An Overview of the Husky Programming Language

Xiyu Zhai

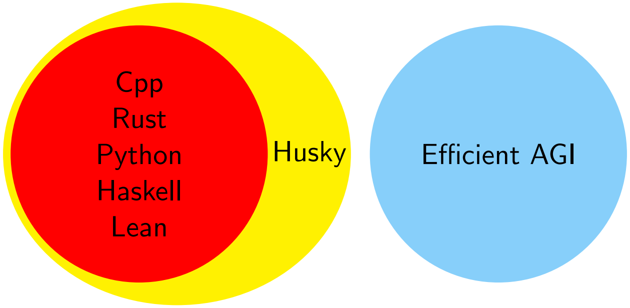
Introduction

A New Programming Language, Husky

Disclaimer: I'm working on Machine Learning Theory and Programming Language for AI, but this talk will focus on Programming Language Design, and won't go into details about Machine Learning and AI.

Why a New Programming Language?

- I was working on better AI algorithms beyond deep learning
- these ideas are impossible to implement in existing languages
- Husky is invented to implementing them with ease
- It pushes the boundary of programming towards efficient AGI



Current-Generation AI

- I'm not against deep learning AIs are AGIs, but I believe they are not so efficient
- Two types of efficiency:
  - Computational. Training/inference time computation cost and requirement, memory, latency
  - Statistical. Number of samples needed.
- Although deep learning can be as accurate as humans in many cases, its efficiency often can't be as good as humans.
- Artificial intelligence prioritizes accuracy over efficiency, whereas natural intelligence evolves under rigorous constraint on efficiency.

Theory for the Gap Between Efficiency

- The inference process of many ML problems are actually NP-problem in disguise:
    - In computer vision, template matching is about finding the configuration with minimal loss
    - In natural language processing, one needs to disambiguate meaning so that the whole text makes sense
- It takes relatively little data to learn the verifier, but maybe much more to learn the solver, and much much more to learn a near-optimal solver
- Human learns how to verify, but current AI not so much
- RLHF can be thought of as a crude verifier

What are Next-Generation AI ideas like?

I have been researching on these ideas for 6 years, complicated enough to give several lectures upon, and not yet reaching a prototype, so I only give impressions rather than details,

- learns a verifier
- modular and domain specific
- models contains operations that are not matrix operations
- more efficient than deep learning only, computationally (both training and inference) and statistically
- merges symbolic AI, neural AI, programming language, formal verification, database, etc

What is Husky

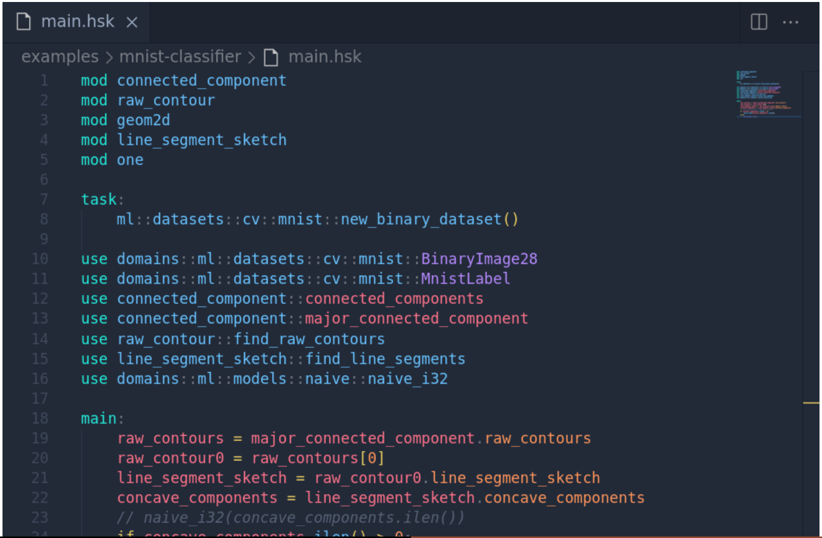
A new language created for next-generation AI/software. It features

- it merges features from modern programming languages.
- it has a novel AI programming paradigm called ascension
- it has a powerful debugging system
- it's highly optimized for incremental computation.

History

- For the 4 years, it went though several stages,
    - Being a Python library. Debugging is like hell.
    - Being a C++ macro library. Debugging is possible by playing with templates. But compilation and error messages are like hell.
    - Becomes a multi-paradigm programming language written in C++ (30k lines of code), with good features from C++ and Rust.
    - Rewritten in Rust, with good features from C++ and Rust and Lean and a new paradigm called Ascension.
- The source code of the last working prototype is 40k lines of Rust code, with limited functionality, tons of bugs, and not so clean code

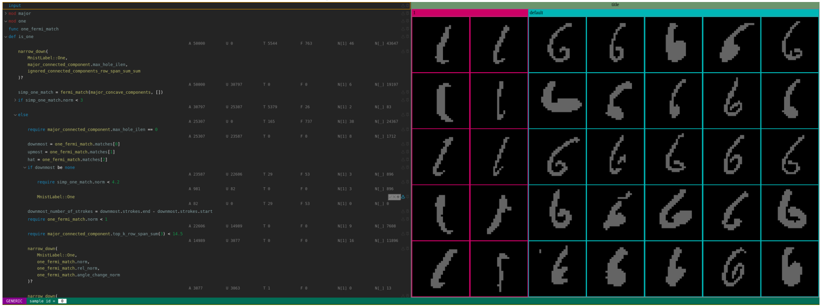
Old Syntax



```
mod connected_component
mod raw_contour
mod geom2d
mod line_segment_sketch
mod one

task:
    ml::datasets::cv::mnist::new_binary_dataset()

use domains::ml::datasets::cv::mnist::BinaryImage28
use domains::ml::datasets::cv::mnist::MnistLabel
use connected_component::connected_components
use connected_component::major_connected_component
use raw_contour::find_raw_contours
use line_segment_sketch::find_line_segments
use domains::ml::models::naive::naive_i32

main:
    raw_contours = major_connected_component.raw_contours
    raw_contour0 = raw_contours[0]
    line_segment_sketch = raw_contour0.line_segment_sketch
    concave_components = line_segment_sketch.concave_components
    // naive_i32(concave_components.ilen())
```

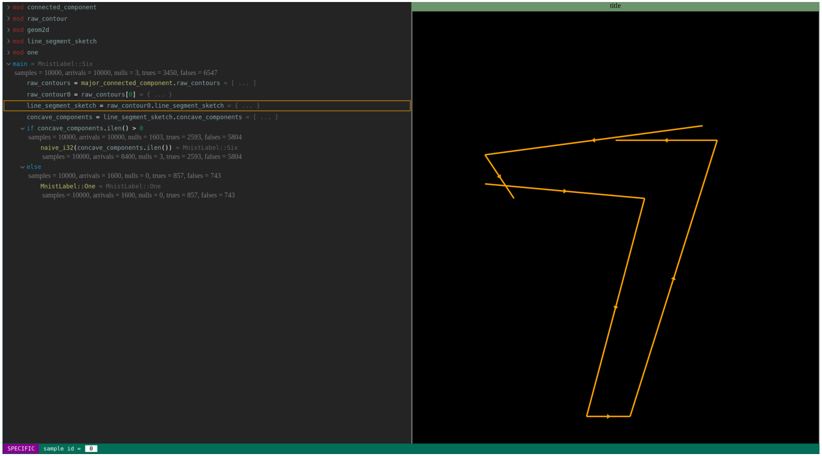
"Achievement"

Hand written code (without using any form of machine learning) for classifying hand-written digits (MNIST), 80% accuracy, not much effort because of bugs



```
input
> mod major
v mod one
  func one_fermi_match
v def is_one
                                                                    A 50000     U 0        T 5544    F 763     N[1] 46    N[_] 43647

    narrow_down(
        MnistLabel::One,
        major_connected_component.max_hole_ilen,
        ignored_connected_components_row_span_sum_sum
    )?
                                                                    A 50000     U 30797    T 0       F 0       N[1] 6     N[_] 19197

    simp_one_match = fermi_match(major_concave_components, [])
  > if simp_one_match.norm < 3
                                                                    A 30797     U 25307    T 5379    F 26      N[1] 2     N[_] 83

  v else
                                                                    A 25307     U 0        T 165     F 737     N[1] 38    N[_] 24367

        require major_connected_component.max_hole_ilen == 0
                                                                    A 25307     U 23587    T 0       F 0       N[1] 8     N[_] 1712

        downmost = one_fermi_match.matches[0]
        upmost = one_fermi_match.matches[1]
        hat = one_fermi_match.matches[2]
      v if downmost be none
                                                                    A 23587     U 22606    T 29      F 53      N[1] 3     N[_] 896

            require simp_one_match.norm < 4.2
                                                                    A 981       U 82       T 0       F 0       N[1] 3     N[_] 896

            MnistLabel::One
                                                                    A 82        U 0        T 29      F 53      N[1] 0     N[_] 0
        downmost_number_of_strokes = downmost.strokes.end - downmost.strokes.start
        require one_fermi_match.norm < 1
                                                                    A 22606     U 14989    T 0       F 0       N[1] 9     N[_] 7608

        require major_connected_component.top_k_row_span_sum(3) < 14.5
                                                                    A 14989     U 3077     T 0       F 0       N[1] 16    N[_] 11896

        narrow_down(
            MnistLabel::One,
            one_fermi_match.norm,
            one_fermi_match.rel_norm,
            one_fermi_match.angle_change_norm
        )?
                                                                    A 3077      U 3063     T 1       F 0       N[1] 0     N[_] 13

        narrow_down(
GENERIC  sample id = 0
```

"Achievement"



```
> mod connected_component
> mod raw_contour
> mod geom2d
> mod line_segment_sketch
> mod one
v main = MnistLabel::Six
  samples = 10000, arrivals = 10000, nulls = 3, trues = 3450, falses = 6547
     raw_contours = major_connected_component.raw_contours = [ ... ]
     raw_contour0 = raw_contours[0] = { ... }
     line_segment_sketch = raw_contour0.line_segment_sketch = { ... }
     concave_components = line_segment_sketch.concave_components = [ ... ]
   v if concave_components.ilen() > 0
     samples = 10000, arrivals = 10000, nulls = 1603, trues = 2593, falses = 5804
        naive_i32(concave_components.ilen()) = MnistLabel::Six
         samples = 10000, arrivals = 8400, nulls = 3, trues = 2593, falses = 5804
   v else
     samples = 10000, arrivals = 1600, nulls = 0, trues = 857, falses = 743
        MnistLabel::One = MnistLabel::One
         samples = 10000, arrivals = 1600, nulls = 0, trues = 857, falses = 743
SPECIFIC  sample id = 0
```

Status Quo

- In the progress towards a new powerful prototype, 116k Lines of well-structured code, 4k todos, 2k warnings.
- This version is very close to an industrial language, with
  - package and module system.
  - powerful type system (variance, generics, dependent type, traits).
  - convenient symbol import (like Rust, better than Haskell and Lean).
  - flexible type inference (like Rust).
  - everything is optimized for incremental computation, refined caching based on region paths and type terms.
- Syntax and semantics suffices for now, I'm working on debugging system, compilation etc.

Near Future: Publish or Perish

- I'm rushing towards a new prototype
- In the following one or two years, I will use it for
  - new computer vision models that run faster on phones or IoT devices
  - machine learning theories for image classification
- it's about time to write papers

Far Future: Keep Agile

- What I don't like is, run into stable version and then discover the features don't satisfy the needs.
- Agility: keep language unstable, test it for various tasks, and gradually adapt.
- 5-10 years from version 0.1, 15-20 years from version 1.0.
- Good enough to apply it for AI research within months.
- For production, transpile into C or Zig or even high level C++ or Rust.

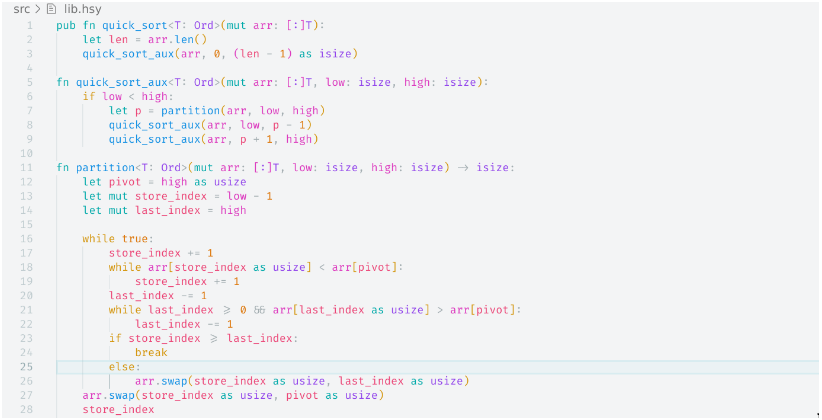
Basic Designs

Husky Merges Features of Modern Languages

- its name ‘Husky’ comes from ‘Haskell’ and ‘Rust’ and ‘python’
- its file extension ‘hsy’ comes from ‘hs’ (Haskell) ‘rs’ (Rust) and ‘py’ (python)
- its package manager called ‘corgi’ is just like ‘cargo’
- its syntax is best described by ‘pythonic Rust’ for runtime and Lean-like for comptime
- type system is influenced by Rust + Lean + ATS, with monad replaced by trait Unveil and effects, and new memory types are introduced
- it has a global computation graph like Haskell, or deep learning libraries

Syntax

Overall it's about system level and functional. No OOP but has methods and traits.
Looks like mojo because the design goals are similar.

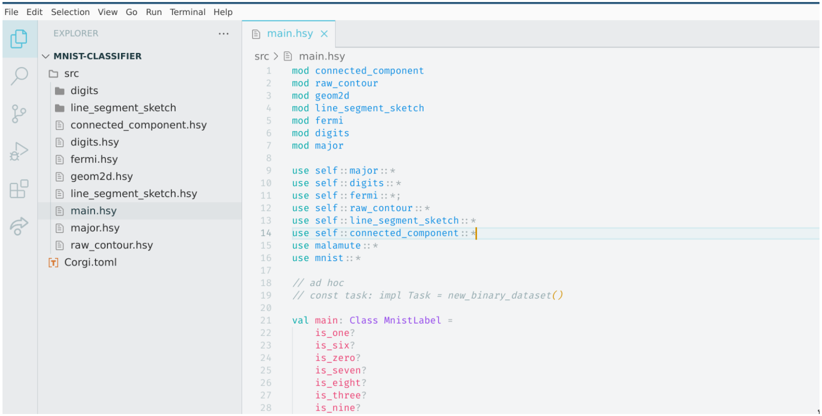


src > lib.hsy

```
pub fn quick_sort<T: Ord>(mut arr: [:]T):
    let len = arr.len()
    quick_sort_aux(arr, 0, (len - 1) as isize)

fn quick_sort_aux<T: Ord>(mut arr: [:]T, low: isize, high: isize):
    if low < high:
        let p = partition(arr, low, high)
        quick_sort_aux(arr, low, p - 1)
        quick_sort_aux(arr, p + 1, high)

fn partition<T: Ord>(mut arr: [:]T, low: isize, high: isize) -> isize:
    let pivot = high as usize
    let mut store_index = low - 1
    let mut last_index = high

    while true:
        store_index += 1
        while arr[store_index as usize] < arr[pivot]:
            store_index += 1
        last_index -= 1
        while last_index >= 0 && arr[last_index as usize] > arr[pivot]:
            last_index -= 1
        if store_index >= last_index:
            break
        else:
            arr.swap(store_index as usize, last_index as usize)
    arr.swap(store_index as usize, pivot as usize)
    store_index
```

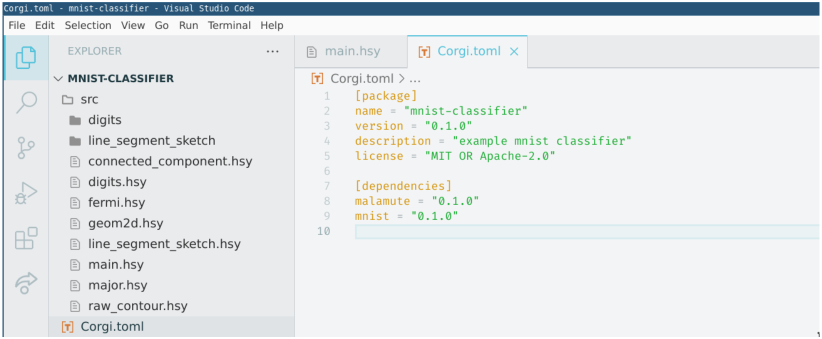
Syntax

Symbol are introduced through ‘use’, basically the same as Rust.
Package, crate, file layout are basically the same as Rust.



```
mod connected_component
mod raw_contour
mod geom2d
mod line_segment_sketch
mod fermi
mod digits
mod major

use self::major::*
use self::digits::*
use self::fermi::*;
use self::raw_contour::*
use self::line_segment_sketch::*
use self::connected_component::*
use malamute::*
use mnist::*

// ad hoc
// const task: impl Task = new_binary_dataset()

val main: Class MnistLabel =
    is_one?
    is_six?
    is_zero?
    is_seven?
    is_eight?
    is_three?
    is_nine?
```

Cargo is called Corgi



```toml
[package]
name = "mnist-classifier"
version = "0.1.0"
description = "example mnist classifier"
license = "MIT OR Apache-2.0"

[dependencies]
malamute = "0.1.0"
mnist = "0.1.0"
```

In Rust, cargo configuration errors will cause compiler to panick. But in Husky, corgi configuration errors are seen as syntax errors and there are customized toml parsers to make things as ergonomic as possible.

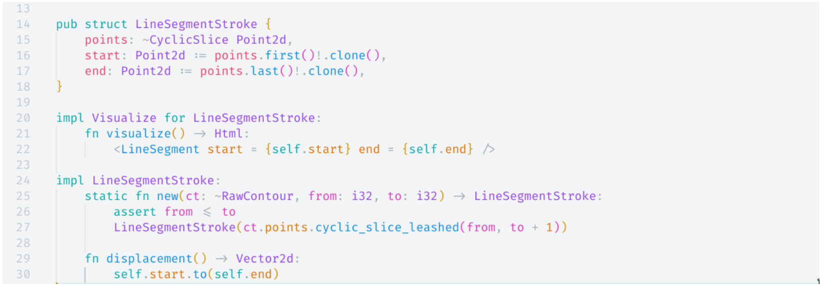
Type Definition

One can define

- regular struct/structure
- tuple struct/structure
- unit struct/structure
- enum/inductive
- extern type

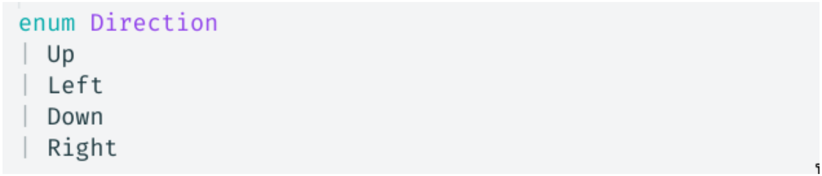
Here struct enum are for runtime values, and structure or inductive are for compile time (term level) values. The details are yet to be determined.

Type Definition Examples



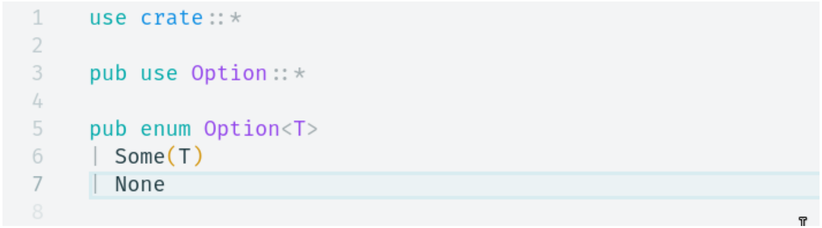
```

pub struct LineSegmentStroke {
    points: ~CyclicSlice Point2d,
    start: Point2d := points.first()!.clone(),
    end: Point2d := points.last()!.clone(),
}

impl Visualize for LineSegmentStroke:
    fn visualize() -> Html:
        <LineSegment start = {self.start} end = {self.end} />

impl LineSegmentStroke:
    static fn new(ct: ~RawContour, from: i32, to: i32) -> LineSegmentStroke:
        assert from <= to
        LineSegmentStroke(ct.points.cyclic_slice_leashed(from, to + 1))

    fn displacement() -> Vector2d:
        self.start.to(self.end)
```

Type Definition Examples

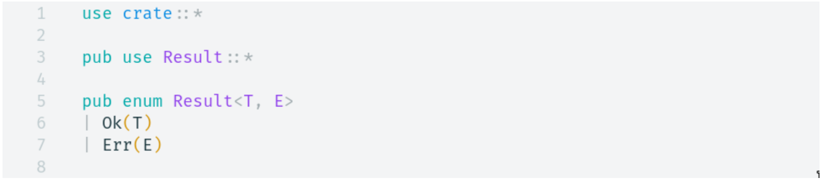


```
enum Direction
| Up
| Left
| Down
| Right
```

Algebraic data type realized just like Rust



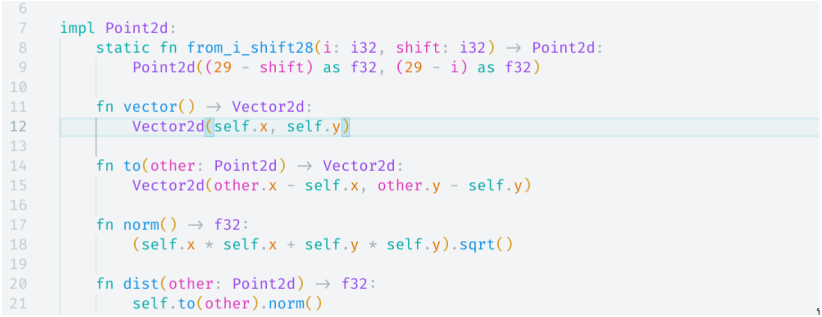
```
use crate::*

pub use Option::*

pub enum Option<T>
| Some(T)
| None
```



```
use crate::*

pub use Result::*

pub enum Result<T, E>
| Ok(T)
| Err(E)
```

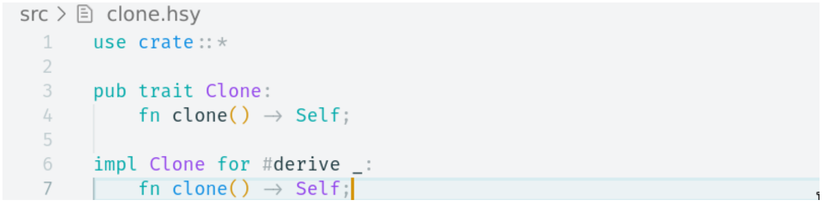
Method

I choose to allow self parameter to be omitted.
Use static for associated functions.



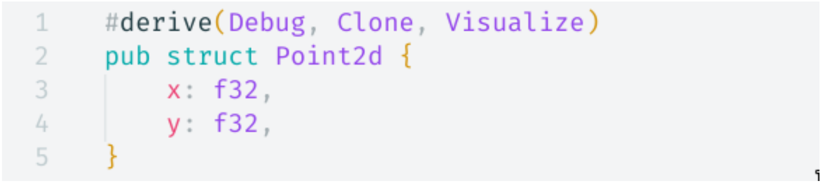
```

impl Point2d:
    static fn from_i_shift28(i: i32, shift: i32) → Point2d:
        Point2d((29 - shift) as f32, (29 - i) as f32)

    fn vector() → Vector2d:
        Vector2d(self.x, self.y)

    fn to(other: Point2d) → Vector2d:
        Vector2d(other.x - self.x, other.y - self.y)

    fn norm() → f32:
        (self.x * self.x + self.y * self.y).sqrt()

    fn dist(other: Point2d) → f32:
        self.to(other).norm()
```

Trait


src > clone.hsy

```
use crate::*

pub trait Clone:
    fn clone() -> Self;

impl Clone for #derive _:
    fn clone() -> Self;
```

Derive Trait



```
#derive(Debug, Clone, Visualize)
pub struct Point2d {
    x: f32,
    y: f32,
}
```

In the above #*derive* is not a macro, but an attribute, serving as a marker. In the original trait definition, there could be a implementation of the trait for all marked types, with a generic representation of a general type, as inpired by Haskell's macro system. Details are to be determined.

Macro

There is no macro in husky.
Macro = undefined behavior of compilers
Husky stays as a research language in foreseeable future, so there is no pressure to be popular.

Type System

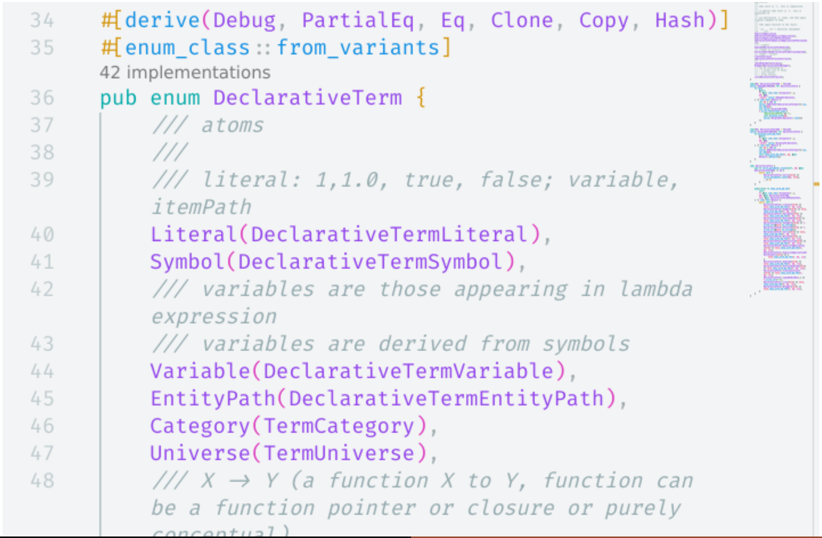
Design goals:

- Easy as python to write
- Safe as ATS, no undefined behaviors,
- Zero-Cost and practical as Rust and C++

Layers of type system (necessarily complicated for efficiency, expressiveness):

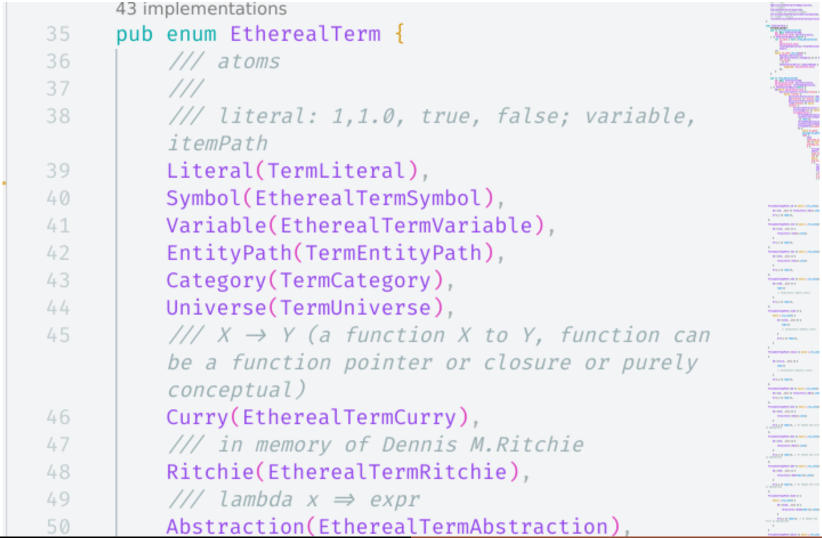
- Declarative Term. Easily read from syntax tree, but can be ambiguous Used in the first step for type signature inference.
- Ethereal Term. Target efficient caching.
- Fluffy Term. Target borrow checking. Not stored globally.
- Hir Type. Close to Rust. Unnecessary terms are thrown away (phantom arguments, univalent values, etc)

Declarative Term for Type Signature Inference



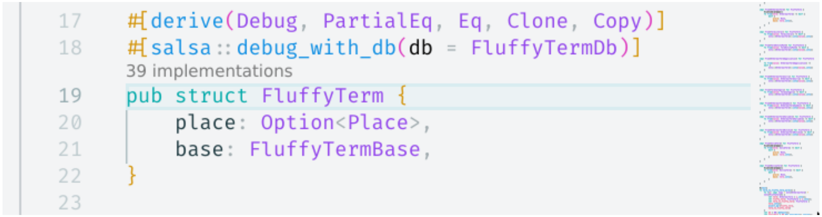
```
#[derive(Debug, PartialEq, Eq, Clone, Copy, Hash)]
#[enum_class::from_variants]
42 implementations
pub enum DeclarativeTerm {
    /// atoms
    ///
    /// literal: 1,1.0, true, false; variable,
    itemPath
    Literal(DeclarativeTermLiteral),
    Symbol(DeclarativeTermSymbol),
    /// variables are those appearing in lambda
    expression
    /// variables are derived from symbols
    Variable(DeclarativeTermVariable),
    EntityPath(DeclarativeTermEntityPath),
    Category(TermCategory),
    Universe(TermUniverse),
    /// X → Y (a function X to Y, function can
    be a function pointer or closure or purely
    conceptual)
```

Ethereal Term for Caching

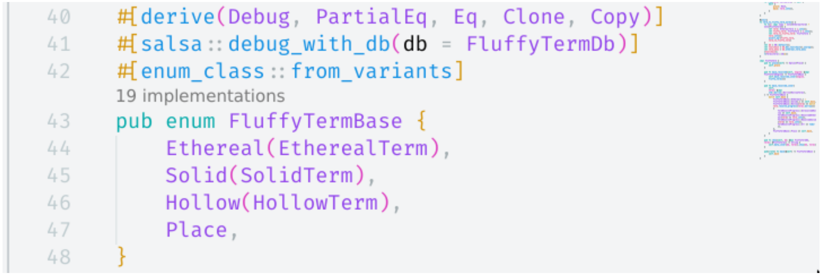

```
43 implementations
pub enum EtherealTerm {
    /// atoms
    ///
    /// literal: 1,1.0, true, false; variable,
    itemPath
    Literal(TermLiteral),
    Symbol(EtherealTermSymbol),
    Variable(EtherealTermVariable),
    EntityPath(TermEntityPath),
    Category(TermCategory),
    Universe(TermUniverse),
    /// X -> Y (a function X to Y, function can
    be a function pointer or closure or purely
    conceptual)
    Curry(EtherealTermCurry),
    /// in memory of Dennis M.Ritchie
    Ritchie(EtherealTermRitchie),
    /// lambda x => expr
    Abstraction(EtherealTermAbstraction),
```

Fluffy Term for Borrow Checking

Fluffy terms are not saved globally because cache hits are hard.

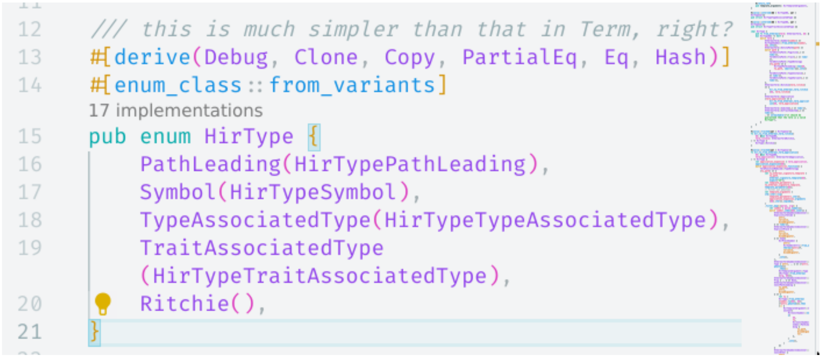


```
#[derive(Debug, PartialEq, Eq, Clone, Copy)]
#[salsa::debug_with_db(db = FluffyTermDb)]
39 implementations
pub struct FluffyTerm {
    place: Option<Place>,
    base: FluffyTermBase,
}

```



```
#[derive(Debug, PartialEq, Eq, Clone, Copy)]
#[salsa::debug_with_db(db = FluffyTermDb)]
#[enum_class::from_variants]
19 implementations
pub enum FluffyTermBase {
    Ethereal(EtherealTerm),
    Solid(SolidTerm),
    Hollow(HollowTerm),
    Place,
}
```

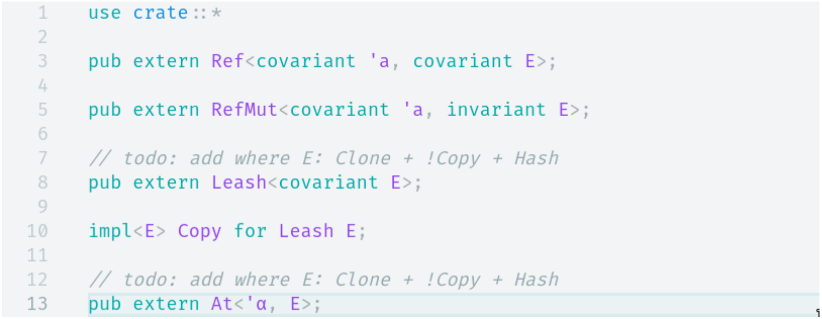
Hir Type for Compilation



```
/// this is much simpler than that in Term, right?
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
#[enum_class::from_variants]
17 implementations
pub enum HirType {
    PathLeading(HirTypePathLeading),
    Symbol(HirTypeSymbol),
    TypeAssociatedType(HirTypeTypeAssociatedType),
    TraitAssociatedType
    (HirTypeTraitAssociatedType),
    Ritchie(),
}
```

Variance

Basically the same as Rust. For extern generic type, variance can be explicitly declared,



```
use crate::*

pub extern Ref<covariant 'a, covariant E>;

pub extern RefMut<covariant 'a, invariant E>;

// todo: add where E: Clone + !Copy + Hash
pub extern Leash<covariant E>;

impl<E> Copy for Leash E;

// todo: add where E: Clone + !Copy + Hash
pub extern At<'α, E>;
```

Of course, having variance makes type inference more complicated.

Memory Types

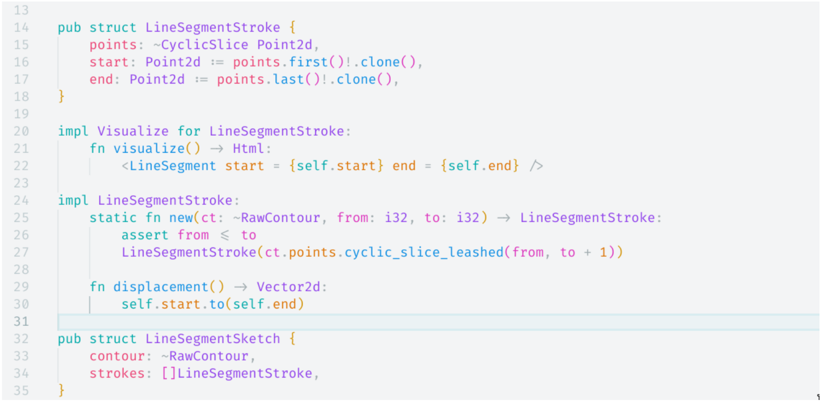
Husky has more memory types than Rust

- Leash type.
- Pure Access.
- At type.

Leash type

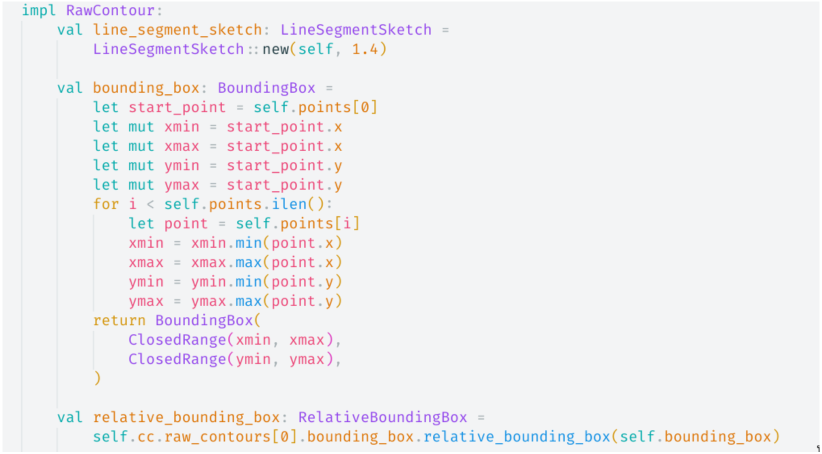
Denoted by $\sim$. This symbol was used for Box in early Rust. Leash type is for access values stored in global database. Similar to salsa. However, Leash type is like a universal interface, one can easily customize its implementation for specific tasks. One can implement it as static reference or garbage collected reference.

Leash In Action



```

pub struct LineSegmentStroke {
    points: ~CyclicSlice Point2d,
    start: Point2d := points.first()!.clone(),
    end: Point2d := points.last()!.clone(),
}

impl Visualize for LineSegmentStroke:
    fn visualize() → Html:
        <LineSegment start = {self.start} end = {self.end} />

impl LineSegmentStroke:
    static fn new(ct: ~RawContour, from: i32, to: i32) → LineSegmentStroke:
        assert from ≤ to
        LineSegmentStroke(ct.points.cyclic_slice_leashed(from, to + 1))

    fn displacement() → Vector2d:
        self.start.to(self.end)

pub struct LineSegmentSketch {
    contour: ~RawContour,
    strokes: []LineSegmentStroke,
}
```

Leash Can Have Memoized Field



```
impl RawContour:
    val line_segment_sketch: LineSegmentSketch =
        LineSegmentSketch::new(self, 1.4)

    val bounding_box: BoundingBox =
        let start_point = self.points[0]
        let mut xmin = start_point.x
        let mut xmax = start_point.x
        let mut ymin = start_point.y
        let mut ymax = start_point.y
        for i < self.points.ilen():
            let point = self.points[i]
            xmin = xmin.min(point.x)
            xmax = xmax.max(point.x)
            ymin = ymin.min(point.y)
            ymax = ymax.max(point.y)
        return BoundingBox(
            ClosedRange(xmin, xmax),
            ClosedRange(ymin, ymax),
        )

    val relative_bounding_box: RelativeBoundingBox =
        self.cc.raw_contours[0].bounding_box.relative_bounding_box(self.bounding_box)
```

Pure Access

It's a type alias or functor, `PureAccess T` for a runtime type `T` is equal to

- itself, if `T` is copyable
- a reference to `T`, otherwise

A function argument type is pure access by default.
In Rust, using get method of HashMap requires a reference to the key, even if the key is copyable. This problem is solved in Husky through pure access.

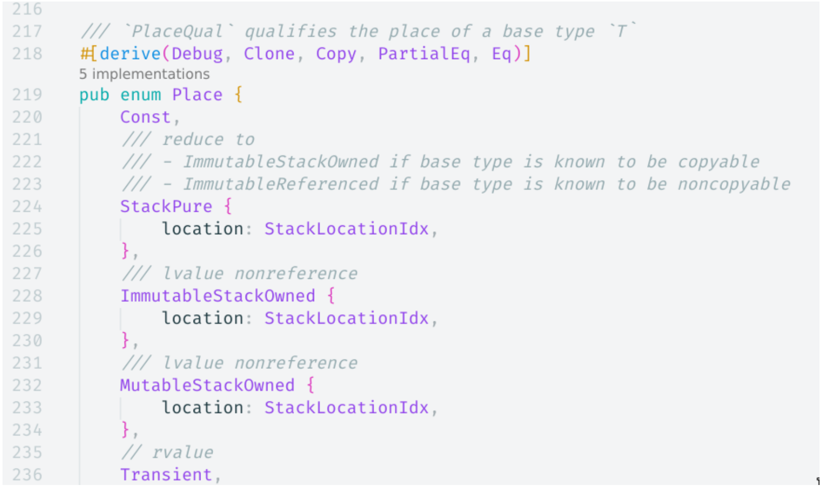
Place and At Type

Rust introduces the concept of lifetime, so does Husky, but Husky also introduces place.

Afterall, time and space is the same thing according to Einstein.

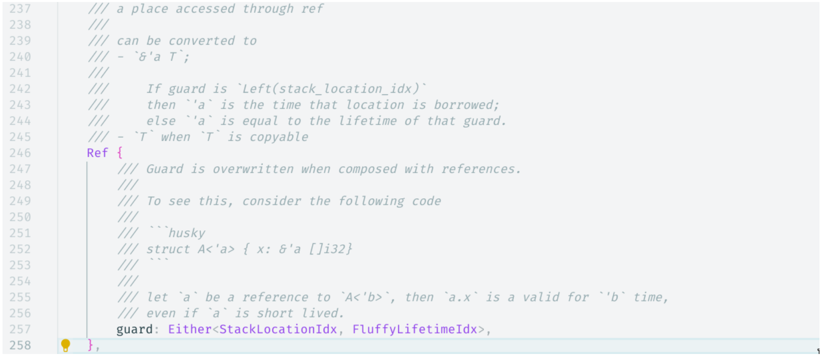
Special thanks to Bjarne Stroustrup. This is influenced by `const` lvalue and rvalue in C++.

Place and At Type: Rust Side Definition



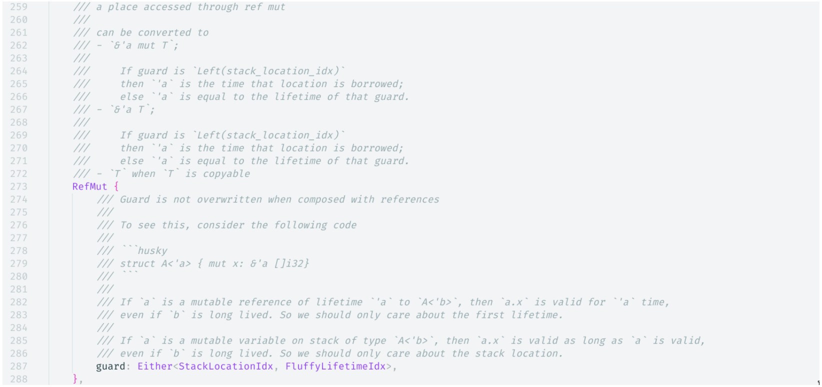
```rust

/// `PlaceQual` qualifies the place of a base type `T`
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
5 implementations
pub enum Place {
    Const,
    /// reduce to
    /// - ImmutableStackOwned if base type is known to be copyable
    /// - ImmutableReferenced if base type is known to be noncopyable
    StackPure {
        location: StackLocationIdx,
    },
    /// lvalue nonreference
    ImmutableStackOwned {
        location: StackLocationIdx,
    },
    /// lvalue nonreference
    MutableStackOwned {
        location: StackLocationIdx,
    },
    // rvalue
    Transient,
```

Place and At Type: Rust Side Definition Continued



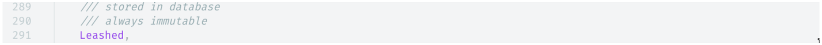
```
    /// a place accessed through ref
    ///
    /// can be converted to
    /// - `&'a T`;
    ///
    ///     If guard is `Left(stack_location_idx)`
    ///     then `'a` is the time that location is borrowed;
    ///     else `'a` is equal to the lifetime of that guard.
    /// - `T` when `T` is copyable
    Ref {
        /// Guard is overwritten when composed with references.
        ///
        /// To see this, consider the following code
        ///
        /// ```husky
        /// struct A<'a> { x: &'a []i32}
        /// ```
        ///
        /// let `a` be a reference to `A<'b>`, then `a.x` is a valid for `'b` time,
        /// even if `a` is short lived.
        guard: Either<StackLocationIdx, FluffyLifetimeIdx>,
    },
```

Place and At Type: Rust Side Definition Continued



```
    /// a place accessed through ref mut
    ///
    /// can be converted to
    /// - `&'a mut T`;
    ///
    ///     If guard is `Left(stack_location_idx)`
    ///     then `'a` is the time that location is borrowed;
    ///     else `'a` is equal to the lifetime of that guard.
    /// - `&'a T`;
    ///
    ///     If guard is `Left(stack_location_idx)`
    ///     then `'a` is the time that location is borrowed;
    ///     else `'a` is equal to the lifetime of that guard.
    /// - `T` when `T` is copyable
    RefMut {
        /// Guard is not overwritten when composed with references
        ///
        /// To see this, consider the following code
        ///
        /// ```husky
        /// struct A<'a> { mut x: &'a []i32}
        /// ```
        ///
        /// If `a` is a mutable reference of lifetime `'a` to `A<'b>`, then `a.x` is valid for `'a` time,
        /// even if `b` is long lived. So we should only care about the first lifetime.
        ///
        /// If `a` is a mutable variable on stack of type `A<'b>`, then `a.x` is valid as long as `a` is valid,
        /// even if `b` is long lived. So we should only care about the stack location.
        guard: Either<StackLocationIdx, FluffyLifetimeIdx>,
    },
```

Place and At Type: Rust Side Definition Continued

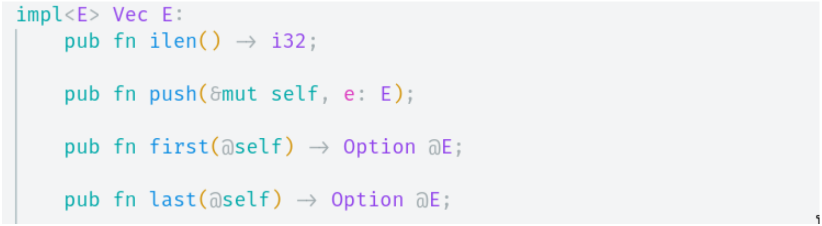


```
    /// stored in database
    /// always immutable
    Leashed,
```

This list is not exhaustive.
Place can include device information such as cpu/gpu.
`At` Type is simply a type together with a place. Written as `@'`α `t` where 'α is a label representing a place and `t` a type. It's customary to use Latin letter labels for lifetime and greek letter labels for place.

- it can be coerced implicitly from/to other types like leash, reference, mutable reference, etc, depending on specific place value.
- place can be generic, thus no need for `last_mut` like in Rust.

Generic Place

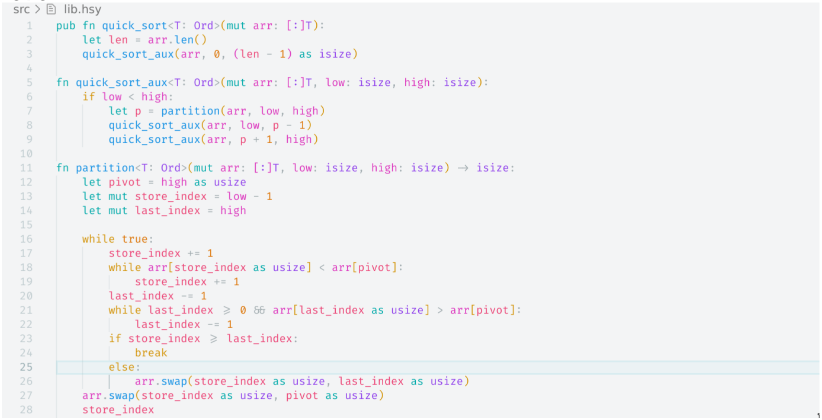


```
impl<E> Vec E:
    pub fn ilen() → i32;

    pub fn push(&mut self, e: E);

    pub fn first(@self) → Option @E;

    pub fn last(@self) → Option @E;
```

In the above, the method `first` and `last` has an implicit place template parameter. There is no need to define `first_mut` and `last_mut` as in Rust.

Place and At Type

At type is central to husky's type inference system. With it, one can write like python but the compiler will understand at a system level.



src > lib.hsy

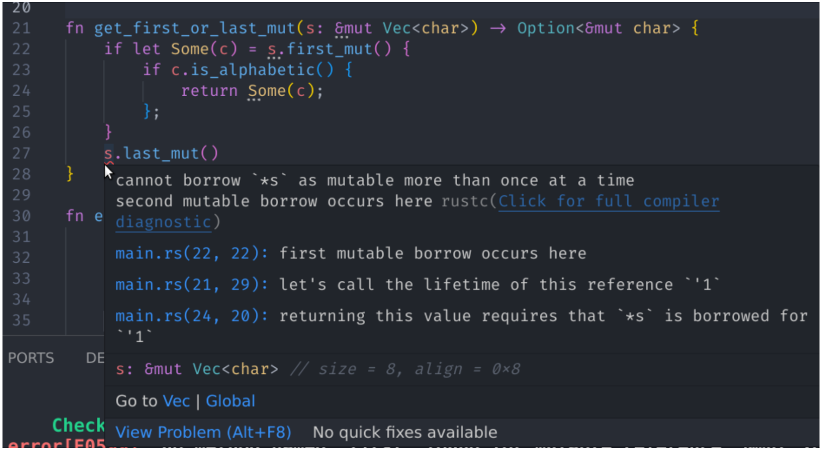
```
pub fn quick_sort<T: Ord>(mut arr: [:]T):
    let len = arr.len()
    quick_sort_aux(arr, 0, (len - 1) as isize)

fn quick_sort_aux<T: Ord>(mut arr: [:]T, low: isize, high: isize):
    if low < high:
        let p = partition(arr, low, high)
        quick_sort_aux(arr, low, p - 1)
        quick_sort_aux(arr, p + 1, high)

fn partition<T: Ord>(mut arr: [:]T, low: isize, high: isize) -> isize:
    let pivot = high as usize
    let mut store_index = low - 1
    let mut last_index = high

    while true:
        store_index += 1
        while arr[store_index as usize] < arr[pivot]:
            store_index += 1
        last_index -= 1
        while last_index >= 0 && arr[last_index as usize] > arr[pivot]:
            last_index -= 1
        if store_index >= last_index:
            break
        else:
            arr.swap(store_index as usize, last_index as usize)
    arr.swap(store_index as usize, pivot as usize)
    store_index
```

Place and At Type

An expression like `a.x` in Rust can be only understood as move or copy.
But Husky interprets the type of `a.x` as a at type, with its place derived from that of a.

Borrow Checking in Rust

In Rust, lifetime is analyzed “statically”.



```
fn get_first_or_last_mut(s: &mut Vec<char>) → Option<&mut char> {
    if let Some(c) = s.first_mut() {
        if c.is_alphabetic() {
            return Some(c);
        };
    }
    s.last_mut()
}

fn e
```

```
cannot borrow `*s` as mutable more than once at a time
second mutable borrow occurs here rustc(Click for full compiler
diagnostic)

main.rs(22, 22): first mutable borrow occurs here

main.rs(21, 29): let's call the lifetime of this reference `'1`

main.rs(24, 20): returning this value requires that `*s` is borrowed for
`'1`

s: &mut Vec<char> // size = 8, align = 0x8
```

Go to Vec | Global

View Problem (Alt+F8)   No quick fixes available

Borrow Checking in Husky

In Husky I've invented a 'dynamic' way of checking, enabling more safe code to pass borrow checking and maybe also more efficient.
However, I have seen a paper from Germany that looks like doing similar things.
In Husky, just collect all lifetime constraints and simply run a 'symbolic simulation' to detect lifetime conflicts, with special handles for branches and loops.

Monad

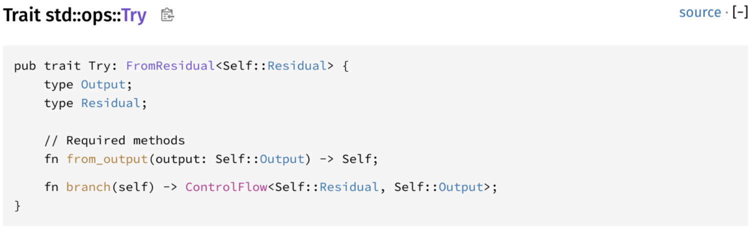
Monad is fun and useful.

- Functional way of defining Monad is too painful.
- Rust way of adopting Monad is much clearer, but can't handle effect and a little cumbersome to implement
- Husky simplifies Rust's way and combine with effectual system

Monad through Effect and Unveil

In Rust,

- effectual monads like IO are not introduced.
- branching monads like Result and Option, however, are realized through traits `std::ops::Try` and `std::ops::FromResidual`



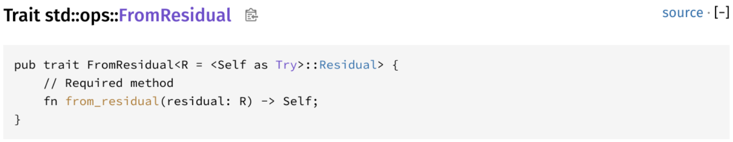
Trait std::ops::Try source · [-]

```
pub trait Try: FromResidual<Self::Residual> {
    type Output;
    type Residual;

    // Required methods
    fn from_output(output: Self::Output) -> Self;

    fn branch(self) -> ControlFlow<Self::Residual, Self::Output>;
}
```

Monad in Rust

In Rust,

- effectual monads like IO are not introduced.
- branching monads like Result and Option, however, are realized through traits `std::ops::Try` and `std::ops::FromResidual`


Trait std::ops::FromResidual

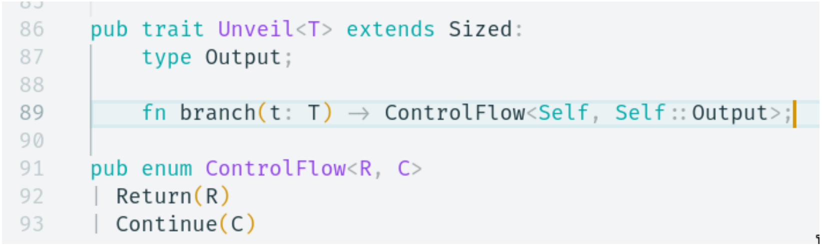
source · [-]

```
pub trait FromResidual<R = <Self as Try>::Residual> {
    // Required method
    fn from_residual(residual: R) -> Self;
}
```

Monad in Husky

In Husky, effects will be handled through attributes on functions and branching is realized through a single trait `core::ops::Unveil`.

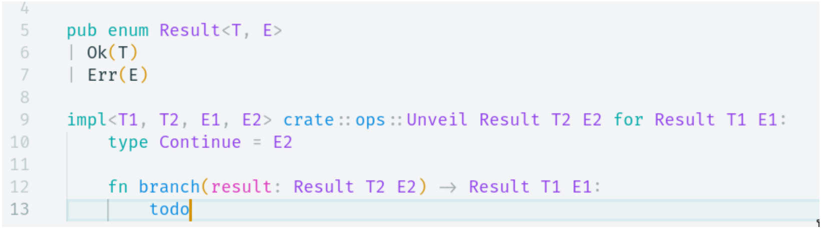
Type `A` implements `core::ops::Unveil B` means that if a function return `A`, then an expression of type `B` in its body can be unveiled.



```
pub trait Unveil<T> extends Sized:
    type Output;

    fn branch(t: T) -> ControlFlow<Self, Self::Output>;

pub enum ControlFlow<R, C>
| Return(R)
| Continue(C)
```

Monad Implementation

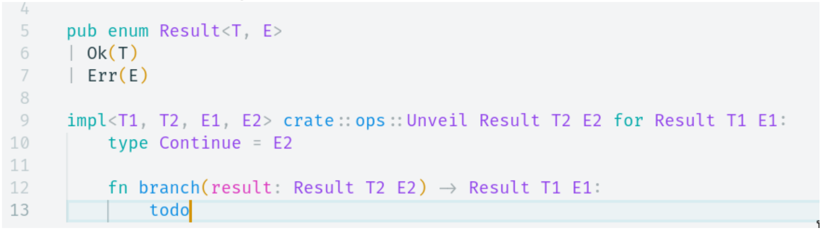
`Result` monad in Husky, from standard library (not complete yet, trait constraints are missing)



```

pub enum Result<T, E>
| Ok(T)
| Err(E)

impl<T1, T2, E1, E2> crate::ops::Unveil Result T2 E2 for Result T1 E1:
    type Continue = E2

    fn branch(result: Result T2 E2) -> Result T1 E1:
        todo
```

Monad Implementation

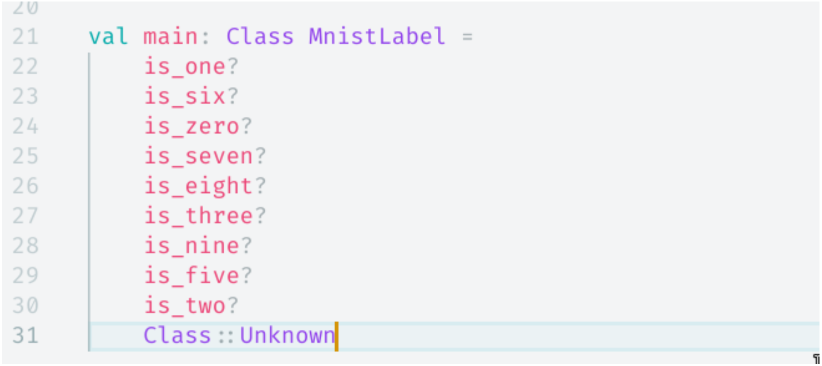
`Class` monad in Husky for classification tasks in machine learning, from custom package



```
pub enum Result<T, E>
| Ok(T)
| Err(E)

impl<T1, T2, E1, E2> crate::ops::Unveil Result T2 E2 for Result T1 E1:
    type Continue = E2

    fn branch(result: Result T2 E2) → Result T1 E1:
        todo
```

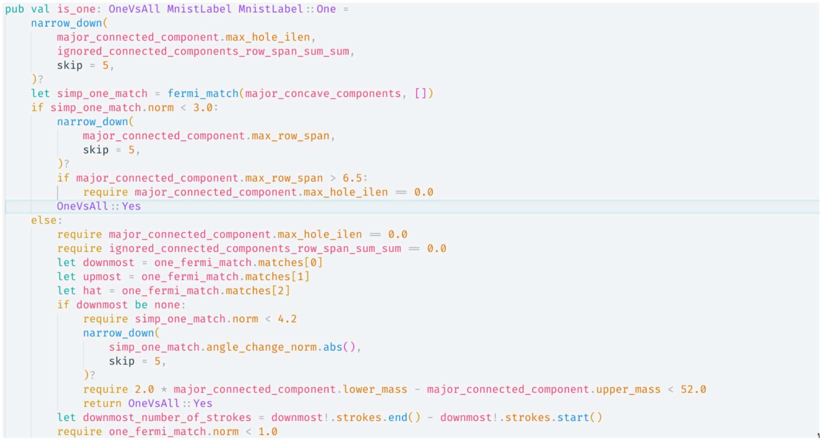
Monad in Action

In Husky, we use the same ? operator like in Rust to express branching



```
val main: Class MnistLabel =
    is_one?
    is_six?
    is_zero?
    is_seven?
    is_eight?
    is_three?
    is_nine?
    is_five?
    is_two?
    Class::Unknown
```

Monad in Action



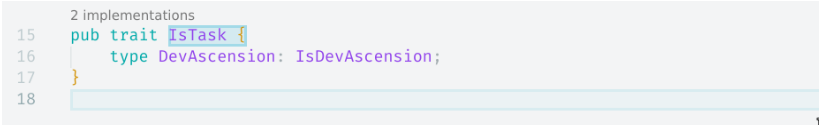
```
pub val is_one: OneVsAll MnistLabel MnistLabel::One =
    narrow_down(
        major_connected_component.max_hole_ilen,
        ignored_connected_components_row_span_sum_sum,
        skip = 5,
    )?
    let simp_one_match = fermi_match(major_concave_components, [])
    if simp_one_match.norm < 3.0:
        narrow_down(
            major_connected_component.max_row_span,
            skip = 5,
        )?
        if major_connected_component.max_row_span > 6.5:
            require major_connected_component.max_hole_ilen == 0.0
        OneVsAll::Yes
    else:
        require major_connected_component.max_hole_ilen == 0.0
        require ignored_connected_components_row_span_sum_sum == 0.0
        let downmost = one_fermi_match.matches[0]
        let upmost = one_fermi_match.matches[1]
        let hat = one_fermi_match.matches[2]
        if downmost be none:
            require simp_one_match.norm < 4.2
            narrow_down(
                simp_one_match.angle_change_norm.abs(),
                skip = 5,
            )?
            require 2.0 * major_connected_component.lower_mass - major_connected_component.upper_mass < 52.0
            return OneVsAll::Yes
        let downmost_number_of_strokes = downmost!.strokes.end() - downmost!.strokes.start()
        require one_fermi_match.norm < 1.0
```

More Advanced Type System

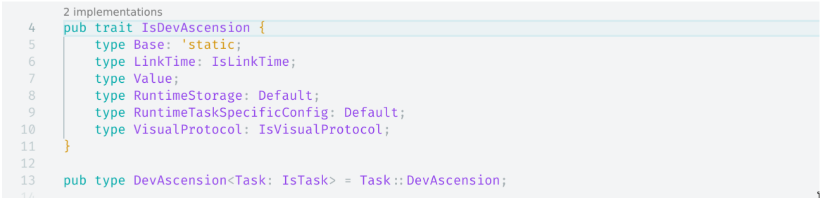
- Dependent type. The type of generic function is actually dependent type. This is needed to have type safe AI models.
- Theorem proving. There are plans for them. Lean proves that there is no need for separate language like Coq and Ocaml. ATS proves that system level programming can be combined with theorem proving. Husky is going to continue this line of work.

Task System

In Husky one can fully customize compilation, runtime, debugging and visualization.
A task is an instance of a type that satisfies `IsTask` trait. This trait exists both in Rust and in Husky.

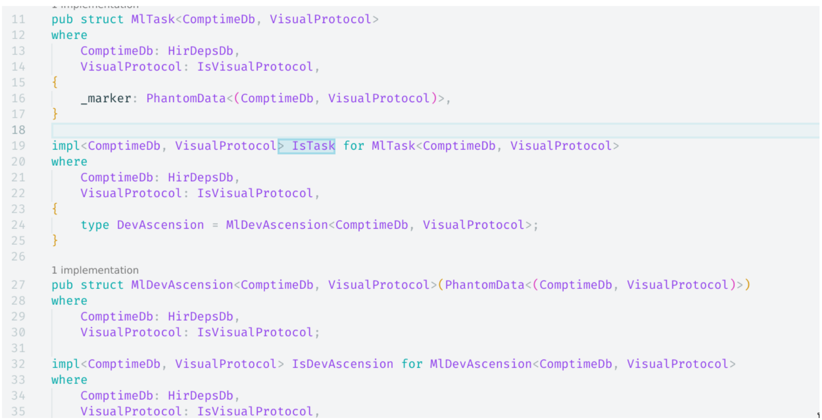


2 implementations
```
pub trait IsTask {
    type DevAscension: IsDevAscension;
}

```



2 implementations
```
pub trait IsDevAscension {
    type Base: 'static;
    type LinkTime: IsLinkTime;
    type Value;
    type RuntimeStorage: Default;
    type RuntimeTaskSpecificConfig: Default;
    type VisualProtocol: IsVisualProtocol;
}

pub type DevAscension<Task: IsTask> = Task::DevAscension;
```

Task Example



```
pub struct MlTask<ComptimeDb, VisualProtocol>
where
    ComptimeDb: HirDepsDb,
    VisualProtocol: IsVisualProtocol,
{
    _marker: PhantomData<(ComptimeDb, VisualProtocol)>,
}

impl<ComptimeDb, VisualProtocol> IsTask for MlTask<ComptimeDb, VisualProtocol>
where
    ComptimeDb: HirDepsDb,
    VisualProtocol: IsVisualProtocol,
{
    type DevAscension = MlDevAscension<ComptimeDb, VisualProtocol>;
}

1 implementation
pub struct MlDevAscension<ComptimeDb, VisualProtocol>(PhantomData<(ComptimeDb, VisualProtocol)>)
where
    ComptimeDb: HirDepsDb,
    VisualProtocol: IsVisualProtocol;

impl<ComptimeDb, VisualProtocol> IsDevAscension for MlDevAscension<ComptimeDb, VisualProtocol>
where
    ComptimeDb: HirDepsDb,
    VisualProtocol: IsVisualProtocol,
```

Task Specification

Task (type and instance) is specified in the root of an executable crate by a zero argument function named task.
For libraries, one might need to specify some traits the task type needs to satisfy.

Why So Much Trouble?

Different tasks require different strategies for compilation, evaluation, debugging. Husky will have very good development experience thanks to this.

- Mixture of interpretation and compilation saves incremental compilation speed by minimizing dependency tree.
- Certain task can have a top level computation graph, and husky support incremental evaluation so that values are automatically kept up to date with little computation. Say goodbye to jupyter notebook
- Support for custom memory allocation and reference strategies.

Husky Stands on Giants' Shoulders

The complicated design of Husky is built upon amazing previous works.

Ascension Paradigm

Functional Programming vs Procedural Programming

- Functional Programming
    - Close to logic
    - Easy to do high level optimization
      incremental computation, laziness, etc.
    - Hard to do low level optimization
- Procedural Programming
    - Close to hardware
    - Easy to do low level optimization
    - Hard to do high level optimization
      mutable state, loops, small vectors
- Compilers are not AGIs. How to get the best of both worlds?

Combining Functional and Procedural Programming

Past efforts are

- Rust salsa. incremental computation. macro heavy.
- Every deep learning framework.
- Reactive programming for UI programming. Svelte, Elm.

In Husky, ascension paradigm is a general way of combining functional and procedural programming. It can express salsa and svelte as special cases.

salsa and deep learning frameworks are just computation graphs, but reactive programming is something more. In Husky's perspective, the latter requires ascension, the former does not.

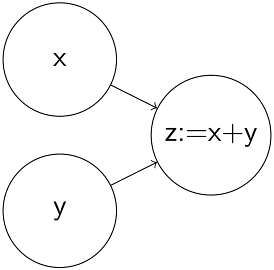
Mathematical Definition of Computation Graph

Let G be a directed graph with vertices V and edges E.
For any $v \in V$, assign a variable x_v of type t_v.
For each source vertex v, x_v is interpreted as input.
For each nonsource vertex v with n incoming vertices $v_1, \cdots, v_n$, we add a relationship

$$x_v = f_v(x_{v_1}, \cdots, x_{v_n})$$

for a function f_v of type $t_{v_1} \to \cdots \to t_{v_n} \to t_v$.

Trivial Example of Computation Graph



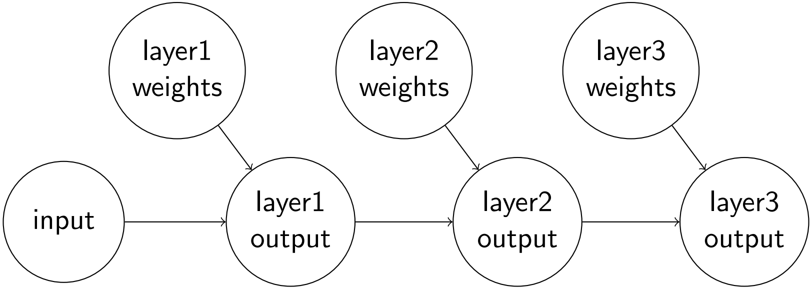
Here $f_v(x, y) = x + y$.

Salsa Example of Computation Graph

$$\begin{tikzcd}\text{input} \arrow[r] & \text{tokens} \arrow[r] & \text{asts} \arrow[r] & \text{typed asts}\end{tikzcd}$$

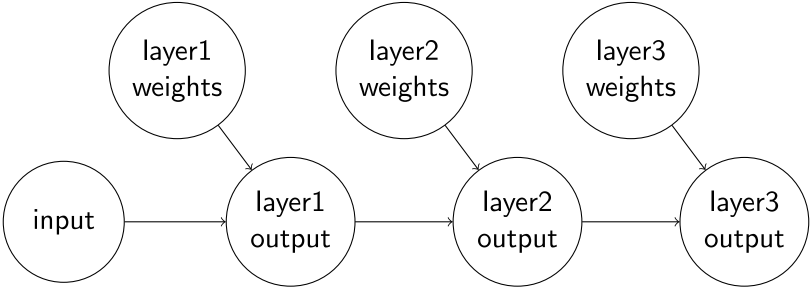
In salsa, f_v are given by Rust functions, compilable to be executed efficiently.

Deep Learning Framework Example of Computation Graph



In deep learning frameworks, t_v is tensor of certain shape, f_v are differentiable tensor operations that can be composed and optimized together.
Here the computation graph describes the inference process, and leaves training to gradient descent because f_v are differentiable.

Deep Learning Framework Example of Computation Graph



In Husky, we are going to have a generalized computation graph that describes inference and training together, with t_v possibly non tensor types and f_v not necessarily differentiable.

A Math Example for Ascension

Let $t \in \mathbb{R}$. t is a placeholder
Let $x = 2t + 1$. the type of x can be $\mathbb{R}$ or $\mathbb{R} \to \mathbb{R}$
Let $u = \int_{-1}^{1} x dt$. the type of x is interpreted as $\mathbb{R} \to \mathbb{R}$

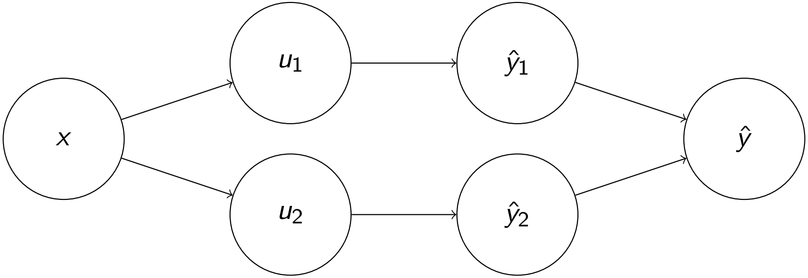
The act of reinterpreting the type of x from $\mathbb{R}$ to $\mathbb{R} \to \mathbb{R}$ is ascension.
In general, a variable of type t can be reinterpreted as of type $t_1 \to \cdots t_n \to t$ where t_i are types of some placeholders.
We denote $\mathscr{F} t := t_1 \to \cdots t_n \to t$, a covariant functor obviously.

Machine Learning Model Selection

Let $\mathcal{X}$ be input space, and $\mathcal{Y}$ be output space, machine learning is about approximating a function from $\mathcal{X}$ to $\mathcal{Y}$ by fitting a dataset of size N $\mathcal{D} = \{(x_i, y_i) : i \in [N]\}$. Let f_1, f_2 be feature maps from $\mathcal{X}$ to $\mathbb{R}$. For simplicity, let $\mathcal{Y} = \mathbb{R}$.

Let $x \in \mathcal{X}$. x is a placeholder
Let $u_1 = f_1(x)$. the type of u_1 can be $\mathbb{R}$ or $\mathcal{X} \to \mathbb{R}$
Let $\hat{y}_1 = \alpha_1 u_1 + \beta_1$, where α_1, β_1 are from least squares.
Let $u_2 = f_2(x)$. the type of u_2 can be $\mathbb{R}$ or $\mathcal{X} \to \mathbb{R}$
Let $\hat{y}_2 = \alpha_2 u_2 + \beta_2$, where α_1, β_1 are from least squares.
Let $\hat{y} = \hat{y}_1$ or $\hat{y}_2$ depending on which one has better loss.

Machine Learning Model Selection


$$\begin{tikzcd} & u_1 \arrow[r] & \hat{y}_1 \arrow[dr] & \\ x \arrow[ur] \arrow[dr] & & & \hat{y} \\ & u_2 \arrow[r] & \hat{y}_2 \arrow[ur] & \end{tikzcd}$$

Without ascension, the above is not a computation graph.

Machine Learning Model Selection

In husky, one can write in a way very close to math,

```
// defines the output node
// keyword 'val' is for defining an item
// that represents a node
val main: f32 =
    let u1 = f1(input)
    let y1 = linear_regression(u1)
    let u2 = f2(input)
    let y2 = linear_regression(u2)
    return select(y1, y2)
```

In the above, `f1`, `f2` are just normal functions, but `linear_regression` and `select` are ‘ascended beings’, called generative functions.

Generative Function

The name comes from the sad reality that ‘generic function’ and ‘generalized function’ or even ‘general function’ already have meanings. But it somehow makes sense because every ‘time’ a generative function is called, a normal function or a constant is generated.

```
// keyword 'gn' denotes generative function
// can be defined through FFI
#[rust_ffi(...)]
gn sum(x: f32) -> f32;
// or be defined through composition
gn ridge_regression_composite(x: f32) -> f32 =
    let y1 = ridge_regression(x, lambda = 0.1)
    let y2 = ridge_regression(x, lambda = 0.2)
    return select(y1, y2)
```

Type Theory for Generative Function in Machine Learning

Let C be the type of contexts containing training dataset and configurations.
Feature of type T is defined by

$$\mathscr{F}T := C \to \text{Input} \to T$$

.
Given a context c, and an input x, it should provide a value that is the feature trained over c and then evaluated on x.

Type Theory for Generative Function in Machine Learning

A generative function

$$\mathrm{gn}(X_1, \cdots, X_n) \to Y$$

can be defined in a coarse way as

$$\mathscr{F} X_1 \to \cdots \to \mathscr{F} X_n \to \mathscr{F} Y, \tag{1}$$

Type Theory for Generative Function in Machine Learning

A generative function

$$\mathrm{gn}(X_1, \cdots, X_n; \tilde{X}_1, \cdots, \tilde{X}_m) \to Y$$

where X_i are normal inputs, and $\tilde{X}_i$ are training-time inputs, can be defined in a more refined way as

$$\begin{aligned} C \to \underbrace{\mathscr{F} X_1 \to \cdots \to \mathscr{F} X_n}_{\text{all-time inputs for training}} \to \underbrace{\mathscr{F} \tilde{X}_1 \to \cdots \to \mathscr{F} \tilde{X}_n}_{\text{training-time inputs}} \\ \to \underbrace{X_1 \to \cdots \to X_n}_{\text{all-time inputs for training}} \to Y \end{aligned} \tag{2}$$

.

Trivally this can be viewed as a subtype of the previous type.

Type Theory for Generative Function in Machine Learning

With all this complexity, the actual type checking is quite simple!
Not much different from checking normal functions.
I never give up on the efficiency of compiler itself!

Ascension: More Advanced Topics

The actual implementation of Husky considers more advanced aspects, including

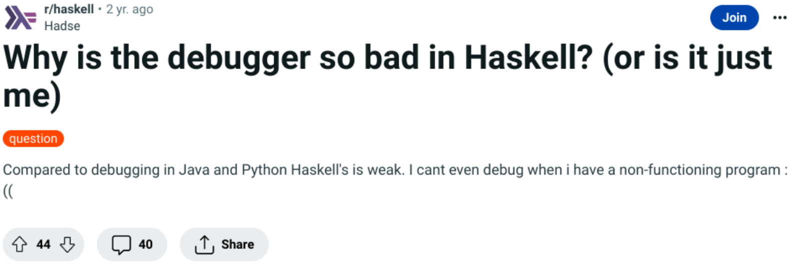
- Partially defined features. Husky has ADTs, so often a feature can only be defined for a portion of the dataset, this is naturally handled by automatically filtering of the dataset.
- Runtime efficiency. Development runtime efficiency is guaranteed by laziness. Release runtime efficiency is guaranteed by compilation, removing unnecessary nodes.
- Memory safety.
- Compile time constants and generative functions.
- Ascension in greater generality, including UI, game development, Reinforcement Learning, meta learning, etc.
- I actually get a lot of inspiration from algebraic geometry.

Ascension: More Advanced Topics

Will be covered in future lectures!

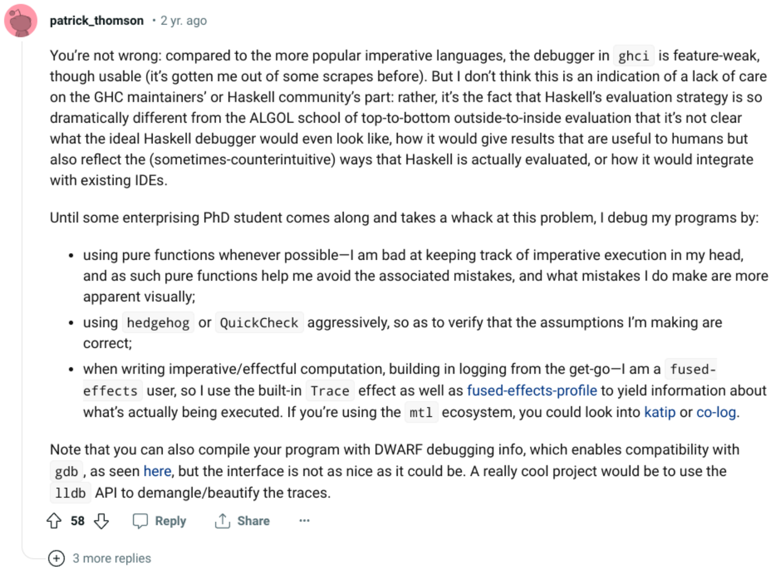
Debugging System

Debugger Bad in Haskell



Why is the debugger so bad in Haskell? (or is it just me)

Compared to debugging in Java and Python Haskell's is weak. I cant even debug when i have a non-functioning program :((



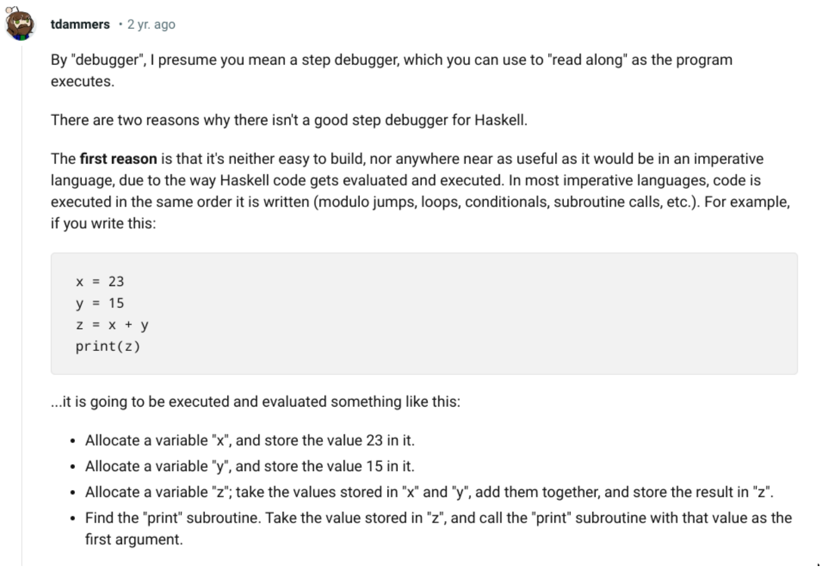
patrick_thomson • 2 yr. ago

You're not wrong: compared to the more popular imperative languages, the debugger in `ghci` is feature-weak, though usable (it's gotten me out of some scrapes before). But I don't think this is an indication of a lack of care on the GHC maintainers' or Haskell community's part: rather, it's the fact that Haskell's evaluation strategy is so dramatically different from the ALGOL school of top-to-bottom outside-to-inside evaluation that it's not clear what the ideal Haskell debugger would even look like, how it would give results that are useful to humans but also reflect the (sometimes-counterintuitive) ways that Haskell is actually evaluated, or how it would integrate with existing IDEs.

Until some enterprising PhD student comes along and takes a whack at this problem, I debug my programs by:

- using pure functions whenever possible—I am bad at keeping track of imperative execution in my head, and as such pure functions help me avoid the associated mistakes, and what mistakes I do make are more apparent visually;
- using `hedgehog` or `QuickCheck` aggressively, so as to verify that the assumptions I'm making are correct;
- when writing imperative/effectful computation, building in logging from the get-go—I am a `fused-effects` user, so I use the built-in `Trace` effect as well as fused-effects-profile to yield information about what's actually being executed. If you're using the `mtl` ecosystem, you could look into katip or co-log.

Note that you can also compile your program with DWARF debugging info, which enables compatibility with `gdb`, as seen here, but the interface is not as nice as it could be. A really cool project would be to use the `lldb` API to demangle/beautify the traces.



tdammers • 2 yr. ago

By "debugger", I presume you mean a step debugger, which you can use to "read along" as the program executes.

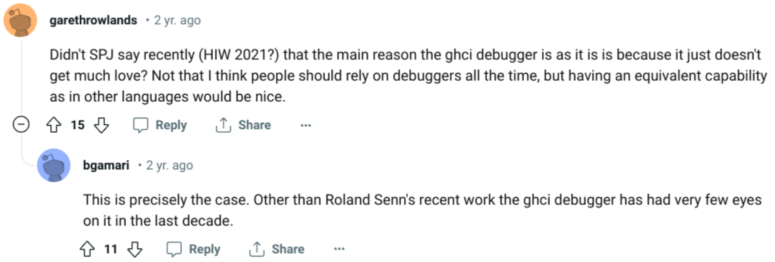
There are two reasons why there isn't a good step debugger for Haskell.

The **first reason** is that it's neither easy to build, nor anywhere near as useful as it would be in an imperative language, due to the way Haskell code gets evaluated and executed. In most imperative languages, code is executed in the same order it is written (modulo jumps, loops, conditionals, subroutine calls, etc.). For example, if you write this:

```
x = 23
y = 15
z = x + y
print(z)
```

...it is going to be executed and evaluated something like this:

- Allocate a variable "x", and store the value 23 in it.
- Allocate a variable "y", and store the value 15 in it.
- Allocate a variable "z"; take the values stored in "x" and "y", add them together, and store the result in "z".
- Find the "print" subroutine. Take the value stored in "z", and call the "print" subroutine with that value as the first argument.



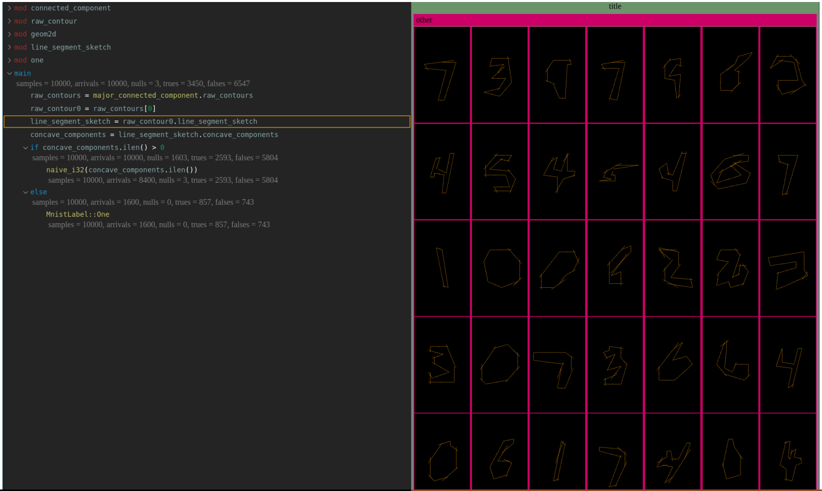
garethrowlands · 2 yr. ago

Didn't SPJ say recently (HIW 2021?) that the main reason the ghci debugger is as it is is because it just doesn't get much love? Not that I think people should rely on debuggers all the time, but having an equivalent capability as in other languages would be nice.

bgamari · 2 yr. ago

This is precisely the case. Other than Roland Senn's recent work the ghci debugger has had very few eyes on it in the last decade.

Husky Debugging System: Expanding the computation graph into a Trace Tree



```
> mod connected_component
> mod raw_contour
> mod geom2d
> mod line_segment_sketch
> mod one
v main
  samples = 10000, arrivals = 10000, nulls = 3, trues = 3450, falses = 6547
      raw_contours = major_connected_component.raw_contours
      raw_contour0 = raw_contours[0]
      line_segment_sketch = raw_contour0.line_segment_sketch
      concave_components = line_segment_sketch.concave_components
    v if concave_components.ilen() > 0
      samples = 10000, arrivals = 10000, nulls = 1603, trues = 2593, falses = 5804
          naive_i32(concave_components.ilen())
          samples = 10000, arrivals = 8400, nulls = 3, trues = 2593, falses = 5804
    v else
      samples = 10000, arrivals = 1600, nulls = 0, trues = 857, falses = 743
          MnistLabel::One
          samples = 10000, arrivals = 1600, nulls = 0, trues = 857, falses = 743
```