Juvix — Language Reference

Christopher Goes, Marty Stumpf, Jeremy Ornelas

14th August 2019 - *Prerelease*

Juvix synthesizes a high-level frontend syntax, dependent-linearly-typed core language, and low-level parallelizable 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 has been purpose-built from the ground up for the particular requirements and economic tradeoffs of the smart contract use case — it prioritizes verifiability, precision, and output code efficiency over compilation speed, syntactical & semantic familiarity, and compatibility with existing blockchain virtual machines. Machine-assisted proof search, declarative deployment tooling, cogent type inference, and alternative spatiotemporal dataflow representations faciliate 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 typesafe 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 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 in principle can provide a high degree of individual protection (no government on Earth can likely break secure encryption), the imprecisions and inaccuracies in large-scale systems have caused the economics of privacy & capability to devolve into base power dynamics, where even individuals possessed of relevant domain expertise stand little chance against a nation state, terrorist group, or 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 construtive & compositional, so that sets of proofs can be imported & reused along with the libraries which comprise most of modern software, and must unify code & proofs into a single efficient language. The approach must result in standards of succinct proofs which can be embedded by the manufacturers of user-facing software, such as web browsers, operating systems, or cryptocurrency wallets, which tightly constrain the opaque behaviour of complex backend codebases. The approach must reduce the costs of formally verifying software to the point where doing so is the economically rational decision for the majority of security-critical projects.

Smart contracts running on distributed ledgers are an archetypal example of a security-critical application, yet results so far have not been promising [15] [3]. 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-recognized. A radically different language is necessary: one that treats verifiability as a design problem, not a feature to be tacked on later, that provides succinct, expressive, and composable proofs over complex logic, and that reduces the cost of verification to the point where not doing so for security-critical software will be considered simply irresponsible. Juvix aimes to realize 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 **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 [2]. 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 are then implemented in another language.

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 including optimizing transformations which take advantage of the expressive typesystem and targeting advanced low-level exeuction models as used by Juvix (understandably so, since Idris is primarily & impressively developed by a lecturer in his free time!). Furthermore, the economics of most standard programs running on the desktop or web favor development & execution speed over perfect safety and correctness, while smart contracts require the opposite.

3.2 Linearly-typed languages

Linear types are included, in limited form but to substantial effect, in the Rust systems programming language, which utilizes them to provide memory safety without garbage collection. 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 them to Haskell is in the discussion stage.

3.3 Dependently-typed smart contracts

One prior work [11] wrote an Idris [2] backend targeting Ethereum's LLL language [4]. Juvix shares many of the goals outlined in that paper, but the approach described therein failed to take advantage of well-known optimizations such as tail-call optimization and handicapped itself by compiling to LLL instead of directly to EVM opcodes. The effects system described therein may be a sensible model for smart contract programs written in Juvix but is out of scope of this paper, which focuses on language & compiler design only.

Formality [6] 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 Core as a compile target.

4 Reasoning

What changes does Juvix make from these prior systems, and why will the solutions proposed fix the problems previously encountered & realise the criteria outlined previously?

Juvix is efficiently executable in machine time through a novel execution model & an expressive typesystem permitting aggressive optimization and efficiently verifiable in developer time through proof bureaucracy automation & composable verification. The rigor 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, parallelizable, higher-order-friendly evaluation

Juvix utilizes 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 subexpressions, parallelizes 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 quantization 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 optimizations. Linear types obviate the need for garbage collection in both the optimal reduction & alternative direct subterm compilation paths, faciliate aggressive imperative optization 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 optimization

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

4.1.4 Proofs become optimizations

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 utilized by the optimizer 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-optimized rewrite rules specialized to the underlying machine, optionally proving equivalence of their hand-optimized rewrite rules to the compiler-generates ones using a formalized 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 utilize multiple tactics to minimize the bureaucracy of writing formal proofs of properties of terms. Generalized 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 finalized. Step-through tactics illuminate the inner desugaring & typechecking steps of the compiler to provide introspection & understandability.

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 rigor 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 optimize 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 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 toolchain 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.

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 nonterminating 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.

- Call-by-name: reduce t_1 first, substitute in t_2 for x without reduction.
- Call-by-value: 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. Doesn't parallelize well.

Find example of term with duplication from either strategy; see Asperti book.

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: $\lambda x_1...x_A.\lambda c_1...c_N.c_i(x_1c_1...c_N)...(x_{A_i}c_1...c_N).$
- 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 $c \in C$.

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

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

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

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

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

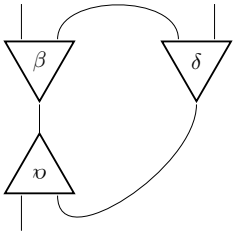
- Explain with chef analogy.

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:



Example: interaction combinators.

5.7 Elementary affine logic

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

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

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

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

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

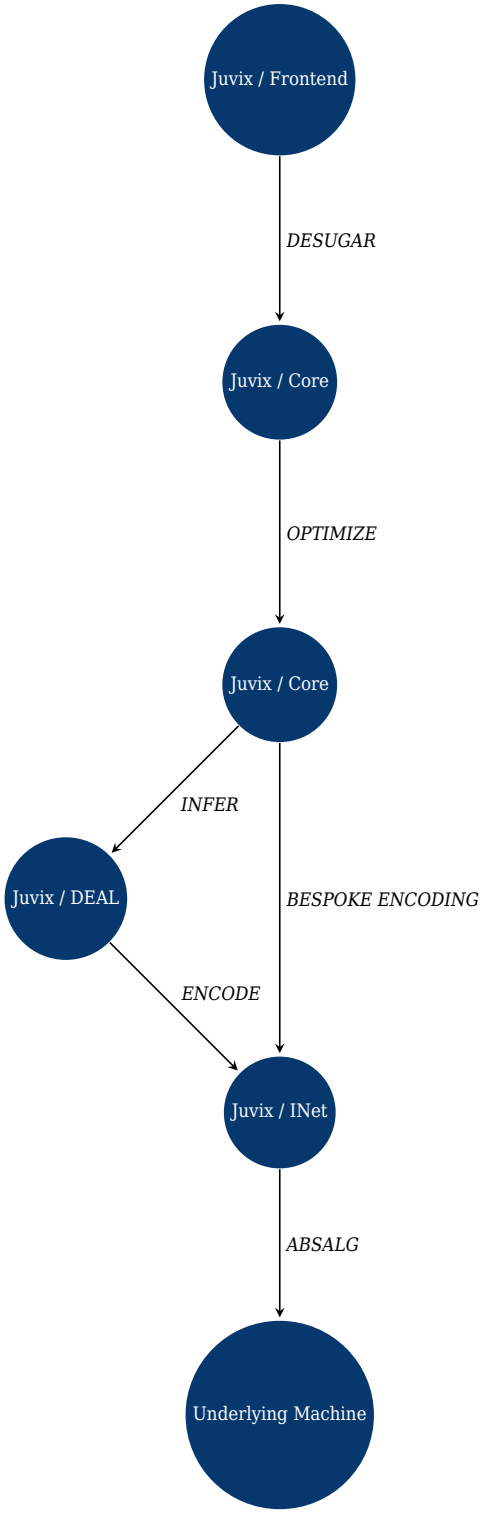
$$\frac{\Gamma \vdash !A \qquad !A, ...,\ !A, \Delta \vdash B}{\Gamma, \Delta \vdash B}\ \text{contr}$$

Figure 4: Typing rules for elementary affine logic

5.8 Optimal reduction

(todo, ref. Asperti book)

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 [2] / Haskell [5] favored
2. Lisp-like [16]

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

(todo: list asymptotic complexities, concrete complexities, encoding algorithm)

7.3.2 Scott

(todo: list asymptotic complexities, concrete complexities, encoding algorithm)

7.4 Desugaring

(todo)

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. Frontend language syntax, frontend language syntactical sugar, and lower-level evaluation choices may safely differ as long as they respect the core semantics.

Juvix Core is primarily inspired by Quantitative Type Theory [1], Formality [6], and Cedille [14]. 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, and defines an extension system for opaque user-defined datatypes & primitives (such as integers, bytestrings, or cryptographic keys).

> Note: at present, the substructural typing in the core language is not required for optimal reduction — separate elementary affine logic assignments are inferred at that stage. Substructural typing is used in Juvix Core to provide additional precision to the programmer and enable optimizations in the bespoke compilation path to custom rewrite rules (such as avoiding garbage collection).

8.2 Syntax

Inspired by the bidirectional syntax of Conor McBride in I Got Plenty o' Nuttin' [7].

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 Juvix the type theory is instantiated over the semiring of natural numbers, which is the most expressive option.

To-do: consider the natural numbers plus ω for flexibility?

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 R, S, T, s, t be types & terms and d, e, f be eliminations.

$R, S, T, s, t ::= *_i$	sort i
$\mid \kappa \in K$	primitive type
$\mid (x \overset{\pi}{\vdots} S) \to T$	function type
$\mid (x \overset{\pi}{\vdots} S) \otimes T$	dependent multiplicative conjunction type
$\mid (x \overset{\pi}{\vdots} S)\ \&\ T$	dependent additive conjunction type
$\mid T \parr T$	non-dependent multiplicative disjunction type
$\mid \iota x.T$	self-type
$\mid \lambda x.t$	abstraction
$\mid e$	elimination
$d, e, f ::= x$	variable
$\mid c \in C$	primitive constant
$\mid fs$	application
$\mid (s, t)$	pair
$\mid s\ \epsilon\ t$	additive conjunction
$\mid s\ \gamma\ t$	multiplicative disjunction
$\mid fst_{\otimes}\ M$	first projection for multiplicative conjunction
$\mid snd_{\otimes}\ M$	second projection for multiplicative conjunction
$\mid fst_{\&}\ M$	first projection for additive conjunction
$\mid snd_{\&}\ M$	second projection for additive conjunction
$\mid let\ (x, y) = d\ in\ e$	dependent pair pattern match
$\mid s \overset{\pi}{\vdots} S$	type & usage annotation

Figure 5: Core syntax

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

Typing judgements have the following form:

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

where $\rho_1 ... \rho_n$ are elements of the semiring and σ is either the 0 or 1 of the semiring (todo: must it be?).

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 $\diamond$ 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 annotations to 0.

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

$$\diamond + \diamond = \diamond$$
$$(\Gamma_1, x \overset{\rho_1}{:} S) + (\Gamma_2, x \overset{\rho_2}{:} S) = (\Gamma_1 + \Gamma_2), x \overset{p_1+p_2}{:} S$$

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 $\sigma \in 0, 1$ (todo: must it be?). 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.

8.3 Typing rules

8.3.1 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.

8.3.2 Primitive constants & types

$$\frac{c \in C \qquad \kappa \in K \qquad c : \kappa}{\vdash c \overset{\sigma}{:} \kappa}\ \text{Prim}$$

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

8.3.3 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:

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

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\pi}{:} S) \to T \qquad \Gamma_2 \vdash N \overset{\sigma'}{:} S \qquad 0\Gamma_1 = 0\Gamma_2 \qquad \sigma' = 0 \Leftrightarrow (\pi = 0 \vee \sigma = 0)}{\Gamma_1 + \pi\Gamma_2 \vdash MN \overset{\sigma}{:} T[x := N]}\ \text{App}$$

- $0\Gamma_1 = 0\Gamma_2$ 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 ($\pi = 0$).

8.3.4 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:

$$\frac{\Gamma_1 \vdash M \overset{\sigma'}{:} S \qquad 0\Gamma_1 = 0\Gamma_2 \qquad \Gamma_2 \vdash N \overset{\sigma}{:} T[x := M] \qquad \sigma' = 0 \Leftrightarrow (\pi = 0 \vee \sigma = 0)}{\pi\Gamma_1 + \Gamma_2 \vdash (M, N) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T}$$

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

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:

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

(should the last be 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?

8.3.5 Multiplicative disjunction

"Both separately in parallel"

Presumably cannot be dependent.

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:

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

To-do: elimination rule

8.3.6 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.

8.3.7 Additive disjunction

"Might be either"

Presumably cannot be dependent.

Formation rules:

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

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

Do we get this for free with lambda-encoded datatypes?

8.3.8 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):

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

8.3.9 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?

8.3.10 Equality judgements

Types are judgementally equal under beta reduction:

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

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

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

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

8.3.11 Sub-usaging

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

8.4 Semantics

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

The reflexive transitive closure of $\rightsquigarrow_\beta$ yields beta reduction $\to_\beta$ as usual.

8.4.1 Confluence

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

8.4.2 Parallel-step reduction

Parallel-step evaluation semantics a la McBride.

8.4.3 Small-step reduction

Small-step evaluation semantics (so that one-step beta equality can be used in the theorem prover).

8.5 Typechecking

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

8.6 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 $\Gamma \vdash t \overset{\sigma}{:} S \blacktriangleright p$.

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

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

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

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

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

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

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.

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

8.7 Examples

8.7.1 Church-encoded natural numbers

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

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

8.7.2 Linear lists

Example from McBride's paper:

$0List : (0X : *) \to *$

$\omega nil : (0X : *) \to ListX$

$\omega cons : (0X : *, 1x : X, 1xs : ListX) \to ListX$

$\omega lit : (0X : *, 0P : (0x : ListX) \to *) \to (1n : P(nilX)) \to (\omega c : (1x : X, 0xs : ListX, 1p : Pxs) \to P(consXxxs)) \to (1xs : ListX) \to Pxs$

$\omega append : (0X : *, 1xs : ListX, 1ys : ListX) \to ListX$

$append = \lambda X.\lambda xs.\lambda ys.litX(\lambda.ListX)ys(\lambda x.\lambda.\lambda xs'.consXxxs')xs$

8.7.3 Linear logic connectives

- Linear logic disjunctions & conjunctions
- Linear induction?

9 Whole-program optimization

Juvix combines compiler-directed & user-directed optimizing transformations into a single optimization 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 optimization is one of the less theoretically risky parts of the compiler design and thus is omitted in the initial release. At present the optimization function ψ is simply the identity. Future releases are expected to incorporate optimizing transformations discussed herein.

9.1 Core-level optimizations

- User can prove extensional equality of functions.
- Compiler can pick which function is cheaper to reduce (& pick differently in different cases)
- Can be specialized 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 optimizations

- Requires formalized 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

Much inspired by the GRIN [10] 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.

Optimizing 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

10.1 Syntax

Define:

10.2 Semantics

- Define evaluation semantics, undefined behaviour.

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 [9].

Why / notes

- Boxes are unintuitive to write, add syntactic bureacracy
- 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. $!^m a = \underline{!...!}\ (m\ times)\ a\ if m \geq 0$
        3. $!^m a = \underline{\bar{!}...\bar{!}}\ (m\ times)\ a\ if\ m < 0$

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 $\bar{!}$ enclosing x, more formally:

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

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

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

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:

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

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.*

(todo: 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 Parameterization

Define **parameterized 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 parameterized 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 **parameterized 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 nonnegative 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 parameterized type assignments for parameterized pseudo-terms with values parameterized types, and simple type assignments for lambda terms with values simple types. Let Σ be a parameterized type assignment for a parameterized pseudo-term t. Denote by $par(\Sigma)$ the parameter set occurring in parameterized types $\Sigma(x)$, for all variables x of t. Let φ be an instantiation for $par(\Sigma)$ which associates a nonnegative 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 parameterized pseudo-terms and parameterized types analogously to those for pseudo-terms and EAL types. Given a lambda term M there exists a unique parameterized 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 $\bar{\Theta}$ pointwise 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 parameterized pseudo-term t and a parameterized type assignment Σ for it, does there exist an instantiation φ such that $\varphi(t)$ has an EAL type deriviation 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 parameterized 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 parameterized 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 parameterized type assignment for a parameterized pseudo-term t. Extend Σ to a partial map from subterms of t to parameterized 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 parameterized pseudo-term and Σ be a parameterized 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 utilizing 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 tradeoffs. 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

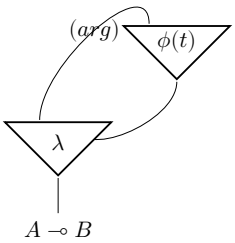
EAL term language $t ::= x \mid \lambda x.t \mid (tu) \mid !t$.

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.

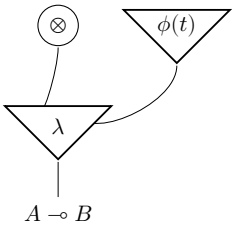
The EAL term is first erased to a simply-typed term, with EAL types and levels of subterms retained in a lookup table for reference during the translation.

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

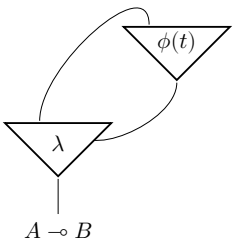


Wiring of the argument x varies depending on variable usage linearity:

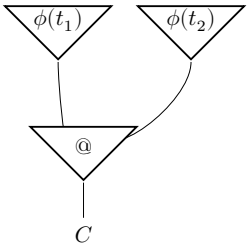
Weakening: If x does not appear in the body t, the λ argument port is connected to an eraser.



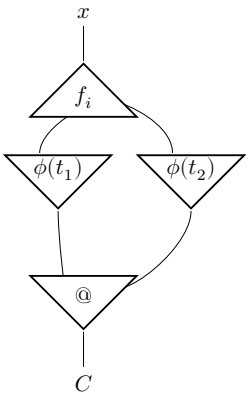
Linear / contraction: If x appears once or more in the body t, the λ argument port is connected to the occurrence(s). If there is more than one occurrence, usages will be shared by a set of fan nodes constructed by the application encoding.



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.

Example from chat & paper (of derivation).

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.

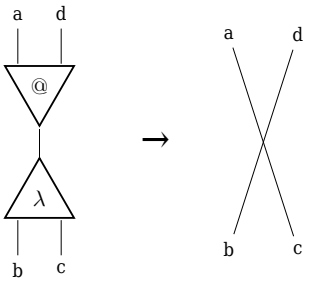


Figure 9: Lambda-application annihilation (beta reduction)

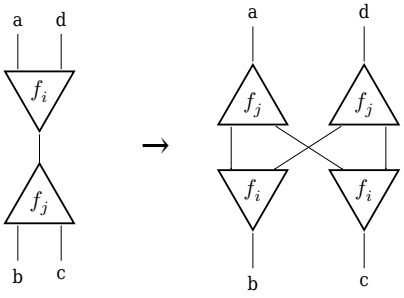


Figure 11: Fan-fan commutation

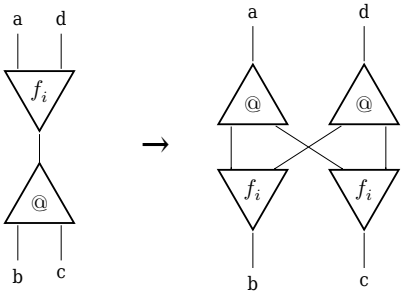


Figure 13: Fan-application commutation

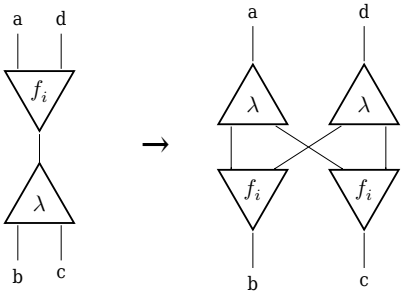


Figure 15: Fan-lambda commutation

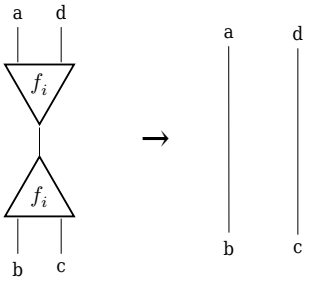


Figure 17: Fan-fan annihilation

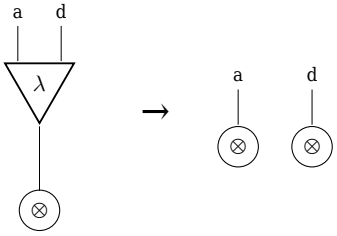


Figure 19: Eraser-lambda commutation

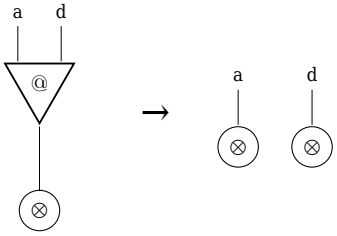


Figure 21: Eraser-application commutation

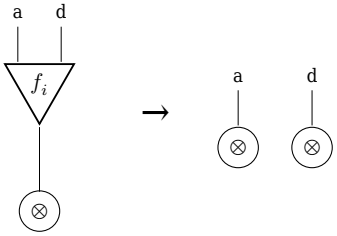


Figure 23: Eraser-fan commutation

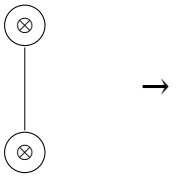


Figure 25: Eraser-eraser annihilation

11.3.1 Argument for correctness

11.4 Bespoke term encoding

- Primitive types (integer, string, bytes) ~> node types w/data
- n-arity primitive functions (+, -, *, concat, index) ~> rewrite rules
- alt-term encoding: $lamx.x + x + x$ ~> $3 * x$ rewrite rule instead of multi-node graph
- Over set of underlying opcodes & von-Neumann-ish VM (e.g. x86, LLVM, EVM).
- Use linear types (e.g. rewrite memory locations in place)
- Must be semantically equivalent to interaction system encoding after read-back
- Must preserve any-order reduction (exists some maximium `n` rewrite steps after which graph read-back must have final form regardless of order)
- Can relax diamond property (strong confluence)? unclear if necessary

11.5 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 utilized by the optimizer.

11.6 Future optimization strategies

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

11.6.1 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.

11.6.2 Speculative execution

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

11.6.3 Stochastic superoptimization

- Utilize 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 Superoptimization [12]
- Will need a lot of clever tricks to avoid getting stuck in local minima (that paper details several).
- See also STOKE [13]

11.6.4 "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 memoization 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 tradeoffs (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 ovearhead - have to increment coutner, 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 optimize (more a problem with using a low-level VM)

12.1.2 Prior cost calculation

Calculute 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 equalence 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 optimizations automagically 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 [8]
2. Could map depths of elementary linear logic terms to spacial depth in an execution visualization
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] Atkey, R. Syntax and semantics of quantitative type theory. https://bentnib.org/quantitative-type-theory.pdf.

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

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

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

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

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

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

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

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

[10] 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.

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

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

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

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

[15] 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/.

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