

The Core language of Juvix

Lukasz Czajka^a

^aHeliix AG

* E-Mail: lukasz@heliix.dev

Abstract

This report describes JuvixCore – a minimalistic intermediate functional language into which Juvix desugars. We provide an abstract yet precise specification of JuvixCore's syntax, evaluation semantics, and optional type system. The correspondence between this specification and the actual implementation is discussed, along with an explanation of JuvixCore's role in the Juvix compilation pipeline. Finally, we compare the language features available in JuvixCore with those in Juvix and other popular functional languages.

Keywords: Juvix; functional programming; compilers; lambda calculus

(Received July 31, 2023; Published: August 21, 2023; Version: August 24, 2023)

Contents

1	Introduction	1
2	JuvixCore specification	1
2.1	Syntax	2
2.2	Evaluation semantics	3
2.3	Type system	4
3	JuvixCore implementation	5
4	Juvix compilation pipeline	6
5	Comparison with other languages	6
	References	6

1. Introduction

Juvix¹ is an open-source functional programming language designed for writing privacy-preserving decentralised applications. Using Juvix, developers can write high-level programs which can be compiled to WASM directly, or through VampIR² to circuits for private execution with Taiga³ on Anoma⁴ or Ethereum⁵.

JuvixCore is a minimalistic intermediate functional language to which Juvix desugars. The relationship between Juvix and JuvixCore is similar to that between Haskell and Haskell Core. After parsing, scoping and type-checking, the Juvix front-end program representation is translated to JuvixCore for further processing. Via different backends, JuvixCore can be compiled to several targets: Geb⁶, VampIR, WASM, native executable.

The main part of this report is a precise and abstract specification of the JuvixCore language (Section 2), including the evaluation semantics and the optional type system. Then, we discuss the implementation of JuvixCore and its relation to the formal specification (Section 3). In Section 4, we describe the Juvix compilation pipeline and the role of JuvixCore in it. Finally, Section 5 compares JuvixCore language features with those of Juvix and other popular functional languages.

2. JuvixCore specification

In this section, we provide a precise and abstract specification of JuvixCore. We specify the syntax, evaluation semantics and the current optional type system.

¹<https://github.com/anoma/juvix>

²<https://github.com/anoma/vamp-ir>

³<https://github.com/anoma/taiga>

⁴<https://anoma.net>

⁵<https://ethereum.org>

⁶<https://github.com/anoma/geb>

2.1. Syntax. A JuvixCore *program* \mathcal{P} is a tuple $(f_m, \mathcal{F}, \mathcal{T}, \mathcal{I})$ where:

- f_m is the main function symbol,
- \mathcal{F} is a mapping from function symbols to closed terms that associates function symbols with corresponding function bodies,
- \mathcal{T} is a mapping from function symbols to types that associates function symbols with the types of the corresponding functions,
- \mathcal{I} is a mapping from type symbols to inductive types.

An *inductive type* is pair (τ_I, \mathcal{C}) where:

- τ_I is a type - the *arity* of the inductive type,
- \mathcal{C} is a nonempty finite set of constructor declarations $c_i : \tau_i$ where c_i is a *constructor* and τ_i is its type.

The constructors are assumed to be unique and associated with exactly one inductive type. For brevity, we will often confuse inductive types with their corresponding type symbols. We write $c \in I$ or $(c : \tau) \in I$ to indicate that c (of type τ) is a constructor in the inductive type I .

Terms t, s, r are defined by the following grammar. The *types* τ, σ are just arbitrary terms.

t, s, r, τ, σ	$::=$	x
		f
		C
		S
		$\text{op}(t_1, \dots, t_n)$
		$c(t_1, \dots, t_n)$
		tt'
		$\lambda x : \tau. t$
		$\text{let } x : \tau := t \text{ in } t'$
		$\text{letrec } \{x_1 : \tau_1 := t_1; \dots; x_k : \tau_k := t_k\} \text{ in } t'$
		$\text{case } t \text{ of } \{c_1(x_1, \dots, x_{n_1}) \Rightarrow t_1; \dots; c_k(x_1, \dots, x_{n_k}) \Rightarrow t_k; _ \Rightarrow t'\}$
		$\epsilon[\tau]$
		$\Pi x : \tau. \tau'$
		Type_n
		$I(t_1, \dots, t_n)$
		Int
		String
		★

We explain the above grammar point by point.

- x is a variable.
- f is a function symbol.
- C is an integer constant, e.g., 1, 20, -5.
- S is a string constant, e.g., "abc", "hello world".
- $\text{op}(t_1, \dots, t_n)$ is a built-in operation application. Available built-in operations op :
 - arithmetic operations on integers: +, -, ·, ÷, mod,
 - integer comparisons: <, ≤,
 - equality: =,
 - string operations: show, concat, strToInt,
 - lazy sequencing: seq,
 - debugging operations: trace, fail.
- $c(t_1, \dots, t_n)$ is a constructor application.
- tt' is an application of t to t' .
- $\lambda x : \tau. t$ is a *lambda-abstraction* (anonymous function).

- $\text{let } x : \tau := t \text{ in } t'$ is a non-recursive *let-expression*. The variable x is bound in t' but not in t or τ .
- $\text{letrec } \{x_1 : \tau_1 := t_1; \dots; x_k : \tau_k := t_k\} \text{ in } t'$ is a *letrec-expression*, or a recursive let-expression. The variables x_1, \dots, x_k are bound in t_1, \dots, t_k, t' , but not in τ_1, \dots, τ_k .
- $\text{case } t \text{ of } \{c_1(x_1, \dots, x_{n_1}) \Rightarrow t_1; \dots; c_k(x_1, \dots, x_{n_k}) \Rightarrow t_k; _ \Rightarrow t'\}$ is a *case-expression*. The c_1, \dots, c_k are constructors of the same inductive type I , and n_i is the number of arguments of c_i . The last clause $_ \Rightarrow t'$ is the an optional *default clause*.
- $\epsilon[\tau]$ is an error node of type τ . Evaluating $\epsilon[\tau]$ results in an error.
- $\Pi x : \tau. \tau'$ is a *dependent function type*. We use the notation $\tau \rightarrow \tau'$ when $x \notin \text{FV}(\tau')$.
- Type_n is a *universe* for $n \in \mathbb{N}$. We often drop the subscript in Type_0 , denoting it by Type .
- $I(t_1, \dots, t_n)$ is an inductive type application. The t_1, \dots, t_n are the parameters of the inductive type I . The number and the types of parameters are determined by the arity of I .
- Int is the primitive type of integers.
- String is the primitive type of strings.
- \star is the dynamic type which can be assigned to any term. This enables the implementation of gradual typing in JuvixCore. See [Siek \(2014\)](#).

We omit the standard definition of the set $\text{FV}(t)$ of variables free in t . We treat terms up to α -conversion. For brevity, we use vector and telescope notation, e.g., we write $\Pi \vec{\alpha} : \vec{\tau}. \sigma$ for $\Pi \alpha_1 : \tau_1 \dots \Pi \alpha_n : \tau_n. \sigma$, and $\Pi \vec{\alpha} : \text{Type}. \tau$ for $\Pi \alpha_1 : \text{Type} \dots \Pi \alpha_n : \text{Type}. \tau$, and $\vec{\tau} \rightarrow \sigma$ for $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \sigma$, and \vec{t} for t_1, \dots, t_n or $t_1 \dots t_n$ depending on the context. By $|\vec{t}|$ we denote the length of the vector \vec{t} .

2.2. Evaluation semantics. Values $v \in \mathcal{V}$ are defined by the following grammar, where t is an arbitrary term. Environments E are finite partial mappings from variables to values.

$$\begin{array}{lcl}
 v \in \mathcal{V} & ::= & C \\
 & | & S \\
 & | & c(v_1, \dots, v_n) \\
 & | & \langle E; t \rangle \\
 & | & \text{Type}_n \mid \text{Int} \mid \text{String} \mid \star \\
 & | & I(v_1, \dots, v_n)
 \end{array}$$

We explain the above grammar point by point.

- C is an integer constant.
- S is a string constant.
- $c(v_1, \dots, v_n)$ is a constructor application with value arguments.
- $\langle E; t \rangle$ is a *closure*. The environment E is required to be *compatible* with t , meaning that $\text{FV}(t) \subseteq \text{dom}(E)$.
- Type_n is a universe and Int , String , \star are types.
- $I(v_1, \dots, v_n)$ is an inductive type application.

A value v can be mapped injectively to a term v^* as follows:

- $C^* = C$,
- $S^* = S$,
- $c(v_1, \dots, v_n)^* = c(v_1^*, \dots, v_n^*)$,
- $\langle E; t \rangle^* = E^*(t)$ where E^* is the homomorphic extension of the mapping $x \mapsto E(x)^*$, avoiding variable capture,
- $\text{Type}_n^* = \text{Type}_n$, $\text{Int}^* = \text{Int}$, $\text{String}^* = \text{String}$, $\star^* = \star$,
- $I(v_1, \dots, v_n)^* = I(v_1^*, \dots, v_n^*)$.

We define the evaluation relation $t \Rightarrow_E r$ in the style of big-step operational semantics (see Nipkow and Klein (2014)), where t is a term, E is an environment compatible with t , and $r \in \mathcal{V} \uplus \{\perp\}$ is either a value v or an error \perp . The evaluation relation is implicitly parameterised by a fixed JuvixCore program $\mathcal{P} = (f_m, \mathcal{F}, \mathcal{T}, \mathcal{I})$. The evaluation strategy is eager (call-by-value).

Evaluation rules:

$$\begin{array}{c}
\frac{}{x \Rightarrow_E E(x)} \quad \frac{}{C \Rightarrow_E C} \quad \frac{}{S \Rightarrow_E S} \quad \frac{\mathcal{F}(f) \Rightarrow_{\emptyset} r}{f \Rightarrow_E r} \\
\\
\frac{t_i \Rightarrow_E v_i}{\text{op}(t_1, \dots, t_n) \Rightarrow_E v} \quad \text{OP} \quad \frac{t_i \Rightarrow_E v_i}{c(t_1, \dots, t_n) \Rightarrow_E c(v_1, \dots, v_n)} \\
\\
\frac{t_1 \Rightarrow_E \langle E'; \lambda x.t \rangle \quad t_2 \Rightarrow_E v \quad t \Rightarrow_{E'[x:=v]} v'}{t_1 t_2 \Rightarrow_E v'} \quad \frac{}{\lambda x.t \Rightarrow_E \langle E'; \lambda x.t \rangle} \\
\\
\frac{t \Rightarrow_E v \quad t' \Rightarrow_{E[x:=v]} v'}{\text{let } x : \tau := t \text{ in } t' \Rightarrow_E v'} \quad \frac{t'_i \Rightarrow_{E'} v_i \quad t' \Rightarrow_{E[x:=v]} v'}{\text{letrec } \{x_1 : \tau_1 := t_1; \dots; x_k : \tau_k := t_k\} \text{ in } t' \Rightarrow_E v'} \quad \text{LR} \\
\\
\frac{t \Rightarrow_E c_i(v_1, \dots, v_{n_i}) \quad t_i \Rightarrow_{E[x_j:=v_j]_{j=1 \dots n_i}} v'}{\text{case } t \text{ of } \{c_1(x_1, \dots, x_{n_1}) \Rightarrow t_1; \dots; c_k(x_1, \dots, x_{n_k}) \Rightarrow t_k; _ \Rightarrow t'\} \Rightarrow_E v'} \\
\\
\frac{t \Rightarrow_E c(v_1, \dots, v_n) \quad t' \Rightarrow_E v' \quad c \notin \{c_1, \dots, c_k\}}{\text{case } t \text{ of } \{c_1(x_1, \dots, x_{n_1}) \Rightarrow t_1; \dots; c_k(x_1, \dots, x_{n_k}) \Rightarrow t_k; _ \Rightarrow t'\} \Rightarrow_E v'} \\
\\
\frac{}{\text{Type}_n \Rightarrow_E \text{Type}_n} \quad \frac{}{\text{Int} \Rightarrow_E \text{Int}} \quad \frac{}{\text{String} \Rightarrow_E \text{String}} \quad \frac{}{\star \Rightarrow_E \star} \\
\\
\frac{}{\Pi x : \tau. \tau' \Rightarrow_E \langle E'; \Pi x : \tau. \tau' \rangle} \quad \frac{t_i \Rightarrow_E v_i}{I(t_1, \dots, t_n) \Rightarrow_E I(v_1, \dots, v_n)} \\
\\
\frac{}{\epsilon[\tau] \Rightarrow_E \perp} \quad \frac{\text{no other rule applies}}{t \Rightarrow_E \perp}
\end{array}$$

Additional requirements:

- Rule OP: n is the arity of the operation op , the types of the values v_1, \dots, v_n match the particular operation, and v is the expected result. For example, the instantiation of this rule with $\text{op} = +$ is:

$$\frac{t_1 \Rightarrow_E C_1 \quad t_2 \Rightarrow_E C_2}{+(t_1, t_2) \Rightarrow_E C_1 + C_2} \quad \text{OP} +$$

- Rule LR:

- $t'_i = t_i[x_j \star / x_j]_{j=1, \dots, k}$,
- $E'(x_i) = \langle E'; \lambda _ . t'_i \rangle$,
- $E'(y) = E(y)$ for $y \notin \{x_1, \dots, x_k\}$.

Note that E' is not a finite object – its definition is not well-founded. Formally, one would define E' using coinduction. To avoid excessive technicalities, we refrain from elaborating on this point any further. The above specification of E' is clear enough for our purposes.

The point with changing x_i to $x_i \star$ and t_i in $E'(x_i)$ to $\lambda _ . t'_i$, is to delay the evaluation of t_i in a closure, so that it can be used with other rules. For example, consider $t = \text{letrec } x := +(3, 4); y := x \text{ in } y$. If we defined $E'(x) = \langle E'; +(3, 4) \rangle$, we would get $x \Rightarrow_{E'} \langle E'; +(3, 4) \rangle$ and since $v_y = \langle E'; +(3, 4) \rangle$ is already a value, that would become the result of evaluating t (which is the result of evaluating y in $E[y := v_y, x := \dots]$). With our approach we take $E'(x) = \langle E'; \lambda _ . +(3, 4) \rangle$, and we have $x \star \Rightarrow_{E'} 7$ according to the rules.

2.3. Type system. JuvixCore does not specify a single type system by itself. Instead, different type systems can be implemented on top of JuvixCore. Evaluation does not depend on type information. All type annotations can be set to \star to represent an untyped program.

Currently, programs translated from Juvix to JuvixCore are all well-typed in a polymorphic type system specified by the rules below. This type system is based on Church-style System F (the polymorphic lambda calculus $\lambda 2$). See (Barendregt, 1992, Section 5).

The typing rules are with respect to a fixed JuvixCore program $\mathcal{P} = (f_m, \mathcal{F}, \mathcal{T}, \mathcal{I})$. The judgements have the form $\Gamma \vdash t : \tau$ where Γ is a set of declarations $x : \tau$ assigning types to free variables. By $\Gamma, x : \tau$ we denote $\Gamma \uplus \{x : \tau\}$ (\uplus is disjoint set sum).

Inductive types can only have parameters which are types, i.e., the arity of any inductive type I has the form $\tau_I = \text{Type} \rightarrow \dots \rightarrow \text{Type} \rightarrow \text{Type}$ with n_I arguments of type Type . Recall that $\text{Type} = \text{Type}_0$. By n_I we denote the number of parameters of I . We assume there exists a fixed inductive type Bool with two constructors true and false .

Typing rules:

$$\begin{array}{c}
\overline{\Gamma, x : \tau \vdash x : \tau} \quad \overline{\Gamma \vdash f : \mathcal{T}(f)} \quad \overline{\Gamma \vdash C : \text{Int}} \quad \overline{\Gamma \vdash S : \text{String}} \\
\\
\frac{\Gamma \vdash t_i : \tau_i}{\Gamma \vdash \text{op}(t_1, \dots, t_n) : \tau} \text{ OP} \\
\\
\frac{\Gamma \vdash \sigma_i : \text{Type} \quad \Gamma \vdash t_i : \tau_i[\vec{\sigma}/\vec{\alpha}] \quad (c : \Pi \vec{\alpha} : \text{Type}. \vec{\tau} \rightarrow I \vec{\alpha}) \in I \quad |\vec{\sigma}| = |\vec{\alpha}| = n_I}{\Gamma \vdash c(\vec{\sigma}, \vec{t}) : I \vec{\sigma}} \\
\\
\frac{\Gamma \vdash t_1 : \Pi x : \tau_1. \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2[t_2/x]} \quad \frac{\Gamma \vdash \tau_1 : \text{Type}_n \quad \Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1. t) : \Pi x : \tau_1. \tau_2} \\
\\
\frac{\Gamma \vdash t : \tau \quad \Gamma, x : \tau \vdash t' : \tau' \quad x \notin \text{FV}(\tau')}{\Gamma \vdash (\text{let } x : \tau := t \text{ in } t') : \tau'} \\
\\
\frac{\Gamma \vdash \tau_i : \text{Type}_n \quad \Gamma, x_i : \tau_i \vdash t_i : \tau_i \quad \Gamma, \vec{x} : \vec{\tau} \vdash t' : \tau' \quad x_i \notin \text{FV}(\tau')}{\Gamma \vdash (\text{letrec } \vec{x} : \vec{\tau} := \vec{t} \text{ in } t') : \tau'} \\
\\
\frac{(c_i : \Pi \vec{\alpha} : \text{Type}. \vec{\tau}^i \rightarrow I \vec{\alpha}) \in I \quad \Gamma \vdash t : I \vec{\sigma} \quad \Gamma, x_1 : \tau_1^1[\vec{\sigma}/\vec{\alpha}], \dots, x_{n_i} : \tau_{n_i}^{n_i}[\vec{\sigma}/\vec{\alpha}] \vdash t_i : \tau \quad \Gamma \vdash t' : \tau}{\Gamma \vdash (\text{case } t \text{ of } \{c_1(x_1, \dots, x_{n_1}) \Rightarrow t_1; \dots; c_k(x_1, \dots, x_{n_k}) \Rightarrow t_k; _ \Rightarrow t'\}) : \tau} \\
\\
\frac{\Gamma \vdash \tau : \text{Type}_n}{\Gamma \vdash \epsilon[\tau] : \tau} \\
\\
\frac{}{\Gamma \vdash \text{Type} : \text{Type}_1} \quad \frac{\Gamma \vdash \tau_1 : \text{Type}_n \quad \Gamma, x : \tau_1 \vdash \tau_2 : \text{Type}_m \quad (n, m) \in \{(0, 0), (1, 0)\}}{\Gamma \vdash (\Pi x : \tau_1. \tau_2) : \text{Type}} \\
\\
\frac{\Gamma \vdash \sigma_i : \text{Type}}{\Gamma \vdash I(\vec{\sigma}) : \text{Type}} \quad \overline{\Gamma \vdash \text{Int} : \text{Type}} \quad \overline{\Gamma \vdash \text{String} : \text{Type}} \quad \overline{\Gamma \vdash \star : \text{Type}}
\end{array}$$

Additional requirements:

- Rule OP: n is the arity of the operation op and the types match the particular operation, e.g., for $\text{op} = <$ we have $n = 2$, $\tau_1 = \tau_2 = \text{Int}$ and $\tau = \text{Bool}$.

3. JuvixCore implementation

The JuvixCore data structure is defined in the Juvix compiler sources in the `Juvix.Compiler.Core.Language` and `Juvix.Compiler.Core.Language.Nodes` modules. The implementation follows closely the abstract definition of terms in Section 2.1. JuvixCore programs $\mathcal{P} = (f_m, \mathcal{F}, \mathcal{T}, \mathcal{I})$, which specify function bodies and inductive type constructors, are represented by the `InfoTable` data structure from the `Juvix.Compiler.Core.Data.InfoTable` module.

In our treatment of binders we have elided the issues with renaming and variable capture, working implicitly up to α -conversion as is standard in textual presentations of lambda-calculi. In the implementation, we use de Bruijn indices to represent binders. The use of de Bruijn indices is common in implementations of dependently typed programming languages and proof assistants. The main advantage is that a de Bruijn representation enables direct manipulation of terms under binders, with overall linear time complexity for most term transformations. Alternative approaches require either repeated renaming of bound variables, substitution or abstraction of free symbols – all of these are linear time operations which when performed repeatedly while processing a single term may result in quadratic runtime. A major disadvantage is that manipulating de Bruijn indices is error-prone. We try to mitigate this by implementing high-level recursors which fold or transform JuvixCore terms while taking care of de Bruijn index adjustments under the hood.

The JuvixCore evaluator is implemented in the `Juvix.Compiler.Core.Evaluator` module. The evaluator directly implements the rules from Section 2.2, using lists to represent environments.

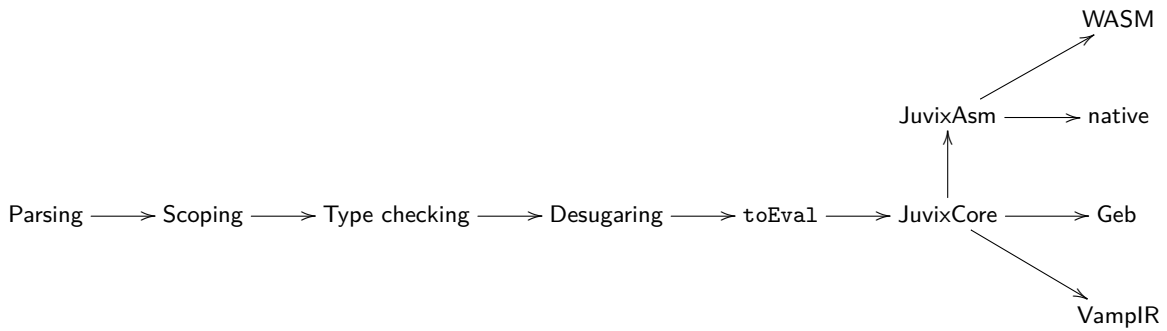
Currently, no type checker is implemented for JuvixCore. Those JuvixCore programs which are translations of Juvix front-end programs are assumed to be well-typed in the type system of Section 2.3. This is guaranteed by the desugaring process but not checked separately. The module `Juvix.Compiler.Core.Transformation.ComputeTypeInfo` implements type inference for already well-typed terms.

JuvixCore programs can be parsed and evaluated by the Juvix compiler directly, either from `*.jvc` files (`juvix dev core eval`) or via the JuvixCore REPL (`juivx dev core repl`). See the `tests/Core/positive` directory in the Juvix compiler sources for examples of `*.jvc` files and the concrete JuvixCore syntax.

4. Juvix compilation pipeline

The JuvixCore language is an intermediate language to which the Juvix front-end language desugars. There are, in fact, several different variants of JuvixCore in the actual implementation. The variant we presented in Section 2 is suitable for evaluation, with pattern matching already compiled to case-expressions. This form of the JuvixCore language corresponds to the Core data structures after performing the `toEvalTransformations` transformations (see module `Juvix.Compiler.Core.Data.TransformationId`), which is the point at which the pipelines for different backends split.

Below we present an overview of the Juvix compiler pipeline.



5. Comparison with other languages

In the table below we provide a short comparison of language features supported by Juvix, JuvixCore, Haskell and OCaml. In the case of JuvixCore, which does not specify a single type system, the “Yes” entries in the rows for polymorphism and data types mean that programs using these features can be directly represented in JuvixCore, not that type checking of such programs is performed by the current JuvixCore implementation.

Feature	Juvix	JuvixCore	Haskell	OCaml
Algebraic data types	Yes	Yes	Yes	Yes
GADTs	No	Yes	Yes	Yes
Prenex polymorphism	Yes	Yes	Yes	Yes
Higher-rank polymorphism	Some	Yes	Yes ^a	No
Hindley-Milner type inference	No	No	Yes	Yes
Type classes	No	No	Yes	No
Modules	Yes	No	Yes	Yes
Parameterised modules	No	No	No	Yes
Eager evaluation	Yes	Yes	Yes ^b	Yes
Lazy evaluation	No	No	Yes	Yes ^c
Metaprogramming	No	No	Yes ^d	Yes ^e

^awith the RankNTypes extension

^bvia strictness annotations

^cvia the Lazy.t type

^dvia Template Haskell

^evia PPXs

Acknowledgements

The author thanks the entire Juvix team, including Jonathan Prieto-Cubides, Jan Mas Rovira and Paul Cadman. The initial design and preliminary implementation of JuvixCore were by the author, but subsequent discussions with the rest of the Juvix team had a decisive impact on the final form of JuvixCore presented in this report.

References

- J. Siek. What is Gradual Typing?, 2014. URL <https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/>. (cit. on p. 3.)
- T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. (cit. on p. 4.)
- H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 118–310. Oxford University Press, 1992. (cit. on p. 4.)

Helix AG. Taiga Implementation, 2023a. URL <https://github.com/anoma/taiga/>.
Helix AG. Geb Lisp Implementation, 2023b. URL <https://github.com/anoma/geb/>.
Helix AG. VampIR Rust Implementation, 2023c. URL <https://github.com/anoma/vamp-ir/>.
Helix AG. Juvix Compiler, 2023d. URL <https://github.com/anoma/juvix/>.