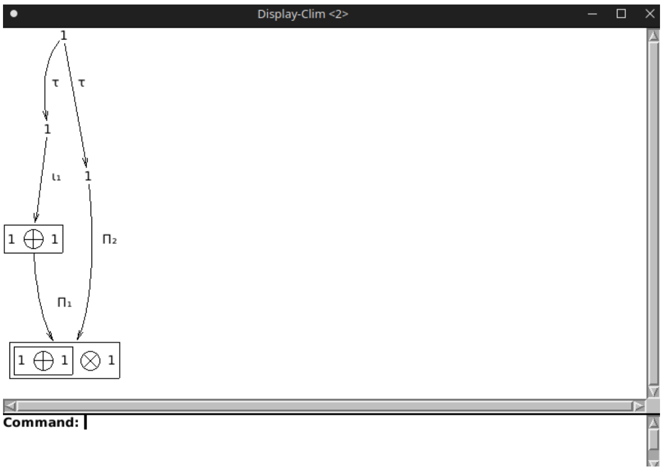

Fig. 1: Visualizer output for the term pair (left so1 (unit))

```
CL-USER> (in-package geb.lambda.trans)
TRANS> (geb-gui::visualize (to-cat nil (pair (left so1 (unit))
                                               (unit))))
```