Juvix — Language Reference

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

23rd February 2020 - *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 [19] [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 [13] 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 [7] 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 Desiderata

What problems within the decentralised application ecosystem at large could be addressed at the language level?

What form should a better contract language take & what requirements must it fulfil?

4.1 Start with a pure functional basis

Many of the observed mistakes & bugs in smart contracts are not specific to the blockchain use-case at all — they are just common errors made in imperative languages by programmers everywhere: unintentional side-effects, type mismatches, and failure to handle exceptions or unexpected states.

Pure functional languages, with strongly-typed terms, immutable data, first-class functions, and referential transparency, are the right basis for a contract language. A strong typesystem enables the typechecker to catch many errors at compile time which would have otherwise resulted in misbehaviour at runtime. Purity, immutable data & referential transparency simplify the mental context a programmer must have, since they no longer need to reason about side effects. First-class functions allow the modularisation of complex logic into individually digestible, composable parts.

4.2 Enable an ecosystem of verification

"Formal verification" has been rendered a bit of a buzzword, but the actual need remains mostly unaddressed: a mathematically sound, legible way for users of decentralised applications to reason about what their interactions with an application will or will not do: whether a token really behaves like a currency, whether an exchange contract could accidentally steal their funds, or whether a derivative will always be sufficiently collataralised.

Exposing formal verification tooling to contract developers is necessary, but not sufficient — it must be possible for third-party software providers, such as wallet authors & application frontend developers, to embed proof-checking into their application and translate success or failure of verifying particular behaviours into legible UI elements — a green checkmark if the token contract is compliant, a red warning otherwise.

Contract property verification must be progressive — proof-construction UX improvements notwithstanding, it will always make little sense to invest time to verify beta applications or experiments. Developers must be able to write contracts, modify them, and progressively verify properties over time, often after the original code has been written.

The state machines of blockchains must eventually understand this verification system, and use that understanding to allow contracts to reason about how other contracts with which they interact will or will not behave, just as users can, and elect only to interact with contracts which can prove that they fulfil certain properties. This will require seamless integration of the verification system into the language itself.

4.3 Eliminate the compositional complexity ceiling

Distributed applications are most compelling, and most competitive with their centralised brethren, when they can benefit from the network effects of permissionless interoperation without the bottlenecks of trusted organisational relationships: anyone in the world can deploy a new contract which interacts with yours, perhaps to provide a new feature or application which you hadn't even conceived of. Alas, this potential network effect is severely curtailed by the sheer difficulty of reasoning about complex, multi-layered compositions of contracts. Because present languages cannot provide checkable guarantees about the behaviour of their own terms, each line of code in a contract must consider what every other line of code might do — to first order, this renders the mental cost of reasoning about the correctness of a contract quadratic in the size of the codebase. Developers must have the ability to articulate predicates which tightly constrain the behaviour of its own terms in the language itself, and thus eliminate entirely the necessity of tracking the behaviour of some particular dependency, since it has been constrained to exactly what is required and no more.

4.4 Provide predictable resource consumption

Resources consumed while executing contracts on replicated ledgers must be paid for. Illegibility and unpredictability of such payments are at present a major UX friction point for contract developers and application users alike. The resource cost of calling a contract must be exactly calculable or at least closely bounded prior to executing the call, so that developers can compare the costs of different code-paths at compile time and users can ensure that they have provided sufficient payment for complete execution. Should the ledger elect to integrate the typechecker into the state machine logic directly, it should be possible to perform these resource cost verifications instead of metering gas at runtime.

4.5 Abstract away from particular backends

Blockchains come & go, and the virtual machines or execution environments which they provide improve & change over time. Developers of decentralised applications should be able to abstract over these implementation particulars and write contracts at a sufficiently high-level to capture instead the essential semantics of their intentions, then redeploy these contracts to multiple ledgers and updated execution environments without rewriting them each time. Optimisations for particular execution environments or ledgers should be logically centralised, handled once by updated compiler pipelines instead of individually by each application developer. Successful abstraction will empower not only contract developers but also protocol engineers: virtual machines can be upgraded more quickly if contracts can easily be recompiled to target new ones, and exit costs of particular ledgers will be reduced if contract code can be re-targeted to other ones.

4.6 Retain efficient execution

Execution of contract code is expensive, whether replicated across validation nodes of a distributed ledger, or simulated within the restricted setting of an interactive prover. Formal verification, resource consumption proofs, and backend abstraction must not come at the cost of runtime performance penalties. The unique properties of the smart-contract use case — small code sizes and infrequent deployment — should be leveraged to enable optimisations which might not be feasible in a more general-purpose language.

5 Ingredients

What theoretical ingredients & practical architectural choices does Juvix make to realise these desiderata?

What changes are made from these prior systems, and why might one expect the solutions proposed to work?

5.1 Dependent types

5.1.1 Basics

The first and most essential ingredient is that of *dependent types*: a typesystem expressive enough to allow terms written in Juvix to express properties about the computation behaviour of other terms.

For example, the fairness of Tendermint proposer election has been partially verified in Idris, which has the same kind of dependent typesystem:

```
fairlyProportional :
  (idA : ProposerId) -> (idB : ProposerId) ->
  (wA : ProposerWeight) -> (wB : ProposerWeight) ->
  (pA : ProposerPriority) -> (pB: ProposerPriority) ->
  (n : Nat) ->
  (wA >= 0 = True) -> (wB >= 0 = True) ->
  (abs(pA - pB) <= (wA + wB) = True) ->
  ((count idA (snd (incrementElectMany n ((idA, wA, pA), (idB, wB, pB)))))
      >= ((n * (wA / (wA + wB))) - 1) = True,
   (count idA (snd (incrementElectMany n ((idA, wA, pA), (idB, wB, pB)))))
      <= ((n * (wA / (wA + wB))) + 1) = True)
```

`incrementElectMany`, the proposer-election function, is the term whose behaviour we want to reason about, and inhabiting the type of `fairlyProportional` with a valid term proves that it behaves in the fashion we intend.

In English, this proof could be read as "a validator, in a sequence of proposer elections where no other power changes take place, proposes no fewer blocks than the total blocks in the epoch multiplied by its fraction of stake less one, and proposes no more blocks than the total blocks in the epoch multiplied by its fraction of stake plus one".

5.1.2 Comparisons to other techniques

Dependent types as implemented in Juvix satisfy all three dimensions of the lambda cube: terms can depend on types, types can depend on types, and types can depend on terms. In accordance with the Curry-Howard correspondence, types in Juvix can express propositions, or properties of terms, and programs which inhabit those types serve as proofs of the propositions.

There are several different formal verification techniques, and they each have different trade-offs. Dependent types are quite expressive, since the conversion rule is just beta reduction, and allow for property verification in the same language, which removes the need for an error-prone translation between a specification and implementation. They allow for progressive verification — contracts can initially be written "unverified", and proofs can be added over time as the logic of an application-in-development solidifies. We have found that dependent types are elegant and become reasonably intuitive with practice, but more tooling support will undoubtedly be necessary to smooth the learning curve and ease some of the proof bureaucracy, which can be onerous at times.

5.1.3 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.

5.1.4 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.

5.2 Usage quantisation

The second theoretical ingredient Juvix brings to the table, used both to allow developers to reason even more precisely about the behaviour of their programs and to ensure that using dependent types for formal verification incurs no runtime cost, is usage quantisation, drawn from the Quantitative Type Theory paper and extended by us.

Typing judgements in Juvix are annotated with precise usages:

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

These usage annotations indicate how many times the subject of the judgement can be "computed with", or used. This is a form of linear types, but much more expressive: usage annotations are numerical and can depend on terms just like types can. Usage annotations are checked at compile-time for correctness. Since proofs of properties need not be computed over and can be constructed in the zero-usage fragment, Juvix can then erase proofs at compile time after checking them, ensuring that property verification creates no runtime overhead. Detailed usage accounting also allows Juvix to avoid garbage collection, optimise stack usage, and safely modify values in-place, in various situations depending on the types and the granularity of usage annotations provided.

5.3 Efficient execution

5.3.1 Whole-program optimisation

Juvix takes advantage of the unique trade-offs of the smart contract use case — small source-code sizes and infrequent deployment, which allow for longer compile times and wider optimisation searches — to enable several kinds of whole-program optimisations. Whole-program optimisation encapsulates a collection of techniques & optimisation passes rather than any specific one, drawn from multiple inspirations including GRIN, STOKE and Morte, including compile-time evaluation, inlining, fusion, case simplification, common sub-expression elimination, various dead code elimination checks, and rewriting provably equivalent terms.

5.3.2 Interaction net evaluation

For certain backends, Juvix can evaluate lambda calculus terms using the abstract model of interaction nets, as outlined by Lamping, Asperti, and Guerrini and inspired by prior work. Interaction nets minimise the number of beta reductions by sharing evaluations, perform runtime fusion, support automatic parallelism, and handle large closures efficiently. They come with some overhead, but this can be mitigated with bespoke encoding — generating custom rewrite rules at compile-time — for small, performance-critical functions.

5.3.3 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.

5.3.4 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.

5.3.5 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.

5.3.6 Programmer-directed optimisations

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-generated ones using a formalised interpreter for the machine model.

5.4 Resource verification

Leveraging the dependent typechecker already in place, Juvix uses techniques from a 2008 POPL paper to check computational resource consumption of terms without executing them by encapsulating computations in a cost monad, where costs can depend on terms in the usual dependent fashion. This requires some annotations by the developer, but tooling can be improved over time, and cost bounds can be added in a progressive fashion just like proofs of properties. In the future, if ledgers elect to integrate a limited version of the typechecker directly into their state machines, this resource verification mechanism can be used to eliminate runtime gas metering entirely: each contract would have a cost function, computable depending on arguments to the call, and the gas (exactly computable) could be deducted at the beginning of the function prior to execution. This resource verification model can also easily incorporate different denominations of resources (compute, storage, etc.) by tracking several costs at once.

5.5 Backend parameterisation

Juvix parameterises the type theory & core language over a set of primitive data types and primitive values, which can include native data types such as strings, integers, or sets, and native functions such as addition, subtraction, string concatenation, set membership, etc. The language & typechecker can then be instantiated over a particular backend which provides concrete sets of primitives and a primitive type-checking relation. Backends are integrated into the language itself as special values, so that the standard library can nicely abstract primitives with different underlying implementations into common typeclasses for numbers, strings, etc. and resolve operators to the correct backend at compile time as instructed by the developer. This will allow contract authors to write a single contract in the frontend language and target different backends with compiler switches or compile to more efficient upgraded versions of a virtual machine later on, with minimal or no code changes, and also eases the path towards integrated typechecking of applications with components on multiple backends (perhaps multiple ledgers and a zkSNARK circuit, for example).

6 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.

6.1 Lambda calculus

6.1.1 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.

6.1.2 Universality

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

6.1.3 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.

6.1.4 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.

6.2 Sequent calculus

Logical deduction ruleset & syntax for first-order logic.

6.3 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

6.4 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

6.5 Linear logic

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

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

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

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

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

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

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

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

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

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

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

Figure 3: Typing rules for linear logic

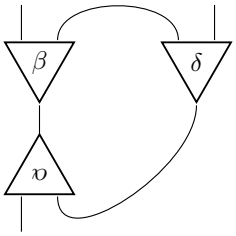
- Explain with chef analogy.

6.6 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 [8].

6.7 Elementary affine logic

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

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

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

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

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

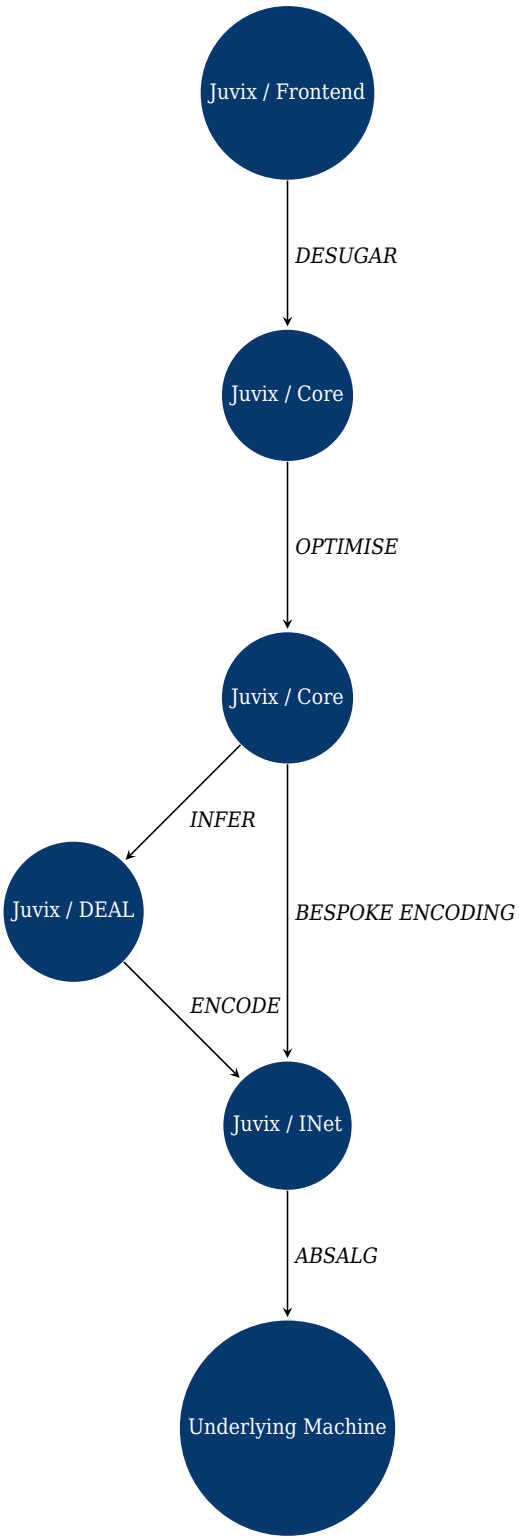
$$\frac{\Gamma \vdash !A \qquad !A, ..., !A, \Delta \vdash B}{\Gamma, \Delta \vdash B}\ \text{contr}$$

Figure 4: Typing rules for elementary affine logic

6.8 Optimal reduction

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

7 Architectural overview



8 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, and how that syntax is elaborated to core. Juvix aims to keep the core language as simple as possible to minimise the chance of mistakes in the type theory or type-checker implementation and faciliate easy optimisation, so most high-level frontend features such as data-types, pattern matching, and type-classes are elaborated into plain core terms in the transformation from the frontend language.

8.1 Syntax

Juvix's frontend syntax is primarily inspired by Haskell & Idris, with alterations for explicit usage accounting & (in the future) for resource accounting.

```
zipWith : n (a -> b -> c) -> Vect n a -> Vect n b -> Vect n c
zipWith _ Nil       Nil       = Nil
zipWith f (x :: xs) (y :: ys) = f x y :: zipWith f xs ys

zipAdd : Num a => Vect n a -> Vect n a -> Vect n a
zipAdd = zipWith (+)

test_zip_add : zipAdd (1 :: 2 :: 3) = 6
test_zip_add = Refl
```

elaborates to

```
zipWith : (a : 0 Type) -> (b : 0 Type) -> (c : 0 Type) -> (n : 0 Nat) -> n (a -> b -> c) -> Vect n a -> Vect n b -> Vect n c
zipWith (a : 0 Type) (b : 0 Type) (c : 0 Type) (Z : 0 Nat)   _ Nil        Nil       = Nil
zipWith (a : 0 Type) (b : 0 Type) (c : 0 Type) (S k : 0 Nat) f (x :: xs)  (y :: ys) = f x y :: zipWith a b c k f xs ys

zipAdd : (a : 0 Type) -> (n : 0 Nat) -> (d : n (Num a)) -> Vect n a -> Vect n a -> Vect n a
zipAdd a n d xs ys = zipWith a a a n ((+) d) xs ys
```

Usage annotations are optional. Implicit usage arguments or constraints are inferred where possible, but explicit annotations may sometimes be required.

```
f : 2 (x : 3 Int) -> Double

f : w (x : 2 Int -> y : () -> IO ())
```

By default, no usage annotation is equivalent to an implicit usage argument. The unification algorithm will attempt to infer constraints involving multiple usage arguments where possible.

8.2 Features

All features desugar to Core (with several passes, typeclass lookup, etc).

8.2.1 Datatypes

Similary to Idris & Agda, datatypes are defined with inductive family `data` declarations:

```
data D Δ : Γ → *_i where
  c_1 : θ_1 → D Δ t_1
  c_n : θ_n → D Δ t_n
```

- `D` is the datatype family.
- `c_1` through `c_n` are the constructors, with given types.
- `t_1` through `t_n` (vectors) are the indices for the constructor return types.
- `Δ` are the parameters, scoped over the types of the constructors, which must be constant in their targets.
- `Γ` are the indices, which can vary over the type of the constructor targets.

In general, constructor types can reference `D` inductively. Different parameterisations of core may represent datatypes in different ways, and might impose restrictions such as positivity (`D` can only occur to the left of an even number of arrows in the constructor type) or strict positivity (`D` cannot occur to the left of an arrow anywhere in the constructor type).

Mutually inductive-recursive definitions will be supported at a later date.

8.2.2 Record types

Record types can be defined with the `record` keyword:

```
record R : Γ → *_i where
  constructor RC
  x : A
  y : B x
  z : C x y
```

This declaration creates a type `R`, constructor `RC`, and accessors `x`, `y`, and `z`. Note that the type of `y` depends on `x`, and the type of `z` depends on `x` and `y`. `Δ` are the parameters, scoped over the types of the fields and the type of the constructor.

Record types & accessor functions desugar to dependent pairs (although the frontend typechecker treats separate record declarations with the same fields as separate types).

8.2.3 Dependent pattern matching

Pattern matching is available in function definitions, case expressions, and let-bindings.

Pattern matching is dependent, so that matching on some arguments may restrict the types of others.

Patterns are separated into accessible & inaccessible patterns following Agda.

8.2.4 Syntactical do-notation

Actions in any monad can be sequenced with `do`-notation:

```
func : m Int
func = do
  one <- funcOne
  two <- funcTwo one
  return (two + 3)
```

`do`-notation desugars into `bind` (`>>=`) and `return` as defined by the monad instance.

8.2.5 Typeclasses

Typeclasses will come in a future frontend language release.

8.2.6 Implicit arguments

Similarly to Idris or Agda, arguments can be omitted in type annotations & function calls, and their values will be inferred. When necessary, implicit arguments can be referenced, captured, or passed with `{}` syntax:

```
id : { a : *_i } → a → a
id x = x
```

There are no syntactic restrictions on usage of implicit arguments. If the compiler cannot infer a value, an error will be thrown.

8.2.7 Type inference

Type inference, where possible, is facilitated by unification.

8.2.8 Usage inference

Usage inference, where possible, is facilitated by unification.

8.2.9 Holes

Holes stand for parts of terms which have not yet been defined, but which the typechecker can still reason about in some limited capacity by their context.

Similarly to Idris, holes can be created by prefixing a question mark to an identifier:

```
term : 1 Int
term = ?term_rhs
```

8.2.10 Modules

The module system, inspired primarily by Agda, is used to split large programs into separately scoped units of code referred to as “modules”.

Modules are not data — they exist only at compile-time. Algebraic structures should instead be modeled with datatypes.

8.2.11 Automatic proof search

Automatic split & fill a la Agda `auto`.

See paper.

Possible alterations/additions:

- Alterations to take into account usage constraints (when known)
- Try case-splitting where applicable (& continue searching)
- Attempt to prove the negation
- Include constants in scope, use A-star search instead of DFS, with a heuristic based on usage correlation across modules

In the future, maybe interface with an automated prover.

8.3 Elaboration to core

8.3.1 Tactics

Tactics as inspired by Idris.

8.3.2 Lambda-encoding of algebraic datatypes

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 $\mathbb{N}$. 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 $(\lambda d.df)$ 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)` [16]. 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.

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[17]. Due to this, Scott encodings can be typed safely in Juvix Core.

Desugaring For 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.

8.3.3 Unification

Unification from: - Higher-Order Dynamic Pattern Unification for Dependent Types and Records - Towards Unification for Dependent Types - Idris 2 codebase - A Tutorial Implementation of Dynamic Pattern Unification

8.4 Future plans

In the future, Juvix might support alternative frontend syntaxes, such as:

- A Lisp-like syntax, inspired in part by Shen
- A visual data-flow representation, inspired in part by Enso

9 Core language

9.1 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 [7], and Cedille [18]. 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.

9.2 Changes from QTT

9.2.1 Primitive types, constants, and functions

Nothing too exciting here. These will vary based on the machine target.

9.2.2 Additional linear logic connectives

Multiplicative disjunction Why?

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

Dependent additive conjunction Why?

- Resourceful production
- Translated as multiplicative conjunction, but only one of fst or snd can be used
- Avoids duplication of resources at runtime
- Can be represented as a function, but wouldn't have the same evaluation semantics

Self types Why?

- Elegant induction for lambda-encoded data
- No runtime presence

9.2.3 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)
```

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 ω 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.

Usage polymorphism in the type theory Add a $\forall u.\, T$, 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 ω).

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 $(x \overset{\pi}{:} S) \to T$ 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)

9.3 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 and usage.

Let F be the set of primitive functions, where each f is related to a function type, including an argument usage annotation, by the $\vdots$ relation and endowed with a reduction operation $\to_f$, 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 Core is erased, so they must be directly supported by the low-level execution model. The core type theory and subsequent compilation pathways are parameterised over K, C, F, $\vdots$, and the reduction operations $\to_f$, which are assumed to be available as implicit parameters.

9.4 Syntax

Inspired by the bidirectional syntax of Conor McBride in I Got Plenty o' Nuttin' [9].

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)
$\mid (x \overset{\pi}{:} S) \to T$	function type	[π] S -> T
$\mid (x \overset{\pi}{:} S) \otimes T$	dependent multiplicative conjunction type	$([\pi]S,\ T)$
$\mid (x \overset{\pi}{:} S)\ \& T$	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)
$\mid f \in F$	primitive function	(varies)
$\mid fs$	application	fs
$\mid (s, t)$	multiplicative conjunction	(s, t)
$\mid s \epsilon t$	additive conjunction	TBD
$\mid s \gamma t$	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 multiplicative conjunction pattern match	$let\ (x, y) = d\ in\ e$
$\mid \odot e$	multiplicative disjunction destructor	$join\ e$
$\mid s \overset{\pi}{:} S$	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 $\diamond$ 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 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 $\sigma \in 0, 1$.

Primitive constant term judgements have the form:

$$\vdash M \overset{\gamma}{:} S \tag{4}$$

where γ is any element in the semiring.

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.

For example, consider 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. x can also be annotated as ω, indicating that it can be used (computed with) an arbitrary number of times.

9.5 Typing rules

9.5.1 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

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

$\sigma = 0$ fragment only.

9.5.2 Primitive constants, functions & types

Constants

Formation & introduction rule

$$\frac{c \in C \qquad \kappa \in K \qquad c : (\gamma, \kappa)}{\vdash c \overset{\gamma}{:} \kappa} \text{Prim-Const}$$

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

Functions

Formation & introduction rule

$$\frac{f \in F \qquad f : (\gamma, (x \overset{\pi}{:} S) \to T)}{\vdash f \overset{\gamma}{:} (x \overset{\pi}{:} S) \to T}\ \text{Prim-Fn}$$

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

Primitive functions can be dependently-typed.

Elimination rule Primitive functions use the same elimination rule as native lambda abstractions.

9.5.3 Dependent function types

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

Formation rule

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

Introduction rule

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

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:

Elimination rule

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

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

9.5.4 Dependent multiplicative conjunction (tensor product)

Colloquially, “pair”. Can be dependent.

Formation rule

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

Type formation does not require any resources.

Introduction rule

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

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

Elimination rules

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

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

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

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

Under the resourceful part, both elements of the conjunction must be matched and consumed.

To-do:

- Simplifies to fst, snd in $\sigma = 0$ fragment (should we combine the rules?)
- If we lambda-encoded pairs, is that isomorphic?

9.5.5 Additive conjunction

Colloquially, “choose either”. Can be dependent.

Formation rule

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

To-do: 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

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

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

9.5.6 Multiplicative disjunction

Colloquially, “both separately in parallel”. 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

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

Elimination rule

$$\frac{\Gamma \vdash M\gamma\ N \overset{\sigma}{:} S \parr T}{\Gamma \vdash \odot\ (M\gamma\ N) \overset{\sigma'}{:} (S \otimes T)}\ \parr\text{-Elim}$$

$\odot$ can be thought of as a syntax-directed “join” operator.

9.5.7 Self types

Formation rule

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

Introduction rule

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

Elimination rule

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

To-do: syntax for self types?

9.5.8 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.

9.5.9 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 need a rule for term equality?

9.5.10 Inductive primitive types

Primitive types & values can also be defined which reference terms & eliminators, respectively, and come with custom derivation rules.

For example, consider the equality type `Eq` and constructor `Refl`.

Formation rule:

$$\frac{\Gamma \vdash t \overset{0}{:} *_i \qquad \Gamma \vdash x \overset{0}{:} t \qquad \Gamma \vdash y \overset{0}{:} t}{\Gamma \vdash Eq\ t\ x\ y \overset{0}{:} *_i}\ \text{Eq-Form}$$

Introduction rule:

$$\frac{\Gamma \vdash t \overset{0}{:} *_i \qquad \Gamma \vdash x \overset{0}{:} t}{\Gamma \vdash Refl\ t\ x \overset{0}{:} Eq\ t\ x\ x}\ \text{Eq-Intro}$$

If $x \to_\beta y$, the type $Eq\ t\ x\ y$ should reduce to $Eq\ t\ x\ x$ by the conversion rule, so $Refl\ t\ x \overset{0}{:} Eq\ t\ x\ y$ should typecheck.

Elimination rule:

$$\frac{\Gamma \vdash t \overset{0}{:} *_i \qquad \Gamma \vdash m \overset{0}{:} (x \overset{0}{:} t) \to (y \overset{0}{:} t) \to (e \overset{0}{:} Eq\ t\ x\ y) \to *_i \qquad \Gamma \vdash n \overset{0}{:} (z \overset{0}{:} t) \to m\ z\ z\ (Refl\ t\ z) \qquad \Gamma \vdash x \overset{0}{:} t, y \overset{0}{:} t, e \overset{0}{:} Eq}{\Gamma \vdash eqElim\ t\ m\ n\ x\ y\ e \overset{0}{:} m\ x\ y\ e}$$

(evaluates to `n x`)

Colloquially, if you have a type dependent on two values & a proof of their equality, and a function from one value to a value of that type, you can eliminate two distinct values & a proof of their equality to derive a value of the type, dependent on both values, which evaluates as the function applied to either one.

9.5.11 Sub-usaging

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

9.6 Reduction semantics

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

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

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

9.6.1 Confluence

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

9.6.2 Parallel-step reduction

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

> Note: The theorem prover will support a special n-step beta equality, where the step count refers to contraction rule applications.

Basic lambda calculus

$$\frac{}{*_i \rhd *_i}$$

$$\frac{}{x \rhd x}$$

$$\frac{S \rhd S' \qquad T \rhd T'}{(x : S) \to T \rhd (x : S') \to T'}$$

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

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

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

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

Linear connectives

Multiplicative conjunction

$$\frac{S \rhd S' \qquad T \rhd T'}{(x : S) \otimes T \rhd (x : S') \otimes T'}$$

$$\frac{s \rhd s' \qquad t \rhd t'}{(s, t) \rhd (s', t')}$$

$$\frac{z \rhd (m, n) \qquad m \rhd m' \qquad n \rhd n' \qquad s \rhd s'}{let\ (x, y)\ =\ z\ in\ s \rhd s'[x := m', y := n']}$$

Reduction takes place inside a multiplicative conjunction.

Multiplicative disjunction

$$\frac{S \rhd S' \qquad T \rhd T'}{S \parr T \rhd S' \parr T'}$$

$$\frac{s \rhd s' \qquad t \rhd t'}{s \gamma t \rhd s' \gamma t'}$$

$$\frac{t \rhd m \gamma n \qquad m \rhd m' \qquad n \rhd n'}{\odot t \rhd (m', n')}$$

Reduction takes place inside a multiplicative disjunction.

Additive disjunction

$$\frac{S \rhd S' \qquad T \rhd T'}{(x : S) \& T \rhd (x : S') \otimes T'}$$

$$\frac{t \rhd m \epsilon n \qquad m \rhd m'}{fst_{\&}\ t \rhd m'}$$

$$\frac{t \rhd m \epsilon n \qquad n \rhd n'}{snd_{\&}\ t \rhd n'}$$

Reduction does not take place until a destructor has been applied.

9.6.3 Self types

$$\frac{T \rhd T'}{\iota x.T \rhd \iota x.T'}$$

9.6.4 Primitives

$$\frac{\kappa \in K}{\kappa \rhd \kappa}$$

$$\frac{c \in C}{c \rhd c}$$

Primitive types and primitive constants reduce to themselves.

$$\frac{f \in F \qquad x \rhd x' \qquad x' \to_f y}{fx \rhd y}$$

Primitive functions reduce according to the reduction operation defined for the function.

9.7 Typechecking

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

9.8 Termination checking

Idris does the obvious things (structural size decrease) + maybe a bit more.

See the Idris paper p.14 and the original paper.

This dissertation seems to cover a variety of tactics.

Also the F* tutorial, section 5 & 5.1.2, may describe a more advanced method.

9.9 Examples

9.9.1 SKI combinators

S combinator The dependent S ("substitution") combinator can be typed as: $\lambda t1.\lambda t2.\lambda t3.\lambda x.\lambda y.\lambda z.xz(yz) : (x : ((a : t1) \to (b : t2) \to t3)) \to (y : ((a : t1) \to t2)) \to (z : t1) \to t3$.

This should also typecheck if the x, y, and z argument usages are replaced with w (instead of 1 and 2).

K combinator The dependent K ("constant") combinator can be typed as: $\vdash \lambda t1.\lambda t2.\lambda x.\lambda y.x : (t1 : *_i) \to (t2 : *_i) \to (x : t1) \to (y : t2) \to t1$.

This should also typecheck if the x and y argument usages are replaced with w (instead of 1 and 0).

I combinator The dependent I ("identity") combinator can be typed as: $\vdash \lambda t.\lambda x.(x : t) : (t : *_i) \to (x : t) \to t$.

This should also typecheck if the x argument usage is replaced with w (instead of 1).

9.9.2 Church-encoded natural numbers

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

This should also typecheck if the s argument usage is replaced with w (instead of n for some specific n).

9.9.3 Primitive equality

Assume a primitive type of naturals, including literals, and a primitive addition function.

Define an `Eq` type:

$$\frac{t \overset{0}{:} *_i \qquad x \overset{0}{:} t \qquad y \overset{0}{:} t}{Eqtxy \overset{0}{:} *_j} Eq - Form$$

And a single `Refl` constructor:

$$\frac{t \overset{0}{:} *_i \qquad x \overset{0}{:} t}{Refl \overset{0}{:} Eqtxx} Refl$$

Then $Refl \overset{0}{:} EqNat(1+1)2$ can be derived, since the conversion rule will be applied when checking the (annotated) $Refl$.

9.9.4 Linear induction

To-do: Linear induction example.

10 Core optimisation

Juvix combines compiler-directed & user-directed optimising transformations into a single whole-program core 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 core 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.

10.1 User-directed 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.

10.2 Optimisation by normalisation

Juvix optimises expressions by normalising — beta-reducing & eta-reducing — them at compile-time, including under lambda expressions.

When using exclusively lambda-encoded datatypes, this makes all abstractions free (at runtime).

10.3 Graph transformations

Primarily inspired by the GRIN [12] 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

11 Erased core language

The erased core captures the operational semantics of the core language, erasing contemplated (zero-usage) terms and dropping type & usage annotations which are not computationally necessary. Core terms can be erased directly or first passed through the elementary affine stratification checker in order to test compatibility with the abstract algorithm.

Analogously to core, the erased core language is parameterised over a set C of primitive constants, a set F of primitive functions, and a set $\rightarrow_f$ of associated reduction rules.

11.1 Syntax

Let u, v, w be terms.

$u, v, w ::= x$	variable
$\mid c \in C$	primitive constant
$\mid f \in F$	primitive function
$\mid \lambda x.u$	abstraction
$\mid u\ v$	application
$\mid (u, v)$	multiplicative conjunction
$\mid u\ \epsilon\ v$	additive conjunction
$\mid u\ \gamma\ v$	multiplicative disjunction
$\mid fst_{\&}\ u$	first projection for additive conjunction
$\mid snd_{\&}\ u$	second projection for additive conjunction
$\mid\ \odot e$	multiplicative disjunction destructor
$\mid let\ (x, y) = u\ in\ v$	multiplicative conjunction pattern match

Figure 7: Erased core syntax

11.2 Erasure from core

Define the core erasure operator $\blacktriangleright$.

Erasure judgements take the form $\Gamma \vdash t \overset{\sigma}{:} S \blacktriangleright u$ with $t \overset{\sigma}{:} S$ a core judgement and u an erased core term.

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

Note that $\sigma / = 0$ must hold, as the erasure of a computationally irrelevant term is nothing.

11.2.1 Primitives & lambda terms

$$\frac{c \overset{\sigma}{:} S \qquad \sigma / = 0}{c \overset{\sigma}{:} S \blacktriangleright c}\ \text{Prim-Const-Erase-+}$$

$$\frac{f \overset{\sigma}{:} S \qquad \sigma / = 0}{f \overset{\sigma}{:} S \blacktriangleright f}\ \text{Prim-Fun-Erase-+}$$

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

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

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

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

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

$$\frac{\Gamma \vdash s \overset{\pi}{:} S \qquad s \blacktriangleright u \qquad \pi / = 0}{\Gamma \vdash s \overset{\pi}{:} S \blacktriangleright u}\ \text{Ann-Erase-+}$$

In the *Lam-Erase-0* rule, the variable x bound in t will not occur in the corresponding u, 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.

11.2.2 Linear connectives

Multiplicative conjunction

Constructor

$$\frac{\Gamma \vdash (s,t) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \qquad \sigma / = 0 \qquad \pi / = 0 \qquad s \blacktriangleright u \qquad t \blacktriangleright v}{\Gamma \vdash (s,t) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \blacktriangleright (u,v)}\ \text{let-Erase-++}$$

If the first element of the pair is used, the constructor is erased to the untyped constructor.

$$\frac{\Gamma \vdash (s,t) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \qquad \sigma / = 0 \qquad \pi = 0 \qquad t \blacktriangleright v}{\Gamma \vdash (s,t) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \blacktriangleright v}\ \text{let-Erase-0+}$$

If the first element of the pair is not used, the constructor is erased completely.

Destructor

$$\frac{\Gamma_1 \vdash s \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \qquad \Gamma_1 + \Gamma_2 \vdash \mathit{let}\ (x,y) = s\ \mathit{in}\ t \overset{\sigma'}{:} M[z := (x,y)] \qquad \sigma, \sigma' / = 0 \qquad s \blacktriangleright u \qquad t \blacktriangleright v}{\Gamma_1 + \Gamma_2 \vdash \mathit{let}\ (x,y) = s\ \mathit{in}\ t \overset{\sigma'}{:} M[z := (x,y)] \blacktriangleright \mathit{let}\ (x,y) = u\ \mathit{in}\ v}\ \otimes\text{-Erase-++}$$

If the pair is used, the destructor is erased to the untyped destructor.

$$\frac{\Gamma_1 \vdash s \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \qquad \Gamma_1 + \Gamma_2 \vdash \mathit{let}\ (x,y) = s\ \mathit{in}\ t \overset{\sigma'}{:} M[z := (x,y)] \qquad \sigma = 0 \wedge \sigma' / = 0 \qquad t \blacktriangleright v}{\Gamma_1 + \Gamma_2 \vdash \mathit{let}\ (x,y) = s\ \mathit{in}\ t \overset{\sigma'}{:} M[z := (x,y)] \blacktriangleright v}\ \otimes\text{-Erase-0+}$$

If the pair is not used, the destructor is erased completely.

Multiplicative disjunction

Constructor

$$\frac{\Gamma \vdash (s\ \gamma\ t) \overset{\sigma}{:} (S \parr T) \qquad \sigma / = 0 \qquad s \blacktriangleright u \qquad t \blacktriangleright v}{\Gamma \vdash (u\ \gamma\ v) \overset{\sigma}{:} (S \parr T) \blacktriangleright (u\ \gamma\ v)}\ \gamma\text{-Erase-+}$$

The constructor is erased to the untyped constructor.

Destructor

$$\frac{\Gamma \vdash \odot\, M \overset{\sigma}{\vdots} (S \otimes T) \qquad \sigma / = 0 \qquad M \blacktriangleright u}{\Gamma \vdash \odot\, M \overset{\sigma}{\vdots} (S \otimes T) \blacktriangleright\ \odot u}\ \odot\text{-Erase-+}$$

The destructor is erased to the untyped destructor.

Additive conjunction

Constructor

$$\frac{\Gamma \vdash (s\ \epsilon\ t) \overset{\sigma}{\vdots} (x \overset{\pi}{\vdots} S)\ \&\ T \qquad \sigma / = 0 \qquad s \blacktriangleright u \qquad t \blacktriangleright v}{\Gamma \vdash (s\ \epsilon\ t) \overset{\sigma}{\vdots} (x \overset{\pi}{\vdots} S)\ \&\ T \blacktriangleright u \epsilon v}\ \epsilon\text{-Erase-+}$$

The constructor is erased to the untyped constructor.

Question: what if $\pi = 0$?

Destructors

$$\frac{\Gamma \vdash fst_{\&}\ t \overset{\sigma}{\vdots} S \qquad \sigma / = 0 \qquad t \blacktriangleright u}{fst_{\&}\ M \blacktriangleright fst_{\&}\ u}\ \text{fst-Erase-+}$$

$$\frac{\Gamma \vdash snd_{\&}\ t \overset{\sigma}{\vdots} S \qquad \sigma / = 0 \qquad t \blacktriangleright u}{snd_{\&}\ M \blacktriangleright snd_{\&}\ u}\ \text{snd-Erase-+}$$

The destructors are erased to the untyped destructors.

11.3 Erasure from elementary affine core

Drop the !s and the $\bar{!}$s, discard type information.

11.4 Reduction semantics

As core, sans the types.

12 Elementary affine core

Intermediary form to check stratification for compatibility with the abstract algorithm.

12.1 Syntax

Let erased core terms be t and core types be T.

Define elementary affine core formulae as $A, B ::= \ T \mid !A$.

$$
\begin{array}{rr}
T ::= *_i & \text{sort } i \\
\mid \kappa \in K & \text{primitive type} \\
\mid (x : S) \to T & \text{dependent function type} \\
\mid (x : S) \otimes T & \text{dependent multiplicative conjunction type} \\
\mid (x : S) \,\&\, T & \text{dependent additive conjunction type} \\
\mid T \parr T & \text{non-dependent multiplicative disjunction type} \\
\mid !T & \text{bang-type}
\end{array}
$$

Define elementary affine core terms as $u, v ::= t \mid !u \mid \bar{!}u$.

Elementary affine core operates on an annotated version of erased core but requires preservation of a type mapping such that the types of free variables can be looked up.

12.2 Typing rules

As core, except:

- Usage annotations erased
- All variables must be linear
- Added weakening rule
- Different rules for contraction & promotion
- Variables can occur twice in additive conjunction (but no more)

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

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

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

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

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

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

Figure 9: Typing rules for EAL

12.3 Box placement inference

Adapted from previous work [11].

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).*

12.3.1 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:

$$path(x, x) = nil \tag{5}$$
$$path(t_1 t_2, x) = path(t_i, x) \text{ where } t_i \text{ contains } x \tag{6}$$
$$path(\lambda y.t, x) = path(t, x) \tag{7}$$
$$path(!t, x) = ! :: path(t, x) \tag{8}$$
$$path(\bar{!}t, x) = \ \bar{!}t :: path(t, x) \tag{9}$$

To-do: changes for full core, should be trivial.

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

$$s(nil) = 0 \tag{10}$$
$$s(! :: l) = 1 + s(l) \tag{11}$$
$$s(\bar{!} :: l) = -1 + s(l) \tag{12}$$

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

$$\forall l \leq path(t, x), s(l) \geq 0 \text{ for any occurrence of a variable } x \text{ in } t \tag{13}$$
$$\forall x \in FV(t), s(path(t, x)) = 0 \text{ (zero sum paths for free variables)} \tag{14}$$

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

$$\forall t_i \in subterms(t), well - bracketed(t_i) \tag{15}$$

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:

$$\Gamma(!u) = !A \text{ if } \Gamma'(u) = A \tag{16}$$
$$\Gamma(\bar{!}u) = A \text{ if } \Gamma'(u) = !A, \text{ undefined otherwise} \tag{17}$$
$$\Gamma(\lambda x.u) = A \multimap B \text{ if } \Gamma'(x) = A, \Gamma'(u) = B \tag{18}$$
$$\Gamma(t_1 t_2) = B \text{ if } \Gamma'(t_2) = A \text{ and } \Gamma'(t_1) = A \multimap B, \text{ undefined otherwise} \tag{19}$$

To-do: changes for full core, trivial except dependent types: function result type is dependent, bangs work the same way.

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

$$\Gamma(t_i) \text{ is defined for all subterms } t_i \text{ of } t \tag{20}$$
$$\text{for any variable } x \text{ of at least 2 occurrences, } \Gamma(x) = !B \text{ for some } B \tag{21}$$

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. $t = \bar{!}u$
    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 $t = !v[x_1 := \bar{!}u_1, ...x_n := \bar{!}u_n]$, 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. $path(t, y) = ! :: path(v, x_i) :: \bar{!} :: path(u_i, y)$
        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 $\Gamma'(x_i)$ = $\Gamma(\bar{!}u_i)$ 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 $v_i' = v_i[x_1 := \bar{!}u_1, ...x_n := \bar{!}u_n]$, 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 $\Gamma(\bar{!}u_i)$ 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

To-do: update for full core. Should be trivial, except possibly dependent type case.

12.3.2 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?

12.3.3 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 $\bar{\Theta}(x) = \Theta(\bar{x})$, 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.

12.3.4 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:

$$U(!^m\alpha, !^n\alpha) = \{m = n\} \tag{22}$$
$$U(!^m(A_1 \multimap A_2), !^n(B_1 \multimap B_2)) = \{m = n\} \cup U(A_1, B_1) \cup U(A_2, B_2) \tag{23}$$
$$U(\_, \_) = \{false\} \tag{24}$$

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:

$$\Sigma(!^n a) = !^m A \text{ if } \Sigma(a) = !^k A \tag{25}$$
$$\Sigma(\lambda x.u) = !^m(A \multimap B) \text{ if } \Sigma(x) = A, \Sigma(u) = B \tag{26}$$
$$\Sigma(u_1 u_2) = B \text{ if } \Sigma(u_1) = A \multimap B \text{ and } \Sigma(u_2) = A \tag{27}$$

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 $m \geq 0$

To-do: update for full core. Changes:

- variables can occur once on each side of an additive conjunction without the constraint
- do we need a constraint for the value in dependent types? can it appear as a computational variable? e.g. a function from types to types (identity), then it would be computed with though. maybe we don't need to do anything here, but should reason about it

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 $C^{typ}(t, \Sigma)$.*

12.3.5 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.

13 Low-level execution model

13.1 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

13.2 Interaction system encoding

13.2.1 Basic lambda calculus

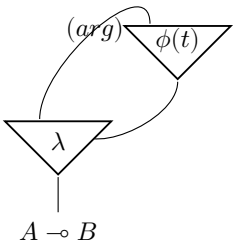
EAL term language $t ::= x \mid \lambda x.t \mid (tu) \mid !t$.

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.

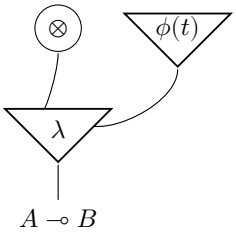
The EAL term is first erased to a simply-typed term, with EAL types and levels of subterms retained in a lookup table for reference during the translation.

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

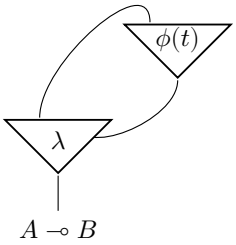


Wiring of the argument x varies depending on variable usage linearity:

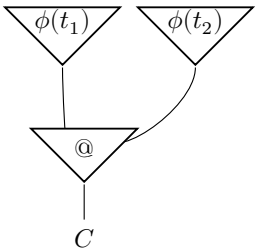
Weakening: If x does not appear in the body t, the λ argument port is connected to an eraser.



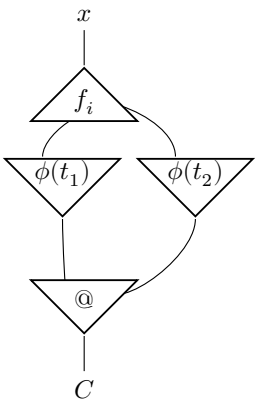
Linear / contraction: If *x* appears once or more in the body t, the λ argument port is connected to the occurrence(s). If there is more than one occurrence, usages will be shared by a set of fan nodes constructed by the application encoding.



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.

Rewrite rules 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.

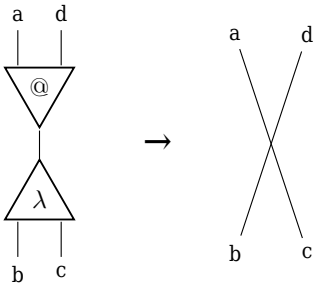


Figure 10: Lambda-application annihilation (beta reduction)

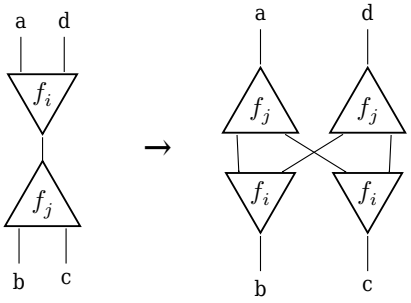


Figure 12: Fan-fan commutation

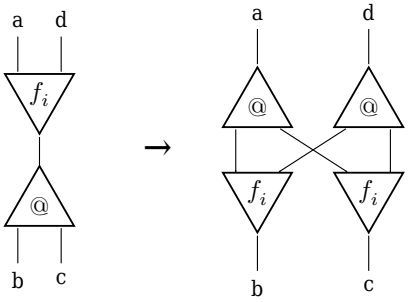


Figure 14: Fan-application commutation

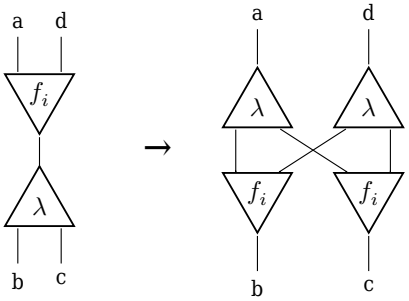


Figure 16: Fan-lambda commutation

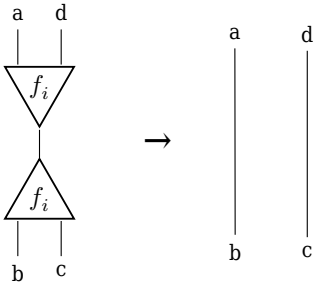


Figure 18: Fan-fan annihilation

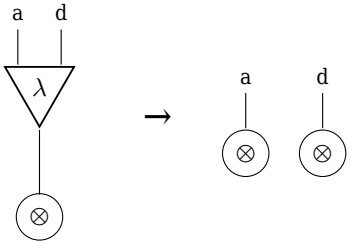


Figure 20: Eraser-lambda commutation

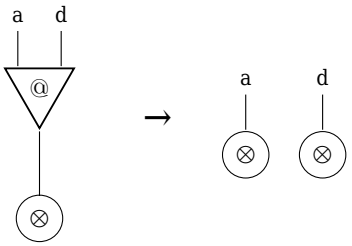


Figure 22: Eraser-application commutation

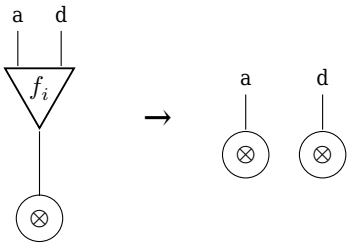


Figure 24: Eraser-fan commutation

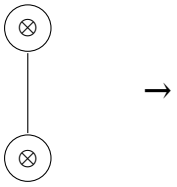


Figure 26: Eraser-eraser annihilation

13.2.2 Linear connectives

Multiplicative conjunction

Constructor

Destructor

Rewrite rules

Multiplicative disjunction Interaction net encoding of $a \parr b$:

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.

Constructor

Destructor

Rewrite rules

Additive conjunction 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.

Constructor

Destructor

Rewrite rules

13.2.3 Primitive constants

Constructors Primitive constants are encoded as simple custom nodes.

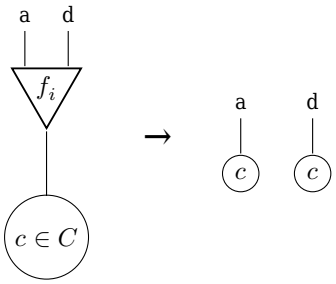


Figure 28: Constant-fan commutation

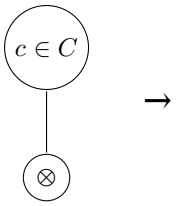
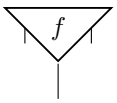


Figure 30: Constant-eraser annihilation

Rewrite rules

13.2.4 Primitive functions

To-do: we need curried functions for this case too. Should we just combine this with bespoke encoding?



Constructors Functions f of arity n are encoded as custom nodes with $n - 1$ auxiliary ports.

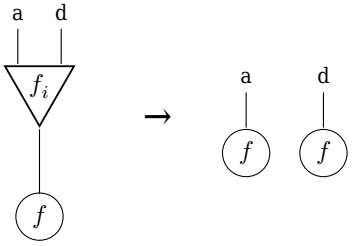


Figure 32: Function-fan commutation

Rewrite rules To-do: this is wrong, deal with n ports.

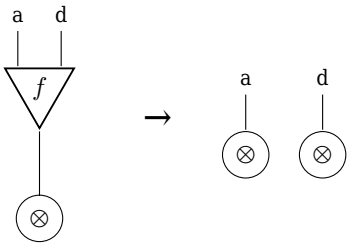


Figure 34: Function-eraser commutation

Generalises to n auxiliary ports - a new eraser node is attached to each.

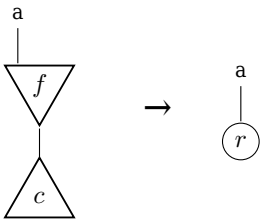


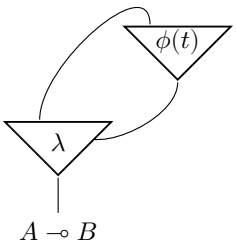
Figure 36: Function-constant application

Where r is the result of reducing fc according to the defined rule $\rightarrow_f$.

13.2.5 Bespoke encoding

Functions 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.

Datatypes

- Primitive types (integer, string, bytes) ~> node types w/data (same as constants)

13.3 Argument for correctness of the abstract algorithm

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 $\rightarrow$ lam x . t (trivial)
    2. x $\rightarrow$ x (trivial)
    3. !x $\rightarrow$!x (trivial)
    4. ! $\bar{!}$ x $\rightarrow$ x (trivial) (defined-ness guaranteed by well-bracketed property)
    5. (lam x . a) b $\rightarrow$ 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)

13.4 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

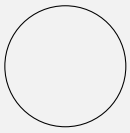
Lemma 13.1: AST→Net has one free port

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.

$\therefore$ AST→Net has one free port

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

Theorem 13.1: AST→Net→Eval has one free port

Let $A = Net(AST)$.

By lemma 13.1 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 ⋯ drawn on the images

Case 1 - Rewire

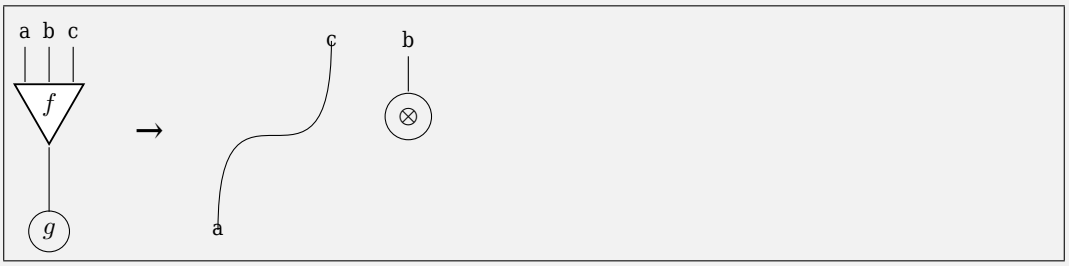
Nodes that fall under this case:
1. And↔T
2. And↔F
3. Or↔T
4. Or↔F

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.

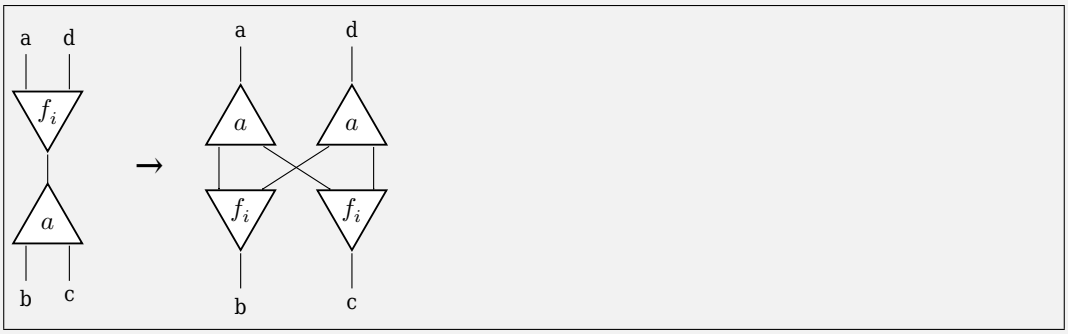
Additionally an eraser node is connected to the b, since the eraser node only has a main port, the number of free ports is not affected.

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 13.1: 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 13.2: 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 13.1 this net originated from a valid EAL-AST which is a subset of BOHM.

Thus by Theorem 13.1 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

13.5 Evaluation strategies

13.5.1 Simple

First version: order-independent sequential reduction. - Reduce primary pairs until we run out of primary pairs. - Track a set of outstanding primary pairs. We will have an initial set when we create the graph. Whenever we connect ports, if the two ports are primary, add the pair to the set. When we reduce a pair, remove it from the set. (I think this will go in `linkPorts` or `relink`, one of those) - Pick primary pairs from the set in any order and reduce them.

(in practice this set can be a stack or something) (but we should try randomly reducing one of the pairs just to test)

Second version: order-independent parallel reduction the simple way, where we just have some thread pool of constant `n` threads and have each reduce a primary pair at once and somehow track the set in a synchronised way. From this we should learn what kind of locks or lock-free data structures or concurrency primitives (e.g. atomics) we need and will work most efficiently for parallel reduction.

Third version: start putting metadata on nodes (e.g. associated with linear logic types where we have some idea that subgraphs can be computed in parallel) and use this metadata at runtime to allocate threads more efficiently, possibly prove non-existence of certain kinds of contention and avoid concurrency primitives (e.g. locks) etc.

13.5.2 Lazy (Levy-optimal)

Algorithm

1. Lazy: stop when root node is a constructor = root node connected by principal port to operator
2. Each step:
    1. Look for leftmost-outermost redex: starting from root of term, traverse any operator reached by an auxiliary port by existing from its principal port until we reach an operator at a principal port
        1. Either the previous operator was the root node, in which case halt
        2. Or we have found a redex (principal pair) - fire the redex and start again from the root
    2. To avoid re-scanning the term from the root, push operators onto a stack while traversing
        1. After firing a redex, pop the last operator and restart search

Properties

1. Minimises number of beta-reductions
2. No parallelism

13.5.3 Parallel

Algorithm

1. Evaluate all primary pairs each step
2. Initially scan for primary pairs, then only examine changed nodes each step

Properties

1. Maximally parallel
2. Scanning may be expensive
3. May reduce unnecessary redexes (compared to lazy algorithm)
4. Unlikely to make sense if possible parallelism exceeds number of available cores

13.5.4 Hybridised

Algorithm

1. As lazy, but when parallel annotations are hit, fork process (or assign thread up to number of processors)

Properties

1. Requires parallel annotations placed by the compiler
    1. Can definitely be placed when ∀ types are used
    2. Use proofs about terms being used to annotate graph?

13.5.5 Implementation concerns

Thread affinity & cache locality

1. Tie a thread to a spacial area of the interaction net.

Safety under concurrency

1. How to avoid locks in rewriting?

14 Cost accounting

14.1 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.

14.2 Accounting approaches

14.2.1 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)

14.2.2 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)

14.2.3 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.

14.3 Monadic cost metering

I think we can utilise the basic monadic system in Danielsson's paper, adding in costs to particular rewrite rules in the evaluation semantics and ensuring that the reduction of an encapsulated term (the cost monad being erased) is bounded by the associated cost.

Some outstanding questions:

- Base primitives & functions will need to have non-unit costs
- How does optimal sharing change the necessary accounting (maybe not much, it already assumes call-by-need, but we need to think about what happens when thunks are copied)
- Can we add in discounts for parallelism (this would need to be understood by the compiler)?
- Can we automate or infer costs? The programmer overhead seems high.
- This doesn't really account for memory usage, which we may need to consider as well.

Additional thoughts after reading McCarthy 2016:

1. Propositional approach is neater, though we could also return 0-usage naturals in QTT.
2. It would be preferable to tie the monadic cost calculations directly to the cost semantics of the evaluator - in particular, we need to capture memory allocations, possibly bound the total graph size, and differentiate between sequential & parallel rewrite steps in some suitable fashion.
3. Allocations may not be unit cost
    1. For primitive data types, cost will depend on the size of the data type
4. Rewrite steps will not be unit cost
    1. Some may have primitive costs (built-in functions), bespoke-encoded rule costs must be calculated according to the term encoded into the bespoke rule
5. Costs assigned to computation & memory allocation must be sufficiently accurate, as a lower bound, to serve as a cost model for a contract on a public blockchain.

6. Cost inference would be nice in the future (the Coq library could perform some transforms, but I think it would be preferable to have explicit costs - just infer them where possible)
7. How best to deal with runtime fusion (e.g. $not.not.not.not$)? Not sure yet.

Example of the runtime fusion case:

```
true t f = t

false t f = f

not b t f = b f t
```

not . not fuses once.

not . not . not . not fuses like this twice in parallel, then once.

$log\ n$ in the general case.

whereas not itself, naively applied n times, would be linear complexity.

Another example: fast exponentiation by squaring.

Conclusions (tentative):

- Elaborator-based approach a la ZF*, with eventual editor support or the like for ergonomics.
- Parameterised costs for flexible machine targeting and easier upgrades.
- Think more about how to deal with runtime fusion, as it results from semantics, not syntax.
- Think more about explicit parallelism and how that will interact with the cost model.
- Think more about how QTT will interact with the cost model or could render inference easier.

14.4 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)
- Must be tied into cost accounting

15 Backends

15.1 Backend requirements

Backend defines:

- Primitive types
- Primitive functions & costs
- Primitive typing relation
- Bespoke encoding path
- (optionally) Interaction net interpreter
- (optionally) Optimisation passes

15.2 Backend fusion

- Higher-level representation of the backend as a value/type in the language?
- It will be possible to typecheck as one project (set of source files which can co-reference each other) a multi-chain / multi-backend application.
- Architecturally, we would do this by having individual parameterisations of core per backend, one ‘meta-parameterisation’ combining them all, and a downcast function, run at compile time, that evaluates the term, walks all primitives, and either succeeds in translating them to a single backend or fails if terms from other backends are still present.

15.3 Currently supported backends

15.3.1 LLVM

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

Experiment with "unrolling" the basic interaction net evaluator case switch by some degree k, such that intermediate allocations can be optimised away at compile-time - code size will be c^k with c the number of branches, but that's still fine for a few levels of unrolling.

This transformation is trivially semantics-preserving in the single-thread case and no additional bookkeeping is necessary. In the multi-thread case, if two threads start reduction less than k hops from each other (hop = wire between nodes), they may reduce overlapping primary pairs, which could be problematic. We could either attempt to eliminate this possibility at compile time, which will require a fair bit of knowledge about where parallel reduction will occur (but perhaps possible, especially with explicit annotations), or reduce optimistically in parallel, insert synchronisation points where necessary, and revert conflicting changes in a semantics-preserving way - an approach which is more complicated to reason about, but doesn't require compile-time knowledge of "distance" between threads rewriting the graph in parallel.

API: - createNet : [Node] -> IO () - readNet : IO [Node] - saveState : IO StateHandle - loadState : StateHandle -> IO () - appendToNet : [Node] -> IO () - reduceUntilComplete : IO ReductionStats

For now, net construction & read-back can be performed in Haskell. saveState and loadState can be used to copy a in-memory net snapshot - this will allow e.g. first calling createNet with a function, saving the state, appending arguments to the net, reducing, reading the net, and restoring the state prior to calling the function again. That way we should be able to call the same function multiple times reasonably efficiently. Probably a more complex API will be required in the future.

Resources for optimisation (outside of the LLVM library itself):

- Grin has a number of simple optimisation passes. Some may be suitable specifically for a model of a lazy functional graph evaluator but I expect many are helpful here.
- MLton is a whole-program-optimising ML compiler.
- Slide deck on functional language compilation from INRIA.
- Intel Haskell research compiler, which has some optimisation passes.

15.3.2 Michelson

Bespoke compilation path only, no interaction nets.

- Stack tracking
- Calling conventions
- Clear stack when variable won't be used again
- Encode all natives as built-in functions, dependently-typed where applicable

Closures: - Scan the body of the function for referenced variables - Compile the function as a lambda on all variables and a tuple of the referenced variables followed by `APPLY` for uniform typing - Compile applications as an unpacking of the referenced variables and then `EXEC`

Outstanding questions: - How to pass higher-level datatypes through core for natural representation? Add them to primitive values & require higher-level language to create them? Do we need special treatment (= overrides for standard library) or can we do this structurally (a la GHC.Generics)? - Same question for case statements.

15.3.3 WASM

WASM is targeted through LLVM.

15.4 Planned backends

15.4.1 EVM

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

15.4.2 Berkeley Packet Filter

Target BPF possibly, Solana uses this.

15.4.3 GPUs (CUDA)

Excellent way to demonstrate parallelism.

Full backend.

Consider fusing with LLVM.

15.4.4 FPGA

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

Full backend.

15.4.5 Rank-1 constraint systems (SNARKs)

Open questions - How to structure the core dialect? Will need to eventually turn into an arithmetic circuit with addition gates, multiplication gates, and an element type (usually a field element) which is converted into a rank-1 constraint system. The circuit encodes a relation between the input & output, i.e. `circuit :: Input -> Output -> Bool` - Input & output types must desugar to vectors of the element type with fixed lengths. - Higher-order functions must be reducible at compile time to relations. - Probably will eventually need Rust bindings to the prover/verifier code (ref, ref). This doesn't matter so much if our primary goal is to output a circuit description in some standard format and 'glue code' for integration into smart contracts, though.

Inspirations: snarky, genSTARK, ZoKrates.

We should only do this if it will actually be useful (compared to existing DSLs). I think it would be useful if and only if: - We can implement a higher level of abstraction without sacrificing performance thanks to the powerful typesystem & usage information, and possible optimisation layers - We can nicely integrate code running in zero-knowledge with regular smart contract code in a way that is very convenient for contract authors, automates all the data conversion tedium, and allows for proofs across multi-backend code (ref https://github.com/cryptiumlabs/juvix/issues/157#issuecomment-552046922) - We can output performant code (circuit descriptions to be executed on dedicated provers/verifiers and smart contracts as appropriate) that can be used in production.

Higher-order functions should be possible if we only operate on closed terms & fully evaluate under lambdas - in particular consider the following case (of a case statement):

```
relation :: Input -> Output -> Bool
relation input output =
  (case input of
     1 -> (==) 2)
     2 -> (==) 3) output
```

If we evaluate under the case (which will desugar to a lambda), we should end up with:

```
relation :: Input -> Output -> Bool
relation input output =
  (input == 1 && output == 2) ||
  (input == 2 && output == 3)
```

which is just fine.

Note that to do this we need some prior knowledge of the type - namely a relation - so that we know that the case statement should turn into an `OR` on the relations of the scrutinee, evaluated clauses & bodies.

Also note that it would be nice to avoid evaluating the scrutinee multiple times for multiple case branches, this should be achievable as an intermediary circuit node.

(to encode the OR in the circuit - `or x y -> x + y - (x * y) = 1`)

Maybe we could also try an evaluation monad which keeps intermediary results (basically 'allocating memory' as necessary) and can output a circuit representation. Might be convenient for avoiding duplication of intermediary evaluations. Advantages: structurally simpler and captures the idiomatic flow better.

15.4.6 Algebraic intermediate representation (STARKs)

Ideally a developer-facing language would operate at a higher level, automatically determining the necessary registers, length of execution trace, transition function & transition constraints. I think this should be possible to do automatically pretty efficiently - possibly in some cases custom constraints could be defined & proved to be equivalent to correct execution of the transition function (e.g., for a layer l_n in a trace, l_n = f(l_n-1) <=> c l_n l_n-1 where c is the constraint relation).

I am not sure if an interaction net transition function can be efficiently encoded in a STARK. I think the transition function itself is less likely to be a problem than the memory accesses (which will require a Merkle tree & proofs for all reads, probably). The concrete numbers here (especially prover time & proof size) may be prohibitively high. Apparently 1-layer recursion in STARKs is possible; I do not know the details. DEEP-FRI should also help.

At minimum, even with "function-to-constraints" compilation, a custom library will be necessary to import STARK-friendly hash functions, signature schemes, etc.

- Construct registers to hold all values
- Use public / private type distinctions for register separation
- ADTs must be completely erased by runtime
- Require primitive type (or tuple, or encodable/decodable ADT) input, output
- Specific primitive types supported by STARKs (field-dependent)
- Unroll loops completely, must have finite length

Questions: - Can QTT provide sufficient precision to allow all register allocation to happen statically? - If we inline everything, can higher-order functions be erased (want to avoid interaction net evaluation, since that requires dynamic memory allocation)? - Transforms to reuse registers over time ~ may be prior literature or forms, e.g. SSA - Figure out rough numbers for size, proof time scaling vs. register count

16 Distributed ledger integration

16.1 Incremental compilation

It might be prudent to consider an "incremental compilation" model where new code is compiled into an existing runtime (e.g. a set of smart contracts, which could be encoded as functions being accepted as input provided by a user at runtime). Of course this is possible with a model like Ethereum's, where code is stored per-contract and loaded only when the contract is run, but that renders cross-contract optimisation quite difficult (I don't think any Ethereum clients do it) and would potentially require expensive data format conversions for Juvix terms.

In principle it should be possible to alter an existing interaction net representation (e.g. in memory) by simply compiling the new term to an interaction net of its own, which then is connected to the existing term at the appropriate location. This will

require, however, a fixed set of choices for ADT encoding, certain optimisation passes, etc. unless we track metadata and can elect to rewrite other parts of the net if a different choice makes sense.

16.2 Compiler & typechecker metering

For future integration with a distributed ledger, it will be helpful to have the ability to run the typechecker on arbitrary user-input terms, which we must meter in order to do safely. Thus it will be required to compile the typechecker (or all of the Juvix library, ideally) to a target VM which can be metered at runtime (so gas can be charged for typechecking terms). This doesn't have to be particularly efficient since terms only need to be typechecked once.

I think the ideal target is WASM, which can be metered in a VM (e.g. Life) and as a bonus runs in the browser. Asterius is a Haskell-to-WASM compiler which appears to be relatively functional & actively developed. We should investigate whether that will work for the Juvix typechecker (both the core typechecker & the EAL typechecker) - or ideally even for the whole compiler (including EAL inference, interaction net interpreter, etc.) - though the latter isn't as critical.

I am not sure whether it is necessary to also meter other semantics-preserving parts of the compiler (e.g. optimisation passes) - it might be possible for nodes to run these individually (and they might be on different platforms), but that could be problematic if particular passes could be DoS attack vectors or if cost semantics (of terms - #45) are dependent on optimisation passes (in which case they would need to be specified & checked in consensus).

16.3 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.

16.4 Architectural optimisations

16.4.1 Fusing across sequentially applied transactions

Consider a sequence of transactions applied in order which read & write various state values. The transactions reads & writes can be fused prior to actually touching state, e.g. if two transactions increment the balance of a destination account the increment amounts can be added prior to applying a single addition operation to state. This should work even if there are dependencies on state reads - it may result in unnecessary evaluation, but the cost of reading & writing Merkleized state is likely to be relatively quite expensive.

This fusion can be performed over any logical block of ordered, subsequent transactions for which computation of the exact following state (e.g. resulting in a Merkle tree root) can be delayed - at minimum, this will be a block, and could be several if finality is delayed or pipelined anyways.

17 Future directions

17.1 Zero-knowledge typing

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

17.2 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.

17.3 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

17.4 Visual spatiotemporal dataflow representation

1. Some (closest?) inspiration: Luna [10]
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.

17.5 Future optimisation strategies

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

17.5.1 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 [14].
- Will need a lot of clever tricks to avoid getting stuck in local minima (that paper details several).
- See also STOKE [15].

17.5.2 "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 (already a concern with optimal reduction), likely low-payoff.

18 Examples

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

Possible ideas (all with proofs of correctness & bounded resource consumption):

18.1 Fungible token contract

Fungible token contract with proofs about constant supply, ownership, fungibility

18.2 Non-fungible token contract

Non-fungible token contract with proofs about constant supply, ownership, non-fungibility

18.3 Multi-signature contract with validity predicate

Multi-signature contract with functional restrictions

18.4 Pooled liquidity exchange

Uniswap exchange contract with proofs of price slippage, fairness

18.5 Pluggable liberal radicalism

Liberalism radicalism implementation which plugs into a funder and stakeholder set

18.6 Shielded circuit connector

Simple shielded circuit with zero-knowledge types

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

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

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

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

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

[12] 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.

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

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

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

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

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

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

[19] 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/.