Anoma Research Topics | TECHNICAL REPORT

The Core language of Juvix

Lukasz Czajka[a]

[a]Heliax AG

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

Abstract

This report describes JuvixCore – a minimalistic intermediate functional language to which Juvix desugars. We provide a precise and abstract specification of JuvixCore's syntax, evaluation semantics, and optional type system. We comment on the relationship between this specification and the actual implementation. We also explain the role JuvixCore plays 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; Language specification; Functional programming; Compilers; Lambda Calculus;

(Received July 31, 2023; Published: August 29, 2023; Version: August 29, 2023)

Contents

1. Introduction

Juvix is an open-source functional programming language designed to write privacy-preserving decentralised applications (Heliax AG, 2023a). Using Juvix, developers can write high-level programs which can be compiled to WebAssembly (WASM) directly, or to circuits via VampIR (Heliax AG, 2023b; Czajka, 2023) or Geb (Heliax AG, 2023c; Gureev and Prieto-Cubides, 2023) for private execution within Taiga[1] on Anoma[2] or Ethereum[3].

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 including Geb , VampIR, WASM, and native executable.

The main part of this report is a precise and abstract specification of the JuvixCore language in Section 2, including the evaluation semantics and the optional type system. Then in Section 3 we discuss the implementation of JuvixCore and its relation to the formal specification. The Juvix compilation process and the role JuvixCore plays in it are discussed in Section 4. Finally, Section 5 compares the features of the JuvixCore language 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/taiga
[2]https://anoma.net
[3]https://ethereum.org

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 $(\tau_I, \mathcal{C})$ 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 arbitrary terms.

$$
\begin{array}{rcl}
t, s, r, \tau, \sigma & ::= & x \\
& | & f \\
& | & C \\
& | & S \\
& | & \mathtt{op}(t_1, \ldots, t_n) \\
& | & c(t_1, \ldots, t_n) \\
& | & tt' \\
& | & \lambda x : \tau . t \\
& | & \mathtt{let}\, x : \tau := t\, \mathtt{in}\, t' \\
& | & \mathtt{letrec}\, \{x_1 : \tau_1 := t_1; \ldots; x_k : \tau_k := t_k\}\, \mathtt{in}\, t' \\
& | & \mathtt{case}\, t\, \mathtt{of}\, \{c_1(x_1, \ldots, x_{n_1}) \Rightarrow t_1; \ldots; c_k(x_1, \ldots, x_{n_k}) \Rightarrow t_k; \_ \Rightarrow t'\} \\
& | & \epsilon[\tau] \\
& | & \Pi x : \tau . \tau' \\
& | & \mathrm{Type}_n \\
& | & I(t_1, \ldots, t_n) \\
& | & \mathrm{Int} \\
& | & \mathrm{String} \\
& | & \bigstar
\end{array}
$$

Fig. 1: JuvixCore syntax grammar.

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"`.
- $\mathtt{op}(t_1, \ldots, t_n)$ is a built-in operation application. Available built-in operations op:
  - arithmetic operations on integers: $+$, $-$, $\cdot$, $\div$, mod,
  - integer comparisons: $<$, $\leq$,
  - equality: $=$,
  - string operations: show, concat, strToInt,
  - lazy sequencing: seq,
  - debugging operations: trace, fail.
- $c(t_1, \ldots, t_n)$ is a constructor application.

- tt' is an application of t to t'.
- $\lambda x : \tau.t$ is a *lambda-abstraction* (anonymous function).
- $\texttt{let}\, x : \tau := t\, \texttt{in}\, t'$ is a non-recursive *let-expression*. The variable x is bound in t' but not in t or τ.
- $\texttt{letrec}\, \{x_1 : \tau_1 := t_1; \ldots; x_k : \tau_k := t_k\}\, \texttt{in}\, t'$ is a *letrec-expression*, or a recursive let-expression. The variables $x_1, \ldots, x_k$ are bound in $t_1, \ldots, t_k, t'$, but not in $\tau_1, \ldots, \tau_k$.
- $\texttt{case}\, t\, \texttt{of}\, \{c_1(x_1, \ldots, x_{n_1}) \Rightarrow t_1; \ldots; c_k(x_1, \ldots, x_{n_k}) \Rightarrow t_k; \_ \Rightarrow t'\}$ is a *case-expression*. The $c_1, \ldots, c_k$ 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 $\tau \to \tau'$ when $x \notin \mathrm{FV}(\tau')$.
- Type_n is a *universe* for $n \in \mathbb{N}$. We often drop the subscript in Type_0, denoting it by Type.
- $I(t_1, \ldots, t_n)$ is an inductive type application. The $t_1, \ldots, t_n$ 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.
- $\bigstar$ 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 $\mathrm{FV}(t)$ 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 $\Pi\alpha_1 : \tau_1 \ldots \Pi\alpha_n : \tau_n.\sigma$, and $\Pi\vec{\alpha} : \mathrm{Type}.\tau$ for $\Pi\alpha_1 : \mathrm{Type} \ldots \Pi\alpha_n : \mathrm{Type}.\tau$, and $\vec{\tau} \to \sigma$ for $\tau_1 \to \ldots \to \tau_n \to \sigma$, and $\vec{t}$ for $t_1, \ldots, t_n$ or $t_1 \ldots t_n$ 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.

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

We explain the above grammar point by point.

- C is an integer constant.
- S is a string constant.
- $c(v_1, \ldots, v_n)$ 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 $\mathrm{FV}(t) \subseteq \mathrm{dom}(E)$.
- Type_n is a universe and Int, String, $\bigstar$ are types.
- $I(v_1, \ldots, v_n)$ is an inductive type application.

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

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

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 $r \in \mathcal{V} \uplus \{\bot\}$ is either a value v or an error $\bot$. 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).

$$\frac{}{x \Rightarrow_E E(x)} \quad \frac{}{C \Rightarrow_E C} \quad \frac{}{S \Rightarrow_E S} \quad \frac{\mathcal{F}(f) \Rightarrow_\emptyset r}{f \Rightarrow_E r}$$

$$\frac{t_i \Rightarrow_E v_i}{\mathtt{op}(t_1, \ldots, t_n) \Rightarrow_E v} \text{ OP} \quad \frac{t_i \Rightarrow_E v_i}{c(t_1, \ldots, t_n) \Rightarrow_E c(v_1, \ldots, v_n)}$$

$$\frac{t_1 \Rightarrow_E \langle E'; \lambda x.t \rangle \quad t_2 \Rightarrow_E v \quad t \Rightarrow_{E'[x:=v]} v'}{t_1 t_2 \Rightarrow_E v'} \quad \frac{}{\lambda x.t \Rightarrow_E \langle E; \lambda x.t \rangle}$$

$$\frac{t \Rightarrow_E v \quad t' \Rightarrow_{E[x:=v]} v'}{\mathtt{let}\, x : \tau := t\, \mathtt{in}\, t' \Rightarrow_E v'} \quad \frac{t'_i \Rightarrow_{E'} v_i \quad t' \Rightarrow_{E[\vec{x}:=\vec{v}]} v'}{\mathtt{letrec}\, \{x_1 : \tau_1 := t_1; \ldots; x_k : \tau_k := t_k\}\, \mathtt{in}\, t' \Rightarrow_E v'} \text{ LR}$$

$$\frac{t \Rightarrow_E c_i(v_1, \ldots, v_{n_i}) \quad t_i \Rightarrow_{E[x_j := v_j]_{j=1\ldots n_i}} v'}{\mathtt{case}\, t\, \mathtt{of}\, \{c_1(x_1, \ldots, x_{n_1}) \Rightarrow t_1; \ldots; c_k(x_1, \ldots, x_{n_k}) \Rightarrow t_k; \_ \Rightarrow t'\} \Rightarrow_E v'}$$

$$\frac{t \Rightarrow_E c(v_1, \ldots, v_n) \quad t' \Rightarrow_E v' \quad c \notin \{c_1, \ldots, c_k\}}{\mathtt{case}\, t\, \mathtt{of}\, \{c_1(x_1, \ldots, x_{n_1}) \Rightarrow t_1; \ldots; c_k(x_1, \ldots, x_{n_k}) \Rightarrow t_k; \_ \Rightarrow t'\} \Rightarrow_E v'}$$

$$\frac{}{\mathrm{Type}_n \Rightarrow_E \mathrm{Type}_n} \quad \frac{}{\mathrm{Int} \Rightarrow_E \mathrm{Int}} \quad \frac{}{\mathrm{String} \Rightarrow_E \mathrm{String}} \quad \frac{}{\bigstar \Rightarrow_E \bigstar}$$

$$\frac{}{\Pi x : \tau.\tau' \Rightarrow_E \langle E; \Pi x : \tau.\tau' \rangle} \quad \frac{t_i \Rightarrow_E v_i}{I(t_1, \ldots, t_n) \Rightarrow_E I(v_1, \ldots, v_n)}$$

$$\frac{}{\epsilon[\tau] \Rightarrow_E \bot} \quad \frac{\text{no other rule applies}}{t \Rightarrow_E \bot}$$

Fig. 2: JuvixCore evaluation rules.

Additional requirements:

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

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

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

Remark 1. Note that E' in the second point above 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.

Remark 2. In the second point above, the purpose of changing x_i to $x_i \bigstar$ 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 $t = \mathtt{letrec}\, x := +(3,4); y := x\, \mathtt{in}\, y$. 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 $E[y := v_y, x := \ldots]$. With our approach we take $E'(x) = \langle E'; \lambda\_.+(3,4) \rangle$, and we have $x \bigstar \Rightarrow_{E'} 7$ 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 $\bigstar$ 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 type parameters, i.e., the arity of any inductive type I has the form $\tau_I = \mathrm{Type} \to \ldots \to \mathrm{Type} \to \mathrm{Type}$ with n_I arguments of type Type. Recall that $\mathrm{Type} = \mathrm{Type}_0$. By n_I we denote the number of parameters of I. We assume that there exists a fixed inductive type Bool with two constructors `true` and `false`.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \quad \frac{}{\Gamma \vdash f : \mathcal{T}(f)} \quad \frac{}{\Gamma \vdash C : \mathrm{Int}} \quad \frac{}{\Gamma \vdash S : \mathrm{String}}$$

$$\frac{\Gamma \vdash t_i : \tau_i}{\Gamma \vdash \mathtt{op}(t_1, \ldots, t_n) : \tau} \ \mathrm{OP}$$

$$\frac{\Gamma \vdash \sigma_i : \mathrm{Type} \quad \Gamma \vdash t_i : \tau_i[\vec{\sigma}/\vec{\alpha}] \quad (c : \Pi\vec{\alpha} : \mathrm{Type}.\vec{\tau} \to I\vec{\alpha}) \in I \quad |\vec{\sigma}| = |\vec{\alpha}| = n_I}{\Gamma \vdash c(\vec{\sigma}, \vec{t}) : I\vec{\sigma}}$$

$$\frac{\Gamma \vdash t_1 : \Pi x : \tau_1.\tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2[t_2/x]} \quad \frac{\Gamma \vdash \tau_1 : \mathrm{Type}_n \quad \Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \Pi x : \tau_1.\tau_2}$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma, x : \tau \vdash t' : \tau' \quad x \notin \mathrm{FV}(\tau')}{\Gamma \vdash (\mathtt{let}\, x : \tau := t\, \mathtt{in}\, t') : \tau'}$$

$$\frac{\Gamma \vdash \tau_i : \mathrm{Type}_n \quad \Gamma, x_i : \tau_i \vdash t_i : \tau_i \quad \Gamma, \vec{x} : \vec{\tau} \vdash t' : \tau' \quad x_i \notin \mathrm{FV}(\tau')}{\Gamma \vdash (\mathtt{letrec}\, \vec{x} : \vec{\tau} := \vec{t}\, \mathtt{in}\, t') : \tau'}$$

$$\frac{\begin{array}{c}(c_i : \Pi\vec{\alpha} : \mathrm{Type}.\vec{\tau}^i \to I\vec{\alpha}) \in I \\ \Gamma \vdash t : I\vec{\sigma} \quad \Gamma, x_1 : \tau_1^i[\vec{\sigma}/\vec{\alpha}], \ldots, x_{n_i} : \tau_{n_i}^i\vec{\sigma}/\vec{\alpha} \vdash t_i : \tau \quad \Gamma \vdash t' : \tau\end{array}}{\Gamma \vdash (\mathtt{case}\, t\, \mathtt{of}\, \{c_1(x_1, \ldots, x_{n_1}) \Rightarrow t_1; \ldots; c_k(x_1, \ldots, x_{n_k}) \Rightarrow t_k; \_ \Rightarrow t'\}) : \tau}$$

$$\frac{\Gamma \vdash \tau : \mathrm{Type}_n}{\Gamma \vdash \epsilon[\tau] : \tau}$$

$$\frac{}{\Gamma \vdash \mathrm{Type} : \mathrm{Type}_1} \quad \frac{\Gamma \vdash \tau_1 : \mathrm{Type}_n \quad \Gamma, x : \tau_1 \vdash \tau_2 : \mathrm{Type}_m \quad (n, m) \in \{(0,0), (1,0)\}}{\Gamma \vdash (\Pi x : \tau_1.\tau_2) : \mathrm{Type}}$$

$$\frac{\Gamma \vdash \sigma_i : \mathrm{Type}}{\Gamma \vdash I(\vec{\sigma}) : \mathrm{Type}} \quad \frac{}{\Gamma \vdash \mathrm{Int} : \mathrm{Type}} \quad \frac{}{\Gamma \vdash \mathrm{String} : \mathrm{Type}} \quad \frac{}{\Gamma \vdash \bigstar : \mathrm{Type}}$$

Fig. 3: JuvixCore optional typing rules.

Additional requirements:

- Rule OP: n is the arity of the operation op and the types match the particular operation, e.g., if op is $(<)$, we have that n is 2, both τ_1 and τ_2 are Int, and τ is 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. 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.

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.

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 described in Section 2.3. This is guaranteed by the

desugaring process but not checked separately. We implement type inference for already well-typed terms in the module `Juvix.Compiler.Core.Transformation.ComputeTypeInfo`.

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 (`juvix 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 present 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 `toEval` transformations (see module `Juvix.Compiler.Core.Data.TransformationId`), which is the point at which the pipelines for different backends diverge. An overview of the Juvix compiler pipeline is depicted in Figure 4.

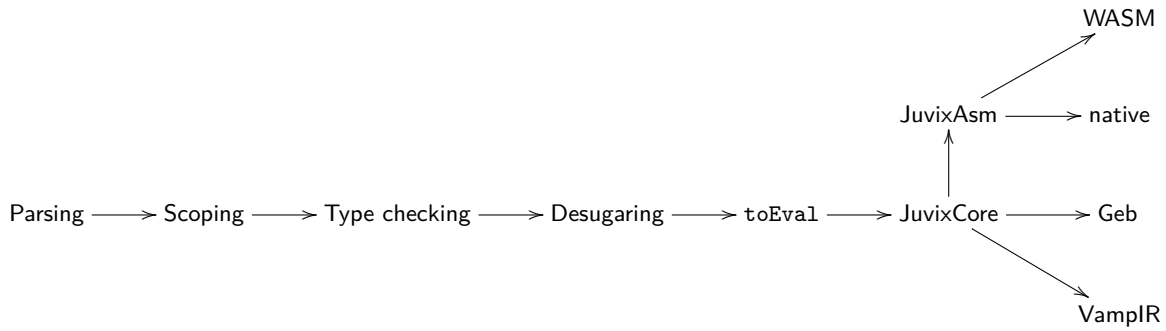

Fig. 4: Juvix compiler pipeline.

5. Comparison with other languages

We provide a short comparison of language features supported by Juvix, JuvixCore, Haskell, and OCaml in Table 1. 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.

Table 1: Comparison between language features supported by Juvix, JuvixCore, Haskell and OCaml.

Feature	Juvix	JuvixCore	Haskell	OCaml
Turing-complete	Yes[4]	Yes	Yes	Yes
Algebraic data types	Yes	Yes	Yes	Yes
GADTs	No	Yes	Yes	Yes
Prenex polymorphism	Yes	Yes	Yes	Yes
Higher-rank polymorphism	Some	Yes	Yes[5]	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[6]	Yes
Lazy evaluation	No	No	Yes	Yes[7]
Metaprogramming	No	No	Yes[8]	Yes[9]

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 done 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. The author thanks Jonathan Prieto-Cubides for reviewing this report.

[4] via `terminating` and `positive` annotations
[5] with the `RankNTypes` extension.
[6] via strictness annotations.
[7] via the `Lazy.t` type.
[8] via Template Haskell
[9] via PPXs.

References

Heliax AG. Juvix Compiler, 2023a. URL https://github.com/anoma/juvix/. (cit. on p. 1.)

Heliax AG. VampIR Rust Implementation, 2023b. URL https://github.com/anoma/vamp-ir/. (cit. on p. 1.)

Lukasz Czajka. Juvix to vampir pipeline, August 2023. URL https://doi.org/10.5281/zenodo.8246535. This document is based on Juvix v0.4.1, Geb v0.4.0, and VampIR v0.1.3. (cit. on p. 1.)

Heliax AG. Geb Lisp Implementation, 2023c. URL https://github.com/anoma/geb/. (cit. on p. 1.)

Artem Gureev and Jonathan Prieto-Cubides. Geb Pipeline. *Anoma Research Topics*, August 2023. doi:10.5281/zenodo.8262815. URL https://doi.org/10.5281/zenodo.8262815. This document is based on Geb v0.4.1. (cit. on p. 1.)

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.)

Heliax AG. Taiga Implementation, 2023d. URL https://github.com/anoma/taiga/.