WIKIPEDIA

Idris (programming language)

Idris is a purely-functional programming language with dependent types, optional lazy evaluation, and features such as a totality checker. Idris may be used as a proof assistant, but it is designed to be a general-purpose programming language similar to Haskell.

The Idris type system is similar to Agda's, and proofs are similar to Coq's, including tactics (theorem proving functions/procedures) via elaborator reflection.[6] Compared to Agda and Coq, Idris prioritizes management of side effects and support for embedded domain-specific languages. Idris compiles to C (relying on a custom copying garbage collector using Cheney's algorithm) and JavaScript (both browser- and Node.js-based). There are third-party code generators for other platforms, including JVM, CIL, and LLVM.[7]

Idris is named after a singing dragon from the 1970s UK children's television program *Ivor the Engine*.[8]

Idris

Paradigm	Functional
Designed by	Edwin Brady
First appeared	2007[1]
Stable release	1.3.3[2] / May 24, 2020
Preview release	0.3.0 (Idris 2)[3] / January 13, 2021
OS	Cross-platform
License	BSD
Filename extensions	.idr, .lidr
Website	idris-lang.org (http://idris-lang.org)
Influenced by	
Agda, Clean,[4] Coq,[5] Epigram, F#, Haskell,[5] ML,[5] Rust[4]	

Contents

Features

Idris combines a number of features from relatively mainstream functional programming languages with features borrowed from proof assistants.

Functional programming

The syntax of Idris shows many similarities with that of Haskell. A hello world program in Idris might look like this:

```
module Main
```

```
main : IO ()
main = putStrLn "Hello, World!"
```

The only differences between this program and its Haskell equivalent are the single (instead of double) colon in the type signature of the main function, and the omission of the word "where" in the module declaration.[9]

Inductive and parametric data types

Idris supports inductively-defined data types and parametric polymorphism. Such types can be defined both in traditional "Haskell98"-like syntax:

```
data Tree a = Node (Tree a) (Tree a) | Leaf a
```

or in the more general GADT-like syntax:

```
data Tree : Type -> Type where
    Node : Tree a -> Tree a -> Tree a
    Leaf : a -> Tree a
```

Dependent types

With dependent types, it is possible for values to appear in the types; in effect, any value-level computation can be performed during typechecking. The following defines a type of lists whose lengths are known before the program runs, traditionally called vectors:

```
data Vect : Nat -> Type -> Type where
  Nil   : Vect 0 a
  (::) : (x : a) -> (xs : Vect n a) -> Vect (n + 1) a
```

This type can be used as follows:

```
total
append : Vect n a -> Vect m a -> Vect (n + m) a
append Nil       ys = ys
append (x :: xs) ys = x :: append xs ys
```

The functions append a vector of m elements of type a to a vector of n elements of type a. Since the precise types of the input vectors depend on a value, it is possible to be certain at compile-time that the resulting vector will have exactly (n + m) elements of type a. The word "total" invokes the totality checker which will report an error if the function doesn't cover all possible cases or cannot be (automatically) proven not to enter an infinite loop.

Another common example is pairwise addition of two vectors that are parameterized over their length:

```
total
pairAdd : Num a => Vect n a -> Vect n a -> Vect n a
pairAdd Nil       Nil       = Nil
pairAdd (x :: xs) (y :: ys) = x + y :: pairAdd xs ys
```

Num a signifies that the type a belongs to the type class Num. Note that this function still typechecks successfully as total, even though there is no case matching Nil in one vector and a number in the other. Since both vectors are ensured by the type system to have exactly the same length, we can be sure at compile time that this case will not occur. Hence it does not need to be mentioned for the function to be total.

Proof assistant features

Dependent types are powerful enough to encode most properties of programs, and an Idris program can prove invariants at compile-time. This makes Idris into a proof assistant.

There are two standard ways of interacting with proof assistants: by writing a series of tactic invocations (Coq style), or by interactively elaborating a proof term (Epigram/Agda style). Idris supports both modes of interaction, although the set of available tactics is not yet as useful as that of Coq.

Code generation

Because Idris contains a proof assistant, Idris programs can be written to pass proofs around. If treated naïvely, such proofs remain around at runtime. Idris aims to avoid this pitfall by aggressively erasing unused terms.[10][11]

By default, Idris generates native code through C. The other officially supported backend generates JavaScript.

Idris 2

Idris 2 is a new self-hosted version of the language which deeply integrates a linear type system, based on quantitative type theory. It currently compiles to Scheme and C.[12]

See also

- Total functional programming

References

1. Brady, Edwin (12 December 2007). "Index of /~eb/darcs/Idris" (https://web.archive.org/web/20080320233322/http://www-fp.cs.st-and.ac.uk/~eb/darcs/Idris/). *University of St Andrews School of Computer Science*. Archived from the original (http://www-fp.cs.st-and.ac.uk/~eb/darcs/Idris/) on 2008-03-20.
2. "Release 1.3.3" (https://github.com/idris-lang/Idris-dev/releases/tag/v1.3.3/). Retrieved 2020-05-25.
3. "Idris 2 version 0.3.0 Released" (https://www.idris-lang.org/idris-2-version-030-released.html). *www.idris-lang.org*. Retrieved 2021-03-17.
4. "Uniqueness Types" (http://docs.idris-lang.org/en/latest/reference/uniqueness-types.html). *Idris 1.3.1 Documentation*. Retrieved 2019-09-26.
5. "Idris, a language with dependent types" (http://www.idris-lang.org/). Retrieved 2014-10-26.
6. "Elaborator Reflection — Idris 1.3.2 documentation" (https://docs.idris-lang.org/en/v1.3.2/reference/elaborator-reflection.html). Retrieved 27 April 2020.
7. "Code Generation Targets — Idris 1.1.1 documentation" (http://docs.idris-lang.org/en/latest/reference/codegen.html). *docs.idris-lang.org*.

8. "Frequently Asked Questions" (http://docs.idris-lang.org/en/latest/faq/faq.html#what-does-the-name-idris-mean). Retrieved 2015-07-19.
9. "Syntax Guide — Idris 1.3.2 documentation" (https://docs.idris-lang.org/en/v1.3.2/reference/syntax-guide.html). Retrieved 27 April 2020.
10. "Erasure By Usage Analysis — Idris 1.1.1 documentation" (http://idris.readthedocs.org/en/latest/reference/erasure.html). *idris.readthedocs.org*.
11. "Benchmark results" (http://ziman.functor.sk/erasure-bm/). *ziman.functor.sk*.
12. "idris-lang/Idris2" (https://github.com/idris-lang/Idris2). *GitHub*. Retrieved 2021-04-11.

External links

- The Idris homepage (http://idris-lang.org/), including documentation, frequently asked questions and examples
- Idris at the Hackage repository (http://hackage.haskell.org/package/idris)
- Documentation for the Idris Language (tutorial, language reference, etc.) (http://docs.idris-lang.org/en/latest/index.html)