Juvix — Language Reference

Christopher Goes, Marty Stumpf, Jeremy Ornelas, Andy Morris, Asher Manning

18th September 2019 - *Prerelease*

Juvix synthesises a high-level frontend syntax, dependent-linearly-typed core language, and low-level parallelisable optimally-reducing execution model into a single unified stack for writing formally verifiable, efficiently executable smart contracts which can be deployed to a variety of distributed ledgers.

Juvix's compiler architecture is purpose-built from the ground up for the particular requirements and economic trade-offs of the smart contract use case — it prioritises behavioural verifiability, semantic precision, and output code efficiency over compilation speed, syntactical familiarity, and backwards compatibility with existing blockchain virtual machines.

Machine-assisted proof search, declarative deployment tooling, type & usage inference, and alternative spatiotemporal dataflow representations facilitate integration of low-developer-overhead property verification into the development process. An interchain abstraction layer representing ledgers as first-class objects enables seamless cross-chain programming and type-safe runtime reconfiguration.

This document is designed to be a first-principles explanation of Juvix. No familiarity with the theoretical background is assumed. Readers previously acquainted with the lambda calculus, sequent calculus, simply-typed lambda calculus, the calculus of constructions, linear logic, interaction nets, elementary affine logic, and Lamping's optimal reduction algorithm may skip the associated subsections in chapter five.

Contents

1 Motivation

Selected out by the twin Darwinian reapers of language network-effect path-dependence and latency-over-correctness content delivery incentives, secure, legible, large-scale, long-running digital systems are a virtually nonexistent breed. Cutting corners on security reduces costs up front, but the total costs of insecure systems are higher, borne later and unevenly — often by individuals who end up rendered vulnerable instead of empowered by poorly engineered technology. Although the underlying cryptographic primitives can in complexity-theoretic principle provide a high degree of individual protection, the imprecisions and inaccuracies in complex networked multi-party protocols have caused the economics of privacy & legibility to devolve into base power asymmetries, where even individuals possessed of relevant domain expertise stand little chance of retaining their privacy & autonomy against a nation state, surveillance capitalist firm, terrorist group, or moderately skilled bounty hunter, and mainstream users stand none at all.

This sorry result is overdetermined & difficult to precisely allocate causal responsibility for, but certainly a substantial contributor is the sheer difficulty and cost of formally verifying complex software systems. Most frequently, security is trial-and-error. At best, models are written and checked separately from the code, an expensive and error-prone process. An approach to security which can be expected to result in a different outcome must be constructive, unifying code & proofs into a single language, and must be compositional, so that sets of proofs can be imported & reused along with the libraries which comprise most of modern software. The approach must provide a typed language for succinct proofs which can tightly constrain the opaque behaviour of complex backend codebases, typecheckers which can be embedded by the manufacturers of user-facing software, such as web browsers, operating systems, or cybercoin wallets. The approach must reduce the costs of formally verifying software, and increase the legible user-facing benefits of doing so, to a degree where formal verification is the economically rational decision for networked software which handles sensitive data or the exchange of value.

Smart contracts running on distributed ledgers are an archetypal example of a security-critical application, and one where the popular conceit of security-through-obscurity holds little sway, yet results so far have not been promising [20] [4]. Luckily, the field has not yet been locked into particular technologies whose network effects could not be overcome, and the necessity of verifiable & verified software systems is widely-recognised. A radically different language is necessary: one that treats verifiability as a design problem, not a feature to be tacked on later, one that provides succinct, expressive, and composable proofs which can tightly constraint the behaviour of complex logic, and one that reduces the cost of verification to the point where not doing so for security-critical software will be considered simply irresponsible. Juvix aims to realise this ideal.

2 Typographical conventions

Throughout this document, `typewriter font` is used for code blocks. Where possible, code blocks are syntax-highlighted.

Mathematical equations are written in the standard LaTeX style — they look like $y = ax^2 + bx + c$.

Definitions of new terms, functions, or properties which will be later referenced are written in a **colored bold**.

Italic text is used for occasional focus on important or counter-intuitive properties.

3 Prior work

3.1 Dependently-typed languages

Three dependently-typed languages have seen substantial contemporary usage: Agda, Coq, and Idris [3]. The first two are focused on theorem proving rather than executable code output, and have been primarily used to verify mathematical formulae or proofs of algorithmic correctness, where the algorithms which have been verified are then implemented in or extracted to another language for execution.

Idris does intend to simultaneously support dependently-typed program verification and produce executable code output, but falls short of the requirements of wide deployment: the compilation output is not efficient enough, too much effort is required to write proofs of properties of terms, and insufficient engineering effort has been dedicated to the inclusion of optimising transformations which take advantage of the expressive typesystem (understandably so, since Idris is primarily & impressively developed by a university lecturer in his free time!). Furthermore, the economics of most standard programs running on the desktop or web favour development & execution speed over safety and correctness.

3.2 Linearly-typed languages

Linear types are included, in a somewhat limited form, in the Rust systems programming language, which utilises them to provide memory safety without runtime garbage collection overhead. No mainstream dependently-typed functional language supports linear types, although the upcoming Idris 2 will (with the same antecedent type theory as Juvix). A proposal to add linear types to Haskell is in the discussion stage.

3.3 Dependently-typed smart contracts

One prior work [14] wrote an Idris [3] backend targeting Ethereum's LLL language [6]. Juvix shares many of the goals outlined in that paper, but the approach described therein failed to take advantage of well-known optimisations such as tail-call optimisation and handicapped itself by compiling to LLL instead of directly to EVM opcodes. The effects system described may be a sensible model for smart contract programs written in Juvix (or other dependently-typed smart contract languages) but is out of scope of this paper, which focuses on language & compiler design only.

Formality [8] was a substantial inspiration for this work, particularly the low-level interaction net execution model. Juvix differs in its decisions to include a frontend language in which programmers will write directly, implement a larger core language and more complex low-level execution model, trade some simplicity in compiler architecture for output performance where the performance gains are substantial, and automate the tedious bureaucracy of elementary affine logic box placement. In the future Juvix may support Formality as a frontend language.

4 Reasoning

What changes are made from these prior systems, and why might one expect the solutions proposed to realise the criteria outlined previously?

Juvix is efficiently executable in machine time through a novel execution model & an expressive typesystem permitting aggressive optimisation and is efficiently verifiable in developer time through proof bureaucracy automation & composable verification. The rigour of the type system enables composition of contracts to a degree of complexity not possible with less precise languages, and the ability to deploy to multiple ledgers provides flexibility to and reduces development costs for language users.

4.1 Machine-time-efficient execution

4.1.1 Optimal, parallelisable, higher-order-friendly evaluation

Juvix utilises a fundamentally different evaluation model (as compared to present functional programming languages), based on recent theoretical advances in optimal lambda calculus reduction using interaction nets, which avoids any unnecessary duplication of reducible sub-expressions, parallelises by default, dynamically fuses composable terms at runtime, and handles higher-order functions & lexical closures efficiently without garbage collection. To take maximal advantage of this evaluation model, Juvix translates high-level algebraic datatypes into pure lambda calculus representations. In cases where optimal reduction requires too much bookkeeping or imposes undesired overhead, Juvix compiles subterms directly into rewrite rules which still enjoy the native parallelism and strong confluence properties of the interaction net model.

4.1.2 Linear dependent types obviate garbage collection and ensure type erasure

The core type theory of Juvix combines linear & dependent types, extending prior research into the combination of the two paradigms with additional linear connectives & pragmatic essentials and instantiating usage quantisation over the natural numbers to provide maximally precise accounting. Dependent types enable the language to verify properties of its own terms in succinct proofs and open up a wide arena of compiler optimisations. Linear types obviate the need for garbage collection in both the optimal reduction & alternative direct subterm compilation paths, facilitate aggressive imperative optimisation transformations, and ensure that dependent types used to enforce properties but not needed at runtime are always erased by the compiler.

4.1.3 Discrete-cost optimisation

Purpose-built for the smart contract use case, Juvix's optimiser requires a discrete instruction cost model of the underlying machine (likely a distributed ledger) which it can utilise to search through semantically equivalent instruction sequences and select the lowest by cost.

4.1.4 Proofs become optimisations

The dependent type system of Juvix Core enables it to express arbitrary properties of its own terms, including equality of functions — proofs of which can be utilised by the optimiser to select between reducible expressions known to be semantically equivalent at compile time.

4.1.5 Hint & bypass compiler when necessary

Primitives are provided to allow developers to bypass the usual compilation pipeline and construct hand-optimised rewrite rules specialised to the underlying machine, optionally proving equivalence of their hand-optimised rewrite rules to the compiler-generates ones using a formalised interpreter for the machine model.

4.2 Developer-time-efficient formal proof construction

4.2.1 Proof-generation bureaucracy automation

Juvix's high-level syntax, compiler, and REPL utilise multiple tactics to minimise the bureaucracy of writing formal proofs of properties of terms. Generalised assisted graph search automates construction of proofs locatable in constrained search spaces, holes enable the developer to type, assert, and prototype now, then prove later when the model is finalised. Step-through tactics illuminate the inner de-sugaring & typechecking steps of the compiler to provide introspection & legibility.

4.2.2 Composable proof ecosystem

As proofs of properties of Juvix terms are simply terms themselves, proofs can be exported & imported from libraries along with the code which will actually be executed. Proof interfaces for common data structures allow swapping out backend components of a higher-level module while retaining verified properties.

4.3 Expanding the frontier of possible complexity

4.3.1 Raise the threshold of possible complexity

Compound smart contract applications constructed using less precise languages invariably hit abstraction limits, where the complexity of writing safe code and verifying safe interoperation scales with the size of the codebase for any individual line, so quadratically in total. The rigour of Juvix's type system, where simple properties of arbitrarily complex terms can be succinctly expressed, enables more complex multi-contract interacting systems to be safely constructed & operated. Juvix can typecheck across contracts and verify compound properties of multi-contract systems. Further integration of the language & compiler into the state machine allows contracts to enforce types & properties of their callers or callees, lets contracts be safely upgraded with new versions which can prove that they satisfy the same properties, and empowers the state machine to safely optimise across contracts in many-contract systems.

4.4 Cross-ledger targeting

The Juvix frontend & core languages are independent of any particular machine-level architecture and can target a variety of models from WASM to the Ethereum Virtual Machine to FPGAs. Sharing all or most of the logic of a smart contract system across ledgers provides flexibility to developers, reduces platform lock-in to particular ledger tool-chain stacks, and reduces development costs of multi-ledger solutions.

5 Theoretical background

This section provides a comprehensive theoretical background which should be sufficient prerequisite for comprehension of the remainder of this language reference. Readers with prior domain experience may skip the appropriate sections.

> Note: this section is a work in progress; it will be finished prior to an official release but perhaps not before then.

5.1 Lambda calculus

5.1.1 Basics

- Define terms $t ::= x \mid \lambda x.t \mid tt$.
- Define the set of free variables of a term t, $FV(t)$, as all referenced but not bound variables in t.
- Define β-reduction as: $(\lambda x.t_1)t_2$ = $t_1[x := t_2]$ (capture-avoiding substitution).
- Two terms are α-equivalent if they are equal up to renaming of bound variables.
- β-reduction is confluent modulo α-equivalence.
- Define η-conversion as $\lambda x.fx$ = f iff. $x \notin FV(f)$.
- de Brujin indices: numbers for bound variables. $\lambda x.x$ = $\lambda.1$, $\lambda x.\lambda y.x$ = $\lambda.2$, etc.

5.1.2 Universality

- Lambda calculus can express non-terminating computations, e.g. $(\lambda x.xx)(\lambda x.xx)$.
- Lambda calculus is Turing-complete.

5.1.3 Efficiency

Beta-reduction is nontrivial, must search for & replace all instances of variable being substituted for.

Choice of reduction strategy: given $(\lambda x.t_1)t_2$, which of t_1 or t_2 to reduce first.

- Normal order: leftmost-outermost redex reduced first (arguments substituted before reduction).
- Call-by-name: as normal order except that no reductions are performed inside abstractions.
- Call-by-value: only outermost redexes reduced, only when right-hand side has reduced to a value (variable or lambda abstraction) - reduce t_2 first, substitute in for x after reduction.
- Call-by-need: create thunk for evaluation of t_2, pass into t_1, if reduced value will be shared. Challenge: "computing under a lambda", sharing a function. Haskell et al. don't do this.

Computing under a function: $(\lambda x.\lambda y.map(\lambda z.z + x + y)zs)23$, $x + y$ should be evaluated once, GHC's STG machine does not do this (only evaluates function once all arguments are passed).

(easy to optimise if 2 & 3 are known at compile time, but that's not the case in general)

e.g. $n2Ia$ with n and 2 Church-encoded naturals, exponential in n because the applications of I will be duplicated.

Example term: $\lambda x.(x(\lambda w.w))\lambda y.(\lambda x.xx)(yz)$.

Define optimality using Levy's framework.

5.1.4 Encoding data structures

Church encoding of natural numbers:

- $n = \lambda s.\lambda z.s...nz.$
- $Z = \lambda s.\lambda z.z.$,$S = \lambda k.\lambda s.\lambda z.(s(ksz)).$
- $plus = \lambda a.\lambda b.\lambda s.\lambda z.as(bsz).$

- $mult = \lambda a.\lambda b.\lambda s.\lambda z.a(bs)z$.
- $exp = \lambda a.\lambda b.\lambda s.\lambda z.(ba)sz$.
- $pred = \lambda a.\lambda s.\lambda z.a(\lambda g.\lambda h.h(gs))(\lambda u.z)(\lambda z.u)$.

Church encoding of booleans:

- $true = \lambda t.\lambda f.t$.
- $false = \lambda t.\lambda f.f$.

Church encoding of pairs:

- $pair = \lambda x.\lambda y.\lambda z.zxy$.
- $fst = \lambda p.p(\lambda x.\lambda y.x)$.
- $snd = \lambda p.p(\lambda x.\lambda y.y)$.

Scott encoding of constructor c_i of datatype D with arity A_i:

- $\lambda x_1 ... x_A . \lambda c_1 ... c_N . c_i x_1 ... x_{A_i}$.
- Compare with Church encoding: $\lambda x_1 ... x_A . \lambda c_1 ... c_N . c_i (x_1 c_1 ... c_N) ... (x_{A_i} c_1 ... c_N)$.
- Scott-encoded datatypes are their own pattern matching functions.

5.2 Sequent calculus

Logical deduction ruleset & syntax for first-order logic.

5.3 Simply-typed lambda calculus

- Eliminate "bad" uses of lambda calculus
- No recursion, always terminates
- No polymorphism
- No guarantees on termination bounds (complexity) or copying
- Extrinsic: assigning type to lambda terms. Intrinsic: type is part of term.
- Set B of base types, set C of term constants (e.g. natural numbers).
- $\tau ::= T \mid \tau \ \rightarrow \ \tau$ with $T \in B$.
- $e ::= x \mid \lambda x : \tau . e \mid e \ e \mid c$ with $c \in C$.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ var}$$

$$\frac{c \in C}{\Gamma \vdash c : T} \text{ const}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . e) : (\tau_1 \rightarrow \tau_2)} \text{ lam}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \text{ app}$$

Figure 1: Typing rules for simply-typed lambda calculus

5.4 Calculus of constructions

- Desiderata
- Lambda cube
- Typing rules
- Examples

Define terms $t ::= T \mid P \mid x \mid t\ t \mid \lambda x : e.e \mid \forall x : e.e$. Let $K = T \mid P$. Let M, N be terms.

$$\frac{}{\Gamma \vdash P : T}$$

$$\frac{\Gamma \vdash A : K}{\Gamma, x : A \vdash x : A}\ \text{var}$$

$$\frac{\Gamma, x : A \vdash B : K \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash (\lambda x : A.N) : (\forall x : A.B) : K}\ \text{lam}$$

$$\frac{\Gamma \vdash M : (\forall x : A.B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}\ \text{app}$$

$$\frac{\Gamma \vdash M : A \qquad A =_\beta B \qquad B : K}{\Gamma \vdash M : B}\ \text{conv}$$

Figure 2: Typing rules for the calculus of constructions

5.5 Linear logic

Key idea: hypotheses are now linear, can only be used once.

Define $A ::= P \mid A \otimes A \mid A \oplus A \mid A \,\&\, A \mid A \parr A \mid A \multimap A \mid 1 \mid 0 \mid \top \mid \bot \mid !A \mid ?A$.

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\ \otimes$$

$$\frac{\Gamma \vdash A, B}{\Gamma \vdash A \parr B}\ \parr$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B}\ \oplus$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\ \text{\&-L}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\ \text{\&-R}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ \text{lam}$$

$$\frac{\Gamma \vdash A \multimap B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}\ \text{app}$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta}\ \text{weak}$$

$$\frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta}\ \text{contr}$$

Figure 3: Typing rules for linear logic

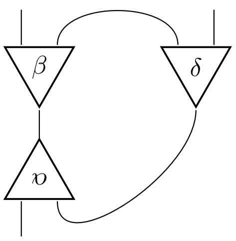
- Explain with chef analogy.

5.6 Interaction nets

An interaction net consists of:

- a finite set X of free ports
- a finite set C of cells
- a symbol $l(c)$ for each $c \in C$
- a finite set W of wires
- a set δw of 0 or 2 ports for each $w \in W$

An example net looks like:



To-do. For now see Damiano Mazza's paper [9].

5.7 Elementary affine logic

$$\frac{}{A \vdash A}\ \text{var}$$

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{weak}$$

$$\frac{\Gamma_1 \vdash A \multimap B \qquad \Gamma_2 \vdash A}{\Gamma_1, \Gamma_2 \vdash B}\ \text{app}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ \text{abst}$$

$$\frac{\Gamma_1 \vdash !A_1, ..., \Gamma_n \vdash !A_n \qquad A_1, ..., A_n \vdash B}{\Gamma_1, ..., \Gamma_n \vdash !B}\ \text{prom}$$

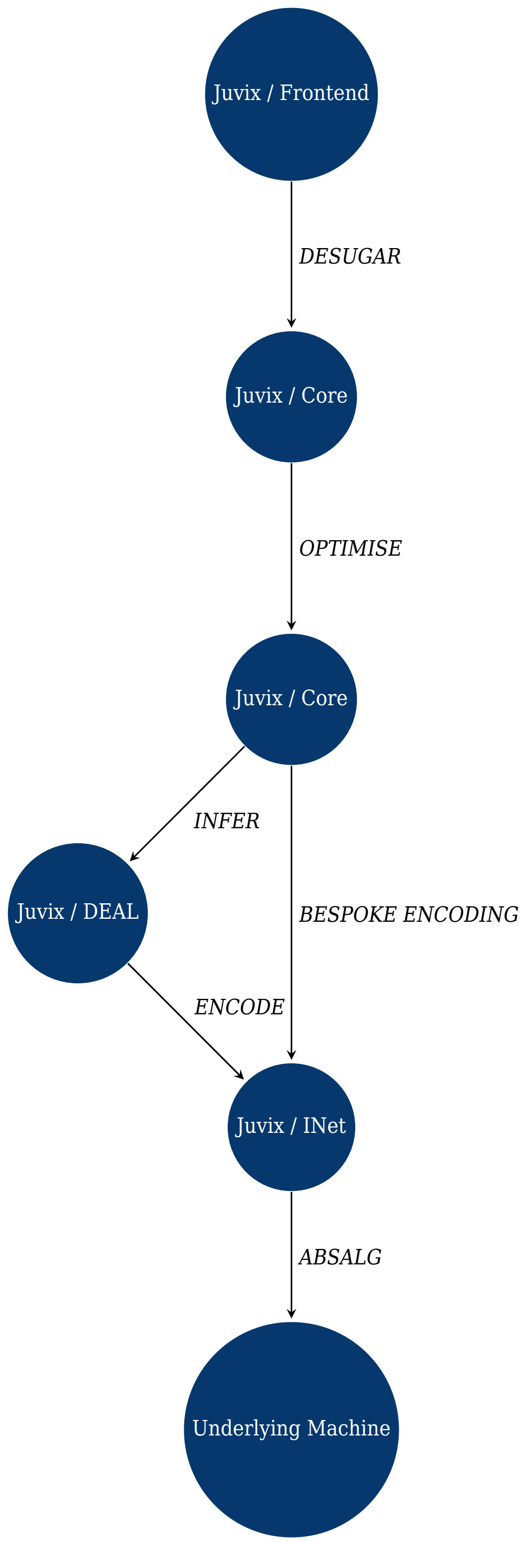
$$\frac{\Gamma \vdash !A \qquad !A, ..., !A, \Delta \vdash B}{\Gamma, \Delta \vdash B}\ \text{contr}$$

Figure 4: Typing rules for elementary affine logic

5.8 Optimal reduction

To-do. For now see the Asperti book [1].

6 Architectural overview



7 Frontend language

This chapter defines the high-level dependently-typed frontend syntax in which developers are expected to write, referred to as the "Juvix frontend language" or merely "Juvix" where unambiguous.

> Note that the frontend language is one of the less theoretically risky parts of the compiler design and thus is omitted in the initial release, except for algebraic datatypes. At present developers code directly in Juvix Core (which will always be possible).

7.1 Syntax

Syntax options:

1. Idris [3] / Haskell [7] flavoured
2. Lisp-like [21]

7.2 Features

- Nested, dependent pattern matching
- Typeclasses a la Idris
- Implicit arguments a la Idris (mostly)
- Linear variable usage annotations over the integer semiring, syntax a bit like Granule perhaps
- Case expressions
- Algebraic datatypes
- Elaboration, tactics a la Idris
- Holes
- Type inference (when possible)

Most features simply desugar to Core.

7.3 Lambda-encoding of algebraic datatypes

7.3.1 Mendler

Mendler style F-algebras have the unique advantage of being strongly normalising for positive and negative inductive datatypes without sacrificing constant time destructors or a linear-space representation. This is achieved through the Mendler representation of the algebraic datatype being a catamorphism with an explicit recursive deconstructor. The strongly normalising property of the Mendler style algebra is only achieved in the absence of deconstructors that are not the internal catamorphism, which shall be referred to as external deconstructors (todo :: find source).

Let us first explore the core of the encoding by first assuming we have access to a non inductive algebraic data type, then strip these assumptions until just base lambda calculus is left.

The first example we will explore is the natural numbers. Recall that the natural numbers can be defined as such

```
-- Normal Haskell way of defining Nat
data Nat = Z
         | S Nat
```

Now, let us strip the recursive nature of this data type using pseudo-Haskell syntax

```
data N r = Z
         | S r

let in = \r. \f. f (\d. d f) r
```

```
type AlgebraM (f :: * → *) (x :: *) = forall (r :: *). (r → x) → (f r) → x

type FixM f = forall x. AlgebraM f x → x

type Nat = FixM N

let zero = in Z : Nat
let succ = \n. in (S n) : Nat → Nat
```

The definitions above that are in need of serious attention are `AlgebraM` and `in`.

`AlgebraM` states that for any F-algebra `f` and result `x`, if we have a function, say `g`, from any `r` to `x` and the F-algebra is over `r`, then we can receive a `x` back through `g`.

The interesting aspect here is that `g` does not work over the outer algebraic structure, but instead the nested data inside of it. In our above example this would be the `r` in the definition of ℕ. This inner recursion on `r` without any other external deconstructors is what allows this encoding to be strongly normalising.

The form that allows us to inhabit `AlgebraM` is the `in` abstraction. To better understand how `in` works, an example of Nat's usage and expansion to `in` would be the most informative.

```
let isEven = \rec. \n.
  case n of
  | Z   → True
  | S n → not (rec n)

let two = succ (succ zero)

two isEven - ===> True

two isEven
= (succ (succ zero))   isEven             - (1) by definition
= (in S (in S (in Z))) isEven             - (2) by definition of succ and zero
= isEven (\d. d isEven) (S (in S (in Z))) - (3) by definition of in
= case (S (in S (in Z))) of               - (4) by definition of isEven
  | Z   → True
  | S n → not ((\d. d isEven) n)
= not ((in S (in Z)) isEven)              - (5) by case expansion
```

In the above example, we can see that `two` inhibits the type FixM ℕ, meaning that the F-Algebra we are working over is `N r` for some `r`. So the algebra must take a function `r → x` and the `f r` itself. this corresponds to the `rec` and `n` in `isEven` respectively. We can see that `rec` only works with the `r` parameter which in the case of `two` is the Mendler encoded `succ zero`. By step `5` of the expansion the definition of `in` finally becomes clear in that $(\lambda d.d f)$ shows itself to satisfy the inner recursive form.

However there is one small problem. In untyped lambda calculus we could define pred as follows:

```
let pred_alg = \rec. \n.
  case n of
  | Z   → zero
  | S n → n
```

We can see that this definition of `pred_alg` is O(1), as the call to `rec` is optional and not forced unlike other encodings. However if we were to type this using the normal Hindley-Milner type system, this does not work out. The Cedille Cast talks about this fact and deals with this issue by having a O(1) cast arising from dependent intersections.

Instead we have to define a function `out` which turns a `FixM f` into a `f (FixM f)` with the additional constraint that `f` must be a functor. This is a hard constraint that type systems without dependent intersection types and O(1) heterogeneous equality must have [5]. The force of using `out` in `pred_alg` makes getting the predecessor of a ℕ O(n).

Another property of this algebra is that this encoding is able to achieve linear space unlike previous encodings that took quadratic if not exponential space to achieve proof of termination and O(1) predecessor [5].

Now that we have some understanding of the inductive nature of the Mendler encoding, we must also strip the ADT tags of `Z` and `S` into base lambda. We will first only consider sum types which contains at most one field, and then investigate how we can modify our representation to include product types with more fields.

```
-- D is added to be illustrative of the effect of adding another case
data Nat n = Z | S n | D n

let inl = \x. \k. \l. k x

let inr = \y. \k. \l. l y

let zero-c = inl ()
let succ-c = \c. inr (inl c)
let dup-c  = \c. inr (inr c)

let zero = in zero-c : Nat
let succ = in succ-c : Nat
let dup  = in dup-c  : Nat
```

There are many valid ways to encode sum types, however for simplicity we have chosen to create a list of cases with our use of `inl` and `inr`. So, we can view inl as `head` and `inr` as tail, thus our encoding is simply the dotted list `(zero succ . dup)` [17]. On the term level, `inl` just applies the first abstraction `k` over the value, and inr simply applies the second abstraction `l` over the inputted value.

One obvious optimisation one can make, is simply turning this dotted list encoding into a balanced tree.

Another issue, is that this encoding can not support fields with multiple arguments, `l` or `k` are simply applied on the first argument given and no more. This can be remedied by swapping the application order of `inl` and `inr`, allowing the first argument to feed `l` or `k` the proper arguments.

```
data Nat n = Z | S n | D n N

let inl-op = \x. \k. \l. x k

let inr-op = \y. \k. \l. y l

let zero-c = inl ()
let succ-c = \c. inr (inl c)
let dup-c  = \a1. \ a2. inr (inr-op (\fun. fun a1 a2))

let zero = in zero-c : Nat
let succ = in succ-c : Nat
let dup  = in dup-c  : Nat
```

As we can also see, this enhancement only affects the last inr/inl in which the representation takes multiple arguments.

7.3.2 Scott

Scott encoding, unlike the Mendler F-Algebra, does not contain an internal catamorphism. Instead Scott encodings are laid out as a simple "case switch". We can see the general layout here for some branch C_i which contains n pieces of data that resides in a sum type with m constructors. Due to this simple "case switch" layout, the encoding takes linear space.

$((\lambda x_1 ... x_n. \lambda C_1 ... C_i ... C_m. C_i\, x_1 ... x_n))$

Since the constructor simply chooses which lambda to apply to the next term, we get O(1) predecessor (or case analysis generally). However since the form is not a catamorphism there is no proof of termination in an unrestricted setting.

A concrete form of the naturals with duplication is listed below to get a concrete understanding of the encoding.

```
data Nat n = Z | S n | D n n

let rec pred Z     = Z
let rec pred S n   = n
let rec pred D x y = D (pred x) (pred y)

let zero =         \zero. \succ. \dup. zero
let succ = \x.     \zero. \succ. \dup. succ x
let dup  = \x. \y. \zero. \succ. \dup. dup x y

let rec pred =
  \nat. nat zero
            (\n. n)
            (\x. \y. D (pred x) (pred y))
```

Another important aspect to Scott encodings is that they can not be typed in `System-F` alone, but instead `System-F` extended with covariant recursive types[18]. Due to this, Scott encodings can be typed safely in Juvix Core.

7.4 Desugaring

or recursive functions that are not too restricted, transforming Mendler algebras into arbitrary recursive function takes some work. As such, the Scott encoding is currently the default encoding at the EAL* level. However, at a future date Mendler encodings will be added.

8 Core language

8.1 Basics

Juvix Core is the core language in the Juvix compiler stack, defining the canonical syntax & semantics on which all compilers & interpreters must agree. Lower-level evaluation choices may safely differ between implementations as long as they respect the core semantics.

Juvix Core is primarily inspired by Quantitative Type Theory [2], Formality [8], and Cedille [19]. It fuses full-spectrum dependent types (types & terms can depend on types & terms) with linear logic using the contemplation — computation distinction introduced by QTT, adds the self-types of Cedille & Formality to enable derivation of induction for pure lambda terms, introduces the full connective set of linear logic, dependent where appropriate, to express different varieties of conjunction & disjunction, and defines an extension system for opaque user-defined datatypes & primitives (such as integers with addition & multiplication, bytestrings with concatenation & indexing, or cryptographic keys with construction & signature checking).

> Note: at present, the substructural typing in the core language is not required for optimal reduction — separate elementary affine logic assignments are inferred in the lower-level stage. Substructural typing is used in Juvix Core to provide additional precision to the programmer and enable optimisations in the bespoke compilation path to custom rewrite rules (such as avoiding garbage collection). In the future a closer fusion allowing the more precise usage information to inform interaction net construction is expected; this is an open research question.

8.2 Changes from QTT

8.2.1 Additional linear logic connectives

Multiplicative disjunction

- Explicit parallelism
- Hints for interaction net runtime
- Translated as multiplicative conjunction, but must be de-structured as a whole

Dependent additive conjunction

- Resourceful production
- Translated as multiplicative conjunction, but only one of fst or snd can be used

8.2.2 Usage polymorphism

An example, with Church-encoded naturals:

```
one :: 1 (1 (a -> a) -> 1 a -> a)

two :: 1 (2 (a -> a) -> 1 a -> a)

three :: 1 (3 (a -> a) -> 1 a -> a)
```

where, ideally, we have

```
succ :: 1 (1 (n (a -> a) -> 1 a -> a) -> ((n + 1) (a -> a) -> 1 a -> a)
```

Several options:

1 - Typeclass-style usage polymorphism in the frontend language

Frontend language level usage polymorphism, where succ must be instantiated for an n known at compile time at each call site (like a typeclass), and must be instantiated with w if n is unknown at the call site. This is easy, but probably not that useful, since often we won't know n for the argument at compile time.

2 - Usage polymorphism in the type theory

Add a $forallu.T$, where u ranges over the semiring, to the core type theory, and can then appear as a variable for any usage in T. This then will require n-variable polynomial constraint satisfiability checks during typechecking, but should have zero runtime cost. It may impact the kinds of memory management we can automate, not sure yet, but we should still have more information than without quantisation at all (or, equivalently, with w).

3 - Dependent usaging in the type theory

Add a $\uparrow u$ term to lift a term to a usage, such that usages in T in a dependent function of type $(x \overset{\pi}{:} S) \rightarrow T$ can depend on x (we must then choose some canonical bijective mapping between semiring elements and terms), and some sort of beta-equivalence proofs of usage correctness will be required of programmers using this kind of lifting (in order for the term to typecheck). (possibly also add a usage-to-term $\downarrow u$, not sure)

8.3 Preliminaries

A *semiring* R is a set R with binary operations $+$ (addition) and $\cdot$ (multiplication), such that $(R, +)$ is a commutative monoid with identity 0, $(R, \cdot)$ is a monoid with identity 1, multiplication left and right distribute over addition, and multiplication by 0 annihilates R.

The core type theory must be instantiated over a particular semiring. Choices include the boolean semiring $(0, 1)$, the zero-one-many semiring $(0, 1, \omega)$, and the natural numbers with addition and multiplication.

In canonical Juvix Core the type theory is instantiated over the semiring of natural numbers plus ω, which is the most expressive option — terms can be 0-usage ("contemplated"), n-usage ("computed n times"), or ω-usage ("computed any number of times").

Let S be a set of sorts (i, j, k) with a total order.

Let K be the set of primitive types, C be the set of primitive constants, and $\vdots$ be the typing relation between primitive constants and primitive types, which must assign to each primitive constant a unique primitive type.

Let F be the set of primitive functions, where each f_i is related to a function type by the $\vdots$ relation and endowed with a reduction operation $\rightarrow_{f_i}$, which provided an argument of the function input type computes an argument of the function output type.

Primitive types, primitive constants, and primitive functions are threaded-through to the untyped lambda calculus to which Juvix Core is erased, so they must be directly supported by the low-level execution model.

8.4 Syntax

Inspired by the bidirectional syntax of Conor McBride in I Got Plenty o' Nuttin' [10].

Let R, S, T, s, t be types & terms and d, e, f be eliminations, where types can be synthesised for eliminations but must be specified in advance for terms.

The three columns, in order, are: syntax utilised in this paper, text description, and syntax utilised in the ASCII parser.

$R, S, T, s, t ::= *_i$	sort i	$*i$
$\mid \kappa \in K$	primitive type	(varies)
$\mid (x \overset{\pi}{\vdots} S) \rightarrow T$	function type	[π] S -> T
$\mid (x \overset{\pi}{\vdots} S) \otimes T$	dependent multiplicative conjunction type	$([\pi]S,\ T)$
$\mid (x \overset{\pi}{\vdots} S) \mathbin{\&} T$	dependent additive conjunction type	$/\backslash$
$\mid T \parr T$	non-dependent multiplicative disjunction type	$\backslash/$
$\mid \iota x.T$	self-type	$@x.T$
$\mid \lambda x.t$	abstraction	$\backslash x.t$
$\mid e$	elimination	e

$d, e, f ::= x$	variable	x
$\mid c \in C$	primitive constant	(varies)
$\mid fs$	application	fs
$\mid (s, t)$	pair	(s, t)
$\mid s \mathbin{\epsilon} t$	additive conjunction	TBD
$\mid s \mathbin{\gamma} t$	multiplicative disjunction	TBD
$\mid fst_{\&}\ M$	first projection for additive conjunction	$fst\ M$
$\mid snd_{\&}\ M$	second projection for additive conjunction	$snd\ M$
$\mid let\ (x, y) = d\ in\ e$	dependent pair pattern match	$let\ (x, y) = d\ in\ e$
$\mid s \overset{\pi}{\vdots} S$	type & usage annotation	$s : [\pi]\ S$

Figure 5: Core syntax

Sorts $*_i$ are explicitly levelled. Dependent function types, dependent conjunction types, and type annotations include a usage annotation π.

Judgements have the following form:

$$x_1 \overset{\rho_1}{:} S_1, ..., x_n \overset{\rho_n}{:} S_n \vdash \ M \overset{\sigma}{:} T$$

where $\rho_1 ... \rho_n$ are elements of the semiring and σ is either the 0 or 1 of the semiring.

Further define the syntactic categories of usages ρ, π and precontexts Γ:

$$\rho, \pi \in R$$
$$\Gamma := \diamond \mid \Gamma, x \overset{\rho}{:} S$$

The symbol ⋄ denotes the empty precontext.

Precontexts contain usage annotations ρ on constituent variables. Scaling a precontext, $\pi\Gamma$, is defined as follows:

$$\pi(\diamond) = \diamond \tag{1}$$
$$\pi(\Gamma, x \overset{\rho}{:} S) = \pi\Gamma, x \overset{\pi\rho}{:} S \tag{2}$$

Usage annotations in types are not affected.

By the definition of a semiring, 0Γ sets all usage annotations to 0.

Addition of two precontexts $\Gamma_1 + \Gamma_2$ is defined only when $0\Gamma_1 = 0\Gamma_2$:

$$\diamond + \diamond = \diamond$$
$$(\Gamma_1, x \overset{\rho_1}{:} S) + (\Gamma_2, x \overset{\rho_2}{:} S) = (\Gamma_1 + \Gamma_2), x \overset{\rho_1 + \rho_2}{:} S$$

Contexts are identified within precontexts by the judgement $\Gamma \vdash$, defined by the following rules:

$$\frac{}{\diamond\vdash}\ \text{Emp}$$

$$\frac{\Gamma \vdash \qquad 0\Gamma \vdash S}{\Gamma, x \overset{\rho}{:} S \vdash}\ \text{Ext}$$

$0\Gamma \vdash S$ indicates that S is well-formed as a type in the context of 0Γ. Emp, for "empty", builds the empty context, and Ext, for "extend", extends a context Γ with a new variable x of type S and usage annotation ρ. All type formation rules yield judgements where all usage annotations in Γ are 0 — that is to say, type formation requires no computational resources).

Term judgements have the form:

$$\Gamma \vdash M \overset{\sigma}{:} S \tag{3}$$

where $\sigma \in 0, 1$. A judgement with $\sigma = 0$ constructs a term with no computational content, while a judgement with $\sigma = 1$ constructs a term which will be computed with.

E.g., for the following judgement:

$$n \overset{0}{:} Nat, x \overset{1}{:} Fin(n) \vdash x \overset{\sigma}{:} Fin(n)$$

When $\sigma = 0$, the judgement expresses that the term can be typed:

$$n \overset{0}{:} Nat, x \overset{1}{:} Fin(n) \vdash x \overset{0}{:} Fin(n)$$

Because the final colon is annotated to zero, this represents contemplation, not computation. When type checking, n and x can appear arbitrary times.

Computational judgement:

$$n \overset{0}{:} Nat, x \overset{1}{:} Fin(n) \vdash x \overset{1}{:} Fin(n)$$

Because the final colon is annotated to one, during computation, n is used exactly 0 times, x is used exactly one time. In our language, x can also be annotated as ω, indicating that it is used an arbitrary number of times.

8.5 Typing rules

8.5.1 Universe (set type)

Let S be a set of sorts i, j, k with a total order.

Formation rule:

$$\frac{0\Gamma \vdash \qquad i < j}{0\Gamma \vdash *_i \overset{0}{:} *_j} \; *$$

Introduction rule ($\sigma = 0$ fragment only):

$$\frac{0\Gamma \vdash M \overset{0}{:} *_i \qquad 0\Gamma, x \overset{0}{:} M \vdash N \overset{0}{:} *_i}{\Gamma \vdash (x \overset{\pi}{:} M) \to N \overset{0}{:} *_i} \; *\text{-Pi}$$

8.5.2 Primitive constants, functions & types

$$\frac{c \in C \qquad \kappa \in K \qquad c \mathrel{\vdots} \kappa}{\vdash c \overset{\sigma}{:} \kappa} \; \text{Prim-Const}$$

Primitive constants are typed according to the primitive typing relation, and they can be produced in any computational quantity.

$$\frac{f \in F \qquad f \mathrel{\vdots} (x \overset{\pi}{:} S) \to T}{\vdash f \overset{\sigma}{:} (x \overset{\pi}{:} S) \to T} \; \text{Prim-Fn}$$

Primitive functions are typed according to the primitive typing relation, and they can be produced in any computational quantity.

Primitive functions can be dependently-typed.

$$\frac{f \in F \qquad \vdash f \overset{\sigma}{:} (x \overset{\pi}{:} S) \to T \qquad \Gamma \vdash M \overset{\sigma'}{:} S \qquad \sigma' = 0 \Leftrightarrow (\pi = 0 \lor \sigma = 0)}{\pi\Gamma \vdash fM \overset{\sigma}{:} T[x := M]} \; \text{Prim-App}$$

Applications of primitive functions impose the same usage constraints as regular abstractions.

8.5.3 Dependent function types

Function types $(x \overset{\pi}{:} S) \to T$ record usage of the argument. The formation rule is:

$$\frac{0\Gamma \vdash S \qquad 0\Gamma, x \overset{0}{:} S \vdash T}{0\Gamma \vdash (x \overset{\pi}{:} S) \to T}\ \text{Pi}$$

The usage annotation π is not used in judgement of whether T is a well-formed type. It is used in the introduction and elimination rules to track how x is used, and how to multiply the resources required for the argument, respectively:

$$\frac{\Gamma, x \overset{\sigma\pi}{:} S \vdash M \overset{\sigma}{:} T}{\Gamma \vdash \lambda x.M \overset{\sigma}{:} (x \overset{\pi}{:} S) \to T}\ \text{Lam}$$

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\pi}{:} S) \to T \qquad \Gamma_2 \vdash N \overset{\sigma\pi}{:} S \qquad 0\Gamma_1 = 0\Gamma_2}{\Gamma_1 + \Gamma_2 \vdash MN \overset{\sigma}{:} T[x := N]}\ \text{App}$$

- $0\Gamma_1 = 0\Gamma_2$ means that Γ_1 and Γ_2 have the same variables with the same types
- In the introduction rule, the abstracted variable x has usage $\sigma\pi$ so that non-computational production requires no computational input
- In the elimination rule, the resources required by the function and its argument, scaled to the amount required by the function, are summed
- The function argument N may be judged in the 0-use fragment of the system if and only if we are already in the 0-use fragment ($\sigma = 0$) or the function will not use the argument ($\pi = 0$).

8.5.4 Dependent multiplicative conjunction (tensor product)

Dependent tensor production formation rule:

$$\frac{0\Gamma \vdash A \qquad 0\Gamma, x \overset{0}{:} S \vdash T}{0\Gamma \vdash (x \overset{\pi}{:} S) \otimes T}\ \otimes$$

$$\frac{0\Gamma \vdash}{0\Gamma \vdash I}\ \text{I}$$

Type formation does not require any resources.

Introduction rule:

$$\frac{\Gamma_1 \vdash M \overset{\sigma\pi}{:} S \qquad \Gamma_2 \vdash N \overset{\sigma}{:} T[x := M] \qquad 0\Gamma_1 = 0\Gamma_2}{\Gamma_1 + \Gamma_2 \vdash (M, N) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T}$$

$$\frac{0\Gamma \vdash}{0\Gamma \vdash * \overset{\sigma}{:} I}$$

This is similar to the introduction rule for dependent function types above.

Elimination rule:

Under the erased ($\sigma = 0$) part of the theory, projection operators can be used as normal:

$$\frac{\Gamma \vdash M \overset{0}{:} (x \overset{\pi}{:} S) \otimes T}{\Gamma \vdash fst_{\otimes}\ M \overset{0}{:} S}$$

$$\frac{\Gamma \vdash M \overset{0}{:} (x \overset{\pi}{:} S) \otimes T}{\Gamma \vdash snd_{\otimes}\ M \overset{0}{:} T[x := fst_{\otimes}(M)]}$$

Under the resourceful part:

$$\frac{0\Gamma_1, z \overset{0}{:} (x \overset{\pi}{:} S) \otimes T \vdash U \qquad \Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \qquad \Gamma_2, x \overset{\sigma\pi}{:} S, y \overset{\sigma}{:} T \vdash N \overset{\sigma}{:} U[z := (x, y)] \qquad 0\Gamma_1 = 0\Gamma_2}{\Gamma_1 + \Gamma_2 \vdash let\ (x, y) = M\ in\ N \overset{\sigma}{:} U[z := M]} \otimes \text{ Elim}$$

- Must pattern match out to ensure both parts of the product are used.

$$\frac{0\Gamma_1, x \overset{0}{:} I \vdash U \qquad \Gamma_1 \vdash M \overset{\sigma}{:} I \qquad \Gamma_2 \vdash N \overset{\sigma}{:} U[x := *] \qquad 0\Gamma_1 = 0\Gamma_2}{\Gamma_1 + \Gamma_2 \vdash let * = M\ in\ N \overset{\sigma}{:} U[x := M]} \otimes \text{ Elim I}$$

- Must explicitly eliminate elements of the unit type in the resourceful fragment.
- Simplifies to fst, snd in $\sigma = 0$ fragment (should we combine the rules?)
- If we lambda-encoded pairs, is that isomorphic?

8.5.5 Additive conjunction

"Choose either"

Can be dependent.

Formation rule:

$$\frac{\Gamma \vdash A \overset{\sigma}{:} S \qquad \Gamma \vdash B \overset{\sigma}{:} T}{\Gamma \vdash (A \overset{\sigma}{:} S)\ \&\ (B \overset{\sigma'}{:} T)} \&$$

- Can we construct with $\sigma' / = \sigma$?

Introduction rule:

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} S \qquad 0\Gamma_1 = 0\Gamma_2 \qquad \Gamma_2 \vdash N \overset{\sigma}{:} T[x := M]}{\pi\Gamma_1 + \Gamma_2 \vdash M\ \epsilon\ N \overset{\sigma}{:} (x \overset{\pi}{:} S)\ \&\ T}$$

Elimination rules:

$$\frac{\Gamma \vdash M\ \epsilon\ N \overset{\sigma}{:} (x \overset{\pi}{:} S)\ \&\ T}{\Gamma \vdash M \overset{\pi\sigma}{:} S}$$

$$\frac{\Gamma \vdash M\ \epsilon\ N \overset{\sigma}{:} (x \overset{\pi}{:} S)\ \&\ T}{\Gamma \vdash N \overset{\sigma}{:} T[x := M]}$$

To-do: terms for elimination rules.

8.5.6 Multiplicative disjunction

"Both separately in parallel"

Presumably cannot be dependent.

Should be able to provide guarantee of parallelism in low-level execution, both in bespoke & interaction net paths.

Formation rule:

$$\frac{\Gamma \vdash (A \overset{\sigma}{:} S), (B \overset{\sigma'}{:} S')}{\Gamma \vdash (A \overset{\sigma}{:} S) \parr (B \overset{\sigma'}{:} S')} \parr$$

Introduction rule:

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} S \qquad 0\Gamma_1 = 0\Gamma_2 \qquad \Gamma_2 \vdash N \overset{\sigma}{:} T}{\Gamma_1 + \Gamma_2 \vdash M \,\gamma\, N \overset{\sigma}{:} S \parr T} \parr\text{-Intro}$$

do we need equality of 0-fragment?

Elimination rule:

$$\frac{\Gamma_1, M \overset{\sigma}{:} S \vdash M' \overset{\sigma'}{:} S' \qquad \Gamma_2, N \overset{\sigma}{:} T \vdash N' \overset{\sigma'}{:} T'}{\Gamma_1 + \Gamma_2 + M\gamma\, N \overset{\sigma}{:} S \parr T \vdash (M', N') \overset{\sigma'}{:} (S \otimes T)} \parr\text{-Elim}$$

reformulate to be syntax-directed, we need some sort of "join"

8.5.7 Self types

$$\frac{\Gamma, x : \iota x.T \vdash T : *_i}{\Gamma \vdash \iota x.T} \text{Self}$$

$$\frac{\Gamma \vdash t : [x := t]T \qquad \Gamma \vdash \iota x.T : *_i}{\Gamma \vdash t : \iota x.T} \text{Self-Gen}$$

$$\frac{\Gamma \vdash t : \iota x.T}{\Gamma \vdash t : [x := t]T} \text{Self-Inst}$$

To-do for self types

- Syntax for self types?

8.5.8 Variable & conversion rules

The variable rule selects an individual variable, type, and usage annotation from the context:

$$\frac{\vdash 0\Gamma, x \overset{\sigma}{:} S, 0\Gamma'}{0\Gamma, x \overset{\sigma}{:} S, 0\Gamma' \vdash x \overset{\sigma}{:} S} \text{Var}$$

The conversion rule allows conversion between judgementally equal types:

$$\frac{\Gamma \vdash M \overset{\sigma}{:} S \qquad 0\Gamma \vdash S \equiv T}{\Gamma \vdash M \overset{\sigma}{:} T} \text{Conv}$$

Note that type equality is judged in a context with no resources.

8.5.9 Equality judgements

Types are judgementally equal under beta reduction:

$$\frac{\Gamma \vdash S \qquad \Gamma \vdash T \qquad S \rightarrow_\beta T}{\Gamma \vdash S \equiv T} \equiv\text{-Type}$$

Terms with the same type are judgementally equal under beta reduction:

$$\frac{\Gamma \vdash M \overset{\sigma}{:} S \qquad \Gamma \vdash N \overset{\sigma}{:} S \qquad M \rightarrow_\beta N}{\Gamma \vdash M \equiv N \overset{\sigma}{:} S} \equiv\text{-Term}$$

To-do: do we need a rule for term equality?

8.5.10 Sub-usaging

To-do: check if we can safely allow sub-usaging if the ring is the natural numbers, discuss here.

8.6 Semantics

Contraction is $(\lambda x.t : (\pi x : S) \rightarrow T)\ s \rightsquigarrow_\beta (t : T)[x := s : S]$.

De-annotation is $(t : T) \rightsquigarrow_\nu t$.

The reflexive transitive closure of $\rightsquigarrow_\beta$ and $\rightsquigarrow_\nu$ yields beta reduction $\rightarrow_\beta$ as usual.

8.6.1 Confluence

A binary relation R has the diamond property iff. $\forall spq.sRp \wedge sRq \implies \exists r.pRr \wedge qRr$.

8.6.2 Parallel-step reduction

Let parallel reduction be $\triangleright$, operating on usage-erased terms, by mutual induction.

To-do: needs to consider cost accounting.

> Note: The theorem prover will support a special one-step beta equality, which stops at the first application of contraction.

$$\frac{}{*_i \triangleright *_i}$$

$$\frac{}{x \triangleright x}$$

$$\frac{S \triangleright S' \qquad T \triangleright T'}{(x : S) \rightarrow T \triangleright (x : S') \rightarrow T'}$$

$$\frac{t \triangleright t'}{\lambda x.t \triangleright \lambda x.t'}$$

$$\frac{t \triangleright t' \qquad T \triangleright T'}{t : T \triangleright t'}$$

$$\frac{f \rhd f' \qquad s \rhd s'}{f s \rhd f' s'}$$

$$\frac{t \rhd t' \qquad T \rhd T'}{t : T \rhd t' : T'}$$

$$\frac{t \rhd t' \qquad S \rhd S' \qquad T \rhd T' \qquad s \rhd s'}{(\lambda x.t : (x : S) \to T) s \rhd (t' : T')[x := s' : S']}$$

8.7 Typechecking

Lay out syntax-directed typechecker following McBride's paper.

8.8 Erasure

- Define erasure to untyped lambda calculus following McBride's paper.
- Define erasure to dependent elementary affine logic analogously (but not erasing the pi types).

Let programs of the untyped lambda calculus be $p ::= x \mid \lambda x.p \mid p\ p \mid c$ where $c \in C$ is a primitive constant.

Define the erasure operator $\blacktriangleright$, such that erasure judgements take the form $\Gamma \vdash t \overset{\sigma}{\vdots} S \blacktriangleright p$.

$$\frac{}{c \blacktriangleright c}\ \text{Prim-Erase}$$

$$\frac{\vdash 0\Gamma, x \overset{\sigma}{\vdots} S, 0\Gamma'}{0\Gamma, x \overset{\sigma}{\vdots} S, 0\Gamma' \vdash x \overset{\sigma}{\vdots} S \blacktriangleright x}\ \text{Var-Erase-+}$$

$$\frac{t \overset{\sigma}{\vdots} T \blacktriangleright p \qquad \sigma\pi = 0}{\lambda x.t : (x \overset{\pi}{\vdots} S) \to T \blacktriangleright p}\ \text{Lam-Erase-0}$$

$$\frac{t \overset{\sigma}{\vdots} T \blacktriangleright p \qquad \sigma\pi / = 0}{\lambda x.t : (x \overset{\pi}{\vdots} S) \to T \blacktriangleright \lambda x.p}\ \text{Lam-Erase-+}$$

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{\vdots} (x \overset{\pi}{\vdots} S) \to T \blacktriangleright p \qquad \Gamma_2 \vdash N \overset{0}{\vdots} S \qquad \sigma\pi = 0}{\Gamma_1 \vdash MN \overset{\sigma}{\vdots} T[x := N] \blacktriangleright p}\ \text{App-Erase-0}$$

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{\vdots} (x \overset{\pi}{\vdots} S) \to T \blacktriangleright p \qquad \Gamma_2 \vdash N \overset{\sigma\pi}{\vdots} S \blacktriangleright p' \qquad \sigma\pi / = 0}{\Gamma_1 + \Gamma_2 \vdash MN \overset{\sigma}{\vdots} T[x := N] \blacktriangleright p\ p'}\ \text{App-Erase-+}$$

In the *Lam-Erase-0* rule, the variable x bound in t will not occur in the corresponding p, since it is bound with usage 0, with which it will remain regardless of how the context splits, so the rule *Var-Erase-+* cannot consume it, only contemplate it.

Other terms erase as you would expect.

To-do:

- Constants $c \in C$ erase to themselves
- Functions $f \in F$ erase to themselves
- Function applications fx of primitive $f \in F$ erase to $f(\blacktriangleright x)$.

Computationally relevant terms are preserved, while terms which are only contemplated are erased.

8.9 Examples

8.9.1 Church-encoded natural numbers

- Representation
- Deriving induction for Church-encoded numerals using self-types

The Church-encoded natural n can be typed as $\vdash \lambda s.z.s...sz \overset{1}{:} (s \overset{n}{:} a \to a) \to (z \overset{1}{:} a) \to a$ where s is applied n times.

8.9.2 Linear logic connectives

- Linear logic disjunctions & conjunctions
- Linear induction?

9 Whole-program optimisation

Juvix combines compiler-directed & user-directed optimising transformations into a single optimisation function ψ, defined in this chapter, which maps core terms to core terms, preserving the evaluation semantics defined in the previous chapter.

> Note that whole program optimisation is one of the less theoretically risky parts of the compiler design and thus is omitted in the initial release. At present the optimisation function ψ is simply the identity. Future releases are expected to incorporate optimising transformations discussed herein.

9.1 Core-level optimisations

- User can prove extensional equality of functions.
- Compiler can pick which function is cheaper to reduce (& pick differently in different cases)
- Can be specialised to properties on arguments, e.g. if $fx|x < 0 = g$, if the compiler can inhabit $x < 0 = True$, it can replace f with g.

9.2 Machine-level optimisations

- Requires formalised VM semantics of particular machine model (in Juvix)
- User can prove semantic equivalence of machine instruction sequences
- Compiler can pick which instruction sequence is cheaper and compile to it (& pick differently in different cases)

9.3 Graph transformations

Primarily inspired by the GRIN [13] paper & implementation.

(todo: determine which of these are rendered unnecessary by interaction net evaluation; keep it as simple as possible)

(todo: some of these need to be applied at a lower layer and only when terms are compiled to custom rewrite rules via the bespoke path)

See this example.

Possible transformations:

- vectorisation
- case simplification
- split fetch operation
- right hoist fetch operation
- register introduction

- evaluated case elimination
- trivial case elimination
- sparse case optimisation
- update elimination
- copy propagation
- late inlining
- generalised unboxing
- arity raising
- case copy propagation
- case hoisting
- whnf update elimination
- common sub-expression elimination
- constant propagation
- dead function elimination
- dead variable elimination
- dead parameter elimination

10 Dependent elementary affine logic

- Alter to simple dependently-typed version
- Figure out what constraints are required for pi types, see https://hal.archives-ouvertes.fr/hal-00021834/document p15, can we do the same thing?
- Add & reason through arguments for correctness

10.1 Syntax

Define:

1. Formulae $A, B ::= \ \alpha \mid A \multimap B \mid \underline{!}A$.
2. Terms \$t, u ::= $x \mid \lambda x.t \mid t\ u \mid !t \mid \overline{!}t$.

10.2 Semantics

For well-typed terms, as the untyped lambda calculus (simply erase the bangs).

10.3 Typing rules

$$\frac{}{x : A \vdash x : A}\ \text{var}$$

$$\frac{\Gamma \vdash t : B}{\Gamma, x : A \vdash t : B}\ \text{weak}$$

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1, \Gamma_2 \vdash (t_1 t_2) : B}\ \text{app}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \text{abst}$$

$$\frac{\Gamma_1 \vdash t_1 : !A_1, ..., \Gamma_n \vdash t_n : !A_n \qquad x_1 : A_1, ..., x_n : A_n \vdash t : B}{\Gamma_1, ..., \Gamma_n \vdash !t[\bar{!}t_i/x_i] : !B}\ \text{prom}$$

$$\frac{x_1 : !A, ..., x_n : !A, \Delta \vdash t : B}{x : !A, \Delta \vdash t[x/x_1, ..., x_n] : B}\ \text{contr}$$

Figure 7: Typing rules for EAL

10.4 Erasure to untyped lambda calculus

Define erasure map $(-)$, just delete the !s.

10.5 Type inference

Adapted from previous work [12].

Why / notes

- Boxes are unintuitive to write, add syntactic bureaucracy
- Inference will sometimes fail. In this case, programmer can be informed and suggested how else to write their function, or alternatively the bespoke encoding route can be taken.
- Most terms one would want to compute (especially in smart contracts) are in the elementary complexity class
- Allows better box placement than programmer might choose, compiler can pick from set of typeable EAL terms & instantiate with fewest number / optimally positioned boxes

Define:

1. Formulae $A, B ::= \ \alpha \mid A \multimap B \mid !A \mid \forall \alpha.A$.
2. Pseudoterms $t, u ::= x \mid \lambda x.t \mid (t)u \mid !t \mid \bar{!}t$.
3. Restricted pseudoterms
    1. $a ::= x \mid \lambda x.t \mid (t)t$
    2. $t ::=!^m a$ where
        1. $m \in \mathbb{Z}$
        2. $!^m a = !...!\ (m\ times)\ a\ if m \geq 0$
        3. $!^m a = \bar{!}...\bar{!}\ (m\ times)\ a\ if\ m < 0$

Theorem 1 *All EAL-typable terms can be converted into restricted pseudo-terms (for proof see the paper).*

10.5.1 Box paths

Let t be a pseudo-term and x be a particular occurrence of a free or bound variable in t.

Define the **path** as an ordered list of the occurrences of $!$ and $\overline{!}$ enclosing x, more formally:

$$path(x, x) = nil \tag{4}$$
$$path(t_1 t_2, x) = path(t_i, x) \text{ where } t_i \text{ contains } x \tag{5}$$
$$path(\lambda y.t, x) = path(t, x) \tag{6}$$
$$path(!t, x) = ! :: path(t, x) \tag{7}$$
$$path(\overline{!}t, x) = \overline{!}t :: path(t, x) \tag{8}$$

Define the **sum** of a path $s(p)$ as:

$$s(nil) = 0 \tag{9}$$
$$s(! :: l) = 1 + s(l) \tag{10}$$
$$s(\overline{!} :: l) = -1 + s(l) \tag{11}$$

Define the **well-bracketed** condition, mapping pseudo-terms to booleans, where $\leq$ is the prefix relation on lists, for a pseudo-term t as:

$$\forall l \leq path(t, x), s(l) \geq 0 \text{ for any occurrence of a variable } x \text{ in } t \tag{12}$$
$$\forall x \in FV(t), s(path(t, x)) = 0 \text{ (zero sum paths for free variables)} \tag{13}$$

Define the **well-scoped** condition, mapping pseudo-terms to booleans, for a pseudo-term t as:

$$\forall t_i \in subterms(t), well - bracketed(t_i) \tag{14}$$

Theorem 2 *If t is a EAL-typed term, t is well-bracketed and well-scoped.*

Let an EAL **type assignment** for a pseudo-term t be a map Γ' from free & bound variables of t to EAL formulae.

Extend that map to a partial map Γ from subterms of t to EAL formulae as:

$$\Gamma(!u) = !A \text{ if } \Gamma'(u) = A \tag{15}$$
$$\Gamma(\overline{!}u) = A \text{ if } \Gamma'(u) = !A\text{, undefined otherwise} \tag{16}$$
$$\Gamma(\lambda x.u) = A \multimap B \text{ if } \Gamma'(x) = A, \Gamma'(u) = B \tag{17}$$
$$\Gamma(t_1 t_2) = B \text{ if } \Gamma'(t_2) = A \text{ and } \Gamma'(t_1) = A \multimap B\text{, undefined otherwise} \tag{18}$$

Let (t, Γ) be a pair of a pseudo-term t and an assignment Γ. t satisfies the **typing condition** if:

$$\Gamma(t_i) \text{ is defined for all subterms } t_i \text{ of } t \tag{19}$$
$$\text{for any variable } x \text{ of at least 2 occurrences, } \Gamma(x) = !B \text{ for some } B \tag{20}$$

Theorem 3 *If t is an EAL-typed term and Γ is an associated assignment then (t, Γ) satisfies the typing condition.*

Theorem 4 *If (t, Γ) satisfies the typing condition and u is a subterm of t, then (u, Γ) also satisfies the typing condition.*

Theorem 5 *If t is a pseudo-term and Γ an assignment such that t is well-bracketed and well-scoped, and (t, Γ) satisfies the typing condition, then t is typable in EAL with a judgement $\Delta \vdash t : A$ such that $\Gamma(t) = A$ and Δ is the restriction of Γ to the free variables of t.*

Proof

Enumerate the bracketing, scope, and typing conditions as (i), (ii), and (iii) respectively. Proceed by induction on the pseudo-term t:

1. $t = x$ is trivial
2. $t = \lambda x.u$
    1. u satisfies the first part of the bracketing condition since t does
    2. u satisfies the second part of the bracketing condition since t satisfies the scope condition for x
    3. u then trivially satisfies (ii), (iii), so by induction we have in EAL* $\Delta, x : A \vdash u : B$ where $\Gamma(x) = A, \Gamma(u) = B$
    4. Apply the abstraction rule to get the judgement for t
3. $t = t_1 t_2$
    1. Subterm t_1 satisfies conditions (i), (ii), (iii), hence by induction $\Delta_1 \vdash t_1 : A_1$, where $\Gamma(t_1) = A_1$ and Δ_1 is the restriction of Γ to the free variables of t
    2. Subterm t_2 satisfies conditions (i), (ii), (iii), hence by induction $\Delta_2 \vdash t_2 : A_2$, where $\Gamma(t_2) = A_2$ and Δ_2 is the restriction of Γ to the free variables of t
    3. As t satisfies the typing condition (iv) A_1 is of the form $A_1 = A_2 \multimap B_1$
    4. If t_1 and t_2 share a free variable y, as t satisfies the typing condition we have $\Gamma(y) =!B$
    5. Rename in t_1 and t_2 the free variables that they have in common, accordingly
    6. Apply an application rule followed by a contraction rule to get the judgement for t
4. $t = \overline{!}u$
    1. t does not satisfy the bracketing condition (i) in the first prefix, so this case is invalid
5. $t = !u$
    1. By the boxing lemma, t can be written as $t =!v[x_1 := \overline{!}u_1, ...x_n := \overline{!}u_n]$, where $FV(v) = x_1...x_n$ and $path(v, x_i)$ is well-bracketed for all x_i
    2. Let y be an occurrence of a variable in u_i
        1. $path(t, y) = ! :: path(v, x_i) :: \overline{!} :: path(u_i, y)$
        2. $path(v, x_i)$ is well-bracketed, so $path(u_i, y)$ satisfies the bracketing condition and u_i satisfies (i).
        3. Since t satisfies (ii) and (iii), u_i, a subterm of t, also satisfies (ii) and (iii).
        4. Therefore there exists an EAL* derivation $\Delta_i \vdash u_i : A_i$, where A_i = $\Gamma(u_i)$, for $1 \leq i \leq n$.
    3. Now examine v
        1. Since t satisfies the bracketing condition, by the boxing lemma, v satisfies (i) and all free variables in v have exactly one occurrence, so as t satisfies (ii) v does also.
        2. Let Γ' be defined as Γ but $\Gamma'(x_i)$ = $\Gamma(\overline{!}u_i)$ for $1 \leq i \leq n$.
            1. If y occurs more than once in v then it also does in t, hence $\Gamma(y) = !A$ so $\Gamma'(y) = !A$.
            2. If $v_1 v_2$ is a subterm of v then $v_1' v_2'$, where $v_i' = v_i[x_1 := \overline{!}u_1, ...x_n := \overline{!}u_n]$, is a subterm of t and $\Gamma'(v_i')$ = $\Gamma(v_i)$
            3. As (t, Γ) satisfies (iii), so does (v, Γ')
            4. $\Gamma(u_i) = A_i$ and $\Gamma(\overline{!}u_i)$ is defined, so $A_i = !B_i$ and $\Gamma'(x) = B_i$
            5. As v satisfies conditions (i), (ii), and (iii), by induction there exists an EAL* derivation $\Delta, x_1 : B_1, ..., x_n : B_n \vdash v : C$ where $C = \Gamma'(v)$.
        3. If u_i and u_j for $i/ = j$ have a common free variable y, as t satisfies the typing condition $\Gamma(y) = !B$.
        4. Rename the free variables in common to the u_is, apply a (prom) rule to obtain the judgements on u_i and the judgement on v
        5. Then apply (contr) rules separately for the final judgement $\Delta' \vdash t : !C$, concluding the proof

10.5.2 Decoration

Consider the **declaration problem**:

Let $x_1 : A_1, ..., x_n : A_n \vdash M : B$ be a simply-typed term. Do there exist EAL decorations A'_i of the A_i for $1 \leq i \leq n$ and B' of B such that $x_1 : A'_1, ..., x_n : A'_n \vdash M : B'$ is a valid EAL judgement for M?

10.5.3 Parameterisation

Define **parameterised restricted pseudo-terms** as restricted pseudo-terms but with unique integer indices for each free parameter: $a ::= x \mid \lambda x.t \mid t\ t,\ t ::=!^n a$ where n is a fresh index chosen monotonically over Z. Given a parameterised pseudo-term t denote by $par(t)$ the set of its parameter indices. An instantiation φ maps t to a restricted pseudo-term by instantiating each indexed parameter n with the integer demarcation $\varphi(n)$.

Define **parameterised types** $A ::=!^n\ \alpha \mid !^n(A \multimap A)$ with n a fresh index chosen monotonically over Z. Denote by $par(A)$ the set of parameters of A. If φ instantiates each index n with an integer demarcation $\varphi(n)$, $\varphi(A)$ is defined only when a non-negative integer is substituted for each parameter (per the type formulae of EAL). Define the size $|A|$ as the structural size of the underlying simple type.

Analogously to EAL type assignments for pseudo-terms consider parameterised type assignments for parameterised pseudo-terms with values parameterised types, and simple type assignments for lambda terms with values simple types. Let Σ be a parameterised type assignment for a parameterised pseudo-term t. Denote by $par(\Sigma)$ the parameter set occurring in parameterised types $\Sigma(x)$, for all variables x of t. Let φ be an instantiation for $par(\Sigma)$ which associates a non-negative integer with each indexed parameter. Then define the map $\varphi\Sigma$ by $\varphi\Sigma(x) = \varphi(\Sigma(x))$. When defined, this map is an EAL type assignment for $\varphi(t)$. Define the size $|\Sigma|$ as the maximum of $|\Sigma(x)|$ for all variables x.

Define the erasure map $(.)^-$ for parameterised pseudo-terms and parameterised types analogously to those for pseudo-terms and EAL types. Given a lambda term M there exists a unique parameterised pseudo-term t, up to renaming of the indices, such that $t^- = M$. Denote t by $\bar{M}$ and call it the **parameter decoration** of M. Note that $|\bar{M}|$ is linear in $|M|$. Given a simple type T, its **parameter decoration** $\bar{T}$ is defined analogously. Finally, given a simple type assignment Θ for a lambda term t, with values simple types, define its parameter decoration $\bar{\Theta}$ point-wise by taking $\bar{\Theta}(x) = \bar{\Theta(x)}$, where all decorations are taken with disjoint parameters.

Thus the decoration problem is reduced to the following instantiation problem:

Given a parameterised pseudo-term t and a parameterised type assignment Σ for it, does there exist an instantiation φ such that $\varphi(t)$ has an EAL type derivation associated to $\varphi\Sigma$?

To answer this question we will translate the bracketing, scope, and typing conditions into a system of linear constraints.

10.5.4 Constraint generation

Bracketing & scope Let t be a parameterised pseudo-term. Define the **boxing constraints** for t as the set of linear equations $C^b(t)$ obtained in the following way:

- Bracketing: for any occurrence of a variable x in t, and any prefix l of $path(t, x)$, add the inequation $s(l) \geq 0$. If $x \in FV(t)$, add the equation $s(l) = 0$.
- Scope: for any subterm $\lambda x.v$ of t, for any occurrence x_i of x in v, add similarly the inequations expressing that $path(v, x_i)$ is well-bracketed.

Typing constraints Define parameterised type unification as:

$$U(!^m\alpha, !^n\alpha) = \{m = n\} \tag{21}$$
$$U(!^m(A_1 \multimap A_2), !^n(B_1 \multimap B_2)) = \{m = n\} \cup U(A_1, B_1) \cup U(A_2, B_2) \tag{22}$$
$$U(\_, \_) = \{false\} \tag{23}$$

Let Σ be a parameterised type assignment for a parameterised pseudo-term t. Extend Σ to a partial map from subterms of t to parameterised types as:

$$\Sigma(!^n a) = !^m A \text{ if } \Sigma(a) = !^k A \tag{24}$$

$$\Sigma(\lambda x.u) = !^m (A \multimap B) \text{ if } \Sigma(x) = A, \Sigma(u) = B \tag{25}$$

$$\Sigma(u_1 u_2) = B \text{ if } \Sigma(u_1) = A \multimap B \text{ and } \Sigma(u_2) = A \tag{26}$$

Define the **typing constraints** for (t, Σ) as the set of linear inequations $C^{typ}(t, \Sigma)$:

- for any subterm of t with the form $\lambda x.u$ with $\Sigma(\lambda x.u) = !^m (A \multimap B)$, add the constraint $m = 0$
- for any subterm of t with the form $u_1 u_2$ with $\Sigma(u_1) = !^m (A_1 \multimap B_1)$ and $\Sigma(u_2) = A_2$ add the constraints $U(A_1, A_2) \cup m = 0$
- for any subterm of t with the form $!^n u$ with $\Sigma(!^n u)$ = $!^m A$ and $\Sigma(u) = !^k A$, add the constraints $m = k + n$ and $m \geq 0$
- for any subterm of t with the form x where x has at least two occurrences and $\Sigma(x) = !^m A$, add the constraint $m \geq 1$
- for any parameter m in $par(\Sigma)$, add the constraint $m \geq 0$

Theorem 6 *Let t be a parameterised pseudo-term and Σ be a parameterised type-assignment for t such that $\Sigma(t)$ is defined. Given an instantiation φ for (t, Σ), $\varphi\Sigma$ is defined and $(\varphi(t), \varphi\Sigma)$ satisfies the typing condition if and only if φ is a solution of $C^{typ}(t, \Sigma)$.*

10.5.5 Constraint solution

Generated constraints are simple integer (in)equalities and can be solved in polynomial time. At present Juvix exports them to Z3.

Multiple solutions may exist, in which case the solution with the least number of box-annotations is selected.

10.5.6 Argument for correctness

Re-run correctness argument from paper, adapt as necessary.

10.5.7 Extensions

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \prod A.B} \text{ abst}$$

$$\frac{\Gamma, x : A \vdash B : *}{\Gamma \vdash \prod x : A.B : *} \text{ pi}$$

$$\frac{\Gamma_1 \vdash t_1 : \prod A.B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1, \Gamma_2 \vdash (t_1 t_2) : B[t_2/A]} \text{ app}$$

Figure 8: Typing rules for DTEAL (TODO)

Q: Can we just split up the context in the (app) rule without problems? Equivalent to zero-usage in QTT?

think about how to correctly translate r = lam-case 1 -> 2Y; 2 -> 3Z, h = (r & x) -> a, constraints should reflect dependently-typed usage calculations (though maybe we don't care for >= 2 usages).

11 Low-level execution model

11.1 Overview

Juvix translates the semantics of a term to equivalent semantics for an interaction system, consisting of node types, rewrite rules, write-forward and read-back algorithms (for translating terms to and from nets, respectively), where elementary-affine-typed terms are in the general case reduced using the oracle-free variant of Lamping's optimal reduction algorithm.

Compared with previous interaction net interpreters for the lambda calculus utilising a static set of node types and fixed rewrite rules, Juvix adds an additional degree of freedom: the node types and rewrite rules of the interaction system can be generated at compile time and even dynamically altered at runtime according to patterns of rewrite combinations and desired time-space complexity trade-offs. Additional type data from the core language, such as exact variable usage counts provided by the instantiation of quantitative type theory with the natural ring, are available to the interaction system construction algorithm.

also - refl (equality) proofs in core language can be used by compiler, e.g. with total supply of a token = constant, for queries on the total supply the constant can be returned; more generally if two expressions are equal the compiler can choose which one to evaluate - will be more effective if graph representation is persistent, instead of written / read-back each contract call. can be used for both code & data

- Define encoding $\phi(t)$ of term t mapping to net n
- Define read-back function $\phi^{-1}(n)$ mapping net n to a term t, where $\phi^{-1}(\phi(t)) = t$ holds
- Define interaction system reduction function $\psi(n)$ mapping nets to nets, where $\phi^{-1}(\psi(\phi(t))) = reduce\ t$ where $reduce$ is as defined in the semantics of Juvix Core

11.2 Interaction system encoding

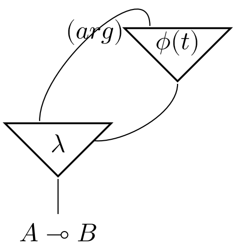
EAL term language $t ::= x \mid \lambda x.t \mid (tu) \mid !t$.

EAL type $A ::= \alpha \mid A \multimap A \mid !A$.

EAL-typed terms can be translated into interaction nets, in accordance with the sequent calculus typing rules, as the function ϕ as follows.

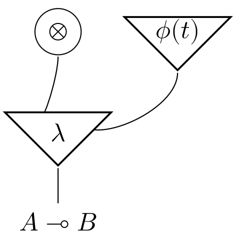
The EAL term is first erased to a simply-typed term, with EAL types and levels of subterms retained in a lookup table for reference during the translation.

Abstraction is applied to terms of the form $\lambda x.t$ and type $A \multimap B$.

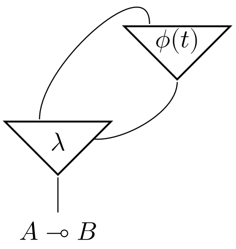


Wiring of the argument x varies depending on variable usage linearity:

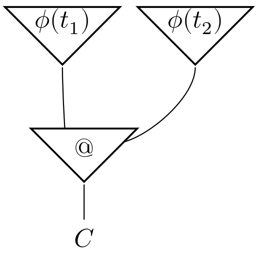
Weakening: If x does not appear in the body t, the λ argument port is connected to an eraser.



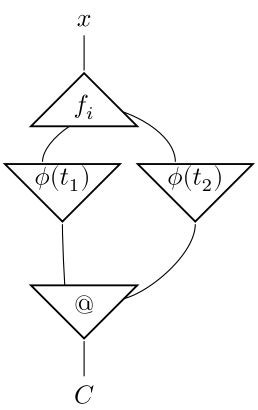
Linear / contraction: If x appears once or more in the body t, the λ argument port is connected to the occurrence(s). If there is more than one occurrence, usages will be shared by a set of fan nodes constructed by the application encoding.



Application is applied to terms of the form $(t_1 t_2)$ and type C.



For each free variable x in $(t_1 t_2)$ occurring more than once, all occurrences of x must be connected by a tree of fan-in nodes, each with a globally unique label (only one fan-in node is shown in the diagram).



That ends the encoding rules for basic lambda terms.

Interaction net encoding of $a\&b$:

1. Primary port as pair, to be connected to destructor (fst / snd)
2. $3n + 2$ auxiliary ports where:
    1. Two are for the subterms a and b
    2. n are for the terms bound to free variables (resources) both subterms use
    3. n are connected to the binding sites in a
    4. n are connected to the binding sites in b
3. Rewrite rule
    1. If the destructor is fst (vice versa if the destructor is snd):
        1. Connects the wires between the n free variables and the n binding sites in a
        2. Attaches erasers to the n binding sites in b and to b itself
        3. Erases $\&$ node
        4. Attaches whatever destructor was connected to to a

That way no duplication of resources need occur, matching the linear logic semantics.

Note that this means reduction within a and b, insofar as it depends on the values of the free variables, will not take place until the caller chooses which variant (a or b) they want.

Interaction net encoding of $a \otimes b$:

1. Primary port as pair, to be connected to destructor ("join")
2. 2 auxiliary ports, where:
    1. Two are for the subterms a and b
3. Rewrite rule
    1. Erases the ⊗ node
    2. Creates a new ⊗ node, attaches its 2 auxiliary ports to a and b
    3. Attaches whatever the destructor was attached to to the new ⊗ node's primary port

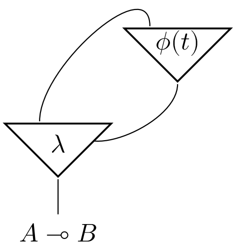
This is structurally identical to the ⊗ encoding (perhaps we can simply erase the "join" destructor prior to runtime), but we should be able to place directives that inform the evaluator to evaluate the two subterms in parallel, as they are guaranteed not to share resources (no duplication required) and be completely disjoint subgraphs.

> TODO: Example term encoding.

11.2.1 Bespoke function encoding

Basics Consider a Core term f of type $A \multimap B$.

In the interaction net encoding compiler path, assuming EAL-typeability, we would encode this (if of form $\lambda x.t$, for example) as:



where φ is the recursive interaction net translation function.

In the bespoke encoding path, we instead create a new node type T and rewrite rule R such that when the primary port of T is connected to an application node to an argument A, we erase T, connect an eraser to A, and connect whatever the application node's primary port was connected to to a new subgraph which is equal to the encoding of $eval(fA)$.

$eval(fA)$ can then be implemented by native evaluation semantics which do not utilise interaction nets. For example, if the Core term in question is a tail-recursive numerical computation, it can be compiled to a native loop (possibly using SIMD).

Furthermore, we can safely encode non-EAL-typable terms this way, such as the Ackermann function, and they can safely interact with the rest of the interaction net (which must have been an EAL-typable term, treating the bespoke-encoded subterm as opaque).

The decision of whether or not to take the bespoke path can be made for all subterms of this form according to some heuristic (or possibly exact cost calculation) in the compiler.

Dealing with various types Where A and B are both types which are encoded as primitive nodes (e.g. integers), this is trivial.

Where A is a primitive type and B is a function of some arity, of only primitive-typed arguments, which then returns a primitive type, this can be implemented as a sequence of node types T, T', T'', etc. which keep the curried arguments and eventually evaluate when all arguments are provided (or even when some are provided, there is a continuum of options here).

Where A and/or arguments of B are non-primitive types (e.g. functions), this becomes more complex, since we must convert between AST and interaction-net form during reduction.

More generally, with our Core term f of type $A \multimap B$, encoding f through the bespoke path would result in a set of new node types T_i with possible curried internal data, and a set of rewrite rules R_i, the first $i-1$ of which just deal with currying (although again, there is a continuum of options, but let's leave that out for now), and the last one of which is interesting, let it be R.

R must then cause, when connected to a primary port of an argument A:

- Erasure of R_i (the prior node).
- Connection of an eraser to A.
- Creation of a new subgraph $\varphi(eval(f(read - backA))$, where φ is the recursive interaction net encoding function (which might itself perform bespoke encoding, although we need to be concerned about runtime costs here), and $read-back$ is the read-back function from nets to Core terms, run starting at A as the root node.
- Connection of the primary port of this subgraph to whatever R_i was previously connected to.

This follows all the interaction net laws and should preserve semantics - but there are oddities:

- Read-back and (complex) encoding algorithms must be executed at runtime
- Read-back must happen over a term A which may be in the progress of parallel reduction

In general, we have no idea of the size (and corresponding read-back cost) of A, and it might be dependent on the order of reduction.

11.2.2 Bespoke datatype encoding

- Primitive types (integer, string, bytes) ~> node types w/data
- User-defined ADTs can be turned into custom nodes

11.3 Oracle-free optimal reduction

The oracle-free abstract algorithm for optimal reduction operates on four node types: λ (lambda), @ (application), f_i (fan, with index i), and $\otimes$ (eraser). Rewrite rules always operate only on primary port pairs and consist of two categories: **annihilation** rules, which remove nodes, and **commutation** rules, which create nodes.

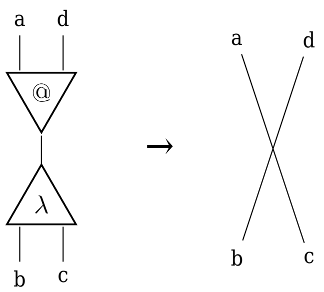


Figure 9: Lambda-application annihilation (beta reduction)

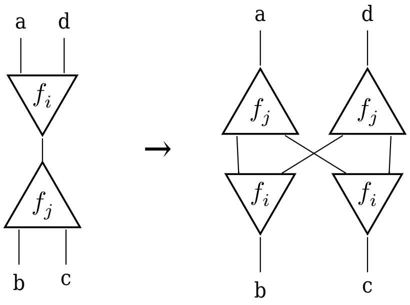


Figure 11: Fan-fan commutation

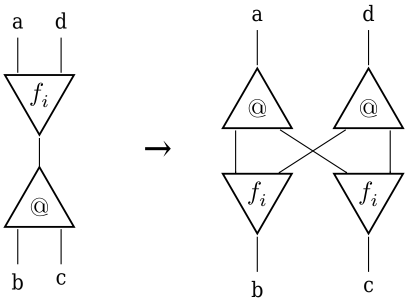


Figure 13: Fan-application commutation

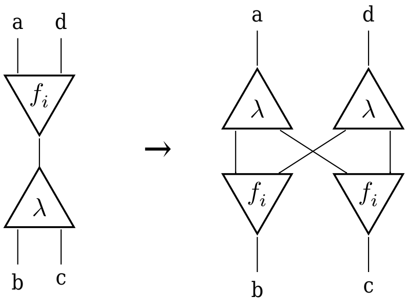


Figure 15: Fan-lambda commutation

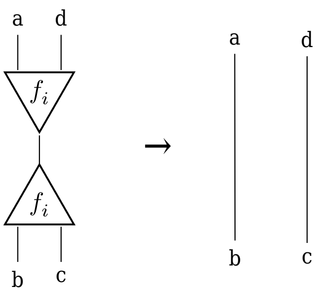


Figure 17: Fan-fan annihilation

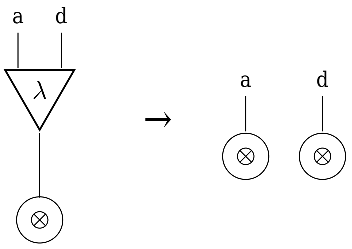


Figure 19: Eraser-lambda commutation

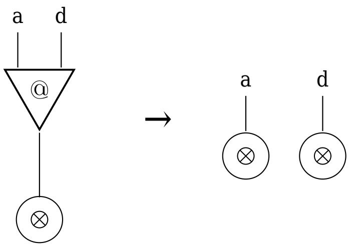


Figure 21: Eraser-application commutation

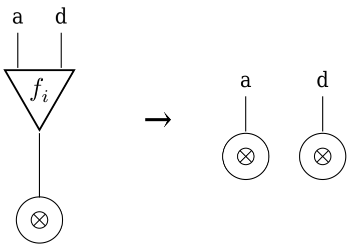


Figure 23: Eraser-fan commutation

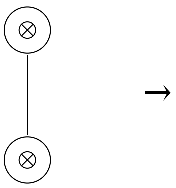


Figure 25: Eraser-eraser annihilation

11.3.1 Argument for correctness

1. Define the level of a subterm
    1. level a a = 0
    2. level lam x . a b = level a b
    3. level a b c = level a c or level b c as appropriate
    4. level !a b = 1 + level a b
    5. level $\bar{!}a$ b = -1 + level a b
2. The level of a subterm is constant through beta reduction
    1. lam x . t → lam x . t (trivial)

   2. x $\longrightarrow$ x (trivial)
   3. !x $\longrightarrow$!x (trivial)
   4. ! $\overline{!}$ x $\longrightarrow$ x (trivial) (defined-ness guaranteed by well-bracketed property)
   5. (lam x . a) b $\longrightarrow$ a [x := b]
      1. If t was in a - trivial
      2. If t was in b - level a x = 0 by well-bracketed property, so level b t = level (a [x := b]) t
3. Map this level to nodes in the interaction net translation
   1. Think concentric boxes with natural number levels
4. Level of node does not change during reduction
   1. Beta reduction only connects nodes on same level
5. Levels can be chosen from contraction nodes of EAL type derivation
   1. Contraction nodes do not change level during proof-net reduction
   2. Also do not change level during abstract algorithm reduction
   3. If fans match label, must have originated from that level
   4. Algorithm is correct with EAL term subset since label indicates level of fan and level does not change according to EAL rules
   5. No loops in reduction of EAL-typable terms that would render labels underspecified

(needs pretty pictures)

11.3.2 Argument for correctness of read-back

We wish to show that a read-back of an EAL-expression that has undergone zero or more reductions gives back the same result as normal evaluation up to alpha equality.

Before we can prove this result, we must first prove a few lemmas and theorems first as well talk about what fan in's with the same label imply.

The first lemma we wish to prove is the following

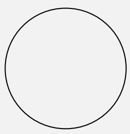
Lemma 11.1: AST→Net has one free port

Let A be a valid BOHM term.

Now Consider the net encoding of A, let N_a be this net.

Now let us consider any sub case of A, $L = P(n_1 \star \cdots \star n_k)$.

The node corresponding to L looks.



TODO :: put ports on this images

Where the labels in N_a are the same as in L, with n_{k+1} being an extra port which connects to the ADT above it.

For an ADT to be realized, we must have all arguments, so all these ports must be connected. Now, if L is A, then we have one free port, however if it is not, then this free port must not be free but instead be connected to the ADT above it.

For free variables which have no λ to connect to, a symbol node is created, maintaining the invariant.

∴ AST→Net has one free port

Now we need to prove that evaluating this net does not change this fact

Theorem 11.1: AST→Net→Eval has one free port

Let $A = Net(AST)$.

By lemma 11.1 A has one free port.

We must now show $Eval(A)$ does not change the number of free ports.

We can show this by simply considering how every reduction rule works. For the sake of brevity, we will only consider the three variations of rewrite rules present in the BOHM system WLOG.

TODO :: Make the ports vary for all example with $\cdots$ drawn on the images

Case 1 - Rewire

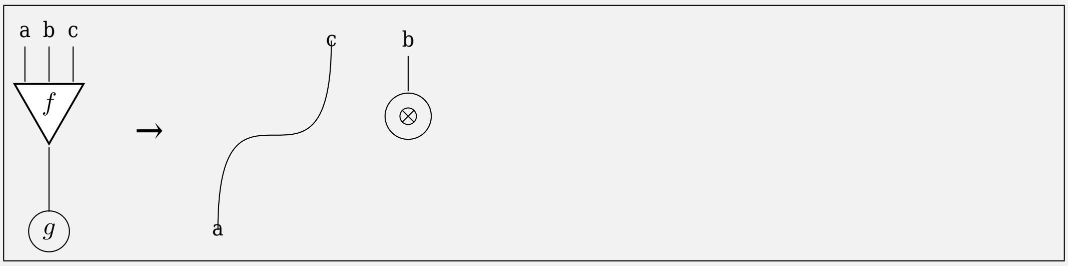
Nodes that fall under this case:
1. And↔T
2. And↔F
3. Or↔T
4. Or↔F

The wire connecting the main ports of node f and node g is eliminated.

The rest of the wires are simply rewired to each other, thus not altering the total number of free ports.

Case 2 - Isolation

Nodes that fall under this case:
1. Cdr↔Cons
2. Car↔Cons
3. TestNil↔Cons
4. IfThenElse↔T
5. IfThenElse↔F



Here we can see that like the first case the wire that connects the main ports are eliminated, and that a and c are rewired.

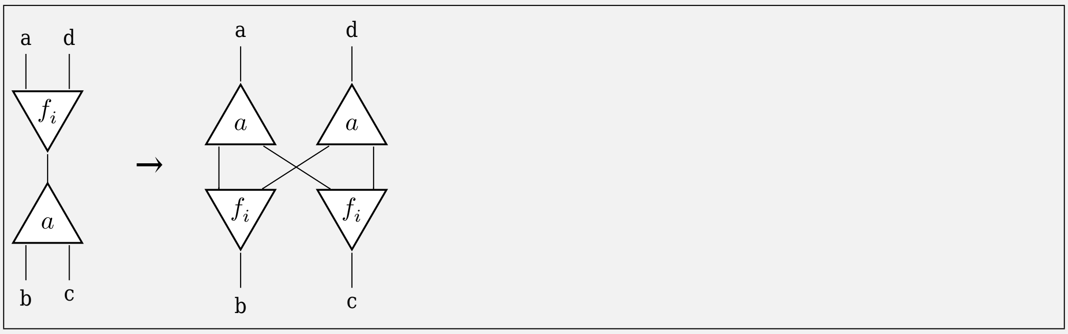
Additionally an eraser node is connected to the b, since the eraser node only has a main port, the number of free ports is not affected.

Case 3 - Creation

Nodes that fall under this case:
1. TestNil↔Nil
2. All nodes which "curry" and eventually become an Int node or T/F
   - add, not, mod, div, sub, more, noteq, eq, meq, less, leq. ↔ intlit
3. Not↔T
4. Not↔F
5. Cdr↔Nil

In this case, we take a node with n ports and construct a node with $n-1$ ports. since the remaining ports are still wired to the same locations, the number of free ports does not change

Case 4 - Duplication



Here we see the case for a fan-in node f_i and some arbitrary node a. Both nodes get duplicated, however the number of free ports is preserved, as each node that is created is internally connected to the duplicated nodes. Additionally the four external ports are preserved among the four duplicated nodes.

∴ since all cases are covered, $Eval(A)$ has only one free port.

We now shall define one more definition before getting to our main theorem.

Definition 11.1: Valid EAL-Net

A **Valid EAL-Net** is a net translated from the EAL subset of BOHM that has undergone zero or more reduction steps.

Theorem 11.2: Read back from a Valid EAL-Net gives is α equivalent to the normal evaluation of the original EAL-term

Let A be the Valid EAL-Net.

By Definition 11.1 this net originated from a valid EAL-AST which is a subset of BOHM.

Thus by Theorem 11.1 A must only have one free port.

Furthermore this free port must be the root of the graph/old AST.

Now to show this evaluates to the same answer, we must reconstruct the ADT considering all cases.

For this we must keep two maps as we do this algorithm.

The first being the variable name from a specific lambda.

The second map is a map from the fan in number to the current status of our traversal of said fan-in.

We can now write a proof by induction, specifically, we shall consider three cases:
1. λ / μ
2. $FanIn$
3. The rest.
We will also consider the inductive case first for the following proof.

Let rec be this recursive algorithm.

Case 1 - rest

The logic for all other cases is simple. Consider some node P with ports $1 \cdots n$.

Let port n be the inhabitant of the BOHM ADT.

By induction we run this algorithm and ports $1 \cdots n - 1$ are α equivalent to normal evaluation.

Thus we can construct ADT P by $P(rec(n_1) \cdots rec(n_{n-1}))$.

$\therefore$ P is well formed and we get an α equivalent answer.

Case 2 - λ / μ

The λ / μ case is special in that in a Valid EAL-Net the node will be traversed via the primary port via the parent AST and then by the second auxiliary port.

The second auxiliary port refers to the variable binding.

When the algorithm traverse the λ node by the primary port first (this must be the case), we add a new var into the first map.

Then we run the recursive algorithm on Auxiliary 1, this subterm will only access the second Auxiliary port of this λ node.

If the term is unused, then we will connect an eraser node to the second Auxiliary port.

This thus preserves the number of free ports.

By induction the subterms on Auxiliary one are also well formed

$\therefore$ the Lambda term is α equivalent to the normal evaluation answer.

Case 3 - FanIn

FanIn's are another interesting case, for this case we allocate a map from each FanIn to the traversal status.

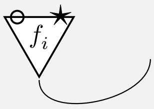
For this, we keep a first-in-last-out stack of the following types.

```
type Status = In FanPort | Complete FanPort
type FanPort = Circle | Star
```

Where circle is the first auxiliary port and star is the second.

The Status type deserves some explanations. So In means that we have visited either circle or star of a FanIn node without visiting the corresponding port on the FanOut.

The $Complete$ case means that we have visited the corresponding FanOut port, and have thus completed the traversal of the specific node. Much like how a) closes a (. This is also why it is first-in-last-out, as we must close the most recent In (parenthesis) before we can close the previous one.



For this analysis we must consider what port we can enter from, however first we should note that before entering the primary port of a FanIn, that we must have first traversed through either $\star$ or $\circ$ first.

This is the case because entering a FanIn in this way is known as a FanOut, and denotes the end of sharing. And since ending sharing before starting is not possible from a valid EAL-Net, these cases never happen.

Enter from Prim

As shown above, we must have already traversed a FanIn node, namely $\star$ or $\circ$.

This means the map from the FanIn number i is not empty, and thus we have a history of completing nodes or being in nodes.

If we have an unfinished In, then we pick the most recent In port left to traverse. This is forced as this node is considered a FanOut and thus closes sharing.

We then mark the node as complete. If we have completed a single node, then we simply choose to leave through the other port.

Since we are working over valid EAL-Net's which require no oracle, there is no ambiguity in this mechanism. Furthermore, this fact also excludes any other configurations from happening.

Enter from Aux

We entered the FanIn through an auxiliary port, so we mark in the second map that we are In this port at FanIn node i.

We then leave through the principle port, denoting the beginning of sharing.

These cases handle all possible configurations in which we can enter and the validity of the expressions are handled by induction.

Now, the base case is quite simple, consider a node, say with zero parameters by itself.

This node is trivially readback as is, being α equivalent to the evaluated term.

$\therefore$ by induction read-back from a valid EAL-Net will give us the same expression up to α equivalence as evaluating the node via a more traditional evaluation methods

11.4 Evaluator cost model

Currently tracked:

- Memory allocations
- Sequential rewrite steps
- Parallel rewrite steps
- Maximum graph size
- Final graph size

In the future we may want to track more granular memory operations (reads/writes in addition to allocations) and computations associated with rewrite rules (all negligible-cost with interaction combinators, but not with e.g. integer arithmetic).

Machine backends are expected to provide a discrete cost model which can be utilised by the optimiser.

11.5 Future optimisation strategies

Juvix does not yet implement these, but the compiler architecture has kept their possibility in mind.

11.5.1 Spacial memory contiguity

Random access O(1) model is imperfect; sequential reads are faster. Ensure correspondence between graphical locality and spacial locality in memory, read nodes in blocks.

11.5.2 Speculative execution

- “Strict” optimal reduction strategies
- Evaluate based on predicting future input (feasible?)

11.5.3 Stochastic superoptimisation

- Utilise sparse sampling (probably Markov-chain Monte Carlo) to search the configuration space of semantically equivalent programs & select the fastest.
- Probably useful at the level of choosing machine implementations of particular rewrite rules.
- See Stochastic Superoptimisation [15]
- Will need a lot of clever tricks to avoid getting stuck in local minima (that paper details several).
- See also STOKE [16]

11.5.4 “Superoptimal” reduction strategies

- Specifically those with the possibility of asymptotically-better performance than Levy’s optimal reduction.
- As far as I can tell, the only candidates here are forms of memoisation which attempt to detect syntactically identical structures during the reduction process which can then be linked and evaluated only once.
- Hash consing may have the most prior research.
- Concerns about space-time trade-offs (may already be concerns).

12 Execution extensions

12.1 Cost accounting

"Gas"

12.1.1 Cost per VM instruction

Gas = cost per instruction on VM - e.g. EVM, Michelson.

1. Advantages
    1. Works, simple
2. Disadvantages
    1. Adds overhead - have to increment counter, check each instruction
    2. UX is pretty terrible - must estimate gas before sending tx, state could change
    3. (~) Cross-contract calls are dumb and hard to optimise (more a problem with using a low-level VM)

12.1.2 Prior cost calculation

Calculate gas cost prior to execution.

1. Need: `cost :: (call, params) => Natural`
    1. Must be (far) cheaper to evaluate than just computing the function
    2. Must be verifiable (once) by the state machine for a particular contract, so need e.g. verified interpreter (for a VM)
    3. Might be worst-case cost, not tight bound (but should be able to make dependent on param values, so should be tight)

12.1.3 Execute off-chain, verify on-chain in constant time

1. Constant gas cost + size of storage diffs
    1. User must do all execution (e.g. in ZK), verification is O(1) or user pays circuit size (easy enough) + proportional cost for state changes
    2. Bottleneck: prover time. TinyRAM runs at ~1Hz for the prover at the moment (old paper though). Maybe inets could be a bit more efficient.

13 Machine targets

In each case, two options: interaction net interpreter & direct compilation.

13.1 LLVM

Target LLVM IR, with parallel (thread?) support.

13.2 Michelson

Target Michelson. Will likely involve a lot of bespoke term compilation.

13.3 WASM

Target WASM. Parallelism?

13.4 EVM

Target EVM v1.x. Maybe pointless and should focus on eWASM instead.

13.5 GPUs (CUDA / OpenCL)

Excellent way to demonstrate parallelism.

13.6 FPGA

Parallelism + versatility, maybe there are large efficiency gains possible.

14 Future directions

14.1 Zero-knowledge execution compression

1. Run interaction nets in ZK
2. Might benefit from the parallelism

14.2 Zero-knowledge typing

1. Earlier than execution compression.
2. Inline ZK verifiers + proofs as custom node types.
3. Type data as "private" or "public" at the monadic level, compiler-enforced.

14.3 Deployment tooling layer

1. Ledgers as first-class objects in declarative deployment scripts.
2. Declarative-stateful deployment system Terraform-style.
3. Blockchains accessible in REPL.

14.4 Persistent interaction system state

1. Defined equivalence semantics but implementation can change later
2. Contracts themselves can call the compiler (needs more R&D)
3. Bounties for proofs, sub-contract-upgrades, etc.
4. More efficient than read-back after execution, just persist the graph of the state machine, many more optimisations automatically happen.
5. Will be helpful for Juvix to be self-hosting or packaged as a runtime which can typecheck untrusted input.

14.5 Interchain abstraction

1. Can run cross-chain over IBC
2. Targets multiple backends (Ethereum, Tezos, Cosmos) initially
3. Avoid lock-in, separate choice of application and choice of consensus

14.6 Visual spatiotemporal dataflow representation

1. Some (closest?) inspiration: Luna [11]
2. Could map depths of elementary linear logic terms to spacial depth in an execution visualisation
3. Goal: isomorphism between textual (AST) and graphical (dataflow graph) representations. Getting the isomorphism right so that they can be switched between for real projects seems like the hard part.

15 Appendices

15.1 Examples

Examples of Juvix high level language, core translation, interaction net evaluation (full pipeline).

Possible ideas (all with proofs):

- Fungible & nonfungible token contracts
- Social recovery - complex predicate account
- Prediction market with pluggable oracle

References

[1] Andrea Asperti, S.G. Optimal implementation of functional programming languages. .

[2] Atkey, R. Syntax and semantics of quantitative type theory. https://bentnib.org/quantitative-type-theory.pdf.

[3] Brady, E. Idris - systems programming meets full dependent types. https://dl.acm.org/citation.cfm?doid=1929529.1929536.

[4] Core Team Post-mortem: 0x v2.0 exchange vulnerability. https://blog.0xproject.com/post-mortem-0x-v2-0-exchange-vulnerability-763015399578.

[5] Denis Firsov, Richard Blair and Stump, A. Efficient mendler-style lambda-encodings in cedille. https://arxiv.org/abs/1803.02473v1.

[6] Edgington, B. Ethereum lisp like language. https://lll-docs.readthedocs.io/en/latest/lll_introduction.html.

[7] Jones, S.P. Haskell - an advanced, purely functional programming language. https://www.haskell.org/.

[8] Maia, V. Formality. https://github.com/moonad/Formality.

[9] Mazza, D. A denotational semantics for the symmetric interaction combinators. https://pdfs.semanticscholar.org/1731/a6e49c6c2afda3e72256ba0afb34957377d3.pdf.

[10] McBride, C. I got plenty o' nuttin'. https://personal.cis.strath.ac.uk/conor.mcbride/PlentyO-CR.pdf.

[11] Order, N.B. Luna. https://github.com/luna/luna.

[12] Patrick Baillot, K.T. A feasible algorithm for typing in elementary affine logic. .

[13] Peter David Podlovics, Csaba Hruska and Penzes, A. A modern look at grin,an optimizing functional language back end. https://nbviewer.jupyter.org/github/Anabra/grin/blob/fd9de6d3b9c7ec5f4aa7d6be41285359a73494e3/papers/stcs-2019/article/tex/main.pdf.

[14] Pettersson, J. Safer smart contracts through type-driven development. https://publications.lib.chalmers.se/records/fulltext/234939/234939.pdf.

[15] Schkufza, E. Stochastic superoptimization. https://theory.stanford.edu/~aiken/publications/papers/asplos13.pdf.

[16] Stanford STOKE: A stochastic superoptimizer and program synthesizer. https://github.com/StanfordPL/stoke.

[17] Steele, G.L. Common lisp the language, 2nd edition. https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node28.html#SECTION00640000000000000000.

[18] Stump, A. Directly reflective meta-programming. http://homepage.divms.uiowa.edu/~astump/papers/archon.pdf.

[19] Stump, A. The calculus of dependent lambda eliminations. https://homepage.divms.uiowa.edu/~astump/papers/cdle.pdf.

[20] Technologies, P. A postmortem on the parity multi-sig library self-destruct. https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.

[21] Common lisp. https://lisp-lang.org/.