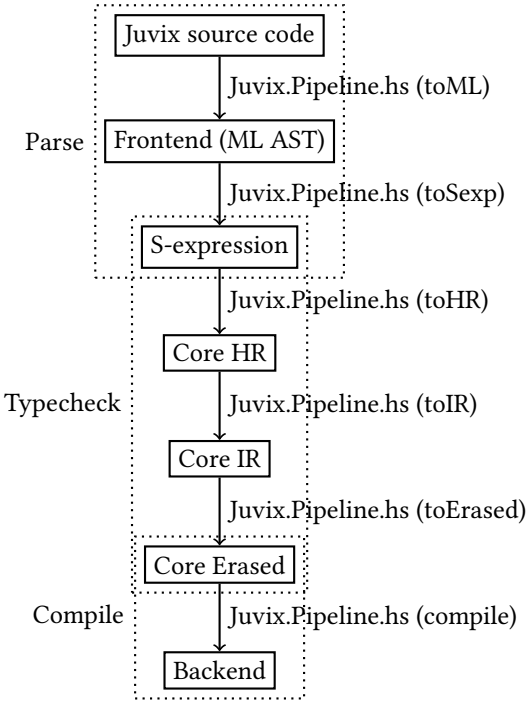
Core Documentation

JUVIX TEAM

Contents

Author's address: Juvix Team.

1 PIPELINE

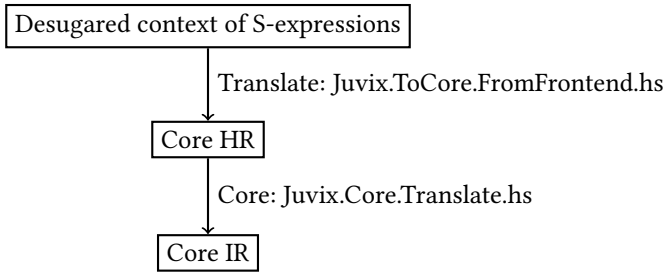

Juvix source code
Juvix.Pipeline.hs (toML)
Parse
Frontend (ML AST)
Juvix.Pipeline.hs (toSexp)
S-expression
Juvix.Pipeline.hs (toHR)
Core HR
Typecheck
Juvix.Pipeline.hs (toIR)
Core IR
Juvix.Pipeline.hs (toErased)
Core Erased
Compile
Juvix.Pipeline.hs (compile)
Backend

2 S-EXPRESSIONS

TODO

3 CORE

3.1 HR



HR stands for Human Readable.

This is a small module. It basically extends the Intermediate Representation (IR) base types to include names rather than De Bruijn indices.

As with any module in Juvix Core, an understanding of the extensible-data library is required.

As mentioned, the goal is to use names instead of De Bruijn indices. Even though HR is a prior step in the pipeline than IR, the code is structured in a way that the *base* types exist in IR and other modules *extend* from it. These base types live in Juvix.Core.IR.Types.Base.

Do all types get affected by this distinction between HR and IR? No. Only *binders* do. A binding is represented by an identifier. In the context of HR, this identifier is a name and this name is bound to a variable. Names must be unique. Binders in Juvix are:

(1) **Pi**

A function whose type of return value varies with its argument is a dependent function and the type of this function is called *pi-type*.

Function types $(x \overset{\pi}{:} S) \to T$ record how the function will use its argument via the π annotation.

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

More details in Syntax and Semantics of Quantitative Type Theory.

We've encoded it in Juvix.Core.IR.Types.Base as follows:

```
data Term primTy primVal =
        ...
        | Pi Usage (Term primTy primVal) (Term primTy primVal)
        | ...
```

Here's an example of equivalent IR and HR Pi types:

```
HR.Pi Usage.Omega "a" (HR.PrimTy ()) (HR.PrimTy ())

                      ⇔

IR.Pi Usage.Omega (IR.PrimTy ()) (IR.PrimTy ())
```

(2) **Lam**

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

Forgetting the resource annotations, this is the standard introduction rule for dependent function types. We require that the abstracted variable x has usage $\sigma\pi$ (multiplication by σ is used to enforce the zero-needs-nothing).

```
data Term primTy primVal =
        ...
        | Lam (Term primTy primVal)
        | ...
```

Here's an example of equivalent IR and HR lambda types:

```
HR.Lam "x" (HR.Lam "y" (HR.Elim (HR.Var "x")))

⇔

IR.Lam (IR.Lam (IR.Elim (IR.Bound 1)))
```

(3) **Sig**

Sig stands for *Sigma* type (not signature!), understood as existential quantifier. It is represented as $(x \overset{\pi}{:} S) \otimes T$. It captures the idea of an ordered pair where the type of the second term is dependent on the value of the first. For example, $\Sigma n : \mathbb{N}$.Vec A n is actually equivalent to *ListA* because we can represent sequences of arbitrary length.

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

In a dependent pair (Σ) type, each half has its own usage. The *usage* (π) of *Sig* in the definition and in the following code refers to how many times the first argument may be used:

```
data Term ty val =
        ...
        | Sig Usage (Term ty val) (Term ty val)
        | ...
```

(4) **Let**

Some constructions allow the binding of a variable to a value. This is called a "let-binder". In a "let-binder", only once variable can be introduced at the same time.

```
data Term ty val =
        ...
        | Let Usage (Elim ty val) (Term ty val)
        | ...
```

The code representing the binders above are written in IR form. The following code extends binder terms, using the extensible-data library. The syntax of the type theory is defined by mutual induction between terms, with types specified in advance, and eliminations with types synthesized. This is why we need to extend *Elim* as well.

```
extTerm :: p1 -> p2 -> IR.ExtTerm
extTerm =
  \_primTy _primVal ->
    IR.defaultExtTerm
      { IR.nameLam = "Lam0",
        IR.typeLam = Just [[t|NameSymbol.T|]],
        IR.namePi = "Pi0",
        IR.typePi = Just [[t|NameSymbol.T|]],
        IR.nameSig = "Sig0",
        IR.typeSig = Just [[t|NameSymbol.T|]],
        IR.nameLet = "Let0",
        IR.typeLet = Just [[t|NameSymbol.T|]]
      }
```

In the snippet above we are extending and renaming *Lam*, *Pi*, *Sig* and *Let* with an additional name represented as *NameSymbol*, which is just a type alias of *NonEmpty Symbol* that encodes a qualified name.

We call this function in *Juvix.Core.HR.Types* using the *extendTerm* function generated by *extensible* (see extensible-data)

```
extendTerm "Term" [] [t|T|] extTerm
```

We rename *Lam* to *Lam*0, *Pi* to *Pi*0, etc. for convenience. This way we can reorder the type parameters using pattern synonyms and only export these patterns from the library

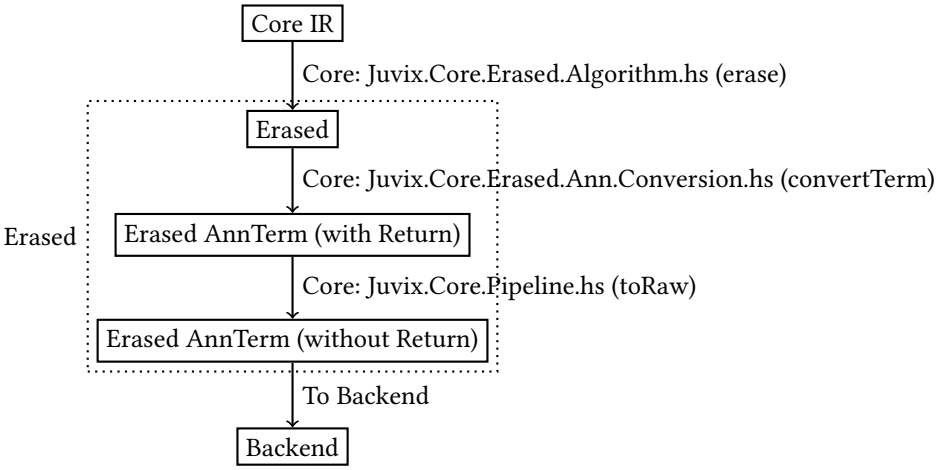
```
pattern Pi pi x s t = Pi0 pi s t x
```

The same procedure follows for *Elim*, although there are a few differences. The data constructors *Bound* and *Free* don't make sense in the context of HR. We want to introduce a new data constructor, *Var*, that holds the qualified name (*NameSymbol*) set by some binder.

```
extElim :: p1 -> p2 -> IR.ExtElim
extElim =
  \_primTy _primVal ->
    IR.defaultExtElim
      { IR.typeBound = Nothing,
        IR.typeFree = Nothing,
        -- | Extend with extra constructor Var
        -- that was not existing before
        IR.typeElimX = [("Var", [[t|NameSymbol.T|]])]
      }

extPattern :: p1 -> p2 -> IR.ExtPattern
extPattern =
  \_primTy _primVal ->
    IR.defaultExtPattern
      { IR.typePVar = Nothing,
        IR.typePatternX = [("PVar", [[t|NameSymbol.T|]])]
      }
```

3.2 Erased



Erased doesn't extend from IR types any more, but from its own types file. In contrast to previous steps in the pipeline, it distinguishes between terms and types.

At this point, terms have already been typechecked.

3.2.1 Ann. We retrieve the usage of a term and annotate the term with it. Certain backends can use the knowledge of a usage's term to optimise compilation.

4 BACKENDS

4.1 LLVM

4.2 Michelson

4.3 Plonk