Idris (programming language)

Idris is a general-purpose purely functional programming language with dependent types. The type system is similar to the one used by Agda.

The language supports interactive theorem-proving comparable to Coq, including tactics, while the focus remains on general-purpose programming even before theorem-proving. Other goals of Idris are “sufficient” performance, easy management of side-effects and support for implementing embedded domain specific languages.

As of October 2014, Idris compiles to C and relies on a custom copying garbage collector using Cheney’s algorithm. There also exist JavaScript and Java backends, and a partial LLVM backend.[3]

The name Idris goes back to the character of the singing dragon in the 1970s UK kids’ program Ivor the Engine.[4]

1 Features

Idris combines a number of features from relatively mainstream functional programming languages with features borrowed from proof assistants, in effect blurring the boundary between the two kinds of software.

1.1 Functional programming

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

```haskell
module Main where
main = putStrLn "Hello, World!"
```

The only differences between this program and its Haskell equivalent are the single colon (instead of two) in the signature of the main function and the omission of the word “where” in the module declaration.

1.1.1 Inductive and parametric data types

Like most modern functional programming languages, Idris supports a notion of inductively-defined data type and parametric polymorphism. Such types can be defined both in traditional “Haskell98” syntax:

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

or in the more general GADT syntax:

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

1.1.2 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 of statically known length, traditionally called ‘vectors’:

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

This type can be used as follows:

```haskell
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 to not enter an infinite loop.

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

```haskell
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.
1.2 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.

1.3 Code generation

Because Idris contains a proof assistant, Idris programs can be written to pass proofs around. If treated naively, such proofs remain around at runtime. Idris aims to avoid this pitfall by aggressively erasing unused terms,[5] with promising results.[6]

By default, Idris generates native code by going through C. Other backends are available for generating JavaScript and Java.

2 See also

- Total functional programming

3 References

[1] “Idris 0.99 released”. 2016-12-03.

4 External links

- The Idris homepage, including documentation, frequently asked questions and examples
- Idris at the Hackage repository
- Documentation for the Idris Language (tutorial, language reference, etc.)
5  Text and image sources, contributors, and licenses

5.1  Text
- **Idris (programming language)**  
  *Contributors:* Tea2min, Raboof, Bywhite, Ori Livneh, Jec, DouglasCalvert, Msnicki, Artemis11, Matěj Grabovský, Yobot, Wickorama, FrescoBot, Rexen, Jshholland, Tredontho, Jonas.betzendahl, Veikk0ma, Soimort, Mister Mormon, HectorAE, Comp.arch, Chai-alice, AviendaBG, Daejo, Chamimi2, Cofunctor, Tchajed and Anonymous: 17

5.2  Images
- **File:Folder_Hexagonal_Icon.svg**  
  *License:* Cc-by-sa-3.0  
  *Contributors:* ?  
  *Original artist:* ?
- **File:Symbol_list_class.svg**  
  *Source:* https://upload.wikimedia.org/wikipedia/en/d/db/Symbol_list_class.svg  
  *License:* Public domain  
  *Contributors:* ?  
  *Original artist:* ?
- **File:Symbol_neutral_vote.svg**  
  *License:* Public domain  
  *Contributors:* ?  
  *Original artist:* ?

5.3  Content license
- Creative Commons Attribution-Share Alike 3.0