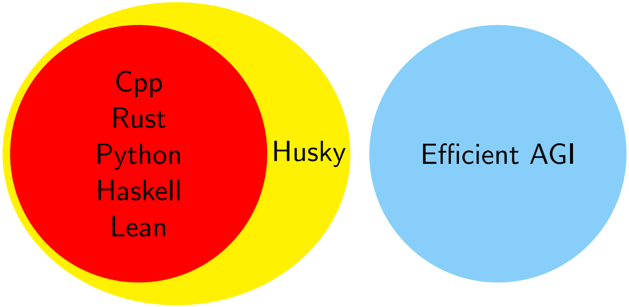
An Overview of the Husky Programming Language

Xiyu Zhai

Introduction

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



What are these 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,

- modular and domain specific
- models contains operations that are not matrix operations
- more efficient than deep learning only, computationally 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 has a novel programming paradigm called ascension
- it has a powerful debugging system
- it merges features from modern regular languages including C/C++, Rust, python, Haskell, Lean, ATS, etc.
  - its name 'Husky' comes from 'Haskell' and 'Rust' and 'python'
  - its file extension 'hsy' comes from 'hs' (Haskell) 'rs' (Rust) and 'py' (python)
- it's implemented by one person with high focus on development efficiency and ergonomity, everything is optimized for incremental computation, including syntax, semantics, ide support, compilation, evalution.

What is Husky

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

- it has a novel programming paradigm called ascension
    - designed for

What is Husky

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

- it has a novel programming paradigm called ascension
- it has a powerful debugging system
    - 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’
- it’s implemented by one person with high focus on development efficient, everything is optimized for incremental computation, including syntax, semantics, ide support, compilation, evalution.

What is Husky

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

- it has a novel programming paradigm called ascension
- it has a powerful debugging system
- it merges features from modern regular languages including C/C++, Rust, python, Haskell, Lean, ATS, etc.
    - its name ‘Husky’ comes from ‘Haskell’ and ‘Rust’ and ‘python’
    - its file extension ‘hsy’ comes from ‘hs’ (Haskell) ‘rs’ (Rust) and ‘py’ (python)

An Ambitious Project

All the following is designed and implemented by one person to play well together and to be optimized for incremental development and customizable for specific coding tasks:

- Syntax
- Semantics
  Type System, Attributes, Compiler Magic, Paradigm, ...
- IDE support
  semantic token, hover, completion, ...
- Computation
  Compiler, Interpreter, Runtime
- Debugger

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 enable by playing with
  - Being 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

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

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.

Future

Notations

Notations: Type Theory

A type is basically a set.

Notations: Curry

X, Y, Z are types.
$X \to Y$ denotes a function from X to Y.
$X \to Y \to Z$ denotes a function from X to $Y \to Z$ because $\to$ is right associative.
Note that $(X, Y) \to Z$ and $X \to Y \to Z$ are "equivalent".

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.

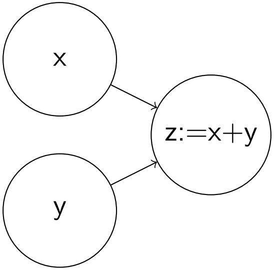
Mathematical Definition of Computation Graph

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

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

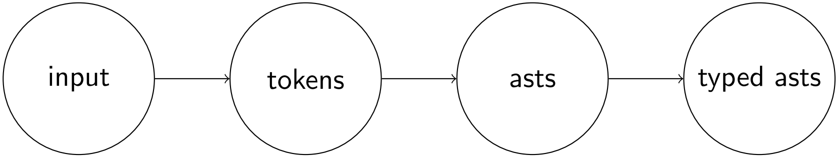
for a function f_v of type $t_{v_1} \to \cdots \to t_{v_n} \to t_v$.

Trivial Example of Computation Graph



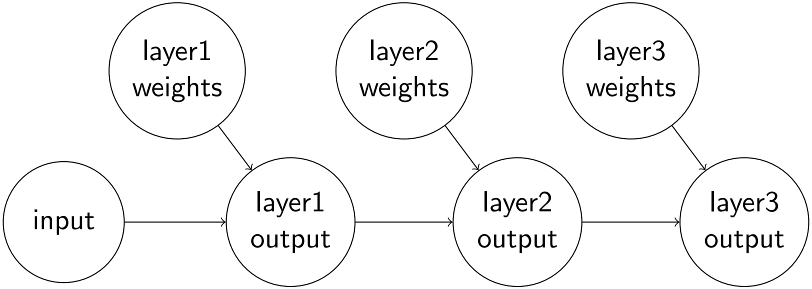
Here $f_v(x, y) = x + y$.

Salsa Example of Computation Graph



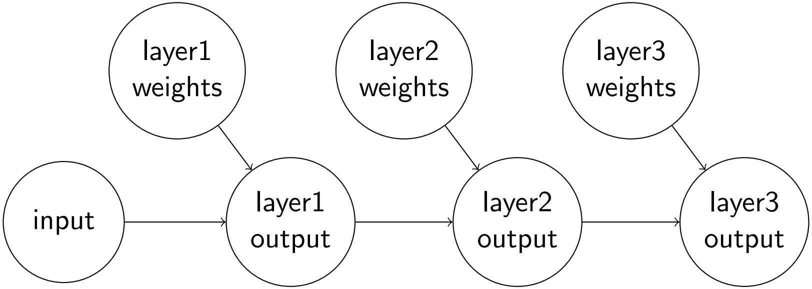
In salsa, f_v are given by Rust functions, compilable to be executed efficiently.

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 $\mathbb{R} \to \mathbb{R}$

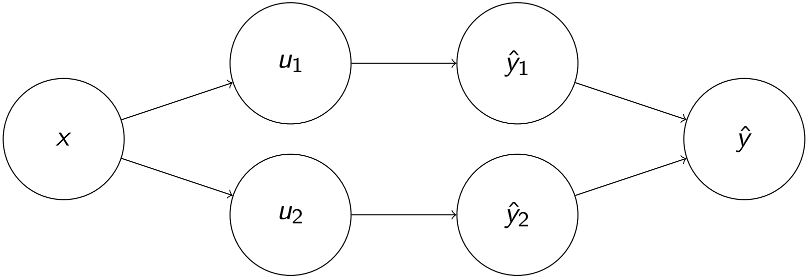
The act of reinterpreting the type of x from $\mathbb{R}$ to $\mathbb{R} \to \mathbb{R}$ is ascension.
In general, a variable of type t can be reinterpreted as of type $t_1 \to \cdots t_n \to t$ where t_i are types of some placeholders.
We denote $\mathscr{F} t := t_1 \to \cdots t_n \to t$, a covariant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

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

.
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!

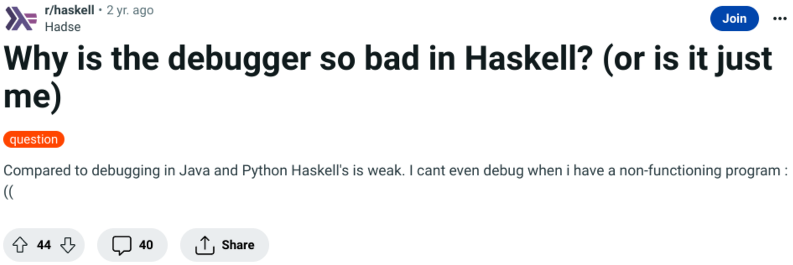
Debugging System

Lessons from Haskell: Debugging Hell

- Monad is hard to debug. Mixing effect and control flow, too concise to be easily understood.

test frame for section one

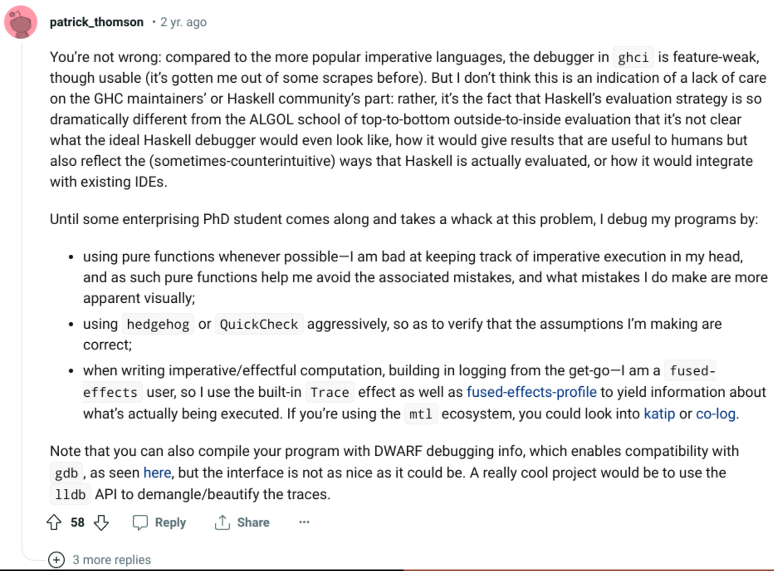
Debugger Bad in Haskell


r/haskell • 2 yr. ago
Hadse

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

question

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



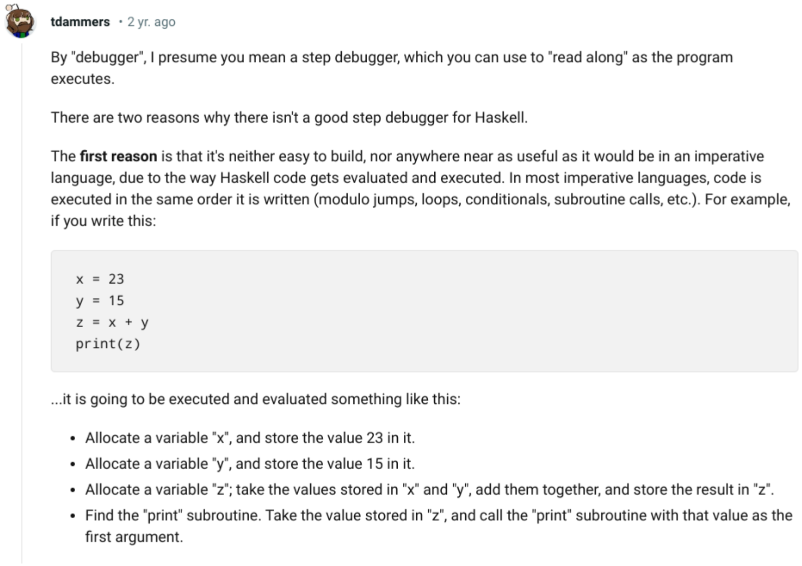
patrick_thomson • 2 yr. ago

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.

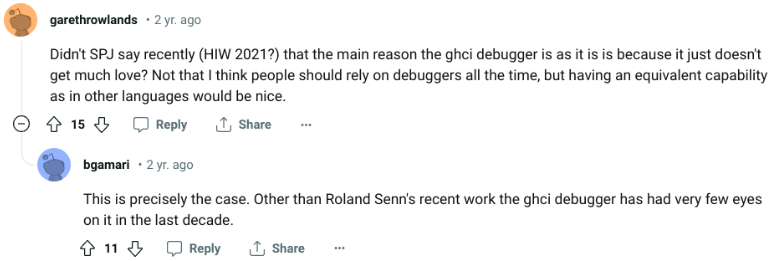
There are two reasons why there isn't a good step debugger for Haskell.

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.

Regular Feautures

Regular Features

This section we will discuss the regular features of Husky.

- functions
- methods
- values
- type definitions

Item

Type Definition

One can define

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

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.

Variance

Basically the same as Rust.
Of course, having variance makes type inference more complicated, but I have the time.

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.

Place and At Type

For linear link list, Husky can do without unsafe, but Rust can't.

Borrow Checking

In Rust, lifetime is tied to a region, which limits its expressive power. In Husky I've invented a better way of checking, maybe also more efficient, enabling more safe code to pass borrow checking. However, I have seen a paper from Germany that looks like doing similar things.

In Husky, I just collect all lifetime constraints and simply run a 'symbolic simulation' to detect lifetime conflicts, with special handles for branches and loops. Unfortunately, they are implemented only in the old C++ codebase, I've not yet got time to do that in the current Rust codebase.

Decorators

must_use, no_discard

Monad through Effect and Unveil

Incremental Code Analysis

Regular Features: Incremental Compilation

Regular Features: Affine Type

Regular Features: Lifetime and Place

Regular Features: Generics

Regular Features: Dependent Types

Development Progress

Agility

- 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.

Solo Project

So far it has been only me.

-

Community

The community should be kept very small for the language development to be agile.

- AI researchers.
- Game developers.
- UI
- students